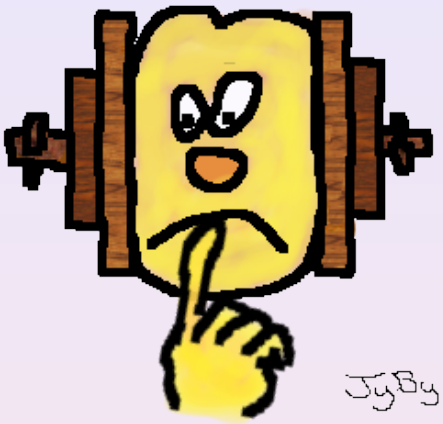
Estructuras de Datos Compactas



Gonzalo Navarro

www.dcc.uchile.cl/gnavarro

gnavarro@dcc.uchile.cl

Departamento de Ciencias de la Computación (DCC)

Universidad de Chile

Sponsors:

fcfm Ciencias de la Computación FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS UNIVERSIDAD DE CHILE

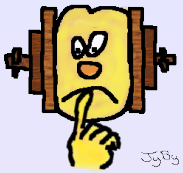
cwr.cl centerforwebresearch

YAHOO! RESEARCH

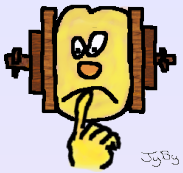
CONICYT COMISIÓN NACIONAL DE INVESTIGACIÓN CIENTÍFICA Y TECNOLÓGICA

GOBIERNO DE CHILE

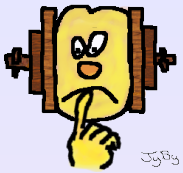
Parte I: Secuencias



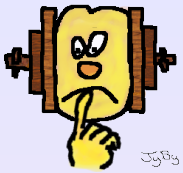
Parte II: Otras Estructuras



Parte I: Secuencias

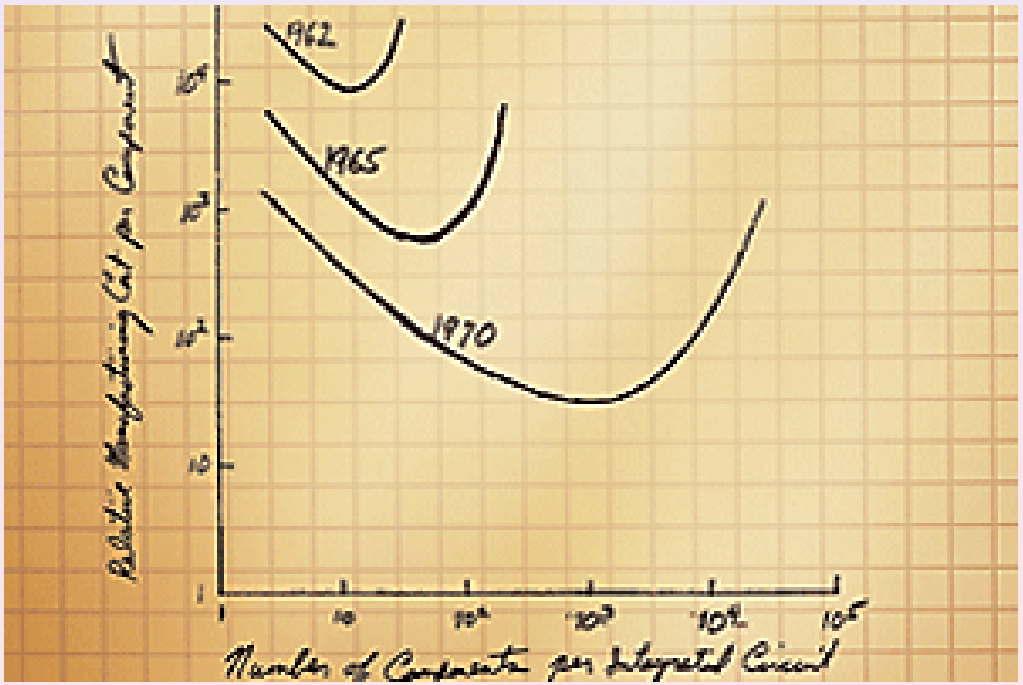


La Jerarquía de Memoria



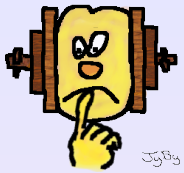
Evolución de los Circuitos: Ley de Moore (1965)

"El número de transistores que consigue el menor costo por transistor en un circuito integrado se duplica cada 24 meses"



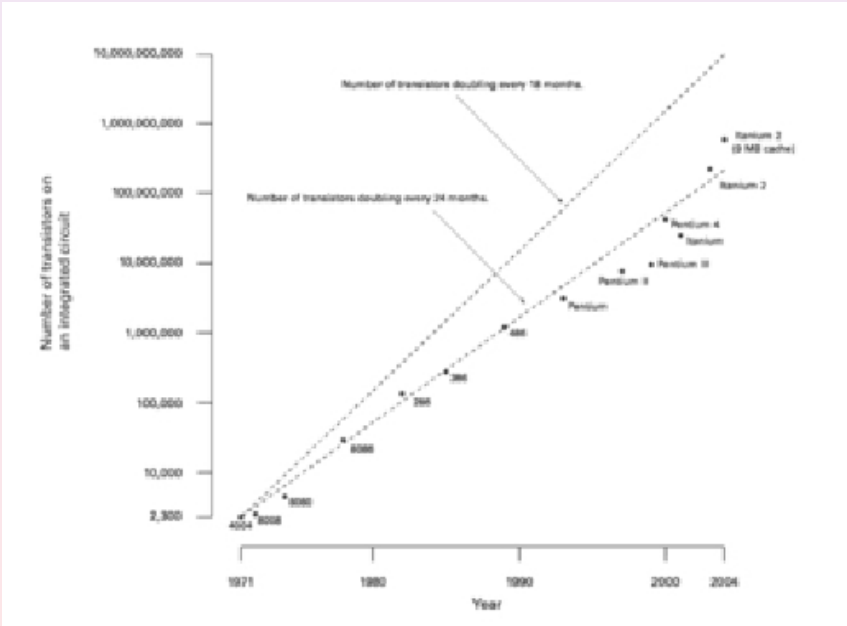
Este gráfico y los siguientes sobre este tema se han extraído de Wikipedia.

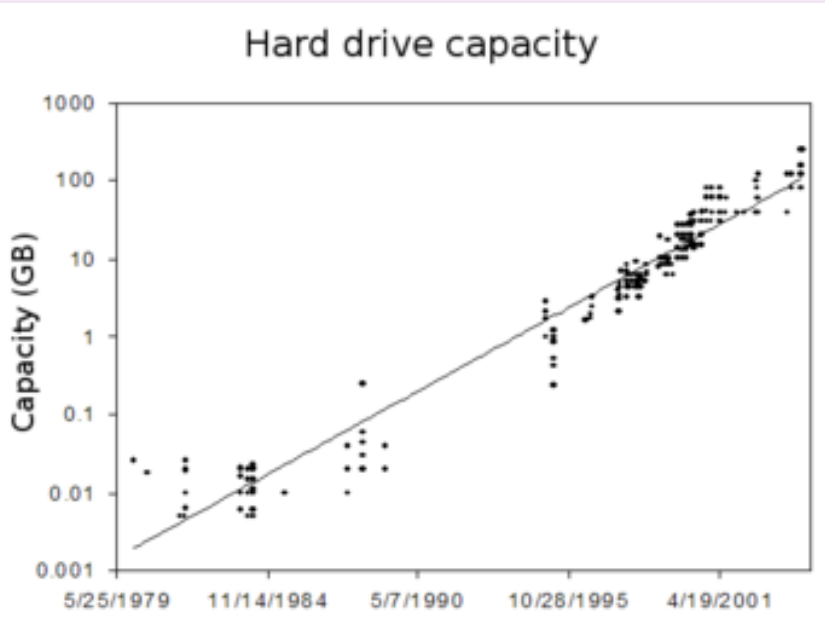
La Jerarquía de Memoria



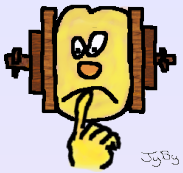
Consecuencias de la Ley de Moore

- Las memorias, a igual costo, son cada vez mayores.
- (Incluso a menor costo son mayores!)
- Las CPUs, igualmente, son más potentes.
- Se cree que valdrá al menos hasta el 2020.





La Jerarquía de Memoria

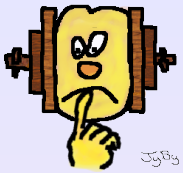


¿Todo sigue la Ley de Moore?

- No!
- Discos (tiempos de seek, por ejemplo).

- Velocidad de acceso a la RAM.

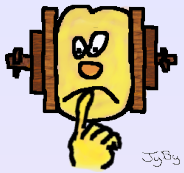
La Jerarquía de Memoria



Situación Actual

- Es posible, en general, tener tanta memoria como se quiera.
- Pero ésta es cada vez más lenta en comparación con la CPU.
- Aparecen nuevas memorias (caches)
  - Más rápidas (por tecnología y distancia a la CPU).
  - Más caras (por tecnología).
  - Más pequeñas (por distancia a la CPU y precio).
- Por el compromiso velocidad/distancia, aparecen múltiples niveles de cache.

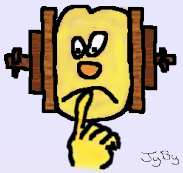
La Jerarquía de Memoria



Situación Actual

- Números gruesos y (relativamente) actuales:
  - Unos pocos registros de CPU, menos de 1 nanosegundo.
  - Unos pocos KBs de cache L1, unos 10 nanosegundos.
  - Unos pocos MBs de cache L2, unos 30 nanosegundos.
  - Unos pocos GBs de RAM, unos 60 nanosegundos.
  - Unos pocos TBs de disco, unos 10 milisegundos de latencia, más unos 500 nanosegundos por palabra transferida.
- La jerarquía de memoria es más relevante que nunca!
  "La diferencia de tiempo entre tener un dato en RAM versus traerlo de disco es comparable a la de tomar el sacapuntas del escritorio donde estoy sentado versus tomarme un avión a la China para ir a buscarlo y regresar."

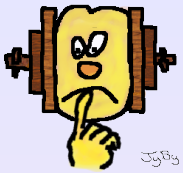
La Jerarquía de Memoria



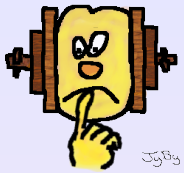
¿Y el Futuro?

- Un poco de ciencia ficción:
  - Vivimos en un universo tridimensional.
  - Si empaqueto n objetos de cierto tamaño...
  - ... la distancia del centro al más lejano es $\Omega(n^{1/3})$.
- Siempre habrá memorias más pequeñas y rápidas...
- ... versus más grandes y lentas.
- Incluso sin considerar razones económicas!

Parte I: Secuencias

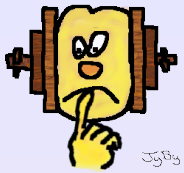


Estructuras de Datos Compactas



Son estructuras de datos...

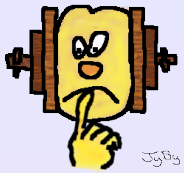
- Modificadas para ocupar poco espacio.
- *Y eso no es compresión?*
- No: deben retener su funcionalidad y acceso directo.
- *Para qué, si la memoria es tan barata?*
- Mejoran el rendimiento debido a la jerarquía de memoria.
- Especialmente si logramos operar en RAM algo que necesitaría del disco!



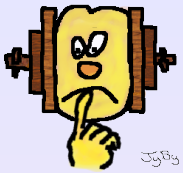
Un ejemplo motivante...

- El genoma humano, recientemente decodificado.
- Contiene 3 mil millones de bases.
- Cada base necesita 2 bits (letras A, C, G, T).
- Cabe holgadamente en una RAM de 1 GB.
- *Pero los biólogos necesitan hacer búsquedas complejas en él!*
- Estas operaciones serían lentísimas en forma secuencial...
  - por ejemplo, obtener la autorrepetición más larga requiere tiempo cuadrático sin un índice apropiado;
  - con el índice adecuado se hace fácilmente en tiempo lineal.

Estructuras de Datos Compactas



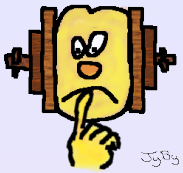
- El índice que resuelve todos esos problemas es el árbol de sufijos.
- Pero éste requiere entre 30 GB y 60 GB de memoria!
- Para peor, no se lleva bien con el disco.
- En la práctica, sólo puede usarse para secuencias de juguete, que hasta podrían tratarse secuencialmente.
- Usando estructuras de datos compactas, cabe en una RAM de 2 GB.
- Es mucho más lento que el árbol de sufijos clásico en una misma memoria...
- pero es infinitamente más rápido corriendo en RAM que el original en disco.



Otro ejemplo motivante...

- El grafo de la Web contenía el 2004 unos 11.5 mil millones de nodos y 150 mil millones de links.
- Crece más o menos según la Ley de Moore.
- Esto considera sólo la Web estática indexada!
- Necesitaría unos 600 GB de RAM para almacenarse.
- Gigantes como Google y Yahoo! lo usan para calcular PageRank, encontrar comunidades, etc.
- Usando estructuras de datos compactas, cabe en unos 100 GB.
- Y con un poco más permite además navegar el grafo hacia atrás, y otras operaciones útiles.

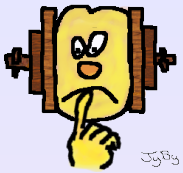
Mapa del Tutorial



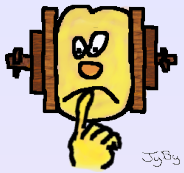
Ahora que están convencidos...

- Revisaremos los avances en la última década en diversas estructuras de datos compactas.
- Estas les darán herramientas teóricas y prácticas para aprovechar la jerarquía de memoria en el diseño de algoritmos y estructuras de datos.
- Veremos estructuras compactas para:
  - Manipular secuencias de bits
  - Manipular secuencias de símbolos
  - Navegar en árboles
  - Buscar en textos
  - Navegar en grafos
- Y aplicaciones a hashing, conjuntos, sumas parciales, geometría, permutaciones, y más.

Parte I: Secuencias



Entropía Empírica



- Entropía binaria: si hay n_0 ceros y n_1 unos en una secuencia de bits B ($n_0 + n_1 = n = |B|$)

$$H_0(B) = \frac{n_0}{n}\log\frac{n}{n_0} + \frac{n_1}{n}\log\frac{n}{n_1} = \frac{1}{n}\log\binom{n}{n_0} + O\left(\frac{\log n}{n}\right)$$

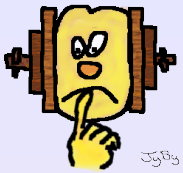
(utilizaremos logaritmos en base 2 por defecto).

- Entropía de orden cero: si hay n_c ocurrencias de c en S (secuencia de símbolos sobre un alfabeto Σ),

$$H_0(S) = \sum_{c\in\Sigma} \frac{n_c}{n}\log\frac{n}{n_c}$$

- Cota inferior a cualquier codificación de Σ que asigne siempre el mismo código al mismo símbolo (ej. Huffman).

Entropía Empírica

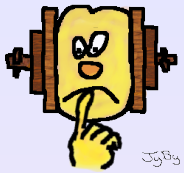


- Entropía de orden k: si S_A es la secuencia de los caracteres que siguen a las ocurrencias de A en S,

$$H_k(S) = \frac{1}{n} \sum_{A \in \Sigma^k} |S_A| H_0(S_A)$$

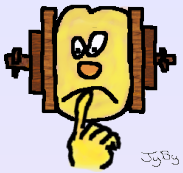
- Cota inferior a codificaciones que consideran los k símbolos precedentes (ej. PPM).

Modelo RAM y Análisis Asintótico

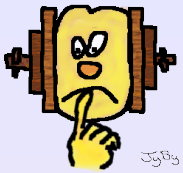


- Mediremos el espacio en bits.
- Modelo RAM:
  - Necesitamos $\log n$ bits para direccionar en una memoria de n bits.
  - Si direccionamos n bits, consideraremos que el computador puede manipular $O(\log n)$ bits en tiempo constante.
- $O(n)$ significa limitado superiormente por $c \cdot n$ para alguna constante (positiva) c a partir de un cierto $n = n_0$.
- $o(n)$ significa que, dividido por n, tiende a cero cuando n tiende a infinito.

Parte I: Secuencias

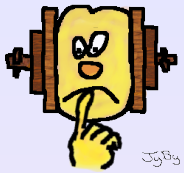


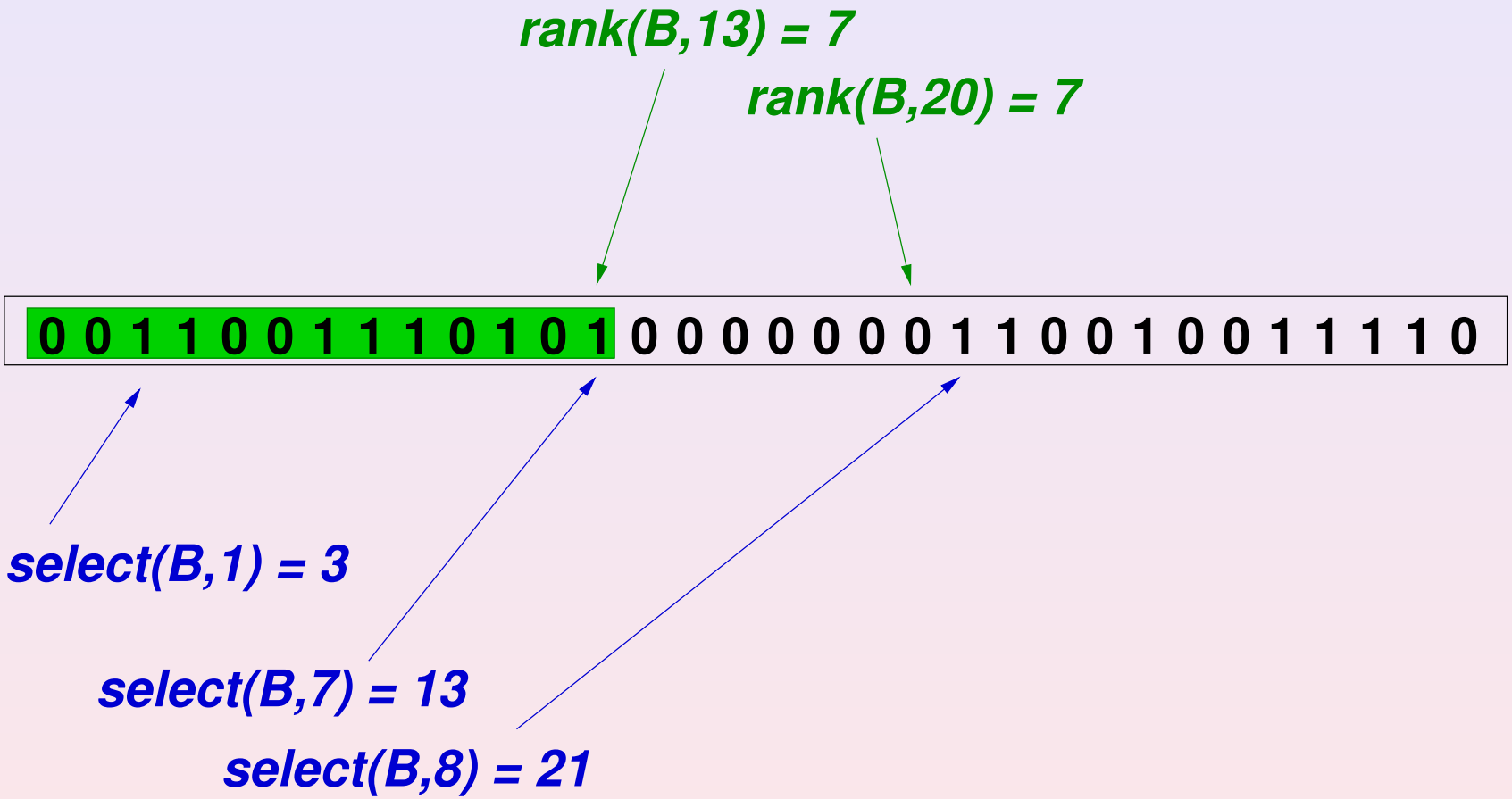
Secuencias de Bits



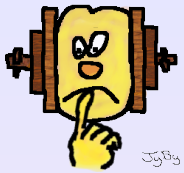
- Consideremos una secuencia de n bits, $B[1, n]$.
- Nos interesan las siguientes operaciones sobre B:
  - $rank_b(B, i)$: ¿cuántas veces aparece el bit b en $B[1, i]$?
  - $select_b(B, i)$: ¿dónde ocurre el bit b por i-ésima vez en B?
- Algunas propiedades simples:
  - $rank_0(B, i) = i - rank_1(B, i)$.
  - $B[i] = rank_1(B, i) - rank_1(B, i - 1)$ (sup. $rank_b(B, 0) = 0$).
  - $rank_b(B, select_b(B, i)) = i$.
  - Si $B[i] = b$, $select_b(B, rank_b(B, i)) = i$.
  - En general, $select_b(B, rank_b(B, i)) \leq i$.
- Cuando no mencionemos b supondremos $b = 1$.

Secuencias de Bits





Secuencias de Bits

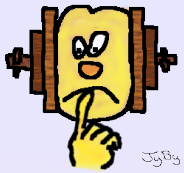


Resultado

- Se puede responder *rank* y *select* en tiempo constante.
- Esto es fácil almacenando todas las respuestas en $O(n \log n)$ bits, pero solamente se necesitan

$$n + O\left(\frac{n \cdot \log\log n}{\log n}\right) \quad = \quad n + o(n)$$

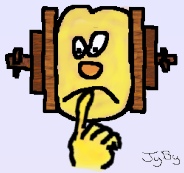
  bits de espacio (los n bits para $B[1, n]$ más un extra sublineal).
- Las soluciones son prácticas (especialmente *rank*).
- Veremos algunas de las muchas aplicaciones antes de mostrar cómo se logra este resultado.



Una aplicación: Hashing perfecto

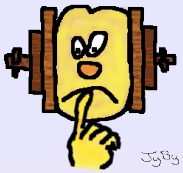
- Si el universo tiene n claves (por ejemplo $[1, n]$),
- y queremos almacenar t elementos con r bits de datos,
- hashing perfecto nos ofrece:
  - $O(tr)$ bits de espacio,
  - tiempo de acceso constante
  - construcción aleatorizada o muy costosa.

Rank y Select

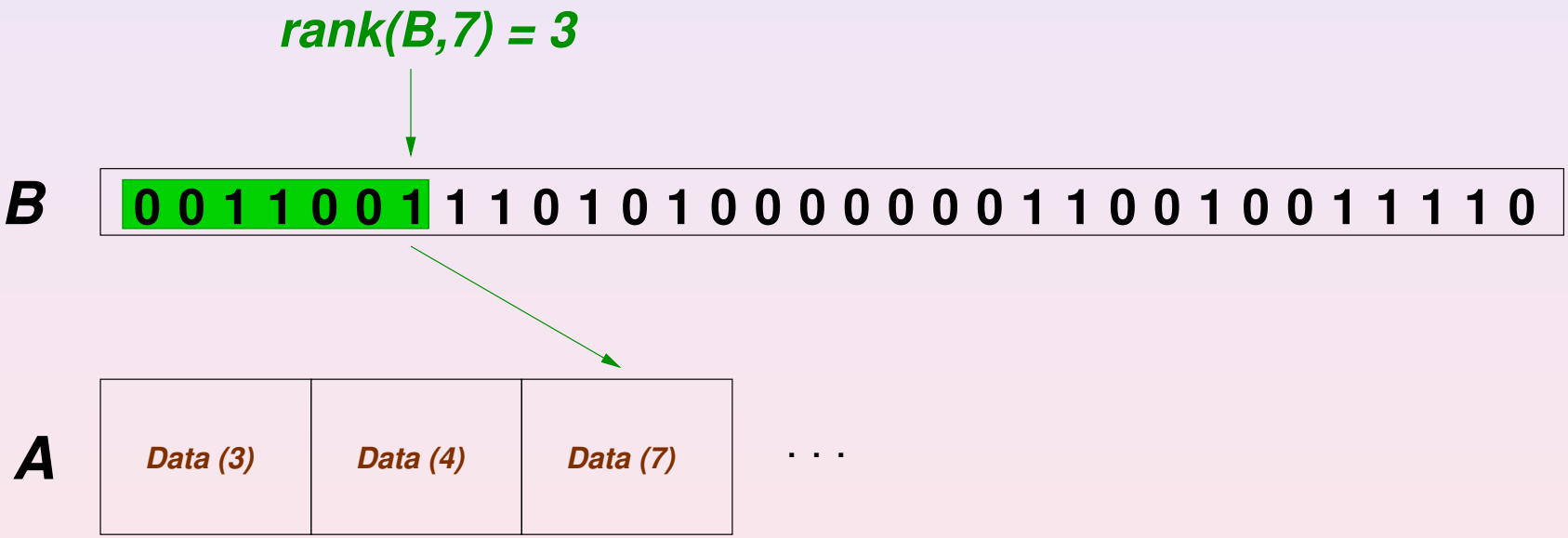


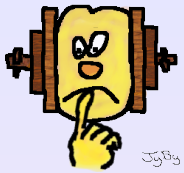
- Consideremos una solución usando *rank*:
  - Tendremos un arreglo $A[1,t]$ con los datos,
  - más un bitmap $B[1,n]$ que marque las claves que existen.
  - Entonces nuestro dato con clave i está en $A[rank(B,i)]$.
  - Y está en el conjunto si $B[i]=1$.
  - Total: $tr+n+o(n)$ bits.
- La solución es interesante si n/t no es muy grande comparado con r.
- Además es mucho más simple.
- Incluso podríamos lograr $tr+O(t\log(n/t))$ bits (menos de un puntero extra por elemento).

Rank y Select



Hashing perfecto

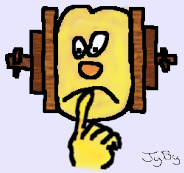




Otra aplicación: Sumas parciales

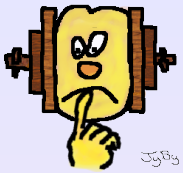
- Supongamos que tenemos t números $A[1,t]$ que suman n,
- y queremos hacer dos tipos de preguntas sobre el arreglo:
  - Sum: Dado r, ¿cuánto es $\sum_{j=1}^{r} A[j]$?
  - Search: Dado s, ¿para qué r ocurre que $\sum_{j=1}^{r} A[j] > s$?
- Es fácil responderlas en tiempo constante usando $t \log n + n \log t$ bits...

Rank y Select

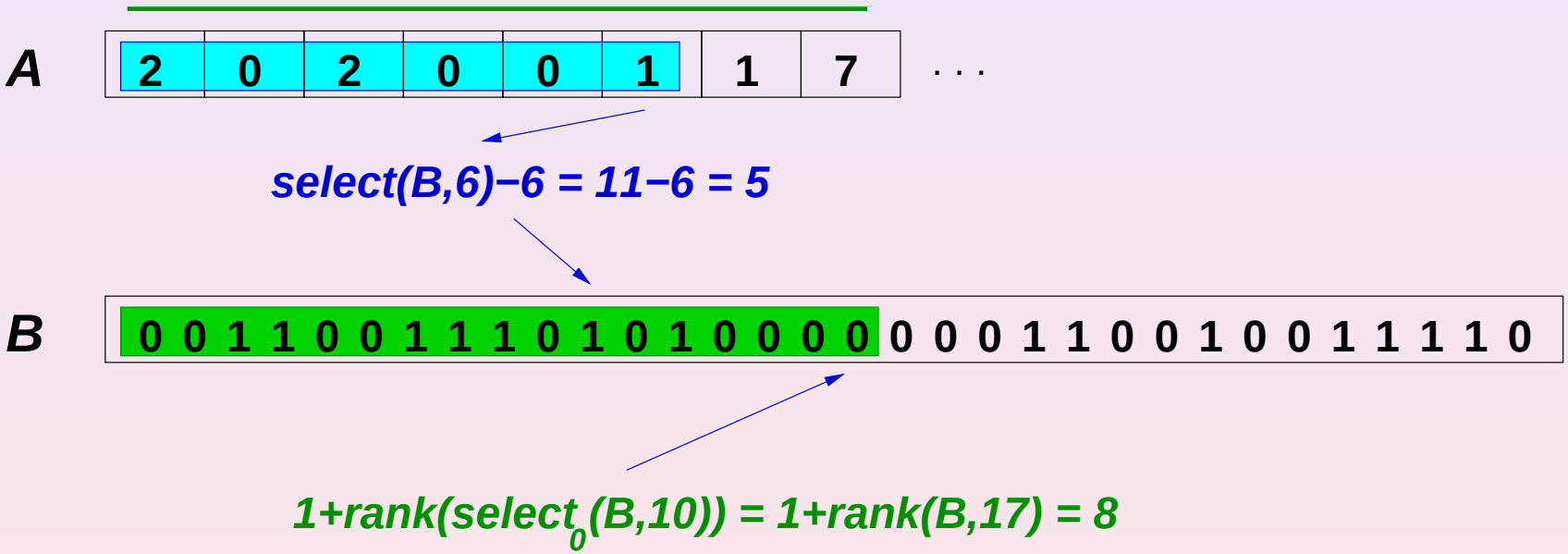


- Supongamos que los $A[i]$ son no negativos.
- Con *rank* y *select* las podemos responder usando $n + t + o(n)$ bits!
- Marcar con 1, en $B[1, n + t]$, las posiciones $r + \sum_{j=1}^{r} A[j]$.
- Entonces
  - Sum: $select_1(B, r) - r$.
  - Search: $1 + rank_1(B, select_0(B, s))$.
- Y si $n + t$ bits parece mucho, mostraremos que también se puede hacer usando $t \log \frac{n+t}{t}$ bits!

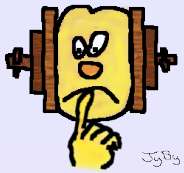
Rank y Select



Sumas parciales



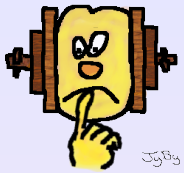
Rank y Select



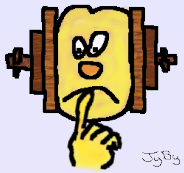
Y otra más: Predecesor y sucesor

- Tenemos t elementos del conjunto $[1, n]$,
- y queremos hacer dos tipos de preguntas sobre el conjunto:
  - Succ: Dado i, ¿cuál es el menor número $\geq i$ en el conjunto?
  - Pred: Dado i, ¿cuál es el mayor número $\leq i$ en el conjunto?
- Nuevamente, se responden fácilmente en tiempo constante usando $O(n \log n)$ bits...

Rank y Select



- Con *rank* y *select* se pueden responder usando sólo $n + o(n)$ bits:
  - Succ: $select(B, rank(B, i-1)+1)$.
  - Pred: $select(B, rank(B, i))$.
- Podríamos fácilmente obtener $Succ^k$ y $Pred^k$ en tiempo constante.
- Y con bitmaps comprimidos usaríamos $t \log(n/t)$ bits!



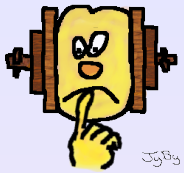
Rank en Tiempo Constante

- Cortamos B en bloques de $b = (\log n)/2$ bits.
- Almacenamos un arreglo $R[1, n/b]$ con los valores de *rank* al comienzo de los bloques.
- Como necesitamos $\log n$ bits para almacenar un valor de *rank*, el total de bits que ocupa R es

$$\frac{n}{b} \cdot \log n = \frac{n}{(\log n)/2} \cdot \log n = 2n$$

- Aceptemos ese precio en espacio por ahora (es más de lo prometido).

Rank

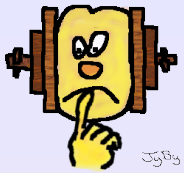


- ¿Cómo calculamos $rank(B, i)$?
  - Descomponemos $i = q \cdot b + r$, $0 \leq r < b$.
  - La cantidad de 1's hasta qb es $R[q]$.
  - Debemos contar los 1's en $B[qb + 1, qb + r]$.
- Supongamos que tenemos una tabla $T[0, 2^b - 1][0, b - 1]$, tal que

$$T[x, r] \quad = \quad \text{total de 1's en } x[1, r]$$

donde vemos x como una tira de b bits.

Rank



- Como cada celda de T puede tomar valores en $[0, b-1]$, T necesita

$$\begin{aligned} 2^b \cdot b \cdot \log b &= 2^{\frac{\log n}{2}} \cdot \frac{\log n}{2} \cdot (\log\log n - 1) \\ &\leq \frac{1}{2}\sqrt{n}\log n \log\log n = o(n) \text{ bits.} \end{aligned}$$

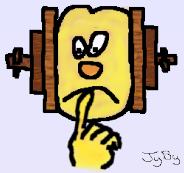
  Por ejemplo, si $n = 2^{32}$, T sólo necesita 512 KB.

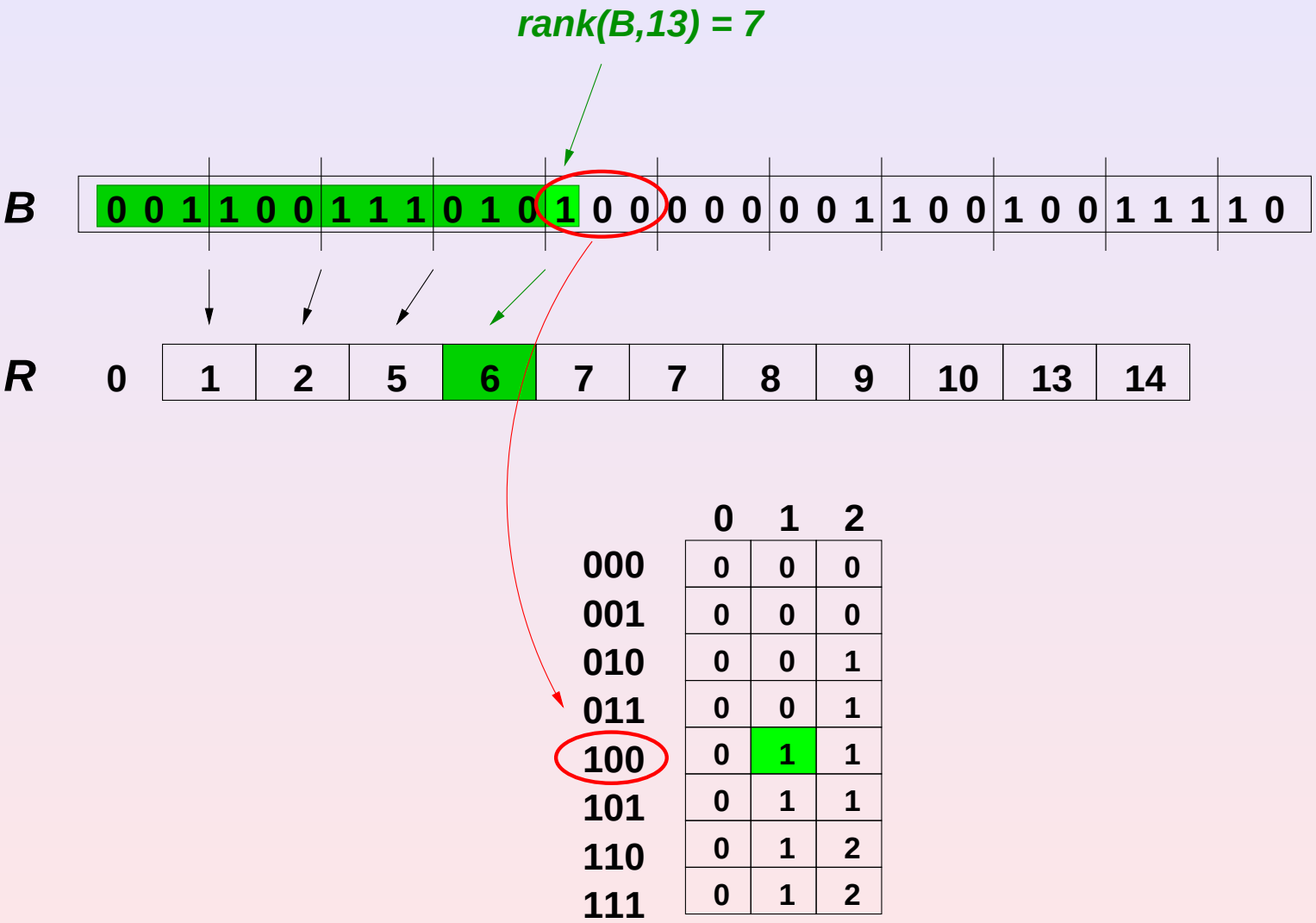
- Entonces, la respuesta final se obtiene en tiempo constante como:

$$R[q] + T[B[qb+1..qb+b], r]$$

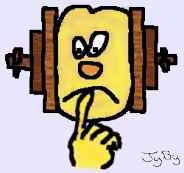
- Muy bien, pero nos gastamos $3n + o(n)$ bits...

Rank




rank(B,13) = 7
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0
R
0 1 2 5 6 7 7 8 9 10 13 14
0 1 2
000 0 0 0
001 0 0 0
010 0 0 1
011 0 0 1
100 0 1 1
101 0 1 1
110 0 1 2
111 0 1 2

Rank



Consiguiendo $o(n)$ bits extra

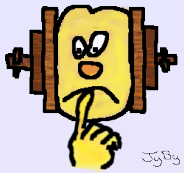
- Cortamos B también en superbloques de

$$s = \frac{(\log n)^2}{2} = b \cdot \log n \text{ bits.}$$

- Almacenamos un arreglo $S[1, n/s]$ con los valores de *rank* al comienzo de los superbloques.
- Ahora los valores de R se almacenan sumando sólo desde el comienzo del superbloque correspondiente:

$$\begin{aligned} R[q] &= rank(B, qb) - S[\lfloor q/\log n \rfloor] \\ &= rank(B, qb) - rank(B, \lfloor q/\log n \rfloor \cdot \log n) \end{aligned}$$

Rank



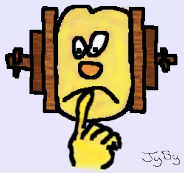
- Como necesitamos $\log n$ bits para almacenar un valor de *rank* en S, el total de bits que ocupa S es

$$\frac{n}{s} \cdot \log n = \frac{n}{(\log n)^2/2} \cdot \log n = \frac{2n}{\log n} = o(n)$$

- Como los valores de R se almacenan relativos al superbloque, sólo pueden llegar a valer $s = O(\log n)^2$, y por ello sólo necesitan $2 \log\log n$ bits. Por ello R ocupa ahora

$$\frac{n}{b} \cdot 2 \log\log n = \frac{n}{(\log n)/2} \cdot 2 \log\log n$$
$$= \frac{4n \cdot \log\log n}{\log n} = o(n) \text{ bits.}$$

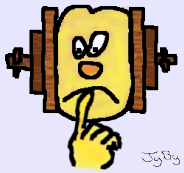
Rank

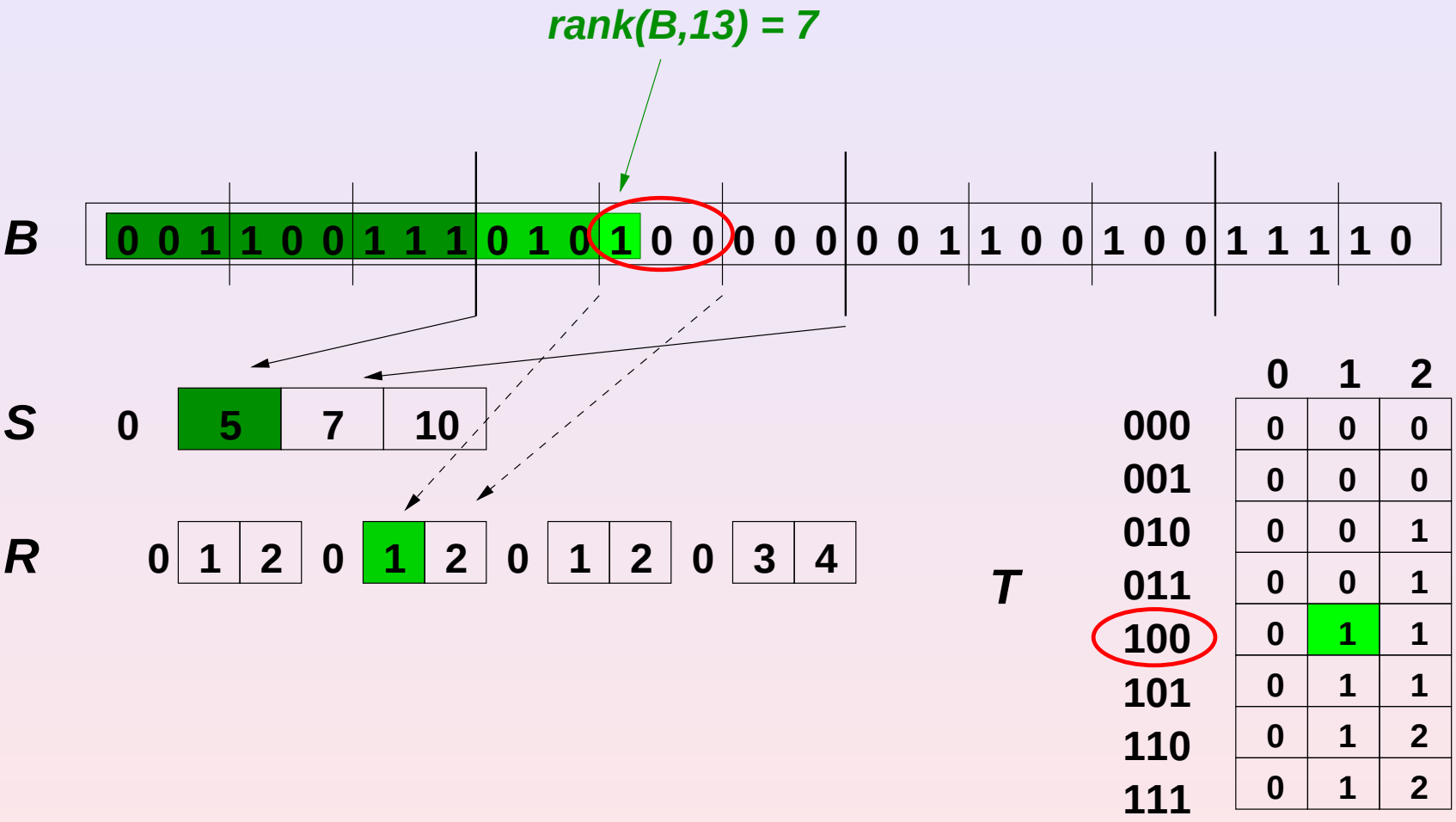


- ¿Cómo calculamos $rank(B, i)$?
  - Descomponemos $i = q \cdot b + r$, $0 \leq r < b$.
  - Descomponemos $i = q' \cdot s + r'$, $0 \leq r' < s$.
  - Y finalmente sumamos los contenidos de tres tablas:

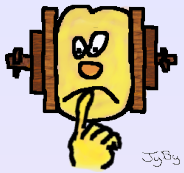
$$rank(B, i) \quad = \quad S[q'] + R[q] + T[B[qb + 1..qb + b], r]$$

Rank



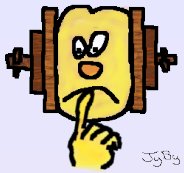

rank(B,13) = 7
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 1 0
S
0
5
7
10
R
0 1 2 0 1 2 0 1 2 0 3 4
T
0 1 2
000 0 0 0
001 0 0 0
010 0 0 1
011 0 0 1
100 0 1 1
101 0 1 1
110 0 1 2
111 0 1 2

Rank

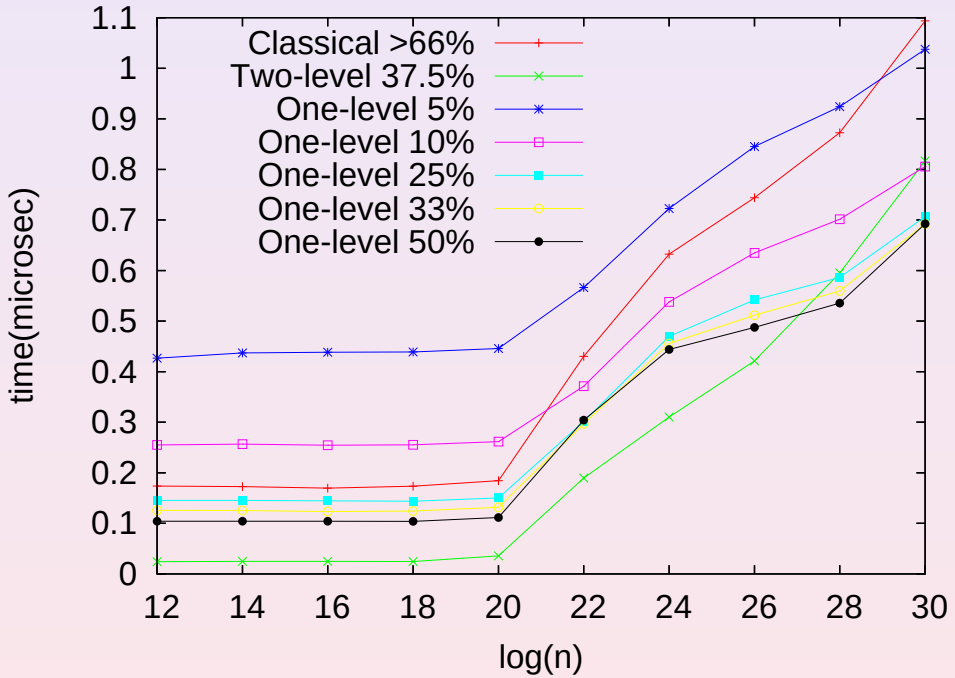


Implementación Práctica

- Evitar divisiones y módulos, usar potencias de 2.
- Tratar de usar valores alineados a bytes, shorts, o ints.
- Los tamaños son parametrizables.
- Aprovechar el cache.
- Ejemplo 1:
  - Usamos solamente superbloques, de $20 \times 32 = 640$ bits.
  - El espacio extra es $\frac{1}{20} = 5\%$.
  - Recorremos secuencialmente el superbloque usando popcount de a bytes (a lo sumo 80 operaciones).
- Ejemplo 2:
  - Usamos superbloques de 256 bits (overhead $\frac{32}{256} = 12.5\%$).
  - Usamos bloques de 32 bits (overhead $\frac{8}{32} = 25\%$).
  - Popcount de a lo más 4 bytes.



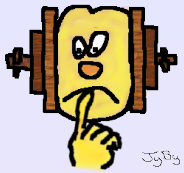
Implementación Práctica



Observar el efecto del cache y el hit ratio.

Gráficos obtenidos por mi alumno Rodrigo González.

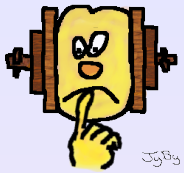
Select



Select en Tiempo Constante

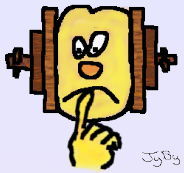
- Describiremos una solución donde $o(n) = O(n/\log\log n)$.
- Notar que necesitamos hacer un sampling regular en los argumentos de select, no regular en B (ambos criterios coincidían para *rank*).
- Un corte en bloques y superbloques como para *rank* no funciona, porque los números dentro de un bloque no se reducen.
- La idea fundamental es: si un bloque tiene pocos 1's, puedo almacenar todas las respuestas a *select* en poco espacio.
- Veremos que *select* es más complejo que *rank*, tanto en la teoría como en la práctica.

Select



- Cortamos los argumentos de *select* en superbloques de $s = \log^2 n$ argumentos.
- Estos superbloques tienen largo variable en B, pero tienen exactamente s 1's.
- Decimos que un superbloque es esparso si su largo en B es a lo menos $s \cdot \log n \cdot \log\log n$; sino es denso.
- Observación fundamental: podemos guardar todas las respuestas de todos los bloques esparsos y el sobrecosto total es $O(n/\log\log n)$.

Select

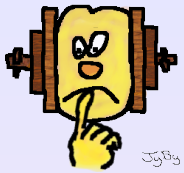


- Almacenamos un bitmap $E[1, n/s]$ indicando qué superbloques son esparsos.
- Almacenamos todas las respuestas a *select* en todos los superbloques esparsos en un arreglo de arreglos R:

$$select(B, i) = R[rank(E, \lfloor i/s \rfloor)]\,[1 + (i \bmod s)]$$

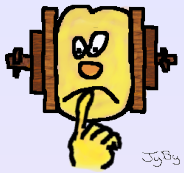
  si $E[\lfloor i/s \rfloor] = 1$.
- Para los superbloques densos, solamente almacenamos un arreglo $P[1, n/s]$ con las posiciones de B donde comienzan.
- Aún no solucionamos el problema de los superbloques densos, pero sabemos dónde empiezan, y su largo es a lo más $s \log n \log\log n = \log^3 n \log\log n$.

Select



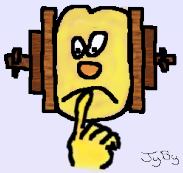
- Dividimos los superbloques densos en bloques de $b = (\log \log n)^2$ argumentos.
- Aplicamos la misma idea nuevamente dentro de cada superbloque denso.
- Necesitamos $\log(\log^3 n \log \log n) \leq 4 \log \log n$ bits para almacenar una posición dentro de un superbloque denso.
- Diremos que un bloque es esparso si su largo en B es a lo menos $4b(\log \log n)^2$, y es denso sino.
- Almacenar todas las respuestas (relativas) de los bloques esparsos dentro de superbloques densos cuesta en total $O(n/\log \log n)$ bits.

Select



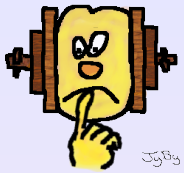
- Almacenaremos un bitmap $E'[1, s/b]$ indicando qué bloques son esparsos.
- Para los bloques densos, sólamente almacenamos un arreglo $P'[1, s/b]$ con las posiciones del superbloque donde comienzan.
- Este P' cuesta también $O(n/\log\log n)$ bits.
- No hemos solucionado el problema de los bloques densos, pero sabemos dónde empiezan, y su largo es a lo más $4(\log\log n)^4 = o(\log n)$ (siempre es $< 82 \log n$).
- Entonces, los bloques densos se pueden procesar en tiempo $O(1)$ usando tablas tipo T.

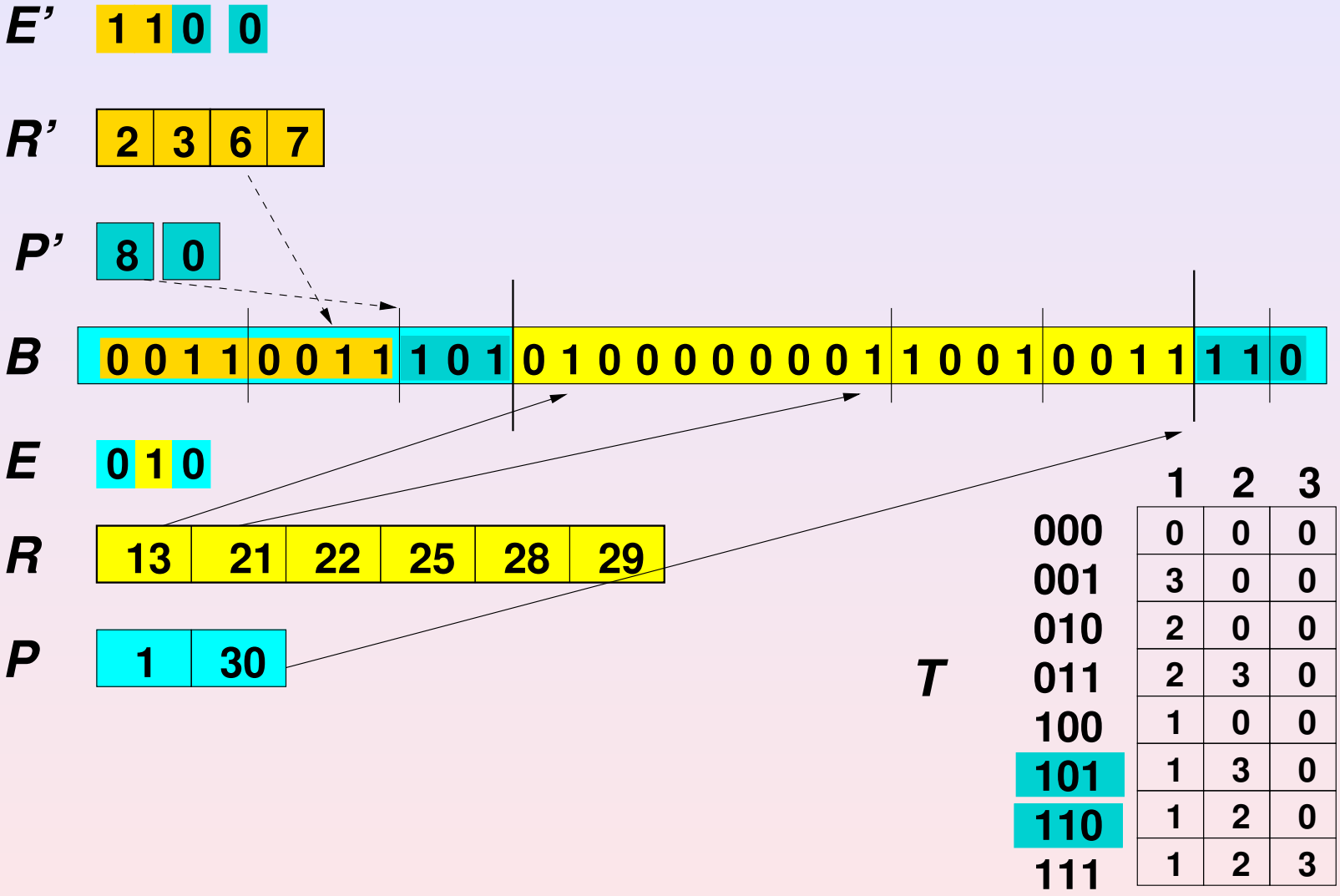
Select



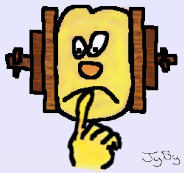
- Cómo calculamos $select(B, i)$?
  - Descomponemos $i = q \cdot s + r$.
  - Si $E[q] = 1$, es un superbloque esparso y retornamos $R[...]$.
  - Sino, es un superbloque denso que comienza en $P[q]$.
  - Descomponemos $r = q' \cdot b + r'$.
  - Si $E'_q[q'] = 1$ es un bloque esparso, retornamos $P[q] + R'[...]$.
  - Sino, es un bloque denso que comienza en $p = P[q] + P'[q']$.
  - Buscamos el r'-ésimo 1 en $B[p...p + 4(\log\log n)^4]$ usando tablas.

Select




E'
1 1 0 0
R'
2 3 6 7
P'
8 0
B
0 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 1 1 0
E
0 1 0
R
13 21 22 25 28 29
P
1 30
T
1 2 3
000 0 0 0
001 3 0 0
010 2 0 0
011 2 3 0
100 1 0 0
101 1 3 0
110 1 2 0
111 1 2 3

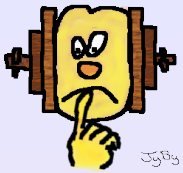
Select



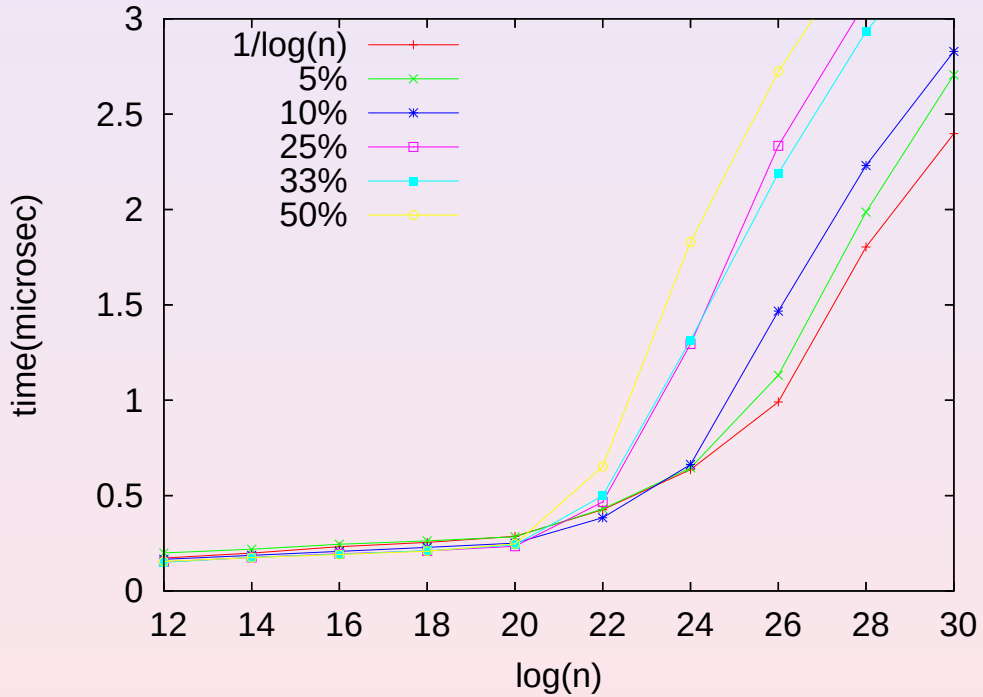
Implementación Práctica

- Implementado directamente es bastante poco práctico.
- Por ejemplo:
  - Si aceptamos recorrer 2048 bits (512 bytes) secuencialmente...
  - ...logramos 78% de espacio extra.
  - Eso debe duplicarse si además queremos $select_0$.
- Una solución práctica aceptable:
  - *select* es el inverso de *rank*.
  - Lo resolvemos con búsqueda binaria en *rank*.
  - Primero en S, luego en R, finalmente en los bits.
  - El mismo espacio de *rank*, y resolvemos $select_0$ también.
  - La mejor alternativa es con un sólo nivel de superbloques.

Select



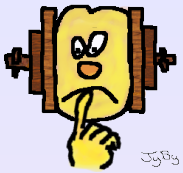
Implementación Práctica



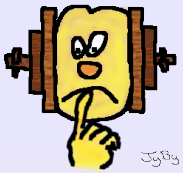
Observar el efecto del cache y el hit ratio.

Gráficos obtenidos por mi alumno Rodrigo González.

Parte I: Secuencias



Rank y Select sobre Secuencias Comprimidas

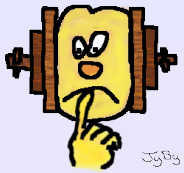


- En muchas aplicaciones, la secuencia tiene pocos o muchos 1's.
- Nos concentraremos en el caso en que hay $m << n$ 1's.
- Por ejemplo, podríamos responder *select* en tiempo constante usando $m \log n$ bits (almacenar todas las respuestas).
- ¿Hay una forma de generalizar esta idea simple?
- Veremos que podemos conseguir *rank* y *select* en tiempo constante utilizando

$$nH_0(B) + o(n) \quad = \quad m \log \frac{n}{m} + O(m) + o(n) \quad \text{bits.}$$

- La solución es teórica pero veremos alternativas prácticas.

Representación Comprimida



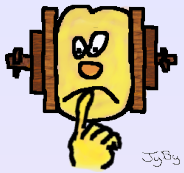
- Dividiremos la secuencia en bloques de $b = (\log n)/2$ bits.
- Sea c_i la cantidad de 1's en un cierto bloque B_i.
- La cantidad de bloques distintos de largo b con c_i 1's es

$$\binom{b}{c_i}$$

y por lo tanto trataremos de representar B_i usando

$$\left\lceil \log \binom{b}{c_i} \right\rceil \text{ bits.}$$

Representación Comprimida

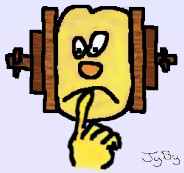


- La representación de B_i será (c_i, o_i), donde
  - La clase, c_i, necesita $\lceil \log(b+1) \rceil$ bits.
  - El offset, o_i, necesita $\lceil \log \binom{b}{c_i} \rceil$ bits.
- Los c_i's son de largo fijo, y ocupan en total

$$\frac{n}{b} \cdot \lceil \log(b+1) \rceil \quad = \quad O\left(\frac{n \cdot \log\log n}{\log n}\right)$$

- Los o_i's son de largo variable. Los concatenaremos todos.
- Después veremos cómo decodificarlos.

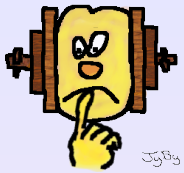
Representación Comprimida



- La concatenación de los o_i's ocupa

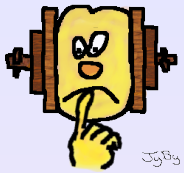
$$\begin{aligned}
\sum_{i=1}^{n/b} \left\lceil \log \binom{b}{c_i} \right\rceil &\leq \sum_{i=1}^{n/b} \log \binom{b}{c_i} + n/b \\
&= \log \prod_{i=1}^{n/b} \binom{b}{c_i} + O(n/\log n) \\
&\leq \log \binom{(n/b)\cdot b}{\sum_{i=1}^{n/b} c_i} + O(n/\log n) \\
&= \log \binom{n}{m} + O(n/\log n) \\
&\leq nH_0(B) + O(n/\log n) \text{ bits.}
\end{aligned}$$

Representación Comprimida



- Hemos representado B usando $nH_0(B) + o(n)$ bits!
- Pero, ¿cómo podemos extraer algún bloque B_i en tiempo constante?
- Primer problema: ¿dónde está o_i?
  - Podemos almacenar las posiciones en $P[1, n/b]$...
  - ... pero eso requeriría $(n/b) \log n = 2n$ bits extra.
  - Definimos superbloques de $s = b \cdot \log n$ bits...
  - ... y almacenamos $P[1, n/s]$ sólo para superbloques.
  - Ahora tendremos un $P'[1, n/b]$ con punteros relativos al superbloque (como en *rank*).
  - Como $|o_i| = O(\log n)$, cada $P'[i]$ requiere $O(\log \log n)$ bits.
  - Total: $O(n \log \log n / \log n)$ bits.

Representación Comprimida

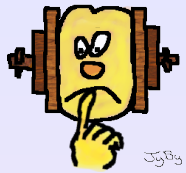


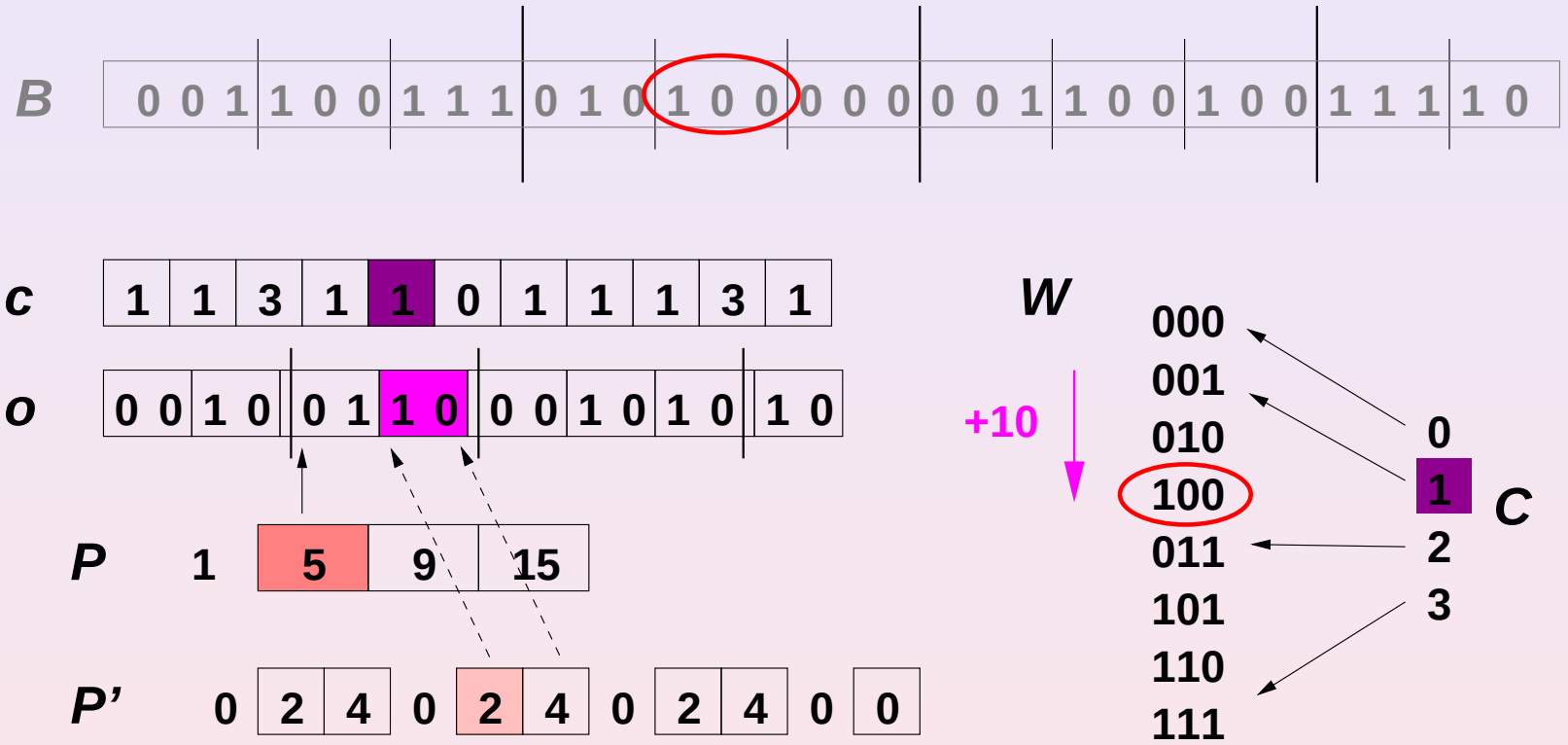
- Segundo problema: ¿qué bloque representa (c_i, o_i)?
  - Tendremos una tabla $W[0, 2^b - 1]$ donde se almacenen todos los bitmaps de largo b agrupados por clase.
  - Las posiciones donde empieza cada clase en W se precalcularán en un arreglo

$$C[c] = 1 + \sum_{i=0}^{c-1} \binom{b}{i}$$

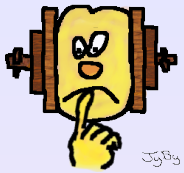
  - W está ordenada de modo que o_i es el índice correspondiente dentro de la zona de la clase c_i.
  - Entonces, (c_i, o_i) representa $W[C[c_i] + o_i]$.
  - Estas tablas ocupan $O(\sqrt{n} \log n)$ bits.

Representación Comprimida



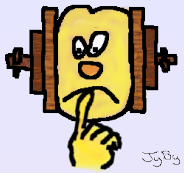


Rank y Select



- Las estructuras para *rank* y *select* ocupan $o(n)$ bits además de B.
- Necesitan acceso en tiempo constante a bloques de B.
- Eso ya lo hemos conseguido con la representación comprimida!
- Por lo tanto, hemos obtenido *rank* y *select* en tiempo constante sobre la representación comprimida (a H_0).
- Más precisamente, usamos $nH_0(B) + O(n \log\log n / \log n)$ bits.

Comprimiendo a Orden k



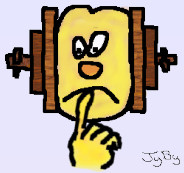
- Notar que cualquier representación de B que permita obtener los bloques en tiempo constante sirve.
- Consideremos el siguiente esquema:
  - Contamos cuántas veces aparece cada bloque en B.
  - Ordenamos los bloques de más a menos frecuente.
  - Les asignamos códigos de largo creciente:

$$"", \ 0, \ 1, \ 00, \ 01, \ 10, \ 11, \ 000 \ldots$$

  - Usamos los códigos en vez de los pares (c_i, o_i) del esquema anterior.
- Se puede probar que esto comprime a orden k:

$$nH_k(B) + o(n), \quad \text{para todo} \quad k = o(\log n)$$

Comprimiendo a Orden k



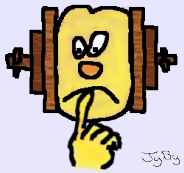
¿Cómo extraemos un bloque?

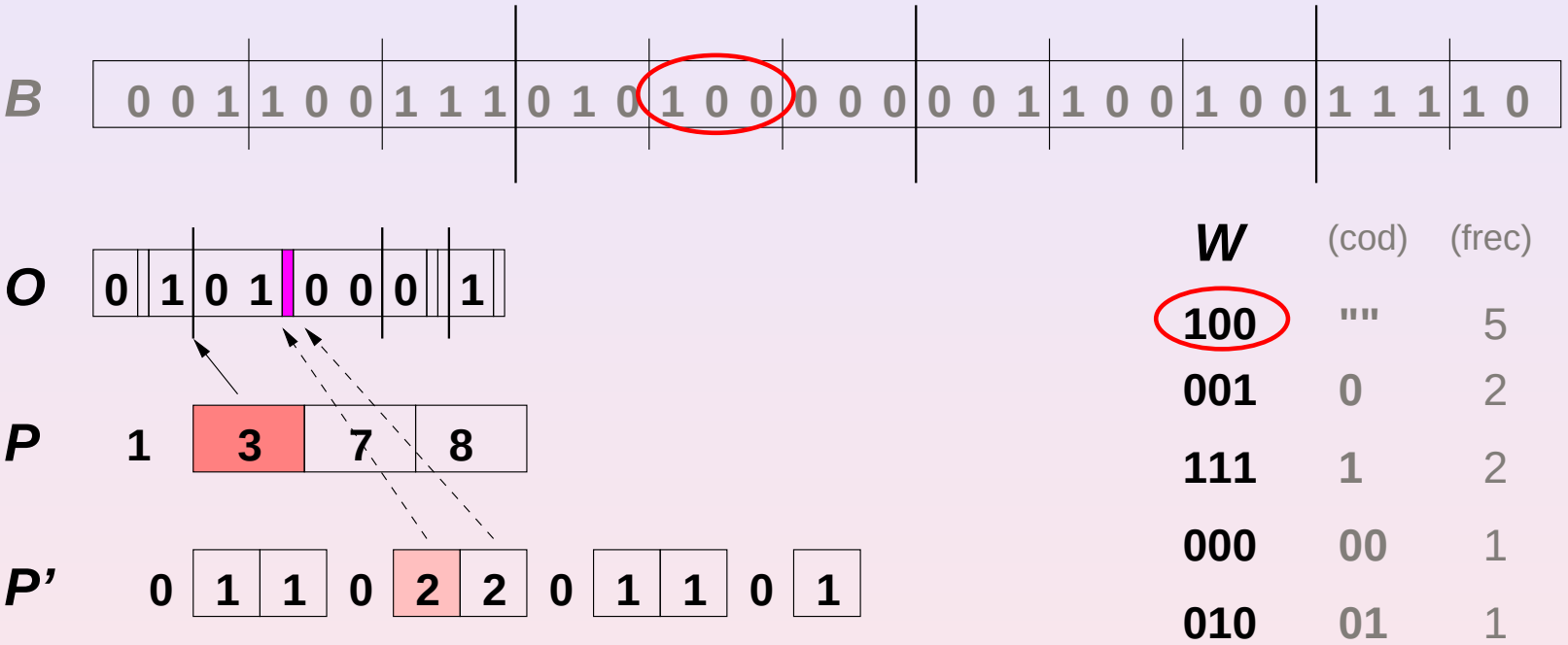
- Almacenamos una tabla W con los bloques ordenados por frecuencia.
- Entonces al código de valor numérico i y de largo t le corresponde el bloque número

$$i + 1 + \sum_{j=0}^{t-1} 2^j \quad = \quad i + 2^t \quad \text{de la tabla.}$$

- Conseguimos *rank* y *select* en tiempo constante y espacio $nH_k(B) + o(n)$ bits!

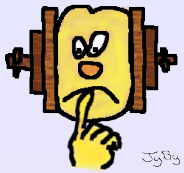
Comprimiendo a Orden *k*





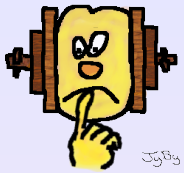
W	(cod)	(frec)
100	""	5
001	0	2
111	1	2
000	00	1
010	01	1

Comprimiendo a Orden k



- Se puede generalizar a secuencias $S[1,n]$ sobre un alfabeto Σ de tamaño σ.
- Se obtiene acceso a $O(\log n)$ bits, es decir $O(\log_\sigma n)$ símbolos, en tiempo constante.
- El espacio es

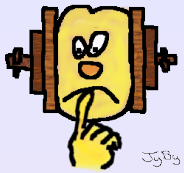
$$nH_k(S) + o(n \log \sigma), \quad \text{para todo} \quad k = o(\log_\sigma n)$$



Implementación Práctica

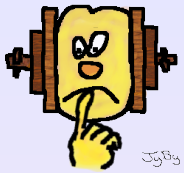
- No funcionan tan bien como uno querría, especialmente si la entropía es muy baja.
- Las estructuras extra (c_i, P', R, etc.) pasan a dominar el uso de espacio.
- Veremos una alternativa práctica llamada sparse array, para el caso de $m << n$.
- Esencialmente, almacenamos todos los valores $S[i] = select(B, i)$, para $1 \leq i \leq m$, y resolvemos *rank* con búsqueda binaria.

Rank y Select



- Los valores $S[i]$ se separan en la parte baja l_i y la parte alta h_i.
- La parte baja son los t bits menos significativos, de modo que $S[i] = h_i \cdot 2^t + l_i$.
- Elegimos $t = \lceil \log \frac{n}{m} \rceil$.
- La secuencia $L[i]$ de partes bajas se almacena en forma normal, usando $t \cdot m = m \log \frac{n}{m} + O(m)$ bits.
- La secuencia de partes altas se marcan en un bitmap $H[1, 2m]$, donde se prenden los bits $H[h_i + i]$.
- Alcanzan $2m$ bits pues $h_m + m \leq n/2^t + m \leq 2m$.
- El espacio total es $m \log \frac{n}{m} + O(m) = nH_0(B) + O(m)$.

Rank y Select

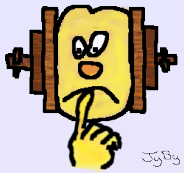


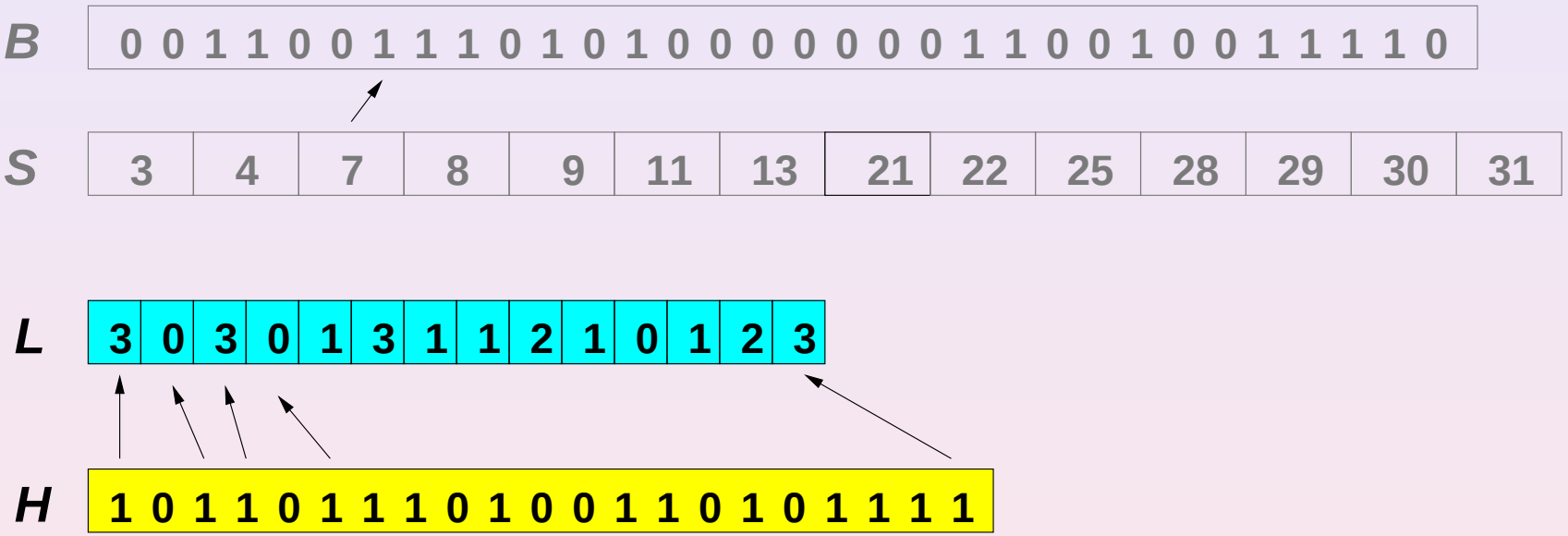
- Es fácil resolver *select* en tiempo constante:

$$select(B, i) \quad = \quad (select(H, i) - i) \cdot 2^t + L[i]$$

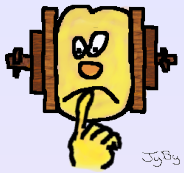
  (hay que preprocesar H para consultas de *select*).
- Para $rank(B, i)$ se usa una búsqueda binaria mejorada.
  - El problema es encontrar dónde está i en H.
  - Descomponemos $i = h \cdot 2^t + l$, $0 \leq l < 2^t$.
  - Cada 0 en H avanza 2^t posiciones en B.
  - De modo que la respuesta está entre los bloques $x = 1 + rank(H, select_0(H, h))$ e $y = rank(H, select_0(H, h + 1))$ de B.
  - Podemos hacer búsqueda binaria de l en $L[x, y]$, respondiendo *rank* en tiempo $O(t) = O(\log \frac{n}{m})$.
  - En la práctica se prefiere la búsqueda secuencial.

Rank y Select





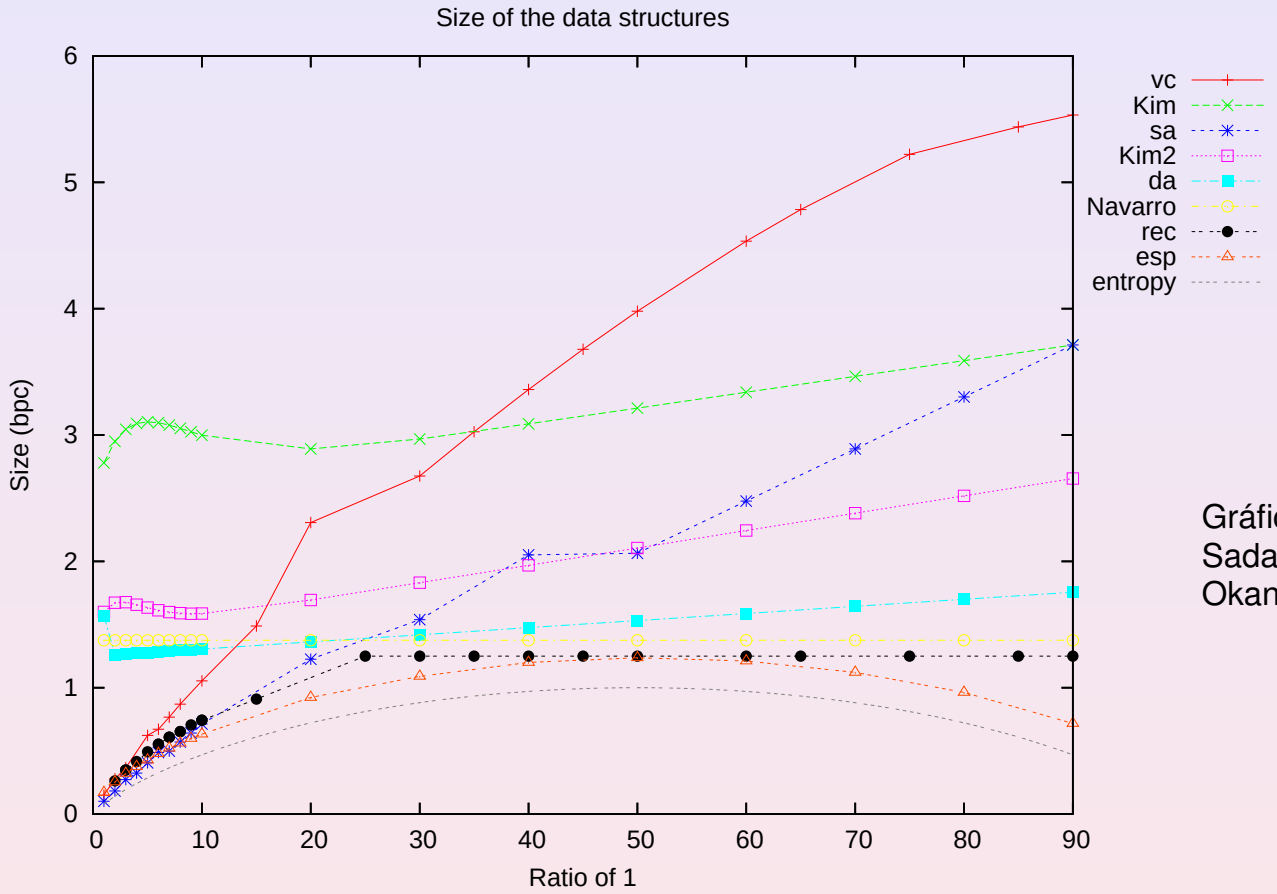
Rank y Select



Arreglos crecientes

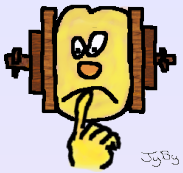
- ¿Han notado que la solución con L y H es general?
- Se puede aplicar a cualquier arreglo $A[1,m]$ de valores crecientes en $[1,n]$.
- La solución permite representarlo usando $m \log \frac{n}{m} + O(m)$ bits...
- ... y acceder a las celdas en tiempo constante.
- Incluso no es difícil modificar los valores de A.

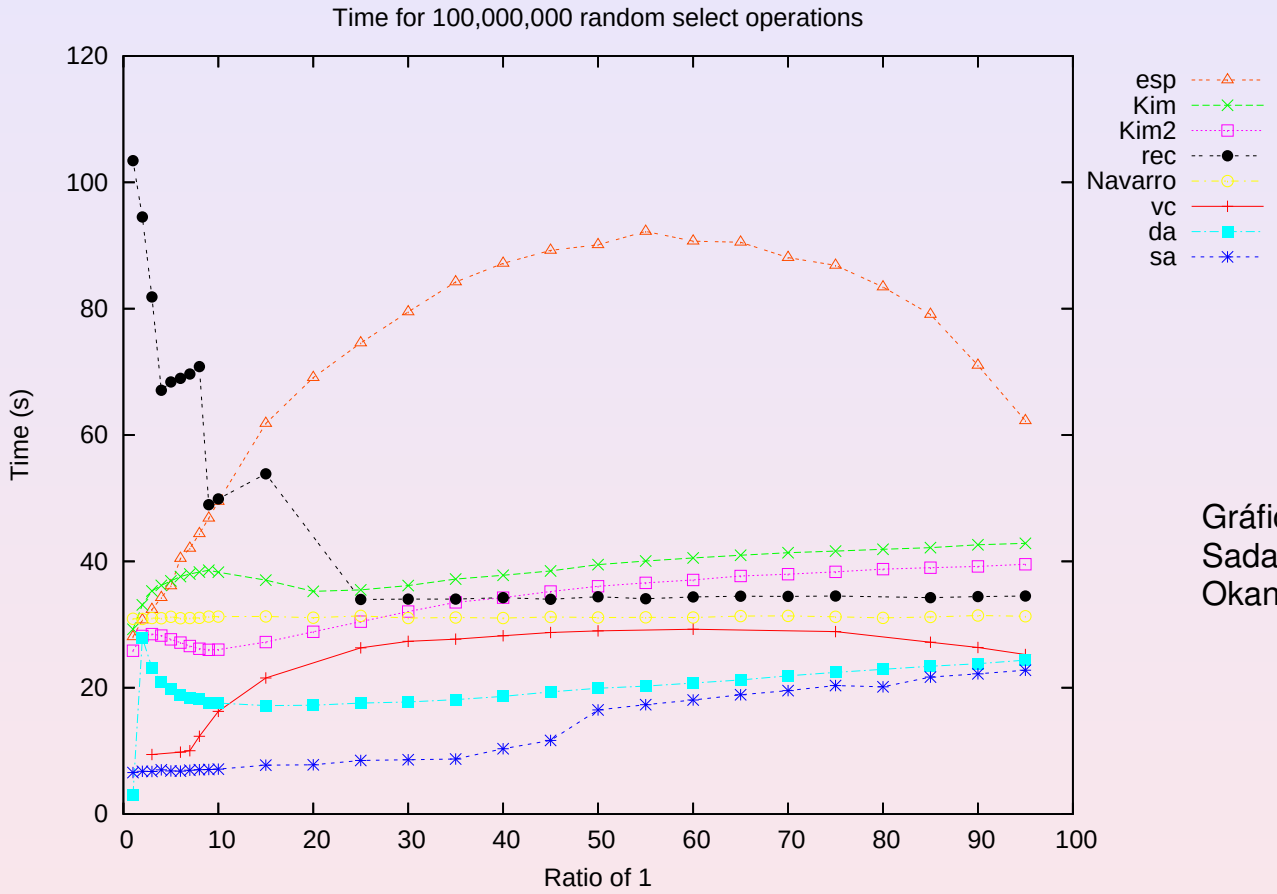
Implementación Práctica: Espacio



Gráficos obtenidos por Sadakane y su alumno Okanohara

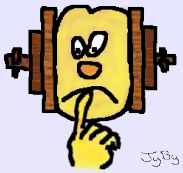
Implementación Práctica: Select

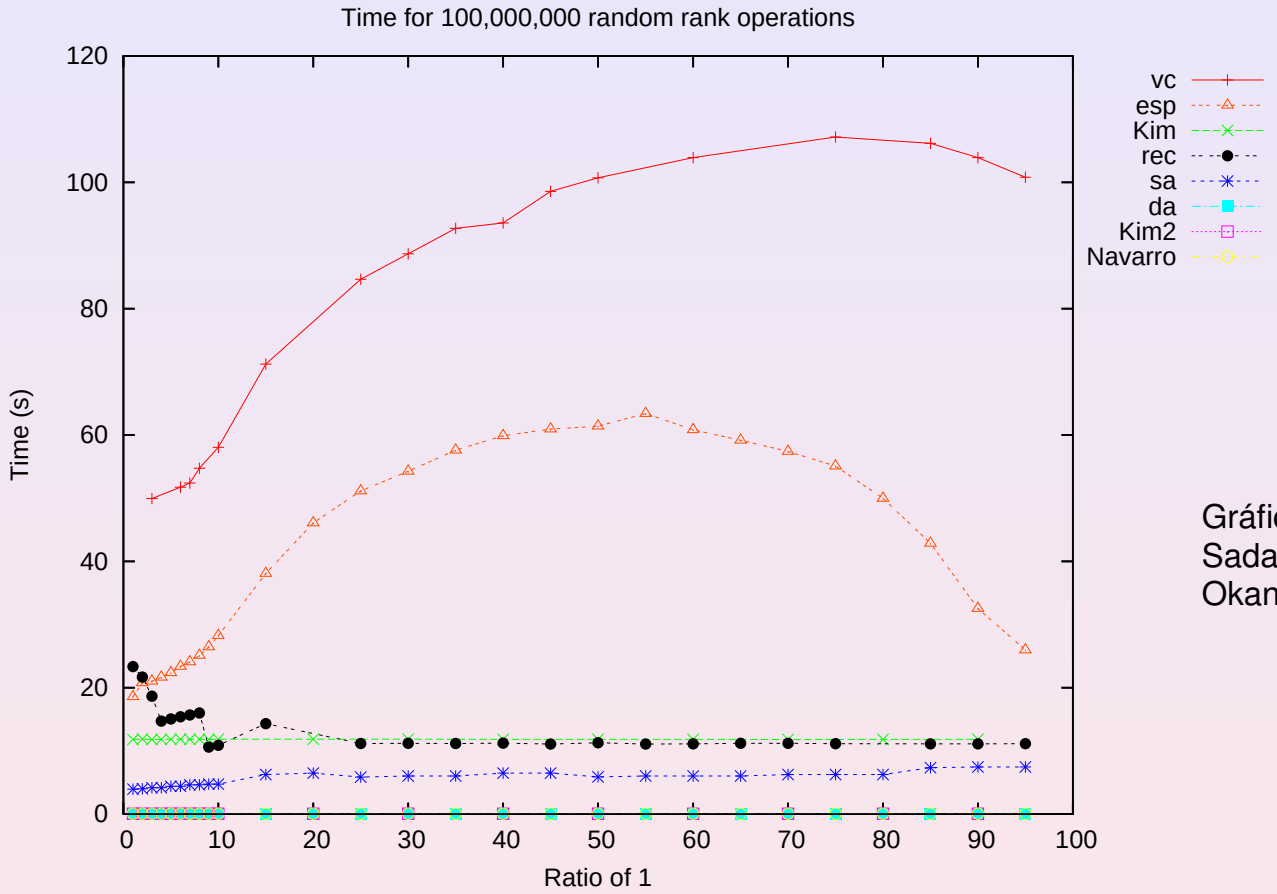




Gráficos obtenidos por Sadakane y su alumno Okanohara

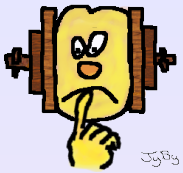
Implementación Práctica: Rank



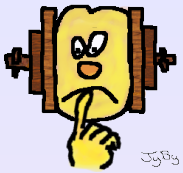


Gráficos obtenidos por Sadakane y su alumno Okanohara

Parte I: Secuencias

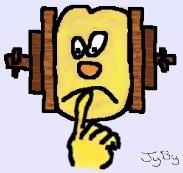


Secuencias Dinámicas



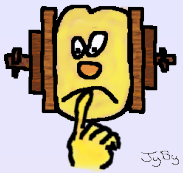
- Hasta ahora hemos considerado secuencias de bits que no cambian.
- ¿Qué ocurre si queremos insertar y eliminar bits?
- Tal como están, funcionan muy mal (recalcular todo).
- En cambio, consideremos esta estructura de árbol binario balanceado:
  - Cada hoja maneja un bit.
  - Cada nodo interno conoce el total de bits y total de 1's en su subárbol.
- Todo se puede resolver en tiempo $O(\log n)$.

Secuencias Dinámicas



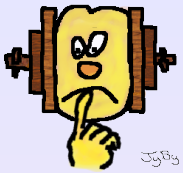
- Para encontrar el i-ésimo bit de B:
  - Si el subárbol izquierdo contiene $l \geq i$ bits,
    - busco el i-ésimo bit en el subárbol izquierdo.
  - Sino,
    - busco el $(i - l)$-ésimo bit en el subárbol derecho.
- Para insertar y borrar en la posición i, la busco e inserto o borro una hoja,
  - y recalculo cantidad de bits y de 1's,
  - y rebalanceo de ser necesario.

Secuencias Dinámicas



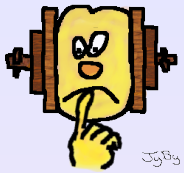
- Para calcular $rank(i)$
  - Si el subárbol izquierdo contiene $l \geq i$ bits,
    - calculo $rank(i)$ en el subárbol izquierdo.
  - Sino,
    - calculo $rank(i - l)$ en el subárbol derecho y le sumo la cantidad de 1's del subárbol izquierdo.
- Para calcular $select(i)$
  - Si el subárbol izquierdo contiene $s \geq i$ 1's,
    - calculo $select(i)$ en el subárbol izquierdo.
  - Sino,
    - calculo $select(i - s)$ en el subárbol derecho y le sumo la cantidad de bits del subárbol izquierdo.

Secuencias Dinámicas



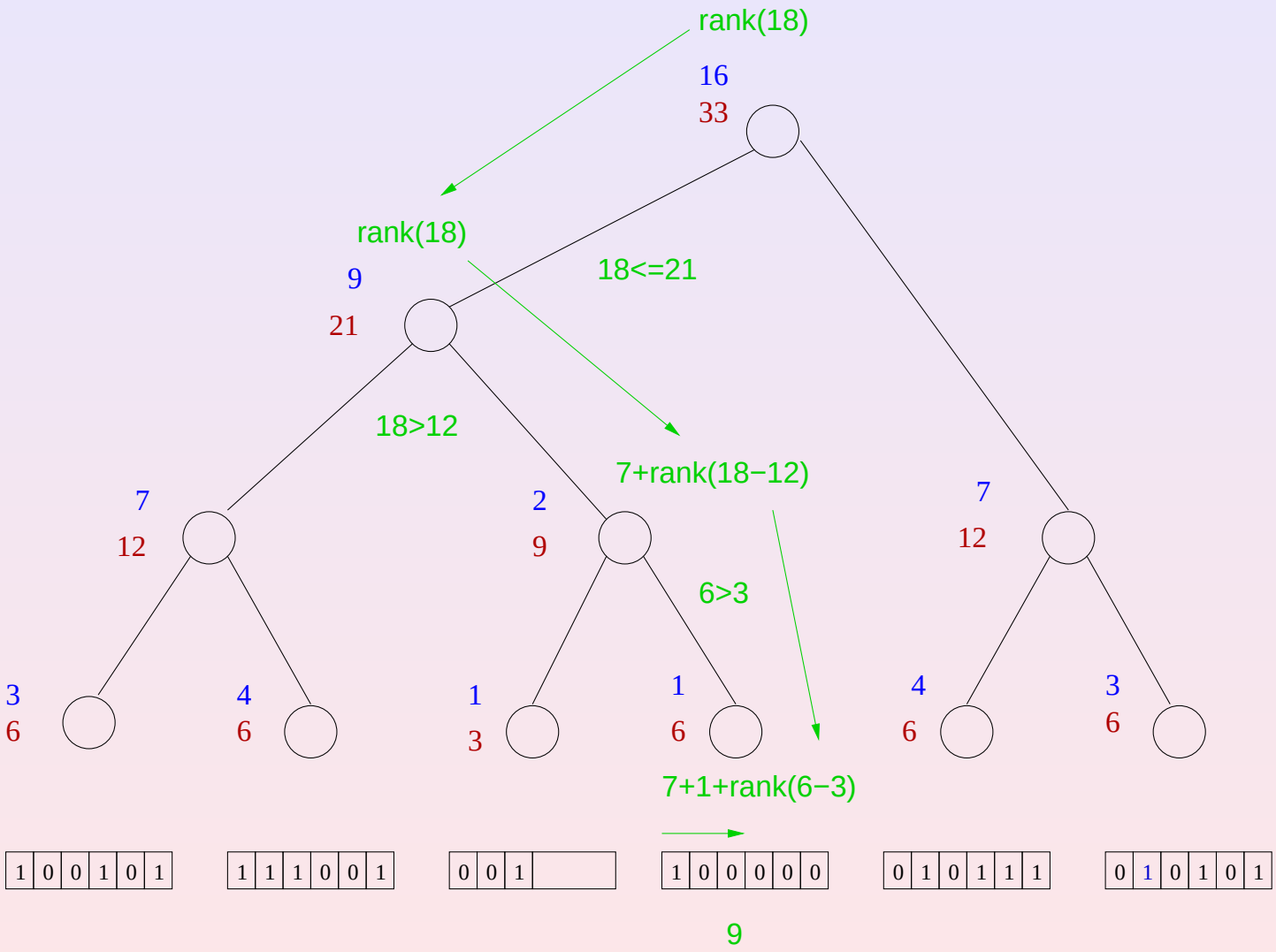
- Pero estamos ocupando $O(n \log n)$ bits!
- Usaremos el truco estándar para dinamizar estructuras compactas:
  - Se construye la estructura dinámica para bloques de datos.
  - Esos bloques se almacenan en forma estática y compacta.
  - Muchas veces se manejan brutalmente (reconstrucción total, recorrido secuencial, etc.).
  - Son suficientemente pequeños para no alterar los tiempos de la estructura dinámica.
  - Son suficientemente grandes para que la estructura dinámica sea pequeña.

Secuencias Dinámicas

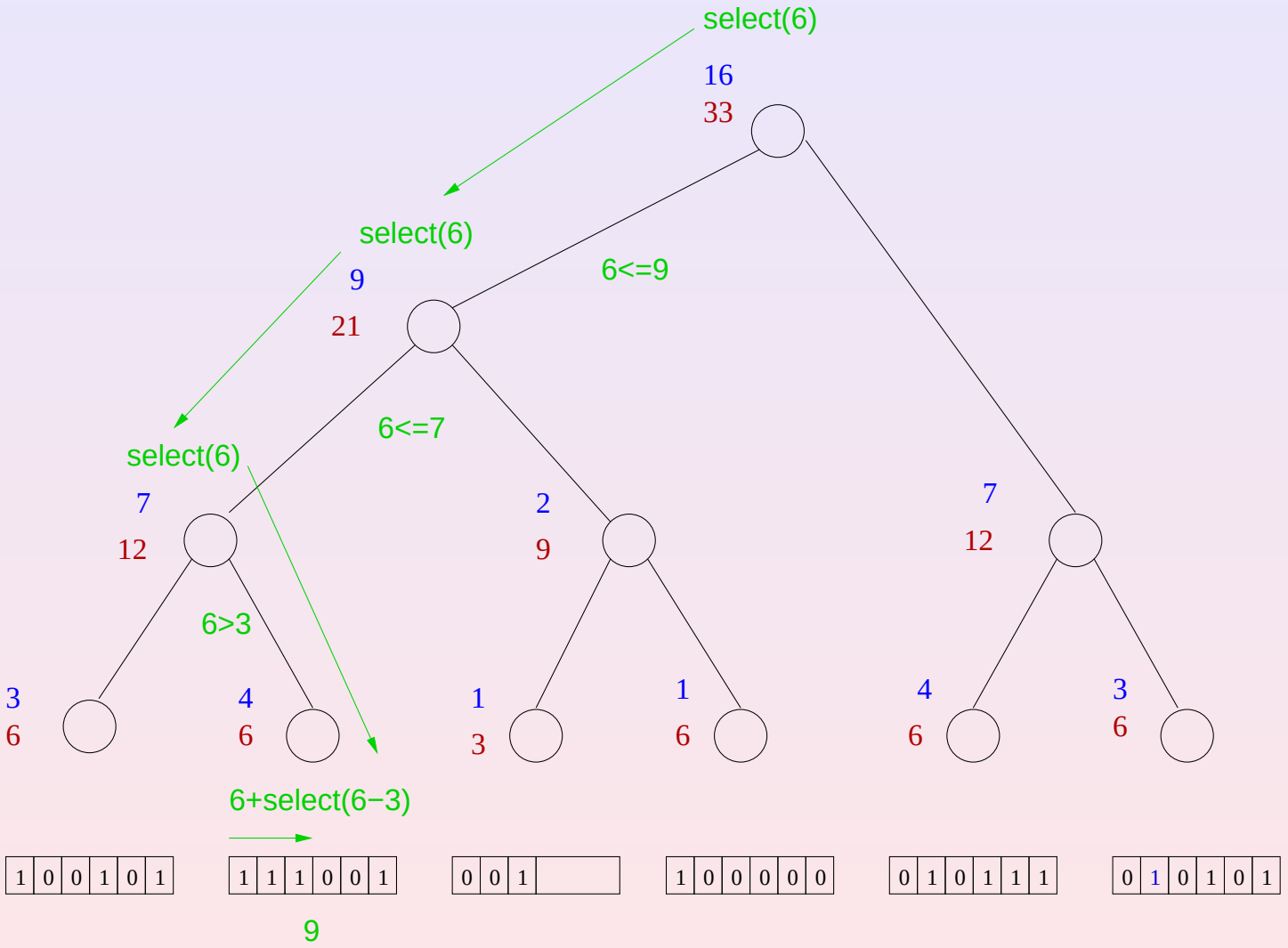


- Agrupamos $s = \log^2 n$ bits en las hojas.
- Formamos el árbol sobre esas hojas.
- Las operaciones en el árbol cuestan tiempo $O(\log n)$...
- ... y otro $O(\log n)$ en las hojas, usando tablas.
- Como hay $O(n/\log^2 n)$ nodos, el árbol ocupa $O(n/\log n)$ bits.
- Se necesita cierto cuidado para no gastarse $O(n)$ bits extra en las hojas
  - No pueden estar llenas a medias, por ejemplo.
- Este esquema puede adaptarse también a una representación comprimida de la secuencia.
- Podemos permitir insertar, borrar, *rank* y *select* en tiempo $O(\log n)$ y $nH_0(B) + o(n)$ bits de espacio.

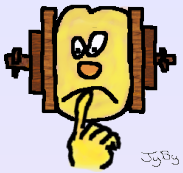
Secuencias Dinámicas



Secuencias Dinámicas

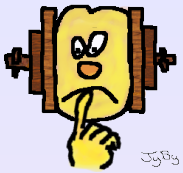


Sumas Parciales Dinámicas

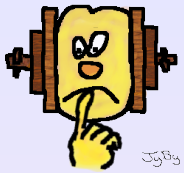


- Tenemos n números de k bits.
- Además de *sum* y *search*, permitiremos insertar y borrar números.
- Es muy fácil extender la técnica de los bitmaps dinámicos.
- En total necesitamos $kn + o(kn)$ bits...
- ... y realizamos todas las operaciones en tiempo $O(\log n)$.
- Con bitmaps comprimidos, se puede lograr espacio igual a la suma total de bits necesarios para representar todos los números.

Parte I: Secuencias

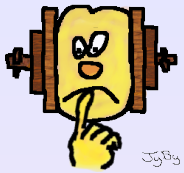


Secuencias de Símbolos



- Consideremos ahora que manejamos una secuencia $S = s_1 s_2 \ldots s_n$ sobre un alfabeto Σ de tamaño σ.
- Una representación plana de S necesita $n \log \sigma$ bits.
- Queremos hacer *rank* y *select* sobre esta secuencia:
  - $rank_c(S, i) =$ número de ocurrencias de c en $S[1, i]$.
  - $select_c(S, i) =$ posición de la i-ésima ocurrencia de c en S.
- ¿Cómo podemos extender nuestros resultados sobre bits?

Secuencias de Símbolos

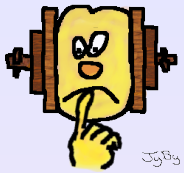


- Supongamos que almacenamos σ bitmaps $B_c[1,n]$, $c \in \Sigma$:

$$B_c[i] = 1 \text{ sii } S[i] = c$$

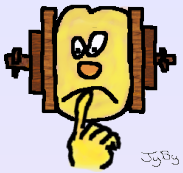
- Entonces
  - $rank_c(S,i) = rank_1(B_c,i)$.
  - $select_c(S,i) = select_1(B_c,i)$.
- Conseguimos tiempo constante... pero el precio es alto:
  - S ocupa $n \log \sigma$ bits.
  - Los bitmaps solos ocupan $n\sigma$ bits!
  - Además conocer $S[i]$ sin almacenarlo cuesta tiempo $O(\sigma)$.

Secuencias Dinámicas



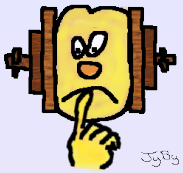
	a	l	a	b	a	r		a		l	a		a	l	a	b	a	r	d	a
$B_{\_}$	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0
B_{a}	1	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1
B_{b}	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
B_{d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
B_{l}	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
B_{r}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Secuencias de Símbolos

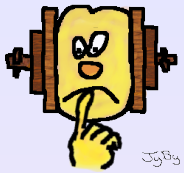


- Y si usáramos bitmaps comprimidos?
  - Los $nH_0(B_c)$ suman $nH_0(S)$.
  - Los espacios extras aún suman mucho, $o(\sigma n)$.
  - Si usáramos la técnica práctica vista, el total sería razonable: $nH_0(S) + O(n)$.
  - El tiempo de *rank* sería cercano a $O(\log \sigma)$, y el *select* sería constante.
  - Pero aún no podemos obtener $S[i]$!
- Estas ideas necesitan que además tengamos S.
- ¿Se puede hacer mejor?

Parte I: Secuencias

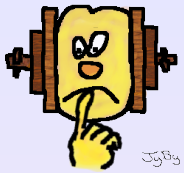


El Wavelet Tree



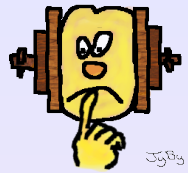
- Es una solución elegante que permite almacenar $S[1,n]$:
  - Usando $n\log\sigma + o(n\log\sigma)$ bits.
  - Resolviendo *rank* y *select* en tiempo $O(\log\sigma)$.
  - Obteniendo $S[i]$ en tiempo $O(\log\sigma)$.
- Se puede mejorar a espacio $nH_0(S) + o(n\log\sigma)$.
- Tiene muchas otras aplicaciones (veremos algunas).

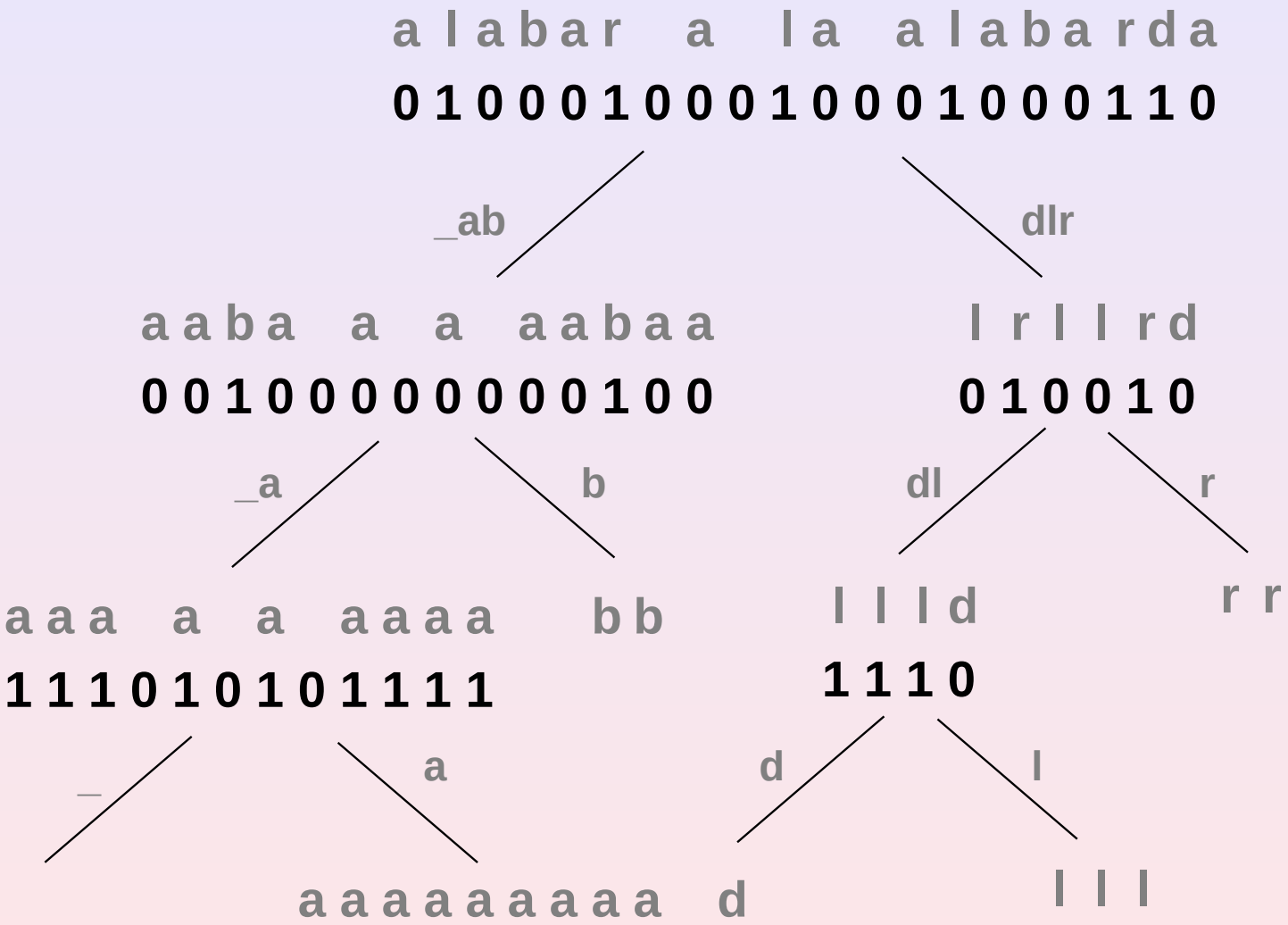
El Wavelet Tree



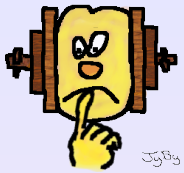
- Supongamos que partimos el alfabeto en dos subconjuntos del mismo tamaño (o casi).
- Creamos un bitmap indicando a qué subconjunto pertence cada letra de S.
- Guardamos ese bitmap en la raíz del wavelet tree.
- Para el subárbol izquierdo/derecho elegimos las letras de S de cada subconjunto.
- Continuamos recursivamente hasta que cada subconjunto tenga una sóla letra.
- Es fácil ver que todos los bitmaps suman $n \log \sigma$ bits.

El Wavelet Tree



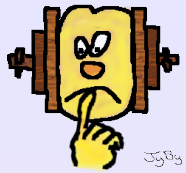


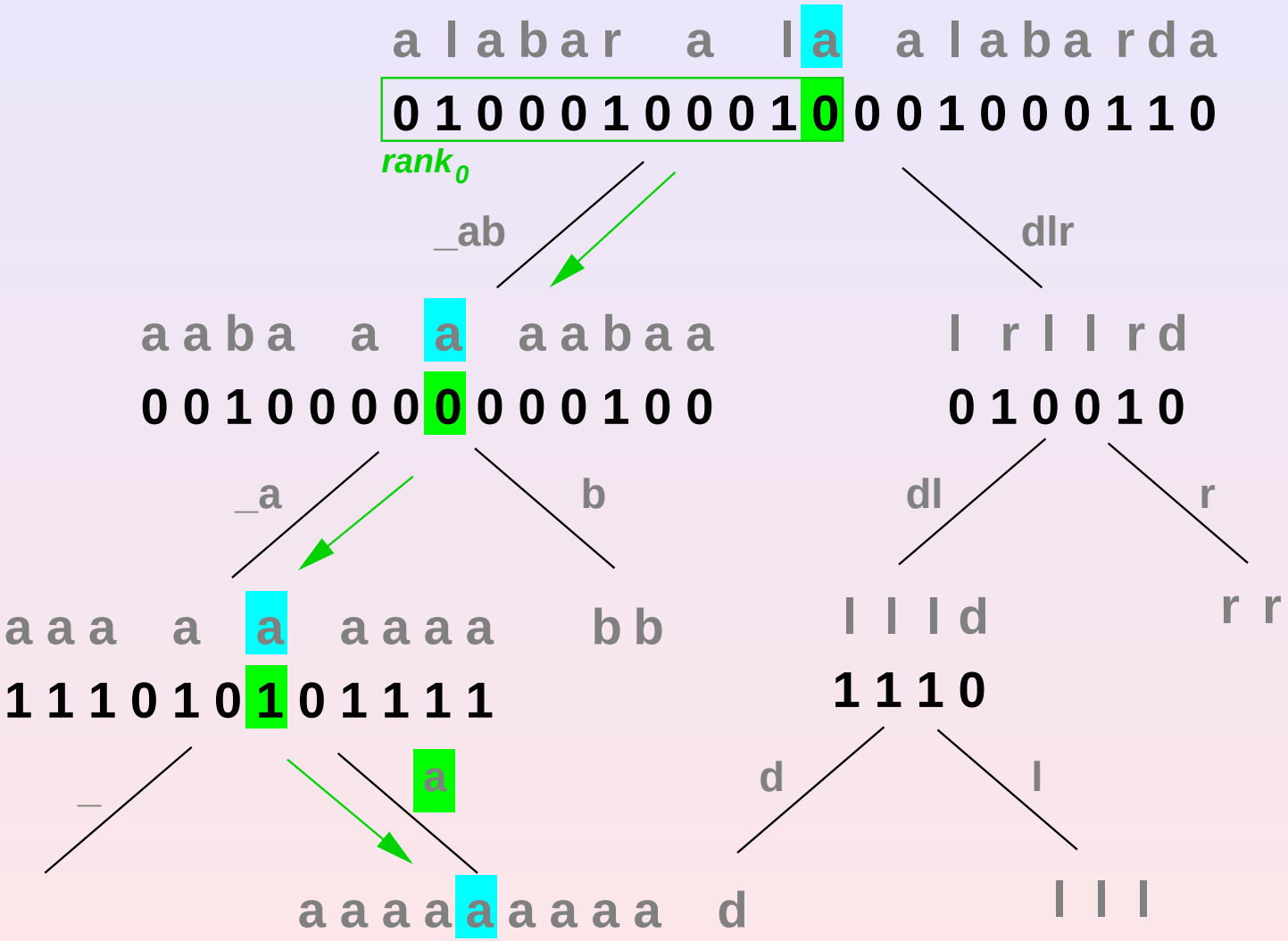
El Wavelet Tree



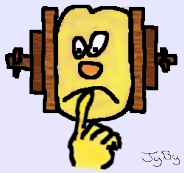
- ¿Cómo recuperamos una letra $S[i]$?
  - Miramos $B[i]$ en la raíz.
  - Si $B[i] = 0$,
    - Nos vamos al subárbol izquierdo.
    - La nueva posición es $i' = rank_0(B, i)$.
  - Si $B[i] = 1$,
    - Nos vamos al subárbol derecho.
    - La nueva posición es $i' = rank_1(B, i)$.
  - Cuando llegamos a una hoja, la letra correspondiente es $S[i]$.
- Tiempo: $\log \sigma$ evaluaciones de *rank*.
- Necesitamos preprocesar los bitmaps para *rank*.

El Wavelet Tree



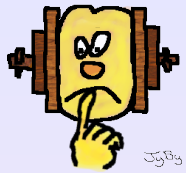


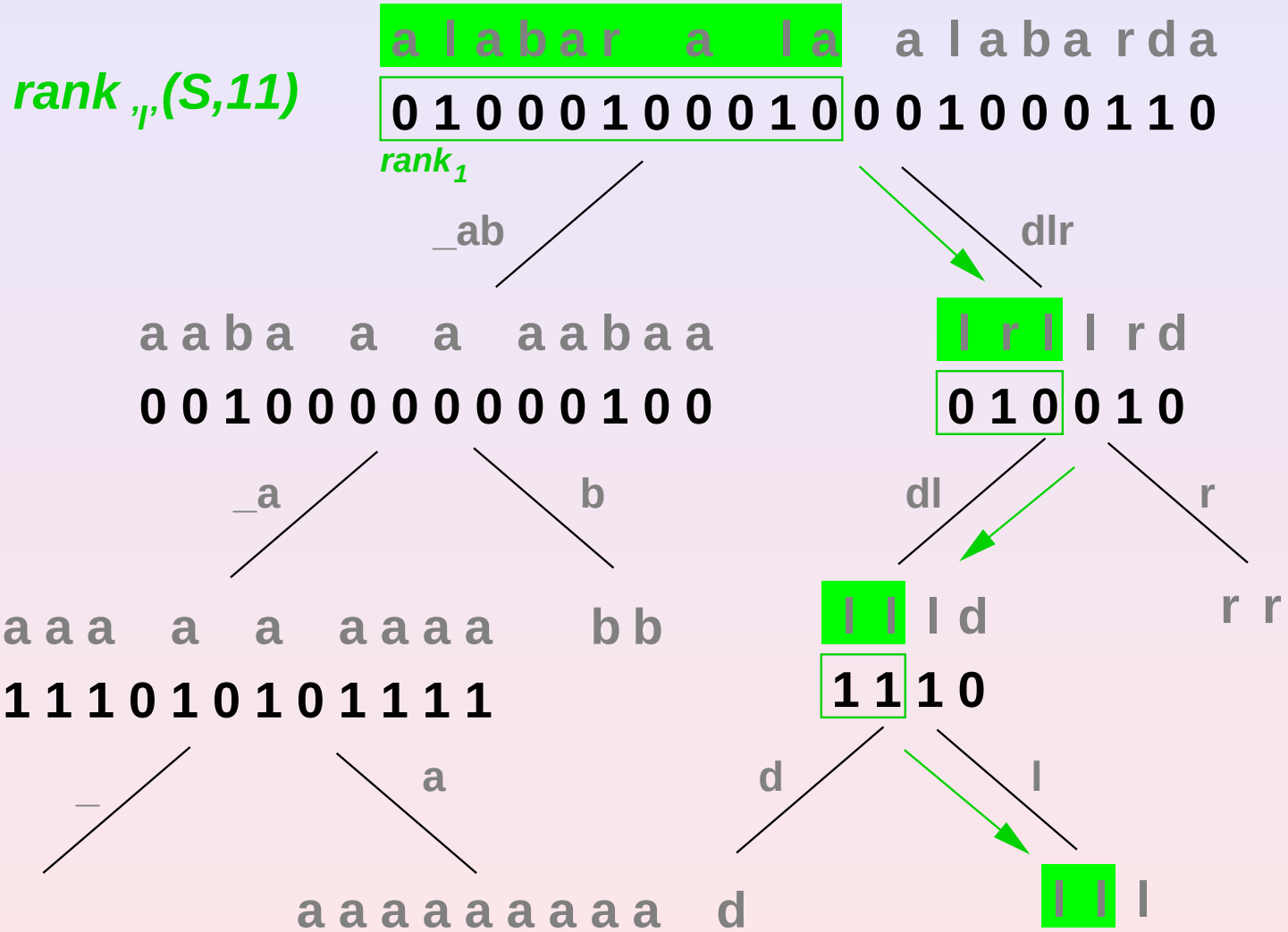
El Wavelet Tree



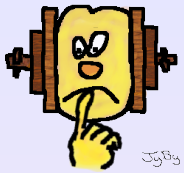
- ¿Cómo calculamos $rank_c(S, i)$?
  - Sea B el bitmap de la raíz.
  - Si c pertenece al subárbol izquierdo,
    - Nos vamos al subárbol izquierdo.
    - La nueva posición es $i' = rank_0(B, i)$.
  - Si c pertenece al subárbol derecho,
    - Nos vamos al subárbol derecho.
    - La nueva posición es $i' = rank_1(B, i)$.
  - Cuando llegamos a una hoja, la respuesta es i.
- Tiempo: $\log \sigma$ evaluaciones de $rank$.

El Wavelet Tree



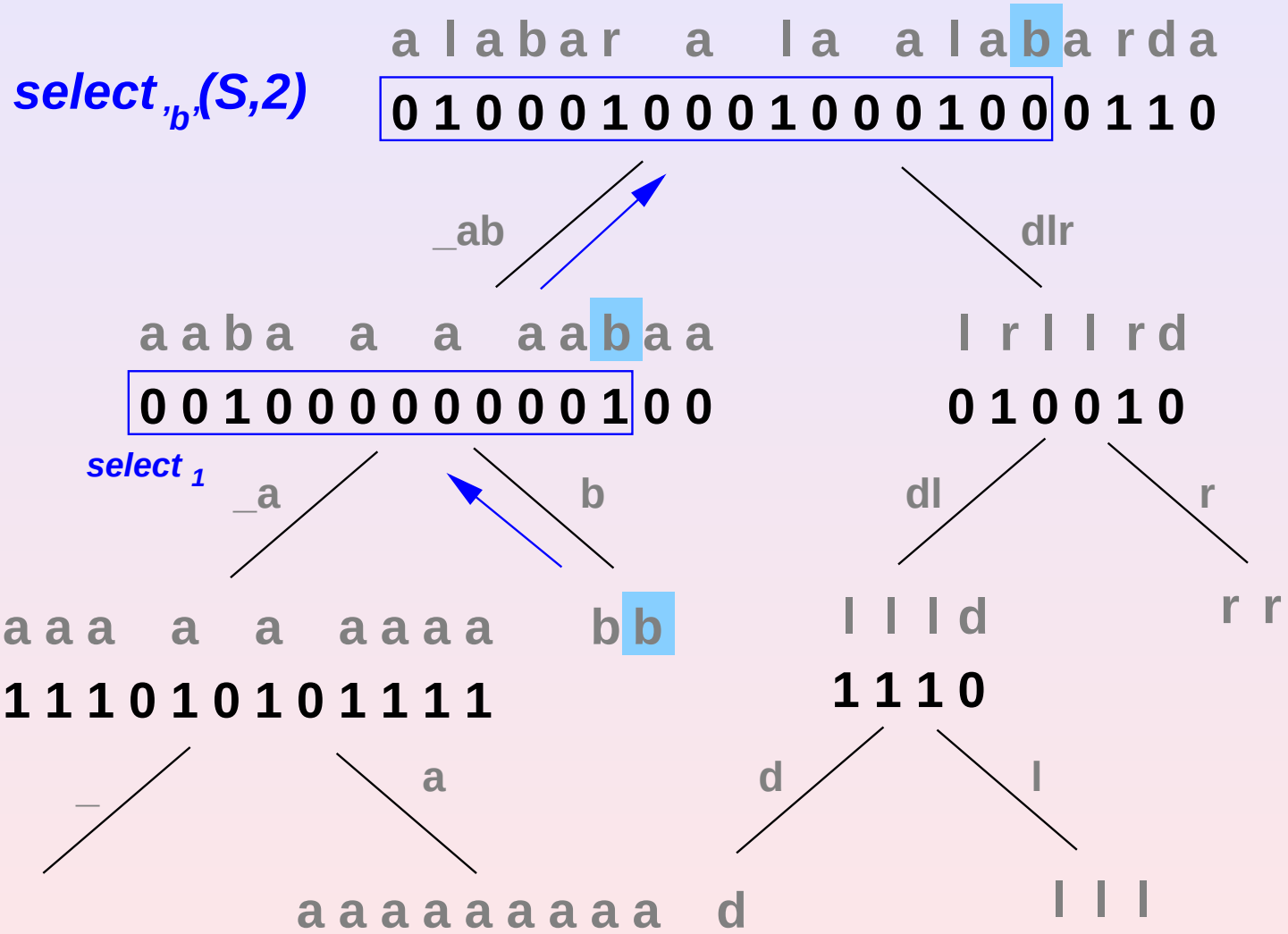


El Wavelet Tree

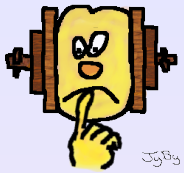


- ¿Cómo calculamos $select_c(S, i)$?
  - Nos vamos a la hoja correspondiente a c.
  - Sea B el bitmap de su padre.
  - Si la hoja es hijo izquierdo de su padre,
    - Nos vamos al padre.
    - La nueva posición es $i' = select_0(B, i)$.
  - Si la hoja es hijo derecho de su padre,
    - Nos vamos al padre.
    - La nueva posición es $i' = select_1(B, i)$.
  - Cuando llegamos a la raíz, la respuesta es i.
- Tiempo: $\log \sigma$ evaluaciones de *select*.
- Necesitamos preprocesar los bitmaps para *select*.

El Wavelet Tree

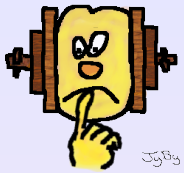


Extensiones



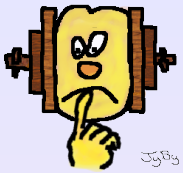
- Si representamos los bitmaps como secuencias comprimidas, obtenemos $nH_0(S) + o(n \log \sigma)$ bits.
- Si utilizamos bitmaps dinámicos, podemos insertar y eliminar letras en tiempo $O(\log n \log \sigma)$.
- En ese caso las otras operaciones también cuestan $O(\log n \log \sigma)$.
- Podríamos mejorar los tiempos con wavelet trees multiarios.
- Para ello necesitaríamos manejar secuencias de símbolos en cada nivel del wavelet tree, en tiempo constante.
- Esto se puede hacer con la técnica de los pares (c, o), para alfabetos pequeños, $o(\log n / \log \log n)$.
- En total, todos los tiempos $O(\log \sigma)$ se pueden mejorar a $O(1 + \frac{\log \sigma}{\log \log n})$.

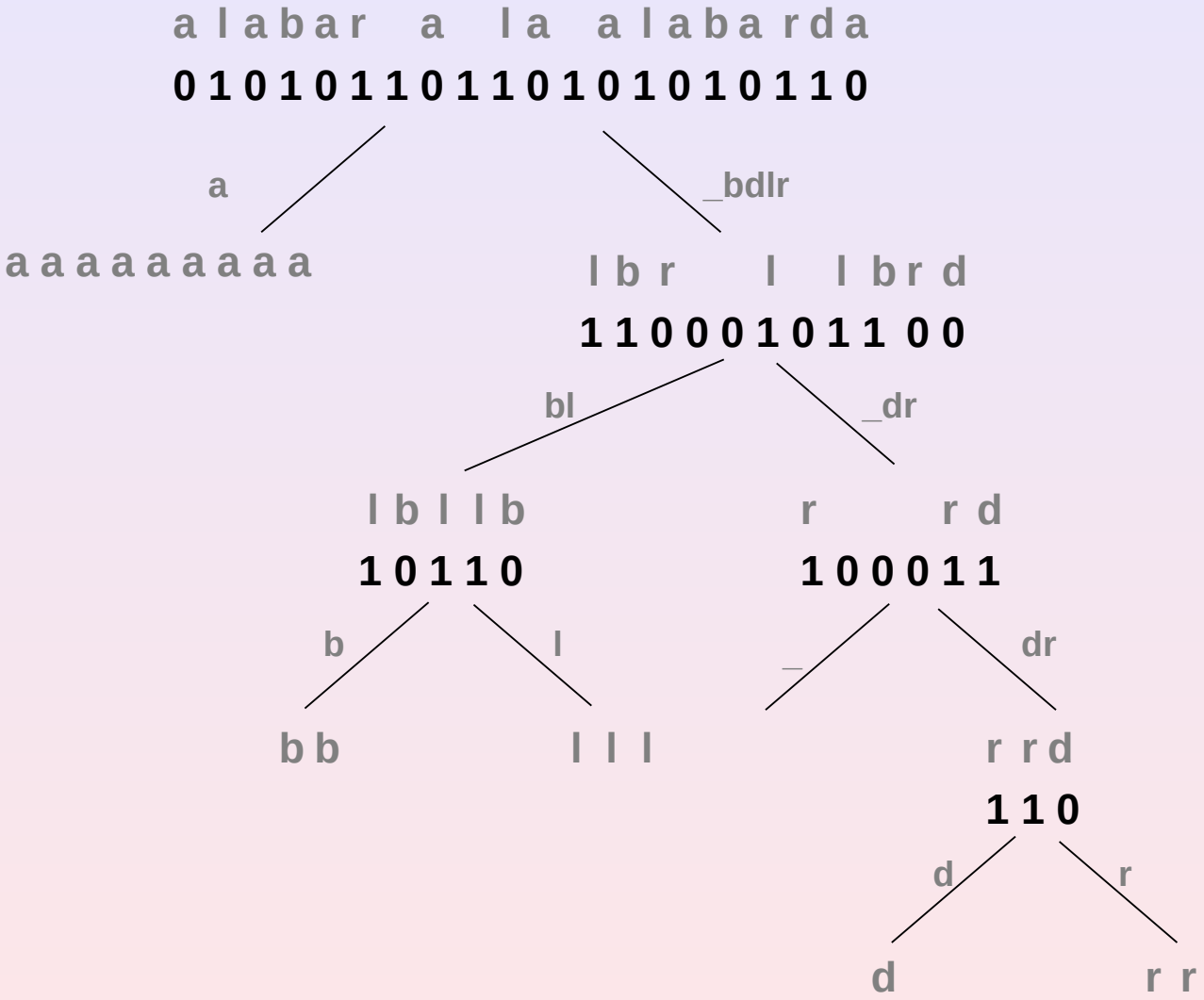
Huffman Wavelet Trees



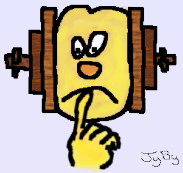
- Veremos una alternativa práctica para obtener casi $nH_0(S) + o(n \log \sigma)$ bits.
- En vez de utilizar bitmaps comprimidos, usaremos los explícitos.
- Pero le daremos forma de árbol de Huffman al wavelet tree.
- No es difícil ver que el largo total de las tiras de bits...
- ... es exactamente el largo de S comprimida con Huffman.
- Este largo es $< n(H_0(S) + 1)$.
- Además, si los caracteres se acceden con la misma frecuencia que tienen en S...
- el tiempo promedio de acceso baja a $O(H_0(S))$.
- Se puede limitar la altura a $O(\log \sigma)$ para el peor caso.

Huffman Wavelet Trees



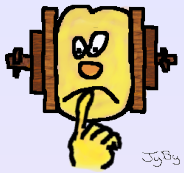


Un Problema Geométrico



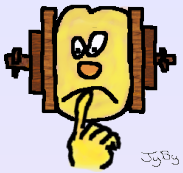
- Tenemos una grilla discreta de $[1, n] \times [1, n]$ (para simplificar, pero podría ser rectangular también).
- Almacenamos t puntos en esa grilla (no más de un punto por celda, para simplificar).
- Queremos responder consultas de rangos:
  - ¿Cuántos puntos hay en $[x_1, x_2] \times [y_1, y_2]$?
  - ¿Qué puntos hay en $[x_1, x_2] \times [y_1, y_2]$?
- Un wavelet tree puede resolver este problema:
  - Usando $(n + t \log n)(1 + o(1))$ bits de espacio.
  - Calculando la cantidad de puntos en tiempo $O(\log n)$.
  - Reportando cada punto en tiempo $O(\log n)$.

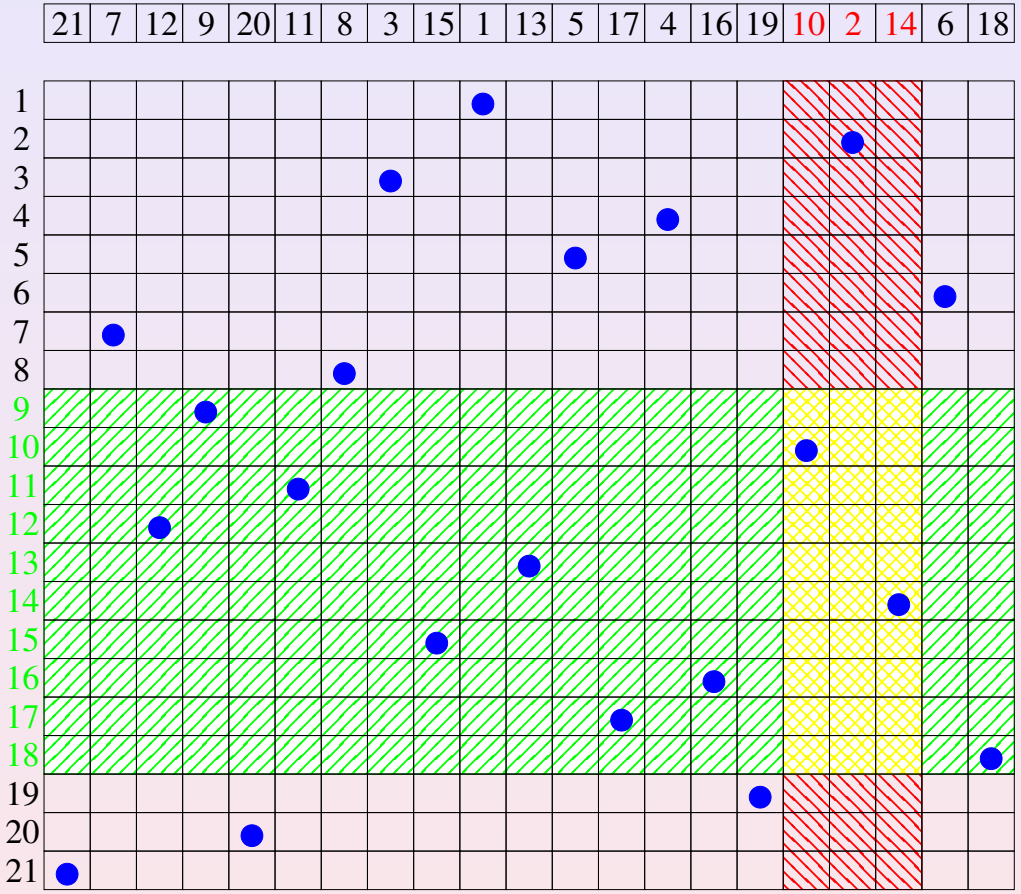
Un Problema Geométrico



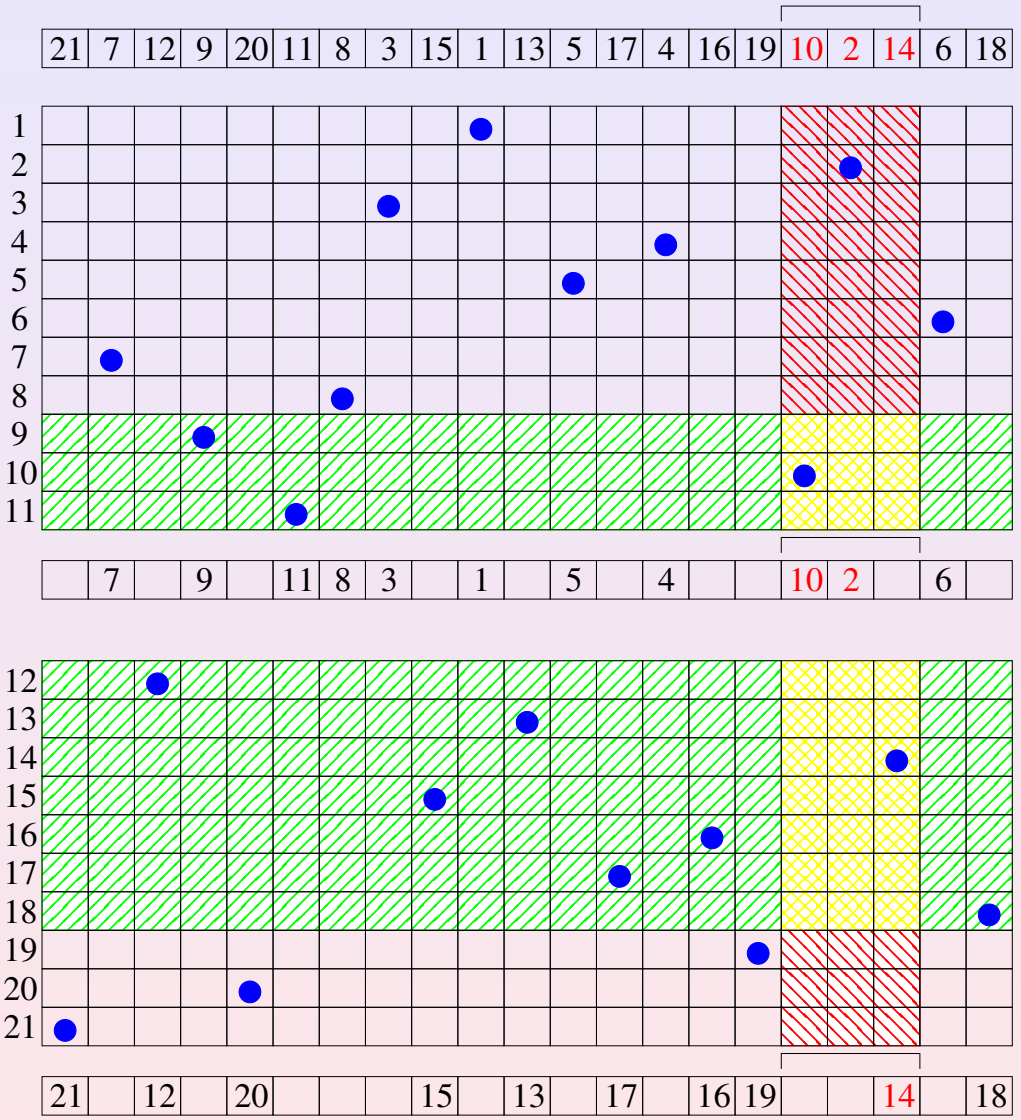
- Comencemos con un caso simplificado: Hay exactamente un punto por valor en coordenada x.
- Entonces las coordenadas y de los puntos, ordenados por coordenada x, forman una secuencia $S[1, n]$.
- Construimos el wavelet tree sobre S.
- Los segmentos en S corresponden a rangos en el eje x.
- Las particiones en mitades que hace el wavelet tree corresponden al eje y.
- Dado un punto $p = (x, y)$, vale que $S[x] = y$.
- Si seguimos a p por el árbol, usando *rank*, llegamos a la y-ésima hoja.
- Si seguimos un valor y desde una hoja, usando *select*, llegamos a la posición x correspondiente en la raíz.

Un Problema Geométrico

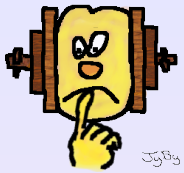


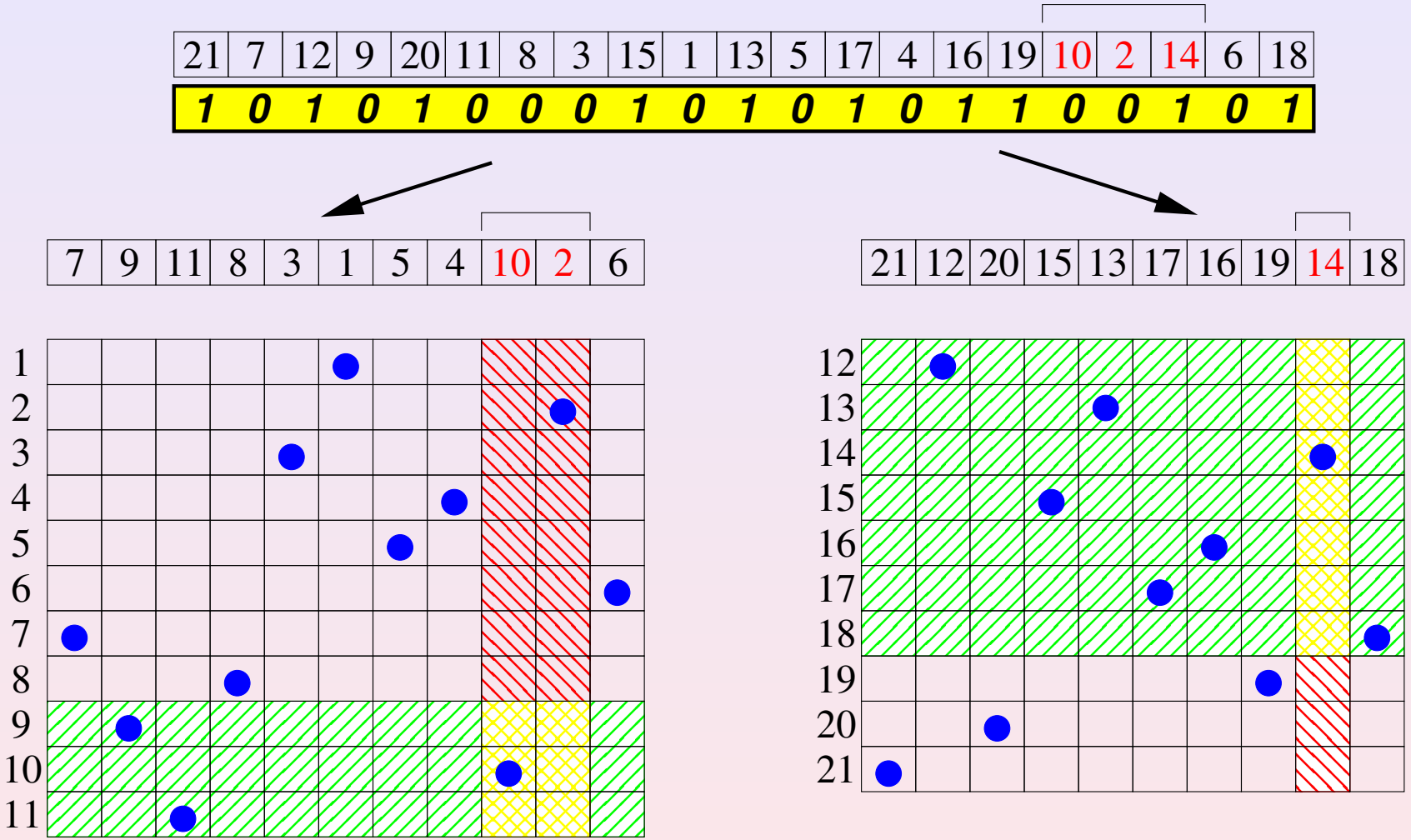


Un Problema Geométrico

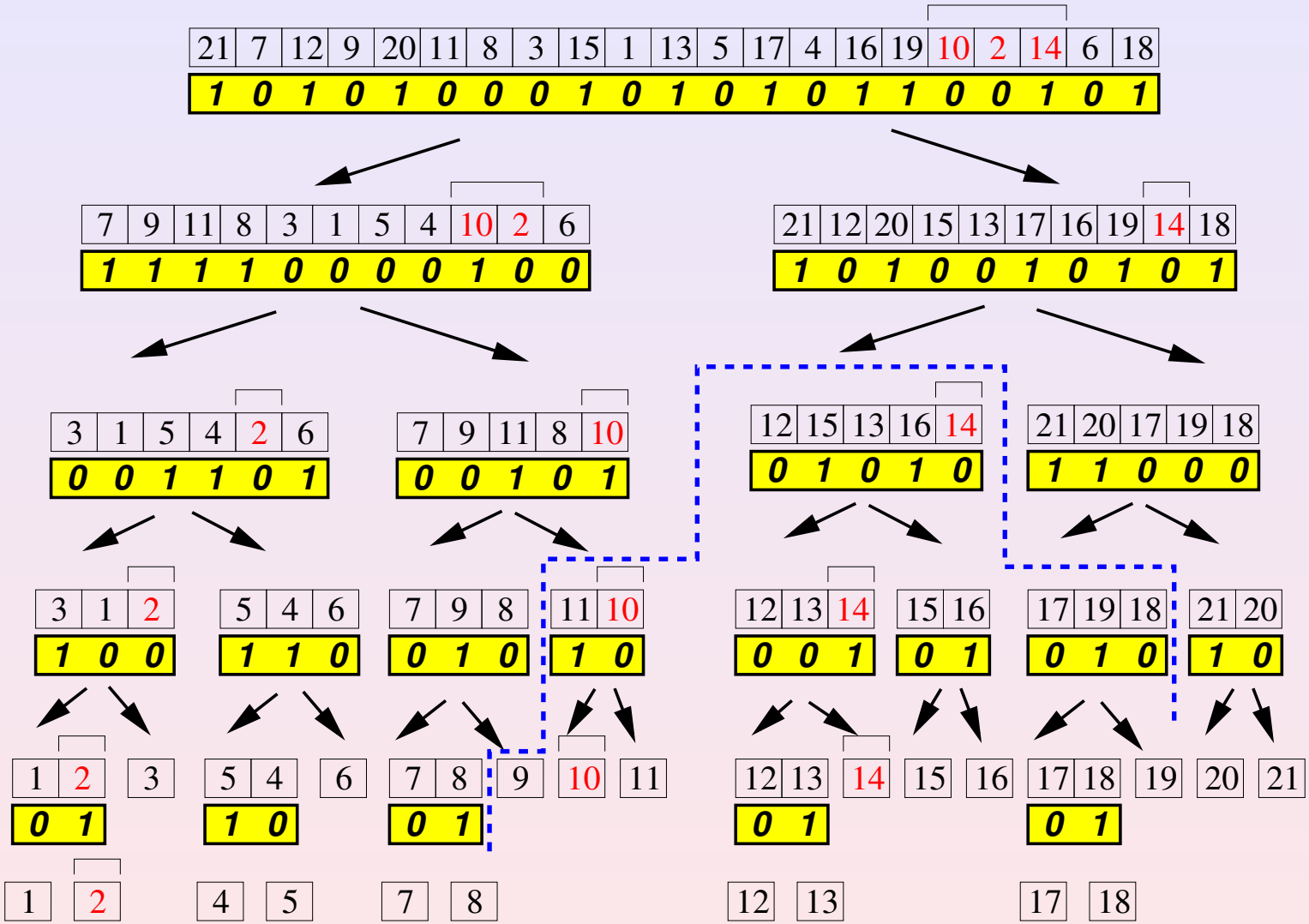


Un Problema Geométrico

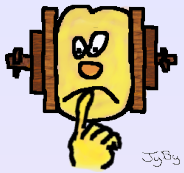




Un Problema Geométrico

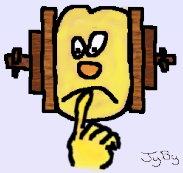


Un Problema Geométrico



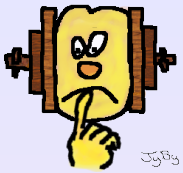
- Más en general, necesitaremos proyectar un rango de valores x:
  - Si en un nodo v, correspondiente al rango $[y, y']$,
  - tenemos un rango $B[x, x']$ en el bitmap de v,
  - entonces $[1 + rank_0(B, x-1), rank_0(B, x')]$ es el rango de los puntos esos que caen en la primera mitad de $[y, y']$,
  - y $[1 + rank_1(B, x-1), rank_1(B, x')]$ es el rango de los puntos esos que caen en la segunda mitad de $[y, y']$.

Un Problema Geométrico



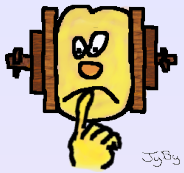
- Para contar cuántos puntos hay en $[x_1, x_2] \times [y_1, y_2]$:
  - Partimos en la raíz con el rango $[x, x'] = [x_1, x_2]$.
  - Proyectamos el rango en ambos subárboles.
  - Continuamos recursivamente, deteniéndonos cuando:
    - El intervalo $[x, x']$ está vacío (no hay puntos del rango original $[x_1, x_2]$ que caen en el rango $[y, y']$ de este nodo).
    - El intervalo $[y, y']$ es disjunto con el original ($[y_1, y_2]$).
    - El intervalo $[y, y']$ está contenido en el original ($[y_1, y_2]$): Sumar $x' - x + 1$ al total
  - Todo subintervalo de $[y_1, y_2]$ se cubre con $O(\log n)$ nodos $[y, y']$ del wavelet tree.
  - El algoritmo hace $O(\log n)$ operaciones.

Un Problema Geométrico



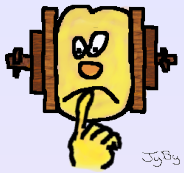
- Para encontrar esos puntos.
  - Partir de cada nodo en el que contamos resultados.
  - Seguir cada punto de $[x, x']$ hacia abajo hasta descubrir su coordenada y.
  - Y/o seguir cada punto de $[x, x']$ hacia arriba hasta descubrir su coordenada x.
- Esto claramente cuesta $O(\log n)$ por cada punto.
- Debemos concatenar los bitmaps de cada nivel para no tener demasiados punteros en el árbol.

Un Problema Geométrico



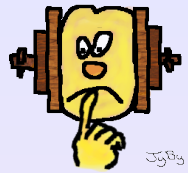
- Eliminemos la simplificación de un punto por coordenada.
- El wavelet tree ahora maneja una posición por cada punto almacenado (t puntos).
- En la raíz los puntos se ordenan columna a columna (y por fila dentro de cada columna).
- Un bitmap $X[1, n + t]$ tiene un 1 por cada cambio de columna y un 0 por cada nuevo punto.
- El árbol va subdividiendo los puntos según coordenada y.
- En las hojas los puntos se encuentran leídos fila a fila (y por columna dentro de cada fila).
- No necesito un equivalente a Y para leer por filas, pues sé en qué hoja del wavelet tree estoy.

Permutaciones



- La variante simplificada de puntos en $n \times n$ también se puede ver como una permutación.
- Una permutación se puede almacenar en $n \log n$ bits y, dado i, calcular $\pi(i)$ en tiempo constante.
- Si quisiera también la permutación inversa $\pi^{-1}(i)$, necesitaría otros $n \log n$ bits.
- Con un wavelet tree sobre S represento implícitamente ambas permutaciones con $n \log n + o(n \log n)$ bits y calculo:
  - $\pi(i) = S[i]$ en tiempo $O\left(\frac{\log n}{\log \log n}\right)$.
  - $\pi^{-1}(i) = \mathit{select}_i(S, 1)$ en tiempo $O\left(\frac{\log n}{\log \log n}\right)$.

Permutaciones

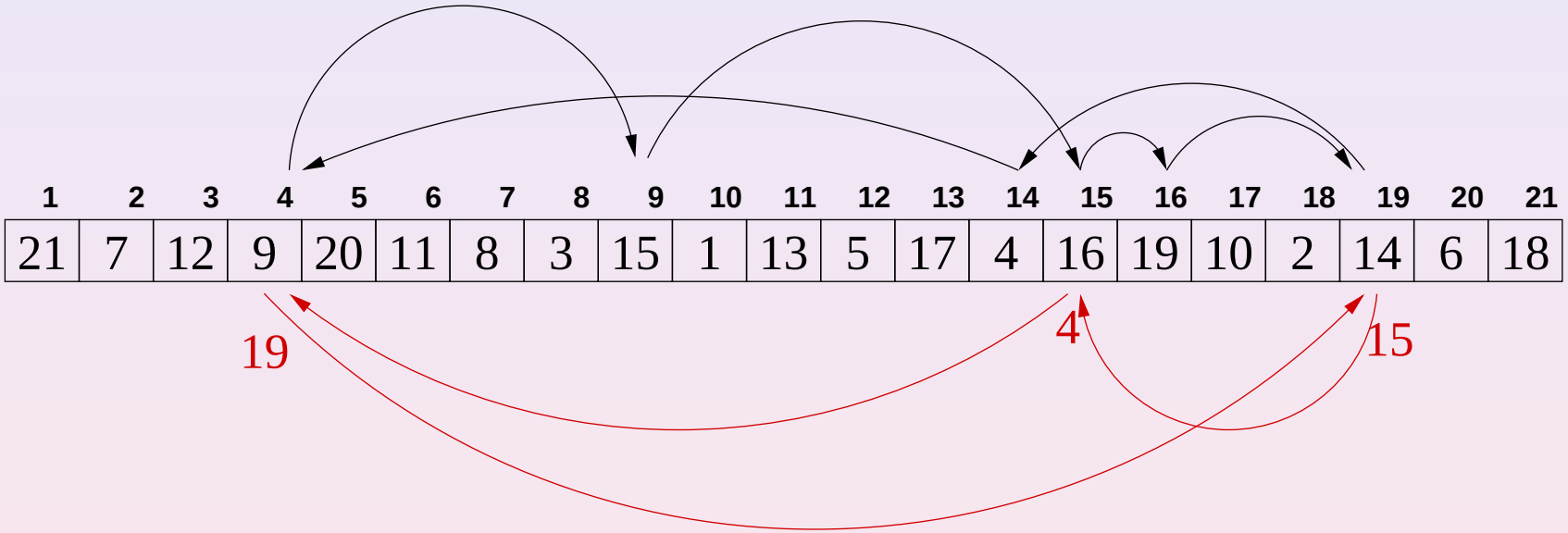


- ¿Se puede hacer mejor? Sí.
- Un ciclo de una permutación es una secuencia

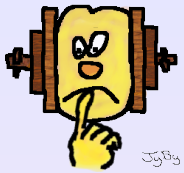
$$i,\ \pi(i),\ \pi(\pi(i)),\ \pi(\pi(\pi(i))),\ \ldots,\pi^{-1}(i),\ i$$

- Toda permutación de $[1,n]$ se descompone en ciclos.
- Para calcular $\pi^{-1}(i)$, basta seguir el ciclo desde i hasta que volvamos a i
  - El último valor visitado antes de volver a i es $\pi^{-1}(i)$.
- Pero el ciclo puede ser muy largo!
- Cada t valores de un ciclo largo, introduciremos punteros reversos que retrocedan t valores en el ciclo.
- Si parto de i, puedo tomar el primer puntero reverso que encuentre, y llegaré a i de vuelta en a lo más $t+1$ pasos.

Permutaciones

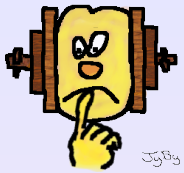


Permutaciones



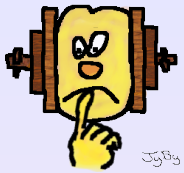
- Cómo almaceno eficientemente los punteros reversos?
  - Marco en un bitmap $R[1,n]$ qué posiciones tienen punteros reversos.
  - Almaceno sus valores en forma compacta en $P[1,s]$, $s \leq n/t$.
  - Si $R[i] = 1$, hay un puntero reverso en i, y su valor está en $P[rank(R,i)]$.
- En total ocupo $n \log n + (n/t) \log n + n + o(n)$ bits.
- Resuelvo $\pi(i)$ en $O(1)$ y $\pi^{-1}(i)$ en $O(t)$.
- Por ejemplo:
  - Con $(1+\epsilon)n \log n + O(n)$ bits, para ϵ constante, resuelvo π en tiempo $O(1)$ y π^{-1} en tiempo $O(1/\epsilon) = O(1)$.
  - Con $n \log n + O(n \log \log n)$ bits, resuelvo π en tiempo $O(1)$ y π^{-1} en tiempo $O(\log n / \log \log n)$ (mejor que con wavelet trees).

Permutaciones



- Y si quisiera resolver π^k y π^{-k} ?
  - Imaginemos que escribimos los ciclos explícitamente en S.
  - $\pi(i)$ sigue a i en S; π^{-1} precede a i.
  - Marcamos en un bitmap $C[1, n]$ dónde comienzan los ciclos.
  - Almacenamos una permutación π' que me lleva de la secuencia original a la posición donde se mapeó en S.
  - Represento π' con la técnica anterior.
  - Eso basta para calcular cualquier π^k y π^{-k} en tiempo $O(t)$.

Permutaciones

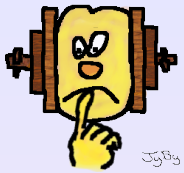


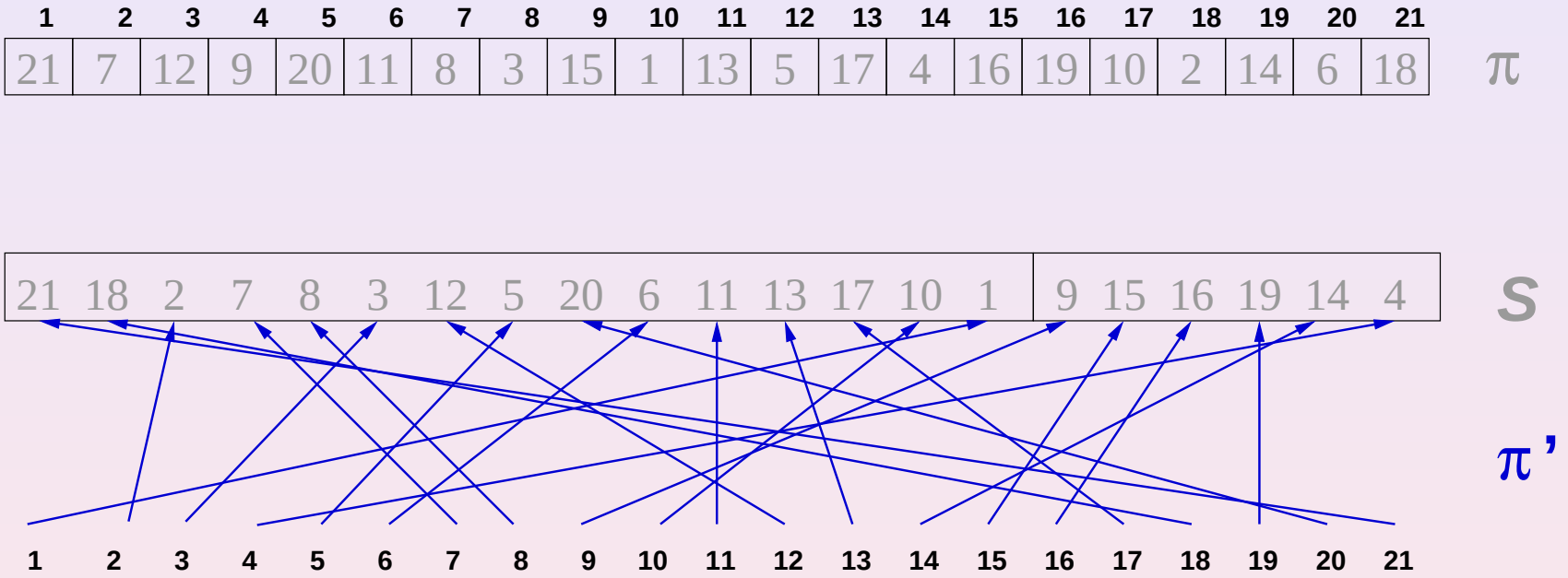
- Para calcular $\pi^k(i)$ (k positivo o negativo):
  - Con $i' = \pi'(i)$ ubico i en S.
  - Calculo los límites del ciclo:
    - $l = select(C, rank(C, l))$
    - $r = select(C, rank(C, l) + 1)$
  - La posición que corresponde a $\pi^k(i)$ es

$$j' = l + (i' + k - l \mod (r - l))$$

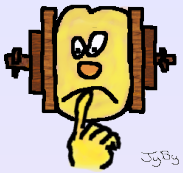
  - Finalmente vuelvo: $\pi^k(i) = \pi'^{-1}(j')$
- Se puede extender a mapeos (no lo veremos).

Permutaciones

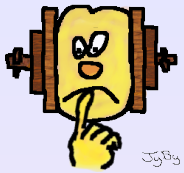




Parte I: Secuencias

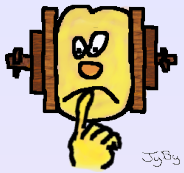


Revisitando las Secuencias de Bits



- Volvamos a la solución de secuencias de bits, que promete mejores tiempos.
- Conseguiremos tiempos de la forma $O(\log\log\sigma)$ ...
- ... y espacio $n\log\sigma + n \cdot o(\log\sigma)$.
- Cortaremos las secuencias de bits en bloques de largo σ.
- Resolveremos separadamente dos subproblemas:
  - Determinar la respuesta a nivel de bloque.
  - Refinar la respuesta dentro del bloque.

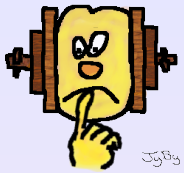
Revisitando las Secuencias de Bits



Nivel de bloques

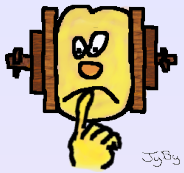
- Consideremos una secuencia de bits particular B_c.
- La representaremos con otra secuencia de bits L_c, donde
  - Comenzaré poniendo un 0.
  - Iré agregando los 1's que aparecen en B_c.
  - Iré agregando un 0 cuando cambie de bloque.
- Sumando sobre todas las L_c, tengo n 1's y $\leq n$ 0's.

Revisitando las Secuencias de Bits



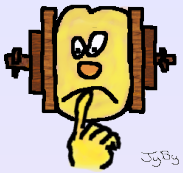
	a	l	a	b	a	r		a		l	a		a	l	a	b	a	r	d	a
$B_{\_}$	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0
B_{a}	1	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1
B_{b}	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
B_{d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
B_{l}	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
B_{r}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
$L_{\_}$	0	0	1	1	1	0	0	0												
L_{a}	0	1	1	1	0	1	1	0	1	1	1	0	1	0						
L_{b}	0	1	0	0	1	0	0													
L_{d}	0	0	0	0	1	0														
L_{l}	0	1	0	1	0	1	0													
L_{r}	0	1	0	0	1	0	0													

Revisitando las Secuencias de Bits



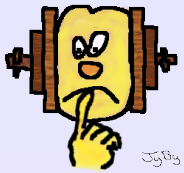
- Para calcular $rank_c(S, i) = rank(B_c, i)$:
  - Calculo $b = 1 + \lfloor i/\sigma \rfloor$.
  - Calculo $r = rank_1(L_c, select_0(L_c, b))$.
  - b es el bloque donde debo completar la respuesta.
  - r es la cantidad de ocurrencias de c antes del bloque b.
  - Dentro del bloque b debo calcular $rank_c(1 + (i \bmod \sigma))$.
- Para calcular $S[i]$:
  - Calculo $b = 1 + \lfloor i/\sigma \rfloor$.
  - b es el bloque donde debo buscar la respuesta.
  - Dentro del bloque b debo obtener el símbolo en la posición $1 + (i \bmod \sigma)$.

Revisitando las Secuencias de Bits



- Para calcular $select(S, i) = select(B_c, i)$:
  - Calculo $j = select_1(L_c, i)$.
  - Calculo $b = rank_0(L_c, j)$.
  - b es el bloque donde debo completar la respuesta.
  - Dentro del bloque b debo calcular $select_c(j - select_0(L_c, b))$.
  - A esa posición debo sumarle $b \cdot \sigma$.

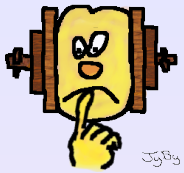
Revisitando las Secuencias de Bits



Resolviendo dentro de un bloque

- Almacenamos las posiciones donde ocurre `"a"` en orden, luego donde ocurre `"b"` en orden, etc.
- El resultado es una permutación π de $[1, \sigma]$.
- Además almaceno un bitmap $P[1, 2\sigma]$ con un 1 por cada ocurrencia almacenada en π de cada símbolo, insertando un 0 al principio y cada vez que se cambia de símbolo.
- Entonces $select_c(i) = \pi(select_0(P, c) - c + i - 1)$.
- Asimismo, $S[i] = rank_0(P, select_1(P, \pi^{-1}(i)))$.

Revisitando las Secuencias de Bits

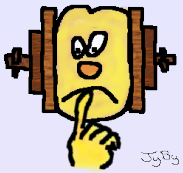


	a	l	a	b	a	r		a		l	a		a	l	a	b	a	r	d	a
$B_{\_}$	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0
B_a	1	0	1	0	1	0	0	1	0	0	1	0	1	0	1	0	1	0	0	1
B_b	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
B_d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
B_l	0	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
B_r	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0

π	1	3	5	4	2	6
P	0	0	1	1	1	0

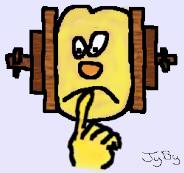
P	0	0	1	1	1	0	1	0	1	0

Revisitando las Secuencias de Bits



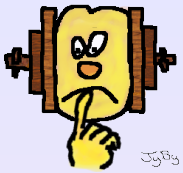
- Lo más difícil es $rank_c(S, i)$:
  - La zona de las ocurrencias de c en π es $[l, r] = [select_0(P, c) - c + 1 \ldots select_0(P, c + 1) - (c + 1)]$.
  - Si hago búsqueda binaria consigo tiempo $O(\log \sigma)$.
  - Para conseguir $O(\log \log \sigma)$:
    - Anoto los valores $\pi[i \cdot \log \sigma]$ en una lista.
    - Esta lista tiene $\frac{\sigma}{\log \sigma}$ números en $[1, \sigma]$.
    - La separo en las sublistas (crecientes) que corresponden a cada c.
    - Necesito calcular el predecesor y sucesor de i en una de esas sublistas.
    - Lo hago en tiempo constante con bitmaps comprimidos.
    - Completo la búsqueda binaria en el tramo de $\log \sigma$ valores.

Revisitando las Secuencias de Bits



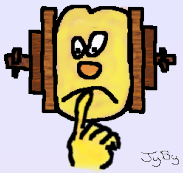
- Los bitmaps L_c ocupan $2n + o(n)$ bits.
- Las permutaciones π ocupan $n \log \sigma$ bits.
- Para obtener π^{-1} en tiempo $O(\log \log \sigma)$ necesitamos punteros reversos que ocupen $O(n \cdot \frac{\log \sigma}{\log \log \sigma}) = n \cdot o(\log \sigma)$.
- Los bitmaps P ocupan $2n + o(n)$ bits (son los L_c reordenados).
- Los bitmaps para predecesor y sucesor ocupan $O(\frac{n}{\sigma} \cdot \frac{\sigma}{\log \sigma} \cdot \log \frac{\sigma}{\sigma / \log \sigma}) = O(n \cdot \frac{\log \log \sigma}{\log \sigma})$ bits.
- Total $n \log \sigma + O(n \cdot \frac{\log \sigma}{\log \log \sigma})$ bits.
- $S[i]$ y $rank_c$ se calculan en tiempo $O(\log \log \sigma)$.
- $select_c$ se calcula en tiempo constante.

Relaciones Binarias



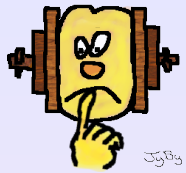
- La solución se puede extender a relaciones binarias $\mathcal{R}$ entre dos conjuntos A y B.
- Si $|A| = n$ y $|B| = m$, el punto (i, j) existe sii $a_i \mathcal{R} b_j$.
- Queremos resolver las siguientes operaciones eficientemente:
  - Dados a_i y b_j, determinar si $a_i \mathcal{R} b_j$.
  - Dado a_i, contar/listar los b_j tal que $a_i \mathcal{R} b_j$.
  - Dado b_j, contar/listar los a_i tal que $a_i \mathcal{R} b_j$.

Relaciones Binarias

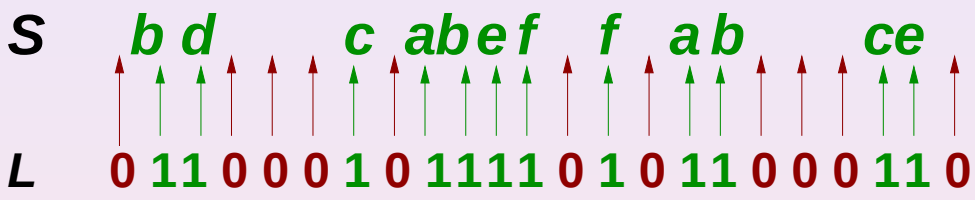


- Si $|\mathcal{R}| = t$, podemos ver la relación binaria como t 1's distribuidos en n bitmaps de largo m.
- Imaginemos que leemos esos 1's columna a columna, y escribimos las filas donde aparecen en una secuencia S.
- Además agregamos a un bitmap L un 1 por cada fila anotada y un 0 por cada nueva columna (más uno final).
- Preprocesamos S y L para *rank* y *select*.
- El espacio total es $t \log n + t + m$ bits.
- O $t \log m + t + n$, lo que sea menor.
- Las operaciones cuestan $O(\log \log t)$ por respuesta.

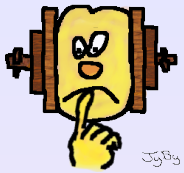
Relaciones Binarias



	1	2	3	4	5	6	7	8	9	10
a	0	0	0	0	1	0	1	0	0	0
b	1	0	0	0	1	0	1	0	0	0
c	0	0	0	1	0	0	0	0	0	1
d	1	0	0	0	0	0	0	0	0	0
e	0	0	0	0	1	0	0	0	0	1
f	0	0	0	0	1	1	0	0	0	0



Relaciones Binarias



- Determinar si un par (a_i, b_j) está relacionado:

$$rank_i(S, select_0(L, j+1) - (j+1)) - rank_i(S, select_0(L, j) - j)$$

- Contar con cuántos pares está relacionado b_j:

$$select_0(L, j+1) - select_0(L, j) - 1$$

- Contar con cuántos pares está relacionado a_i:

$$rank_i(S, t)$$

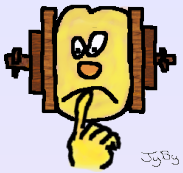
- Listar con quiénes se relaciona b_j

$$S[k + (select_0(L, j) - j)], \quad k = 1, 2, \ldots$$

- Listar con quiénes se relaciona a_i

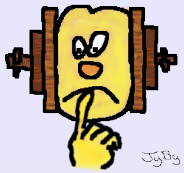
$$rank_0(L, select_1(L, select_k(S, i)))], \quad k = 1, 2, \ldots$$

Relaciones Binarias



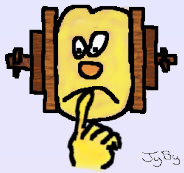
- Con técnicas un poco más sofisticadas...
  - Ocupamos $t \log \frac{mn}{t}$ bits (orden cero).
  - Resolvemos todas las consultas en tiempo $O(\log \log \frac{mn}{t})$.
- Se particiona horizontalmente y luego cada partición se re-particiona verticalmente.
- Eso garantiza una cantidad conveniente de 1's en cada bloque.
- No veremos más detalles.

Relaciones Binarias



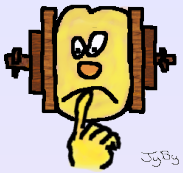
- Aplicación inmediata 1: Grafos $G = (V, E)$
  - $A = B = V$ son los nodos, $E = \mathcal{R}$ son las aristas.
  - El espacio es básicamente $e \log n$, como una lista de adyacencia.
  - Permite saber si una arista está presente en el grafo.
  - Permite listar los vecinos directos y reversos.
  - Permite contarlos eficientemente.
  - Cada respuesta en tiempo $O(\log \log n)$.

Relaciones Binarias

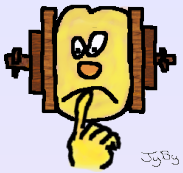


- Aplicación inmediata 2: Indices Invertidos
  - A son los d documentos de una colección y B las p palabras distintas (vocabulario).
  - El espacio es básicamente $t \log p$ bits, menos que una representación plana de la colección.
  - O bien $t \log d$ bits, igual que un índice invertido tradicional.
  - Podemos saber si una palabra aparece en un documento.
  - Podemos listar los documentos donde aparece una palabra, y las palabras que aparecen en un documento.
  - Podemos saber en cuántos documentos distintos aparece una palabra (llamado df y muy importante para calcular relevancia).
  - Y también cuántas palabras distintas tiene un documento.
  - El tiempo es $O(\log \log p)$ ó $O(\log \log d)$.

Parte II: Otras Estructuras

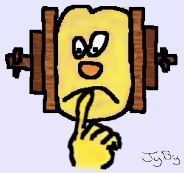


Arboles Generales



- Una estructura clásica de punteros ocupa $O(n \log n)$ bits para un árbol de n nodos.
- Permite navegar en el árbol en tiempo constante.
- Pero realmente hay sólo $\approx 2^{2n}/\sqrt{n}$ árboles distintos con n nodos.
- Se debería poder almacenar con $2n - o(n)$ bits.
- En realidad no es difícil representar un árbol en ese espacio...
- ... el verdadero desafío es poderlo navegar sin descompactarlo.
- Veremos la forma de lograrlo en tiempo constante por operación, usando $2n + o(n)$ bits.

Representación con Paréntesis

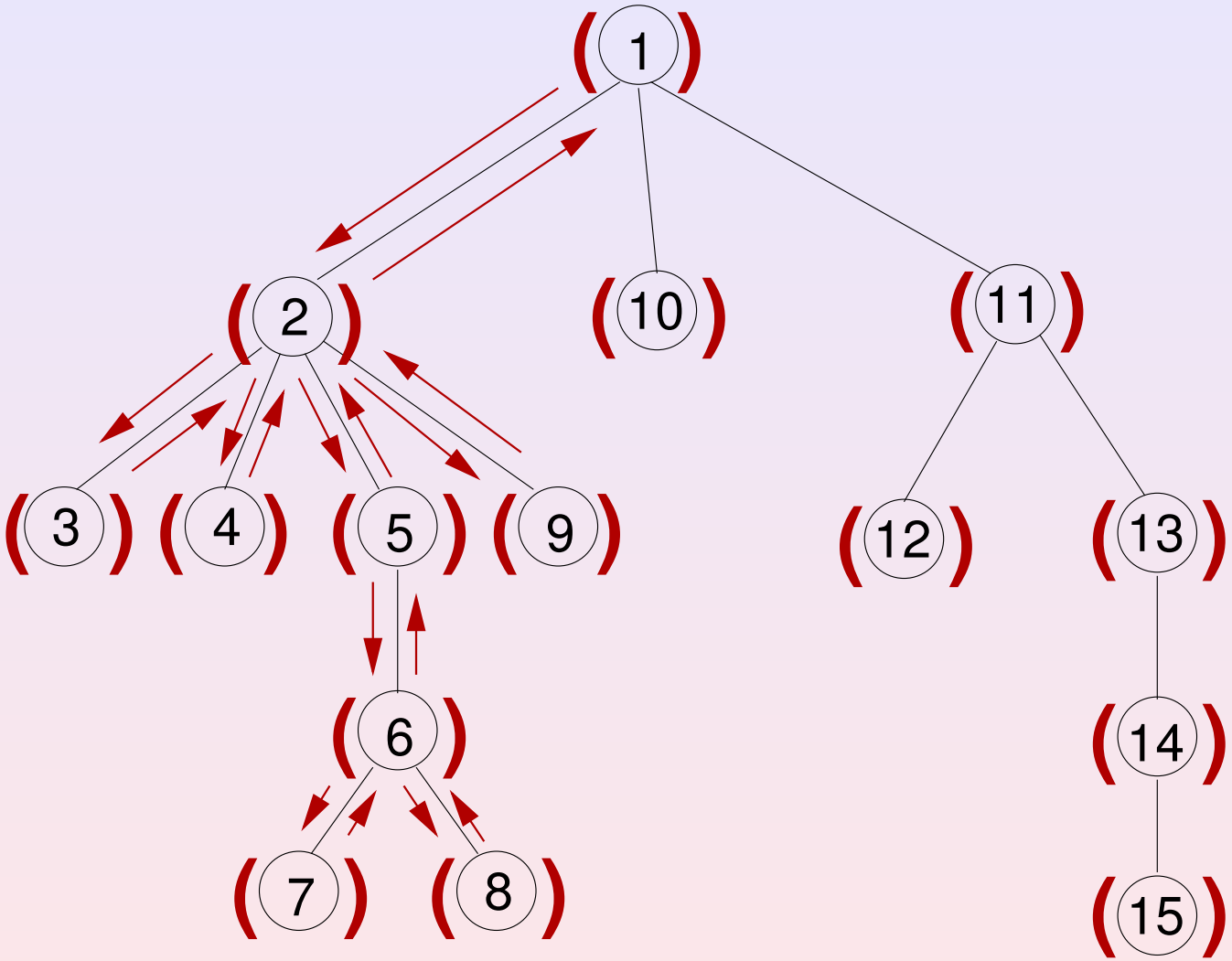


- Una representación muy conocida de árboles generales utiliza paréntesis balanceados.
- Recorremos el árbol en preorden.
- Es decir, al llegar a un nodo:
  - Escribimos un `'('`.
  - Recorremos sus hijos en orden.
  - Escribimos un `')'`.
- O, expresado de otra manera, la representación $R(v)$ de un nodo v con hijos $v_1,\ v_2 \ldots v_k$ es

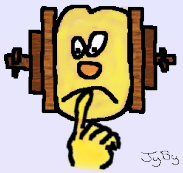
$$R(v) \quad = \quad (\ R(v_1)\ R(v_2)\ \ldots\ R(v_k)\ )$$

- El resultado es una secuencia balanceada de $2n$ paréntesis.

Representación con Paréntesis



Representación con Paréntesis

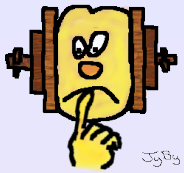


- Una secuencia balanceada de paréntesis cumple:
  - Es una secuencia sobre los símbolos `'('` y `')'`.
  - En total hay tantos `'('` como `')'`.
  - En cualquier punto i de la secuencia, la cantidad de `'('` en $S[1,i]$ es $\geq$ a la cantidad de `')'` en $S[1,i]$.
- Esto último tiene nombre: se llama exceso en i:

$$exceso(S,i) \quad = \quad rank_{'('}(S,i) - rank_{')'}(S,i)$$

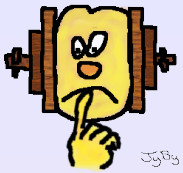
y la propiedad establece simplemente que $exceso(S,i) \geq 0$ para todo i, y que $exceso(S,2n) = 0$.

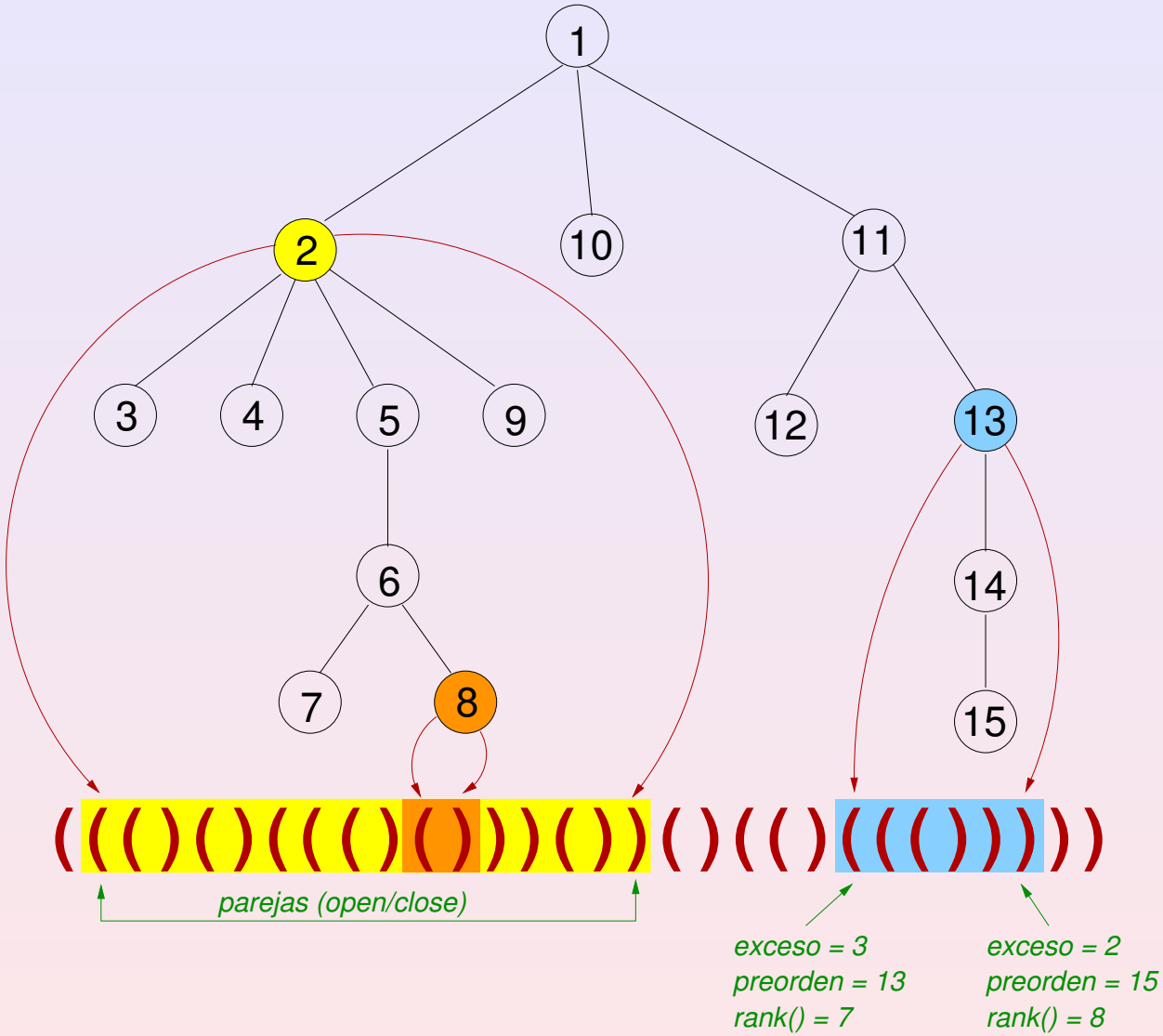
Representación con Paréntesis



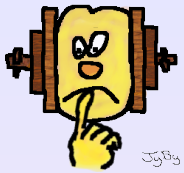
- Llamaremos pareja de un `'('` al `')'` que lo cierra, y pareja de un `')'` al `'('` que lo abre.
- Si $S[v] =' ('$, llamaremos $close(v)$ a la posición de su pareja.
- Si $S[v] =')'$, llamaremos $open(v)$ a la posición de su pareja.
- Observar:
  - $close(v)$ es el menor $v' > v$ tal que $exceso(v') = exceso(v) - 1$.
  - $open(v')$ es el mayor $v < v'$ tal que $exceso(v) = exceso(v') + 1$.

Representación con Paréntesis





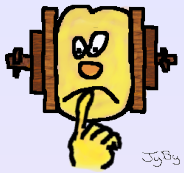
Representación con Paréntesis



Varias propiedades interesantes

- Identificaremos a un nodo v con su `'('` correspondiente.
- Una hoja se ve así: `'()'`.
- u es ancestro de v, sii $[u, close(u)]$ contiene estrictamente a $[v, close(v)]$.
- La profundidad de v en el árbol es $exceso(v)$.
- La posición de v en una enumeración en preorden del árbol es $rank_{'('}(S, v)$.
  - Esto es útil para almacenar información asociada a los nodos.
  - Se puede igualmente precalcular un $rank_{'()'}(S, v)$ para almacenar información asociada a las hojas.
- Con eso tenemos también tamaño de subárbol.

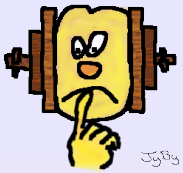
Representación con Paréntesis



Navegación

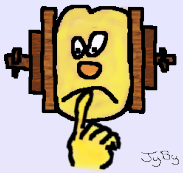
- El siguiente hermano de v es $close(v) + 1$
  - Pero si es un `')'`, v no tiene siguiente hermano.
- El hermano previo de v es $open(v - 1)$
  - Pero si $S[v - 1] =$ `'('`, v no tiene hermano previo.
- El primer hijo de v es $v + 1$
  - Pero si es un `')'`, v no tiene hijos.
- El padre de v no es tan fácil de calcular.
  - Se crea la operación $enclose(v)$.
  - Es el `'('` del nodo más profundo que contiene a v.

Representación con Paréntesis

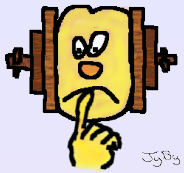


- Por lo tanto, basta con implementar *open*, *close* y *enclose*...
- ... para tener bastantes operaciones de navegación en un árbol representado con paréntesis.
- Conseguiremos tiempo constante usando $2n + o(n)$ bits.
- Hay muchas otras operaciones interesantes:
  - i-ésimo padre, ancestro común más bajo,
  - i-ésimo hijo, hijo con rótulo l,
  - búsqueda de caminos rotulados, ...
- También pueden hacerse eficientemente, con técnicas más avanzadas.
- Asimismo puede comprimirse el árbol.
- Veremos sólo algunas de estas cosas.

Parte II: Otras Estructuras



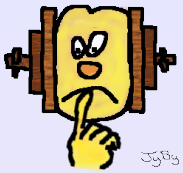
Navegando entre Paréntesis



Operación *close*

- Dividimos la secuencia de paréntesis en bloques de largo $b = \frac{\log n}{2}$.
- Un paréntesis es cercano si su pareja está en el mismo bloque, sino es lejano.
- Resolver $close(v)$ para un v cercano es fácil:
  - Tenemos una tabla $T[x, v]$, $0 \leq x < 2^b$, $1 \leq v \leq b$.
  - x es el bitmap de un bloque y v una posición en él.
  - $T[x, v]$ precalcula $close(v)$ si éste está dentro de x.
  - Sino, indica que v es lejano.
  - T requiere $\sqrt{n} \log n \log \log n = o(n)$ bits.
- Nuestro verdadero problema son los paréntesis lejanos.

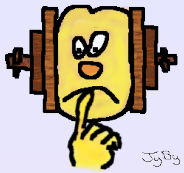
Navegando entre Paréntesis

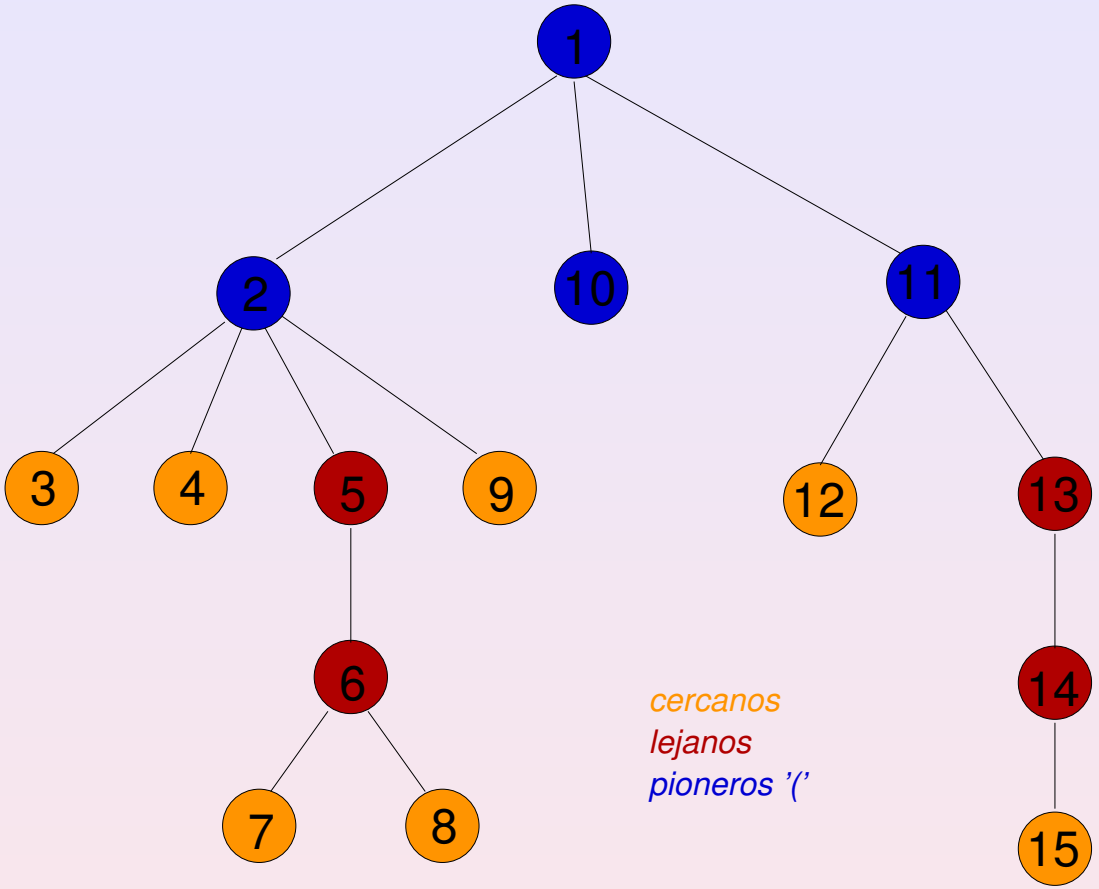


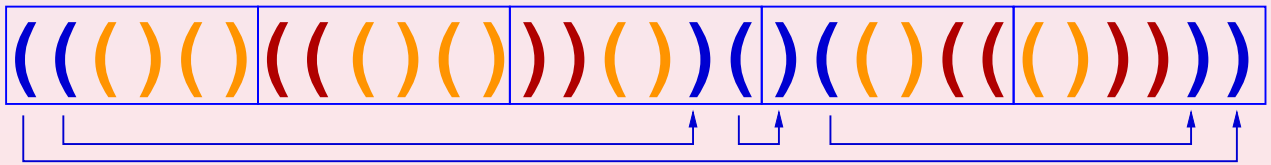
Paréntesis lejanos y pioneros

- Distinguiremos algunos paréntesis lejanos como pioneros.
- Un paréntesis lejano p (que abre) es pionero si el previo lejano que abre se cierra en un bloque distinto al del que cierra p.
- Un paréntesis lejano p (que cierra) es pionero si el siguiente lejano que cierra se abre en un bloque distinto al del que abre p.
- La pareja de un pionero también es pionero.

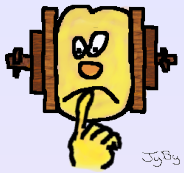
Navegando entre Paréntesis







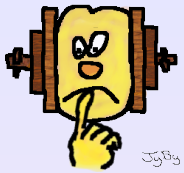
Navegando entre Paréntesis



Dos propiedades esenciales sobre pioneros

- Hay a lo sumo $4n/b$ paréntesis pioneros.
  - Pues cada pionero (por derecho propio) que abre debe cerrarse en un bloque distinto.
  - Con los que cierran, y con las parejas de los pioneros, podemos cuadruplicar la cantidad.
- El paréntesis que cierra a v está en el mismo bloque que el que cierra al último pionero que abre en $S[1, v]$.
  - Si ese pionero no es v mismo, se cierra en el mismo bloque que v, por definición (y se cierra después de v).
  - El último pionero en $S[1, v]$ debe abrir, pues si cerrara, el primero lejano que abre luego de él sería pionero.
  - Por lo tanto el último pionero representa un ancestro de v.

Navegando entre Paréntesis



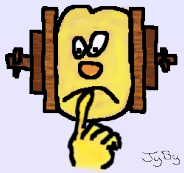
Estrategia general para hallar $v' = close(v)$

- Buscaremos el último pionero p que abre en $S[1, v]$.
- Resolveremos $p' = close(p)$ de alguna manera.
- Buscaremos v' con tablas en el bloque de p'.

Resolviendo *close* para pioneros

- La secuencia de paréntesis pioneros es una nueva secuencia balanceada.
- Supongamos por un momento que en esa secuencia puedo responder en tiempo constante $open'$, $close'$, y $enclose'$.
- Luego lo resolveremos usando recursión.

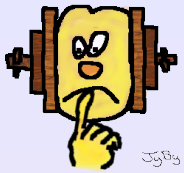
Navegando entre Paréntesis



Hallando el último pionero

- Marco en una secuencia P con un 1 los paréntesis que son pioneros.
- Representada en forma comprimida, P requiere $O(n \cdot \frac{\log b}{b}) = O(n \cdot \frac{\log \log n}{\log n}) = o(n)$ bits.
- Encontramos el paréntesis pionero que precede a v mediante $p = select(P, rank(P, v))$.
- Su posición en la secuencia reducida es $q = rank(P, p)$.
- Entonces $p' = select(P, close'(q))$.

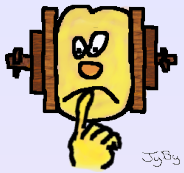
Navegando entre Paréntesis



Hallando *close* en el bloque donde cierra el pionero

- Tenemos ubicado a p', en el mismo bloque de v'.
- Sabemos que $exceso(v') - exceso(p') = exceso(v) - exceso(p)$.
- Con eso, podemos encontrar v' a partir de p' usando una tabla parecida a T.

Navegando entre Paréntesis



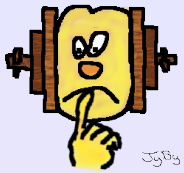
Recursión

- En la secuencia de pioneros (de largo $\leq 4n/b$) hacemos exactamente lo mismo, en forma recursiva (mantenemos el mismo b).
- Al siguiente nivel, ya hay $O(n/\log^2 n)$ paréntesis, y todas las respuestas se pueden almacenar directamente con $O(n/\log n)$ bits.
- Son sólo 2 niveles de recursión, tiempo total constante.

Resolviendo $open(v)$

- Es completamente simétrico a $close(v)$.

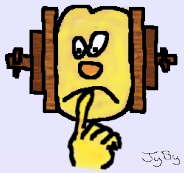
Navegando entre Paréntesis



Resolviendo $enclose(v)$

- Es el último $u < v$ tal que $exceso(u) = exceso(v) - 1$.
- Primero se ve con tablas si está en el mismo bloque de v o de $close(v)$.
- Calculamos el primer pionero c' en $S[v \ldots]$.
- Si c' cierra, entonces calculamos $p' = open(c')$, sino $p' = enclose(c')$ en la secuencia de pioneros.
- p' es el pionero más cercano que encierra a v.
- Entonces $u = enclose(v)$ debe estar en el mismo bloque que p', después de p' (o puede ser el mismo p').
- Se busca con tablas $u =$ último paréntesis que abre en el bloque de p' con el exceso correcto.

Navegando entre Paréntesis

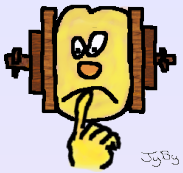


Espacio total

- La secuencia de paréntesis S usa $2n$ bits.
- La secuencia P que marca pioneros, comprimida, usa $O(n \cdot \frac{\log\log n}{\log n})$ bits.
- Las tablas (universales) usan $O(\sqrt{n}\,\mathrm{polylog}(n))$ bits.
- El segundo nivel requiere $O(n/\log n)$ bits.
- El tercer nivel requiere $O(n/\log n)$ bits.
- Total:

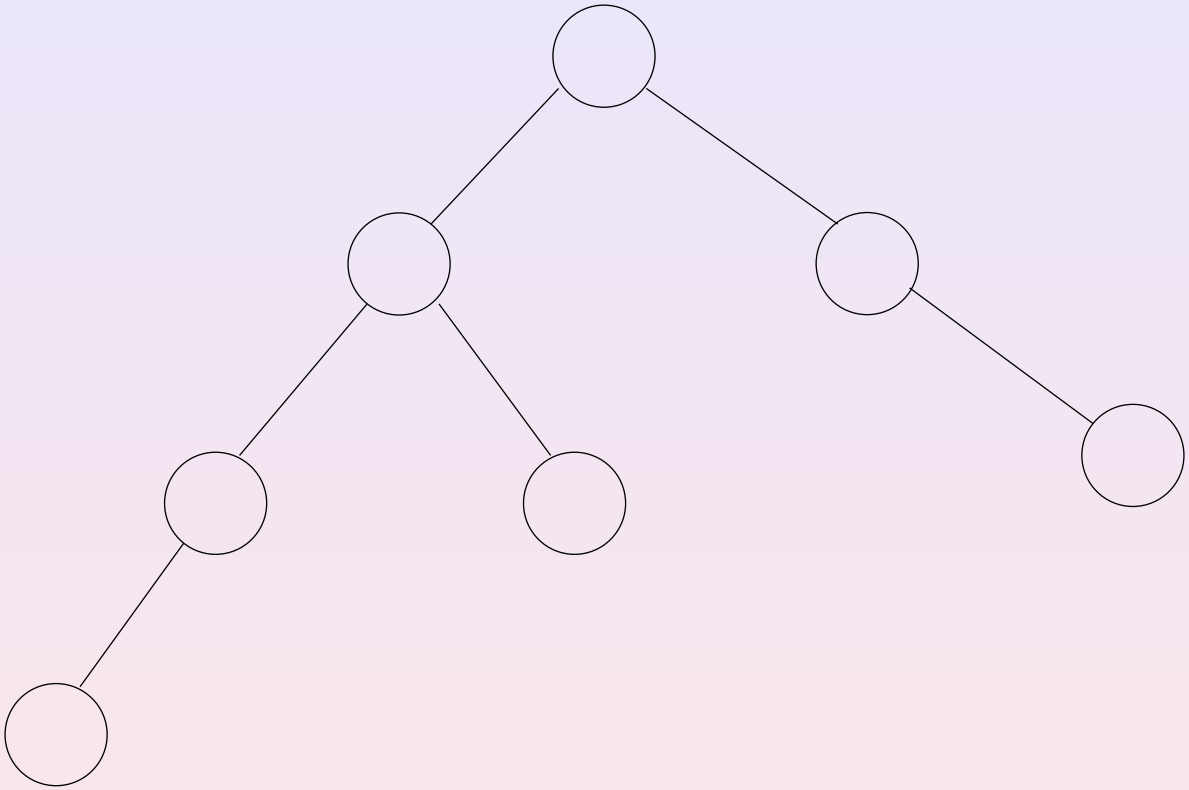
$$2n \;+\; O\left(n \cdot \frac{\log\log n}{\log n}\right) \;=\; 2n + o(n) \quad \text{bits.}$$

Arboles Binarios

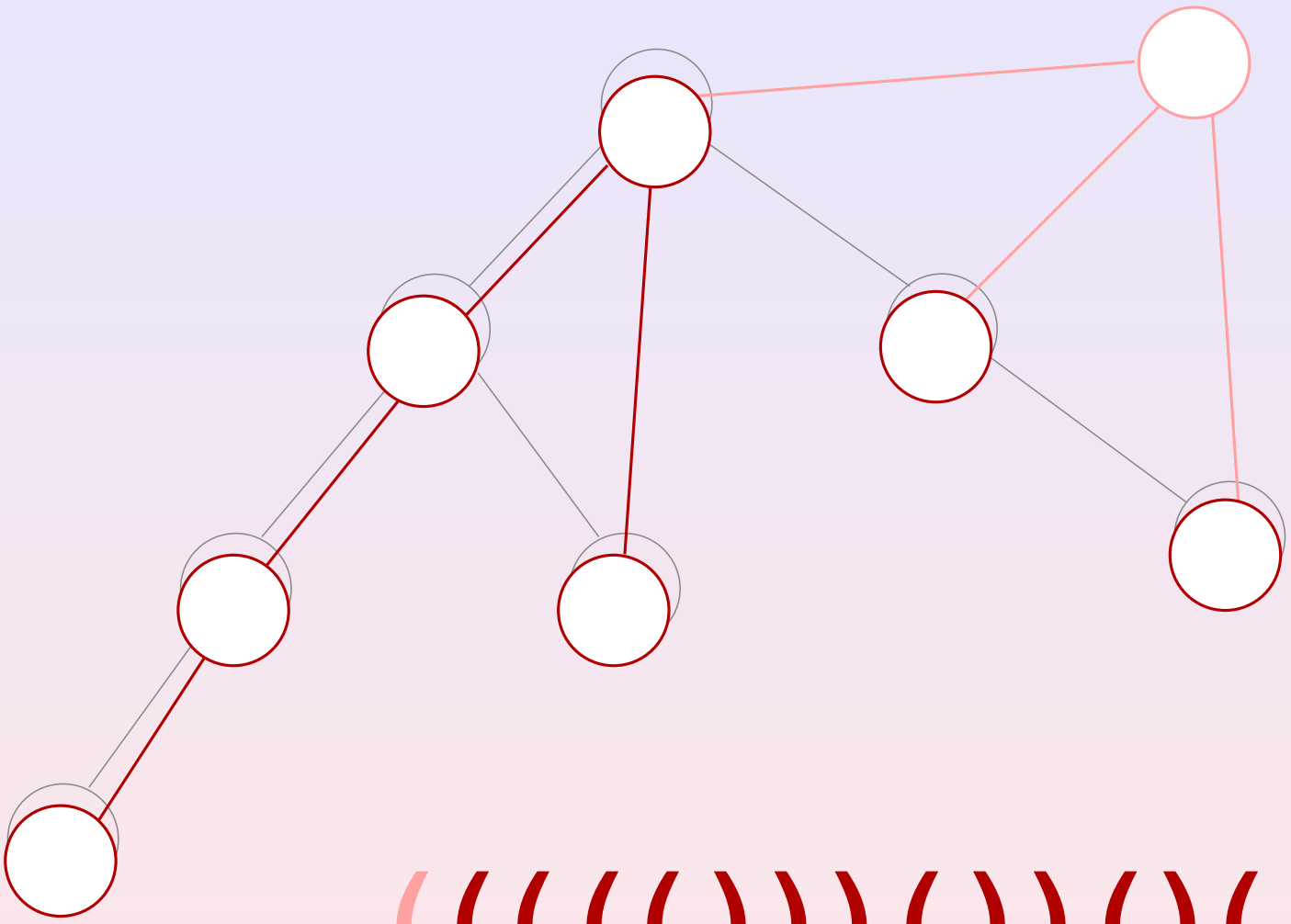


- Se pueden representar como árboles generales, pero requieren $4n$ bits (marcar hojas explícitamente).
- Pero se puede representar con $2n$ bits.
- La idea es usar un mapeo conocido entre árboles binarios y generales.
- El árbol general tiene una raíz ficticia (que no pondremos en los paréntesis)...
- ... y todo el camino más derecho del árbol binario son los hijos de la raíz del árbol general.
- Luego se transforma recursivamente.

Arboles Binarios

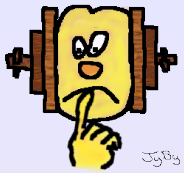


Arboles Binarios



((((()))())()())

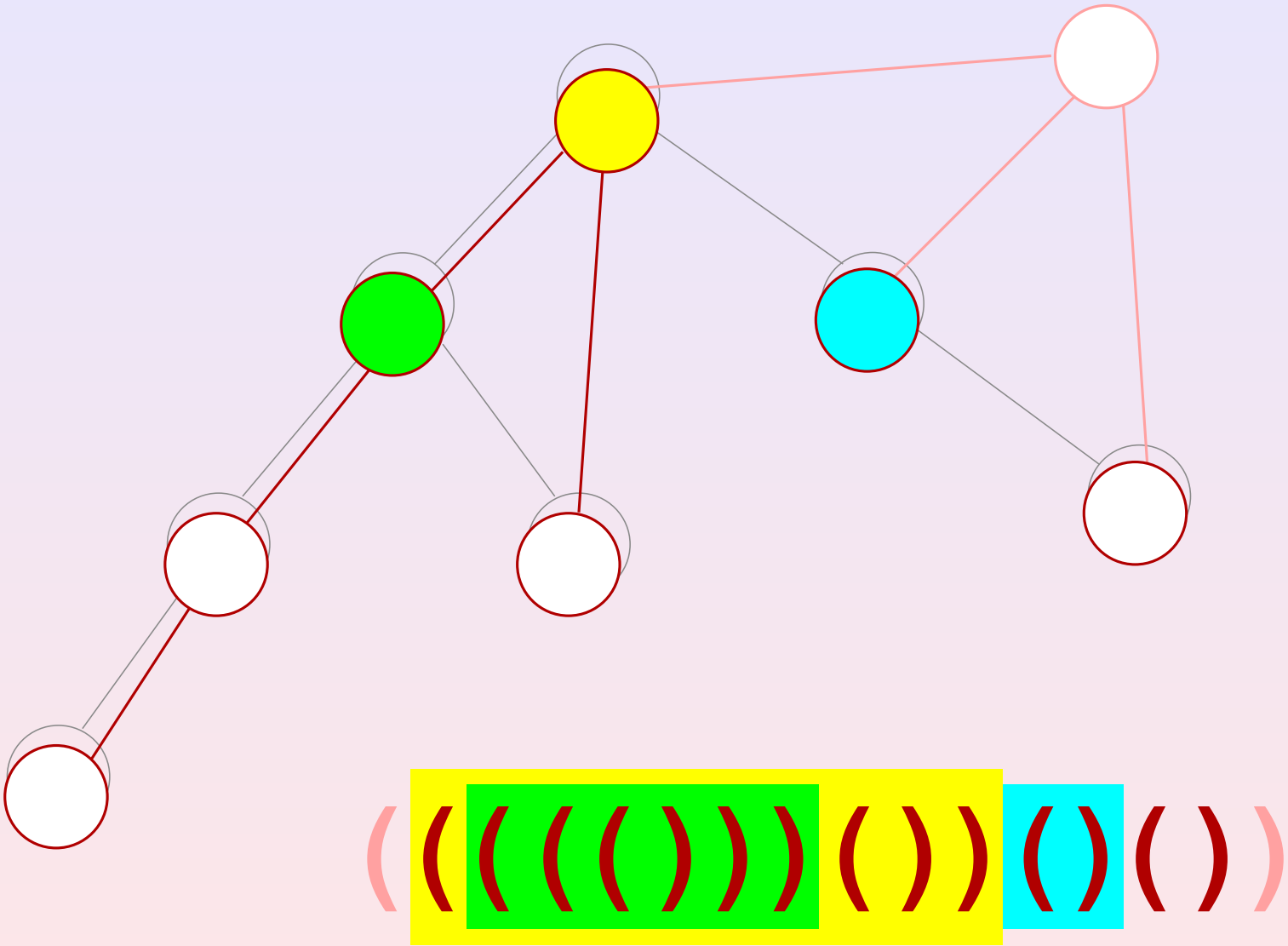
Arboles Binarios



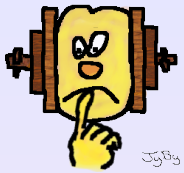
Navegación

- La raíz del árbol binario es el primer hijo del árbol general.
- Hijo izquierdo de v: primer hijo de v ($= v + 1$).
- Hijo derecho de v: siguiente hermano de v.
- Padre de hijo izquierdo v: padre de v ($= v - 1$).
- Padre de hijo derecho v: hermano previo de v.
- v es hijo izquierdo sii $S[v-1] = '('$.
- v es hoja sii $S[v+1] = ')'$ y $S[v+2] = ')'$.

Arboles Binarios

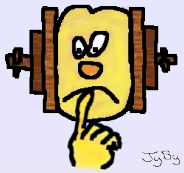


LCA: Ancestro Común más Bajo



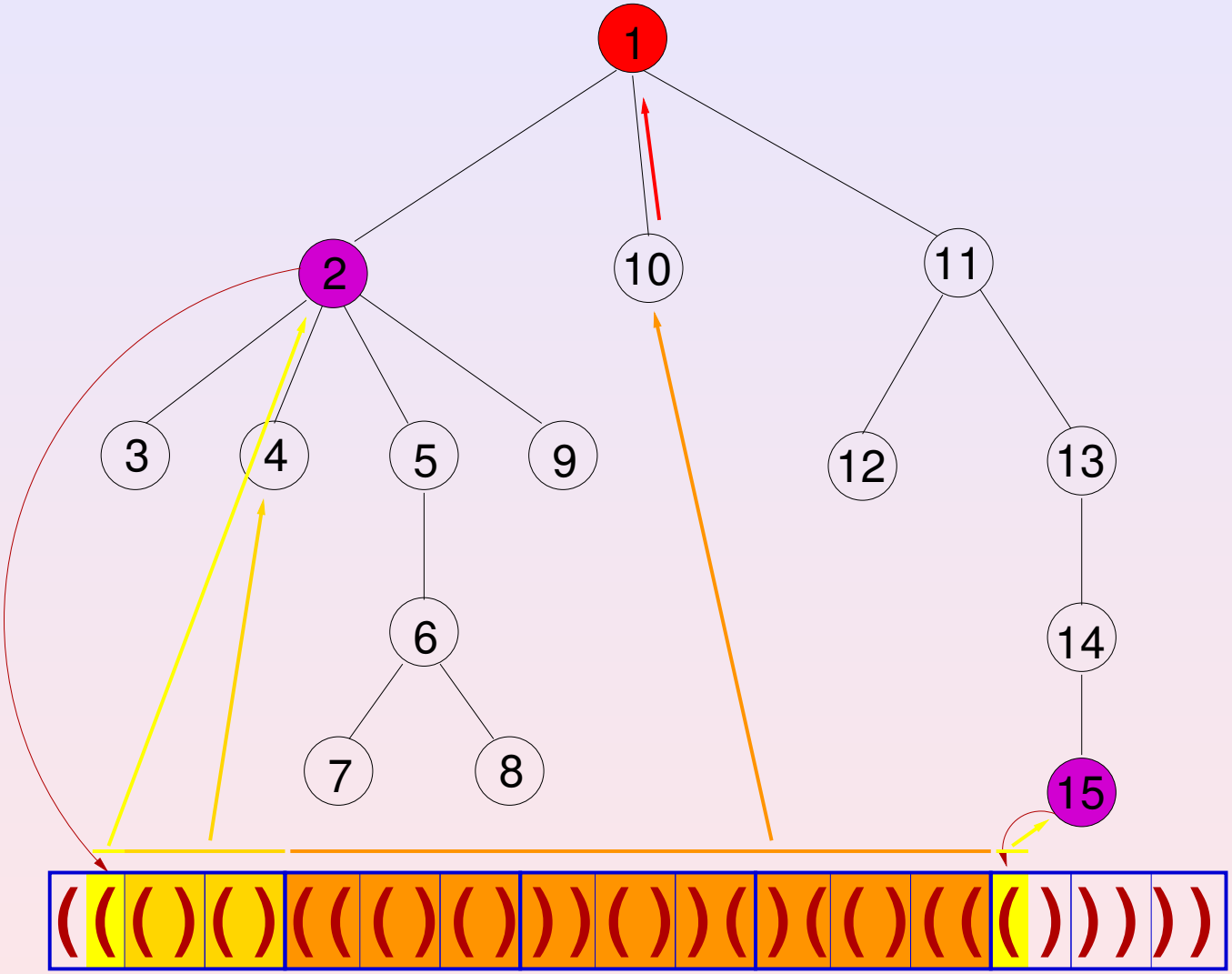
- $LCA(v, v')$ es el nodo más profundo que es ancestro de v y de v'.
- Tiene muchas aplicaciones importantes (veremos una).
- Se puede calcular en tiempo constante.
- Si v es ancestro de v', entonces $LCA(v, v') = v$, y simétricamente con v'.
- Dediquémonos al caso en que eso no ocurre:
  - Propiedad: v y v' descienden de hijos distintos de $LCA(v, v')$.
  - Propiedad: $LCA(v, v')$ es el padre del nodo con menor exceso en $[x, y] = [\min(v, v'), \max(v, v')]$.
    - pues el $'('$ del hijo del LCA que es ancestro de $\max(v, v')$ debe estar en el rango, mientras que el del LCA no.

LCA: Ancestro Común más Bajo

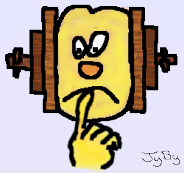


- Dividiremos la secuencia de paréntesis en bloques y superbloques.
- Los bloques miden $b = \frac{\log n}{2}$ bits y los superbloques $s = 2b \cdot \log^2 n = \log^3 n$ bits.
- La idea general es encontrar separadamente los menores excesos en los superbloques, bloques, y posiciones individuales involucrados en el rango $[x, y]$.

LCA: Ancestro Común más Bajo



LCA: Ancestro Común más Bajo

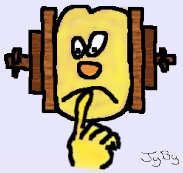


- Primero calculamos los límites de bloques y superbloques involucrados:

$$s_x = \lfloor x/s \rfloor, \quad s_y = \lfloor y/s \rfloor, \quad b_x = \lfloor x/b \rfloor, \quad b_y = \lfloor y/b \rfloor.$$

- Luego tomamos el mínimo exceso de 5 rangos:
  - Los superbloques contenidos en $[x, y]$, es decir $[1 + s_x \cdot s, s_y \cdot s]$.
  - Los bloques contenidos en $[x, y]$ que preceden a s_x, es decir $[1 + b_x \cdot b, s_x \cdot s]$.
  - Los bloques contenidos en $[x, y]$ que siguen a s_y, es decir $[1 + s_y \cdot s, b_y \cdot b]$.
  - Las posiciones contenidas en $[x, y]$ que preceden a b_x, es decir $[x, b_x \cdot b]$.
  - Las posiciones contenidas en $[x, y]$ que siguen a b_y, es decir $[1 + b_y \cdot b, y]$.

LCA: Ancestro Común más Bajo



Resolviendo los Superbloques

- Si almacenáramos el mínimo exceso dentro de cada par de superbloques, ocuparíamos demasiado espacio.
- Utilizaremos la propiedad siguiente: Si $a < b < c < d$, entonces $\min[a,d] = \min(\min[a,c], \min[b,d])$.
- Almacenaremos una tabla

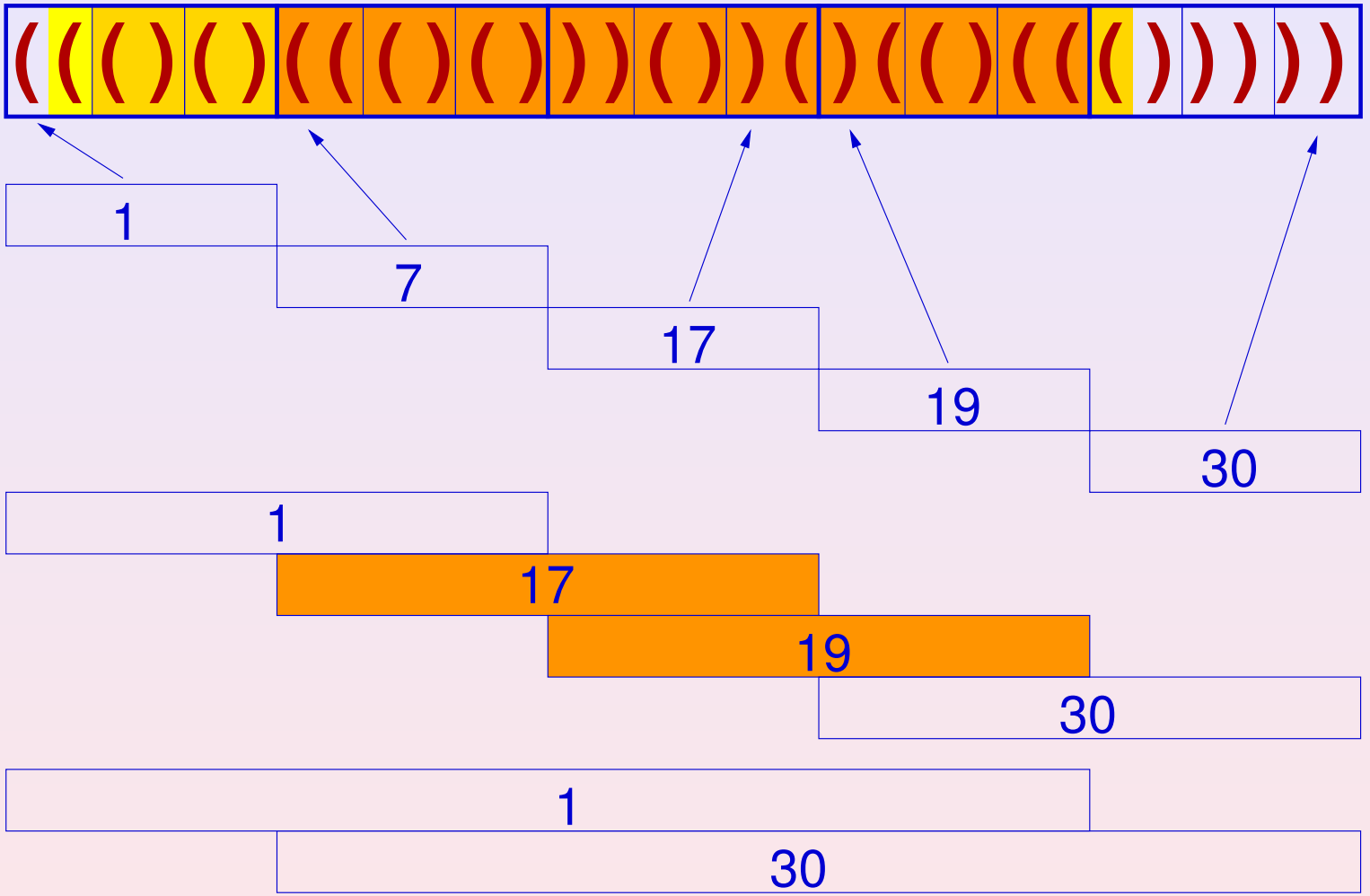
$$M[i,j] \quad = \quad \text{posición del mínimo exceso en los superbloques } [i, i+2^j-1]$$

- Entonces

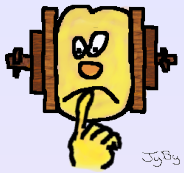
$$\min[1+s_x \cdot s, s_y \cdot s] \quad = \quad \min(M[s_x, j], M[s_y - 2^j + 1, j])$$

donde $j = \lfloor \log(s_y - s_x + 1) \rfloor$.

LCA: Ancestro Común más Bajo



LCA: Ancestro Común más Bajo

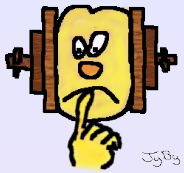


Espacio para los Superbloques

- Notar que almacenamos las soluciones sólo para rangos cuyo largo es potencia de 2.
- Todo rango de largo arbitrario se resuelve superponiendo dos rangos de largo potencia de dos que lo cubran.
- Si hay t superbloques, M tiene $t \log t$ celdas.
- El total de bits que ocupa es

$$t \log t \log n \leq \frac{n}{\log^3 n} \cdot \log^2 n = O\left(\frac{n}{\log n}\right)$$

LCA: Ancestro Común más Bajo

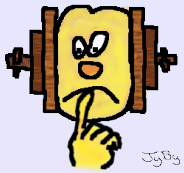


Resolviendo los Bloques

- Usamos el mismo esquema a nivel de bloques, internamente a cada superbloque.
- Como las posiciones son internas a los superbloques, caben en $3 \log\log n$ bits.
- El total de bits que se ocupa es

$$\frac{n}{s} \cdot \frac{s}{b} \cdot \log \frac{s}{b} \cdot 3 \log\log n \;=\; 12\, n \cdot \frac{(\log\log n)^2}{\log n}\,(1 + o(1))$$

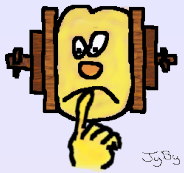
LCA: Ancestro Común más Bajo



Resolviendo dentro de un Bloque

- Usamos una tabla tipo T que almacene la posición del mínimo exceso dentro de cada bloque posible.
- T necesita $O(\sqrt{n} \log\log n)$ bits.

LCA: Ancestro Común más Bajo



Finalmente...

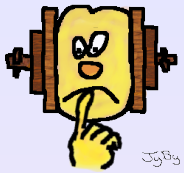
- Una vez que tenemos los 5 candidatos $p_1 \ldots p_5$ a posición del mínimo exceso...
- ... calculamos

$$m = \text{argmin}_{p_i}\ exceso(p_i)$$

- ... y respondemos

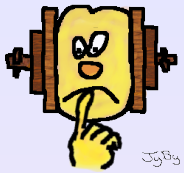
$$LCA(v, v') = enclose(m)$$

RMQ: Mínimos en Rangos



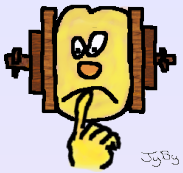
- Un problema en principio sin relación con árboles.
- Tengo un arreglo de números $A[1, n]$, que puedo preprocesar.
- Luego, dados rangos $[i, j]$, necesito saber $\min A[i \ldots j]$.
- Se puede resolver en tiempo constante usando solamente $4n + o(n)$ bits extra!

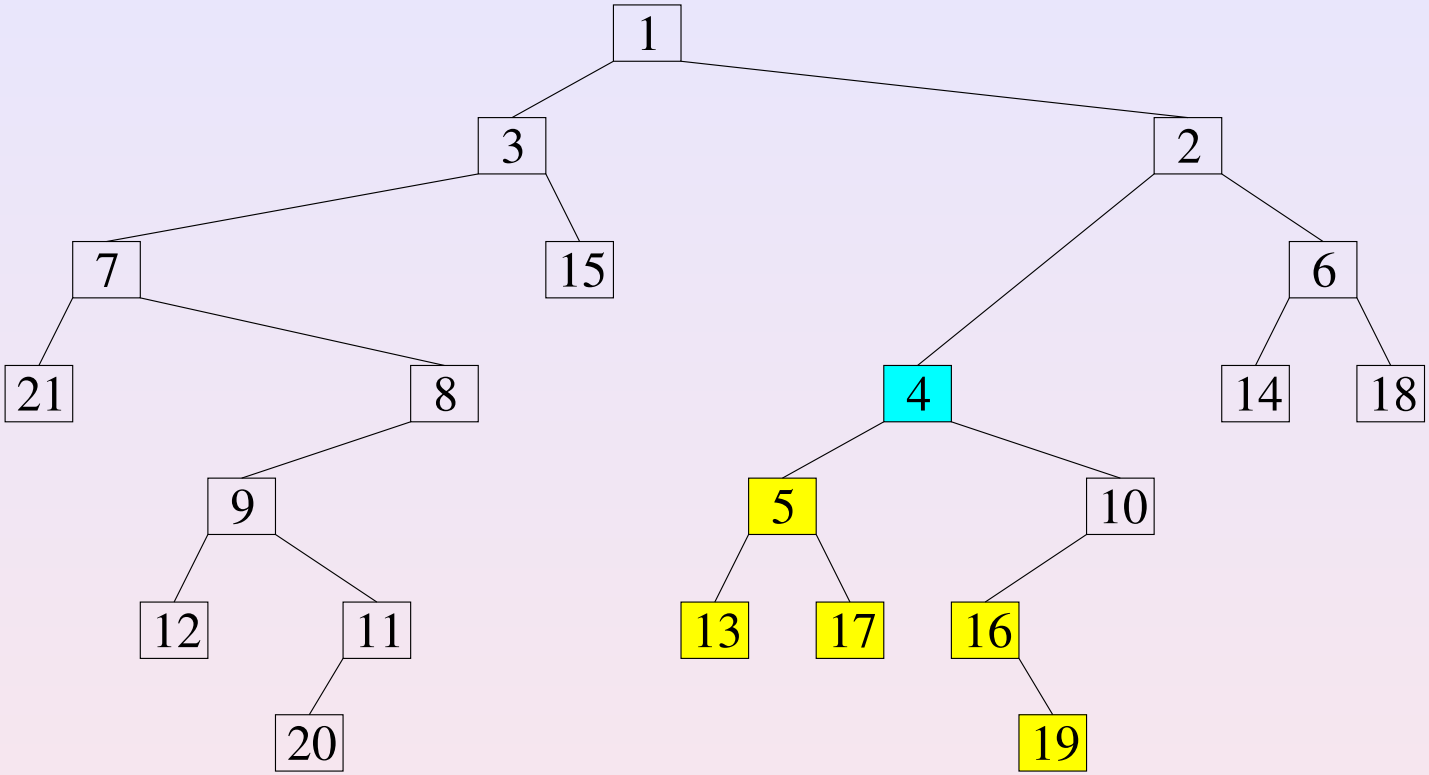
RMQ: Mínimos en Rangos



- El árbol cartesiano de $A[1,n]$ se define así:
  - Sea m la posición del $\min A[i \ldots j]$.
  - Entonces el árbol tiene una raíz con hijos T_1 y T_2.
  - T_1 es el árbol cartesiano de $A[1, m-1]$.
  - T_2 es el árbol cartesiano de $A[m+1, n]$.
- El árbol tiene n nodos.
- Sea $H[1,n]$ la secuencia de las profundidades de los nodos.
- La posición del mínimo en $A[i \ldots j]$ es la misma posición del mínimo en $H[i \ldots j]$.
  - Pues corresponde al $LCA(i,j)$ en el árbol cartesiano.

RMQ: Mínimos en Rangos





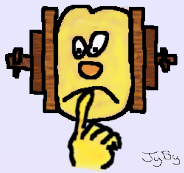
H

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
3	2	5	4	6	5	3	1	2	0	4	3	4	2	4	5	3	1	3	2	3

A

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18

RMQ: Mínimos en Rangos

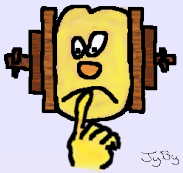


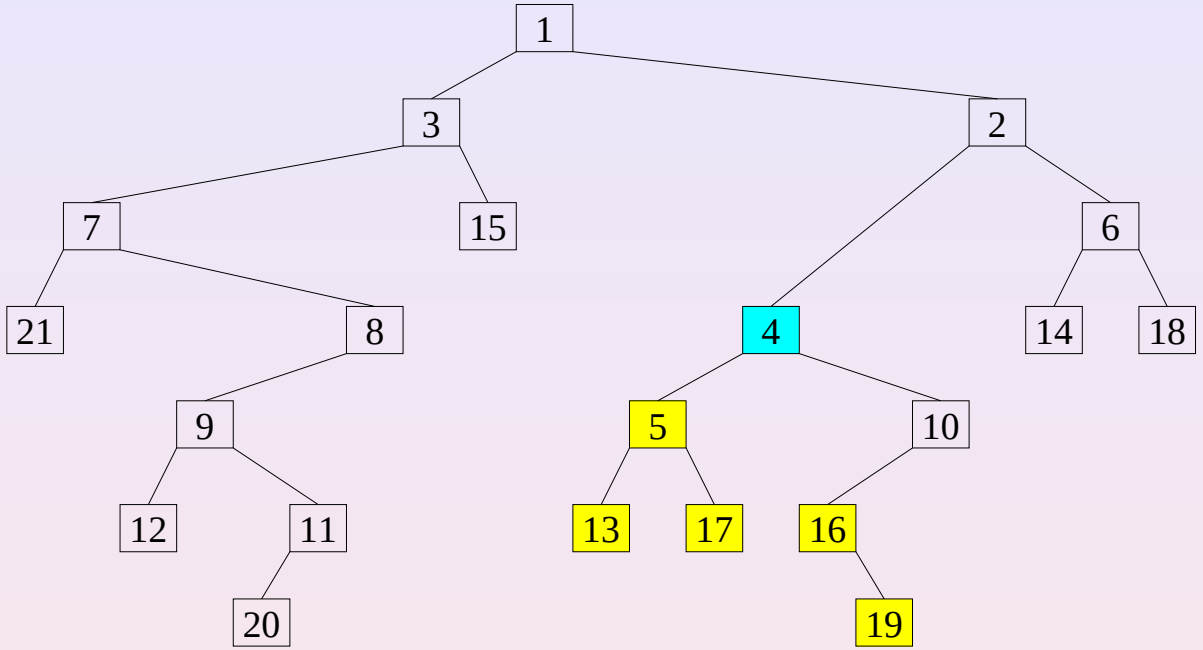
- Supongamos $H[0] = H[n+1] = 0$ y generemos una secuencia a partir de H:
  - Si $H[i] > H[i-1]$, agregamos $H[i] - H[i-1]$ símbolos '(' .
  - Si $H[i] < H[i-1]$, agregamos $H[i-1] - H[i]$ símbolos ')' .
- El resultado es $S[1, 2n]$, la representación en paréntesis del árbol cartesiano.
- El mínimo en H corresponde al $\min[x, y]$ en S.
- Marcamos en otro bitmap $P[1, 2n]$ las posiciones donde comienza la codificación de cada valor de H en S.
- Entonces,

$$RMQ(i,j) \;=\; A[rank(P, \min_{S}[select(P,i), select(P,j+1)-1])]$$

se resuelve en tiempo constante.

RMQ: Mínimos en Rangos



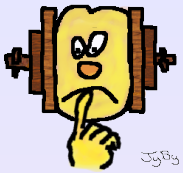


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
H	3	2	5	4	6	5	3	1	2	0	4	3	4	2	4	5	3	1	3	2	3

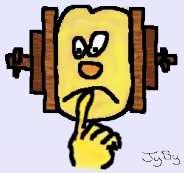
S ((( ) ((( ) (( ))))) ( )) (((( ) ( )) ((( )))) (( ) ( )))

P 1 0 0 1 1 0 0 1 1 0 1 1 0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 0 0

Parte II: Otras Estructuras

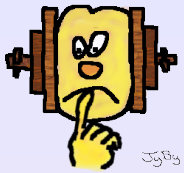


Otra Representación usando Paréntesis



- La representación usada tiene muchas bondades.
- Pero es ineficiente para una operación importante: $hijo(v, i)$ es el i-ésimo hijo de v.
- Lo podemos hacer en tiempo $O(i)$, pero eso puede ser muy lento en ciertos árboles.
- Veremos una representación alternativa que resuelve esto en tiempo constante.
- También puede calcular la cantidad de hijos (aridad) de v en tiempo constante.
- Y también permitirá comprimir el árbol!
- A cambio, no permite conocer la profundidad de v.

Otra Representación usando Paréntesis

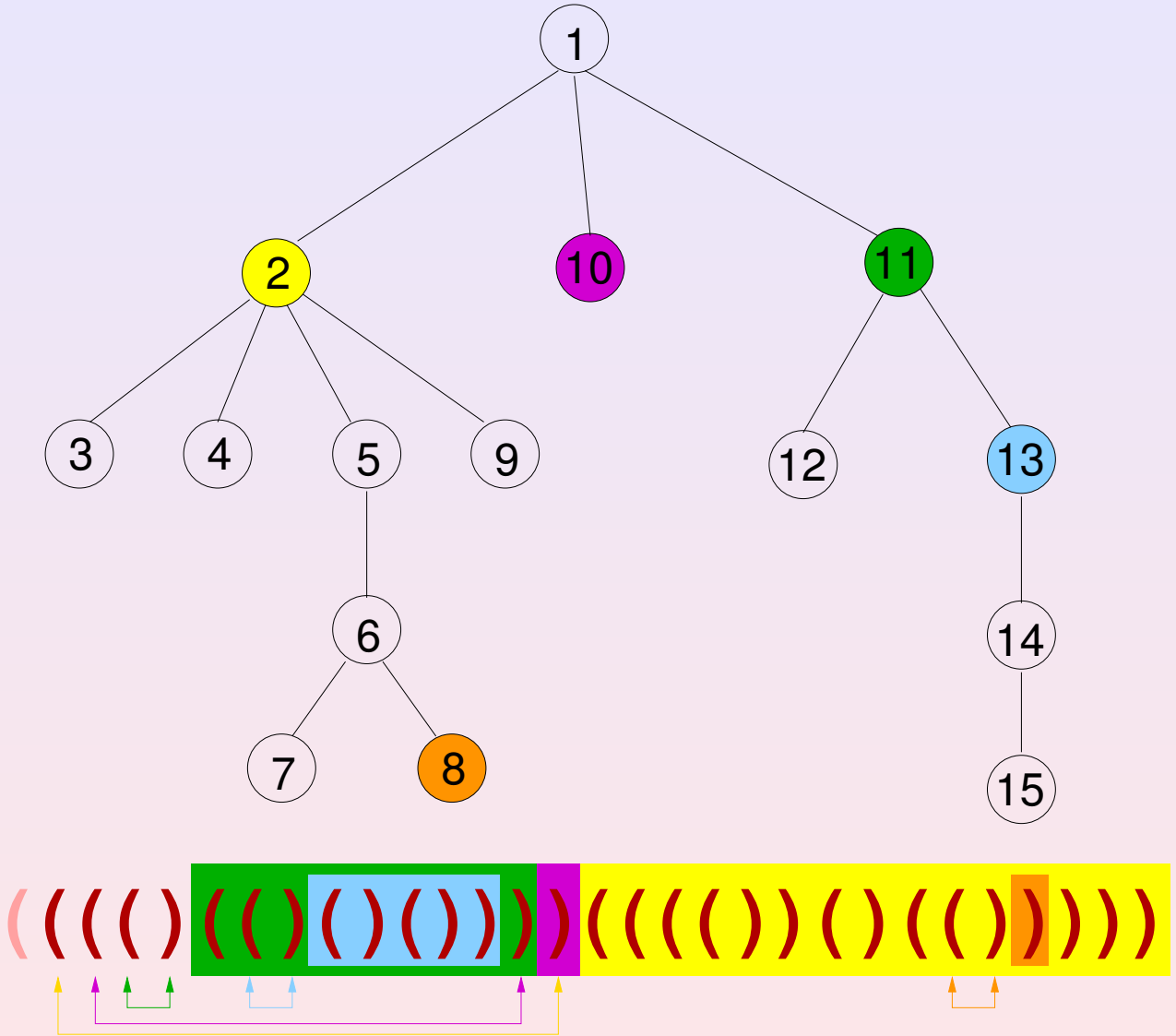


- Usaremos paréntesis balanceados, pero con otro significado.
- Un nodo v con hijos $v_1, v_2, \ldots, v_k$ se representará como

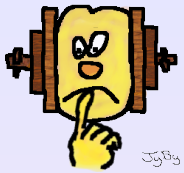
$$R(v) = \underbrace{((\ldots(}_{k})\ )\ R(v_k)\ldots R(v_2)\ R(v_1)$$

- El nodo v corresponde a su primer paréntesis.
- Una hoja se representa como `')'`.
- Propiedad importante: Todo árbol termina con `')'`, y su exceso interno es -1.
- Agregamos un `'('` a S para que quede balanceada.
- En total S queda con $2n$ paréntesis balanceados!

Otra Representación usando Paréntesis

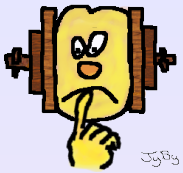


Otra Representación usando Paréntesis



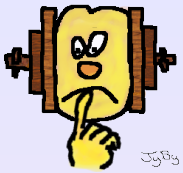
- Utilizaremos las mismas operaciones *open*, *close* y *enclose* para navegar en esta nueva representación.
- La aridad de un nodo es $select_{')'}(S, rank_{')'}(S, v-1)+1)-v$.
- El i-ésimo hijo de v es $close(v+i-1)+1$
- El padre de v es $1+select_{')'}(S, rank_{')'}(S, open(v-1)))$.
- Puedo saber qué hijo soy de mi padre: $open(v-1)-padre(v)+1$

Otra Representación usando Paréntesis



- Con las operaciones anteriores tengo automáticamente los hermanos.
- Los ancestros siguen conteniendo a sus descendientes.
- Con $rank_{'(')'}(S, v)$ tengo numeración tipo preorden.
- Por lo tanto, tengo también tamaño de subárbol.
- Contando $'()'$ tengo cantidad de nodos internos.
- Con ambos, tengo cantidad de hojas.
- También puede resolverse *LCA* prácticamente del mismo modo.

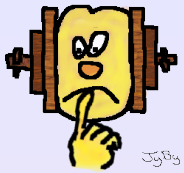
Otra Representación usando Paréntesis



Comprimiendo

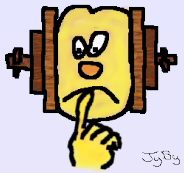
- Realmente esta representación es la secuencia de las aridades en preorden, escritas en unario.
- Se podrían representar como números y comprimir la secuencia a orden cero.
- Eso comprime si existe regularidad en las aridades del árbol.
- No veremos el detalle, es demasiado técnico.

Textos: Otra Visión de Secuencias



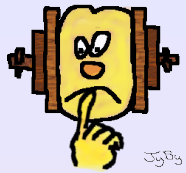
- Dado un alfabeto Σ de tamaño σ...
- ... un texto $T[1,n]$ es una secuencia sobre Σ.
- *¿Eso no es lo mismo que una secuencia?*
- Sí, pero nos interesan otras operaciones.
- Dado un patrón $P[1,m]$:
  - Contar la cantidad de ocurrencias de P en T (occ).
  - Ubicar esas occ ocurrencias en T.

Textos: Otra Visión de Secuencias



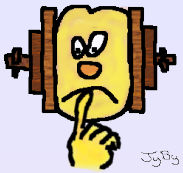
- Cuando el texto no es muy largo, o cambia muy frecuentemente, lo mejor es la búsqueda secuencial.
- Pero en el mejor caso esto cuesta $O(\frac{n \log_\sigma m}{m})$.
- Cuando el texto es de lenguaje natural y sólo queremos buscar palabras y frases, lo mejor es un índice invertido.
- Esto es un vocabulario de las palabras distintas, y una lista de ocurrencias de cada una.
- No es difícil conseguir poco espacio con bitmaps comprimidos para las listas.
- Otras variantes se especializan en cálculo de relevancia.

Textos: Otra Visión de Secuencias



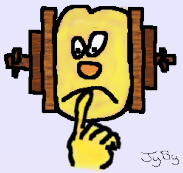
- Los índices invertidos funcionan muy bien.
- Son usados por buscadores como Google y Yahoo!.
- Uno de sus grandes desafíos es unir e intersectar listas.
- Y en eso son útiles los resultados que vimos para relaciones binarias.
- Pero no todo se puede resolver con índices invertidos.
- La expresión “lenguaje natural” implica muchas suposiciones:
  - El texto se puede cortar automáticamente en “palabras”.
  - El usuario sólo querrá buscar palabras o secuencias de palabras.
  - El conjunto de palabras distintas (vocabulario) crece sublinealmente con n (ley de Heaps).
  - Las frecuencias de las palabras se distribuyen muy desigualmente (leyes de Zipf o Mandelbrot).

Textos: Otra Visión de Secuencias



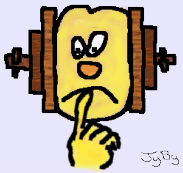
- “Lenguaje natural” excluye lenguajes muy naturales como el chino, japonés, coreano...
- ... e incluso occidentales (aglutinantes) como alemán y finlandés!
- Hay secuencias donde no existe el concepto de palabra en absoluto:
  - Secuencias biológicas (ADN o proteínas).
  - Secuencias musicales (valores de pitch en archivos MIDI).
  - Señales discretizadas.
- Hay otras donde podrían definirse palabras, pero no interesa buscar sólo palabras, o no siguen las leyes:
  - Código fuente en lenguajes de programación.
  - Secuencias numéricas o de códigos de algún tipo.

Arboles y Arreglos de Sufijos



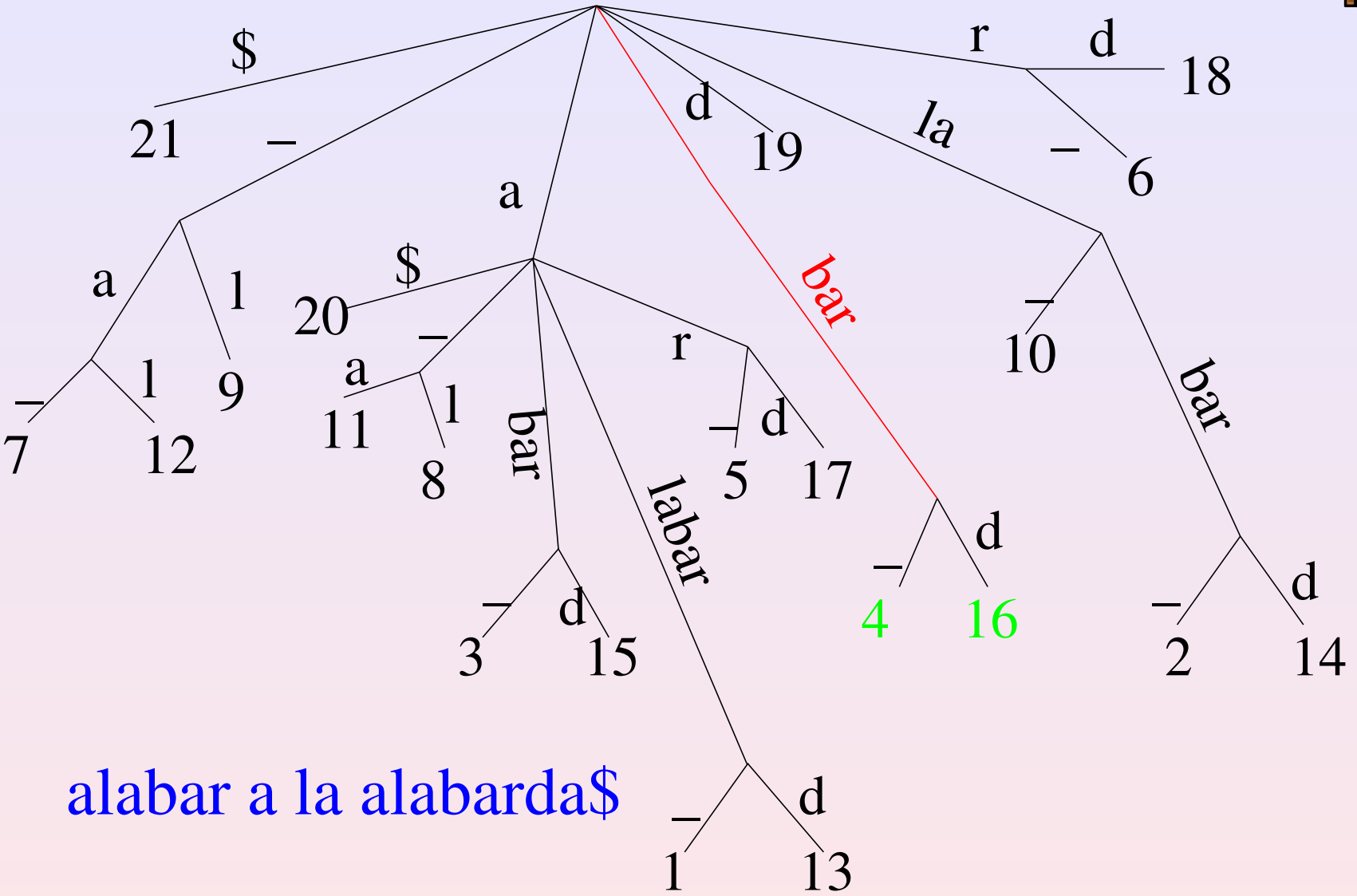
- Veremos un tipo de índice mucho más potente que los índices invertidos.
- No hacen ninguna suposición sobre el texto.
- Permiten buscar cualquier substring del texto.
- Modelo general: considerar los n sufijos $T[i, n]$ de T.
- Todo substring de T es el prefijo de un sufijo de T.
- Estas estructuras indexan el conjunto de sufijos de T y permiten encontrar todos los que comparten un cierto prefijo.

Arboles y Arreglos de Sufijos



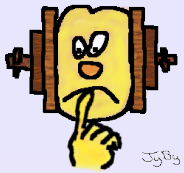
Arbol de Sufijos

- Es un árbol digital que contiene todos los sufijos de T.
- Consideraremos que T termina con un símbolo especial \$.
- Cada hoja del árbol corresponde a un sufijo distinto de T.
- Y cada nodo interno a un substring repetido de T.
- Se comprimen caminos unarios para garantizar $O(n \log n)$ bits.
- Se puede construir en tiempo $O(n)$ si $\sigma = O(n)$ es discreto.



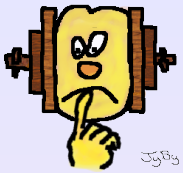
alabar a la alabarda$

Arboles y Arreglos de Sufijos

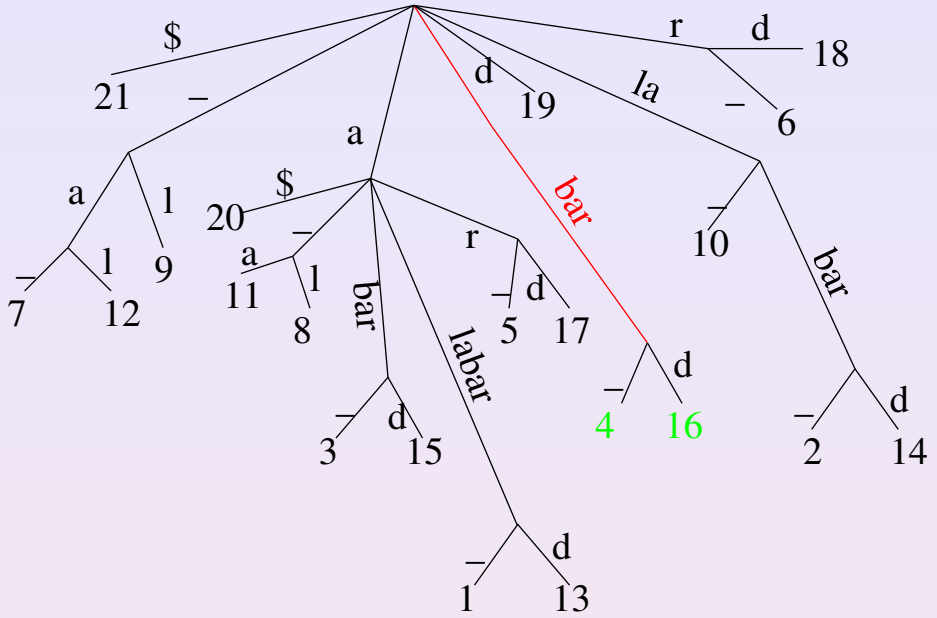


- Para buscar P, se intenta bajar en el árbol usando sus caracteres.
  - Si agoto P en una arista, el subárbol que desciende agrupa todas las ocurrencias de P.
  - Si no puedo bajar por un cierto $P[i]$, entonces P no aparece en T.
  - Si llego a una hoja, hay a lo sumo una ocurrencia, que debo terminar de verificar en T.
- Cuenta en tiempo $O(m)$, ubica en tiempo $O(m + occ)$.
- Puede resolver muchos otros problemas más complejos.

Arboles y Arreglos de Sufijos



- El árbol de sufijos es una de las estructuras de datos más elegantes y útiles, pero ocupa muchísimo espacio.
- Puede requerir de $10n$ a $20n$ bytes, por ejemplo 30-60 GB para el genoma humano.
- Una solución bastante exitosa para reducir su espacio es el arreglo de sufijos.
- Este es el arreglo de las hojas del árbol de sufijos.
- Con las técnicas para representar árboles...
- ... podemos representar el árbol de sufijos como
  - El arreglo de sufijos.
  - $2n$ paréntesis para la estructura de árbol.
  - El espacio extra para la navegación que se desee.
- Por lo tanto, tiene interés comprimir el arreglo de sufijos.



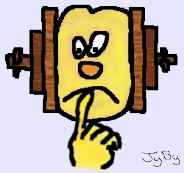
(()((()())())(()(()())(()())(()())(()())(()()))(()())()(()(()()))(()()))

alabar a la alabarda$

bar

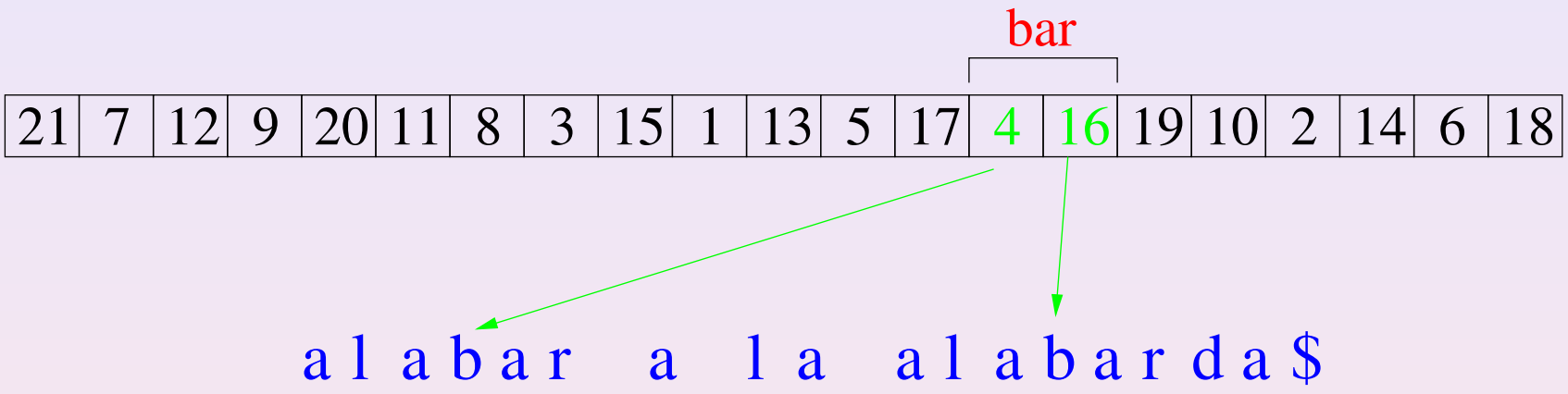
21	7	12	9	20	11	8	3	15	1	13	5	17	4	16	19	10	2	14	6	18

Arboles y Arreglos de Sufijos

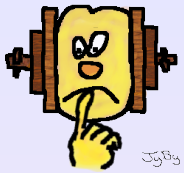


Arreglo de Sufijos

- Un arreglo de sufijos también puede funcionar sin el árbol.
  - No puede hacer todo lo que un árbol de sufijos.
  - Pero sí puede buscar patrones, en tiempo $O(m \log n + occ)$,
  - ... y varias otras cosas.
- Todo subárbol del arbol de sufijos corresponde a un intervalo del arreglo de sufijos.
- Se puede hacer una búsqueda binaria de los sufijos que empiezan con P.
- El rango resultante contiene todas las respuestas...
- ... y corresponde al subárbol que habríamos encontrado en el árbol de sufijos.

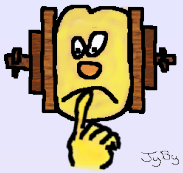

bar
21 7 12 9 20 11 8 3 15 1 13 5 17 4 16 19 10 2 14 6 18
a l a b a r a l a a l a b a r d a $

Arboles y Arreglos de Sufijos

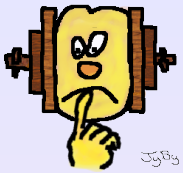


- Aún solo, el arreglo de sufijos presenta problemas de espacio.
- Ocupa unos $4n$ bytes (12GB para el genoma humano).
- Nuevamente, es de interés comprimirlo.
- Pero... ¿se puede comprimir una permutación?.

Parte II: Otras Estructuras

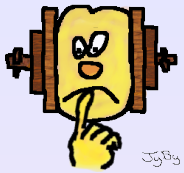


El Arreglo de Sufijos Comprimido



- Un arreglo de sufijos no es cualquier permutación.
- Hay $n!$ permutaciones posibles pero sólo σ^n arreglos de sufijos posibles.
- *¿Cómo se refleja la compresibilidad de un texto en su arreglo de sufijos?*
- *¿De qué tipo de compresibilidad de textos estamos hablando?*
- Concentrémosnos en una que es importante en la práctica: entropía de orden k.
  - Orden cero captura tamaño de alfabeto y sesgo en la frecuencia de los símbolos.
  - Ordenes mayores capturan la predictibilidad del texto que sigue en base a lo que lo precede.

El Arreglo de Sufijos Comprimido



La función Ψ

- Llamemos $A[1,n]$ al arreglo de sufijos de $T[1,n]$.
- Podemos pensar en A como una permutación y A^{-1} como su inversa.
- Definiremos una función $\Psi(1..n)$ como

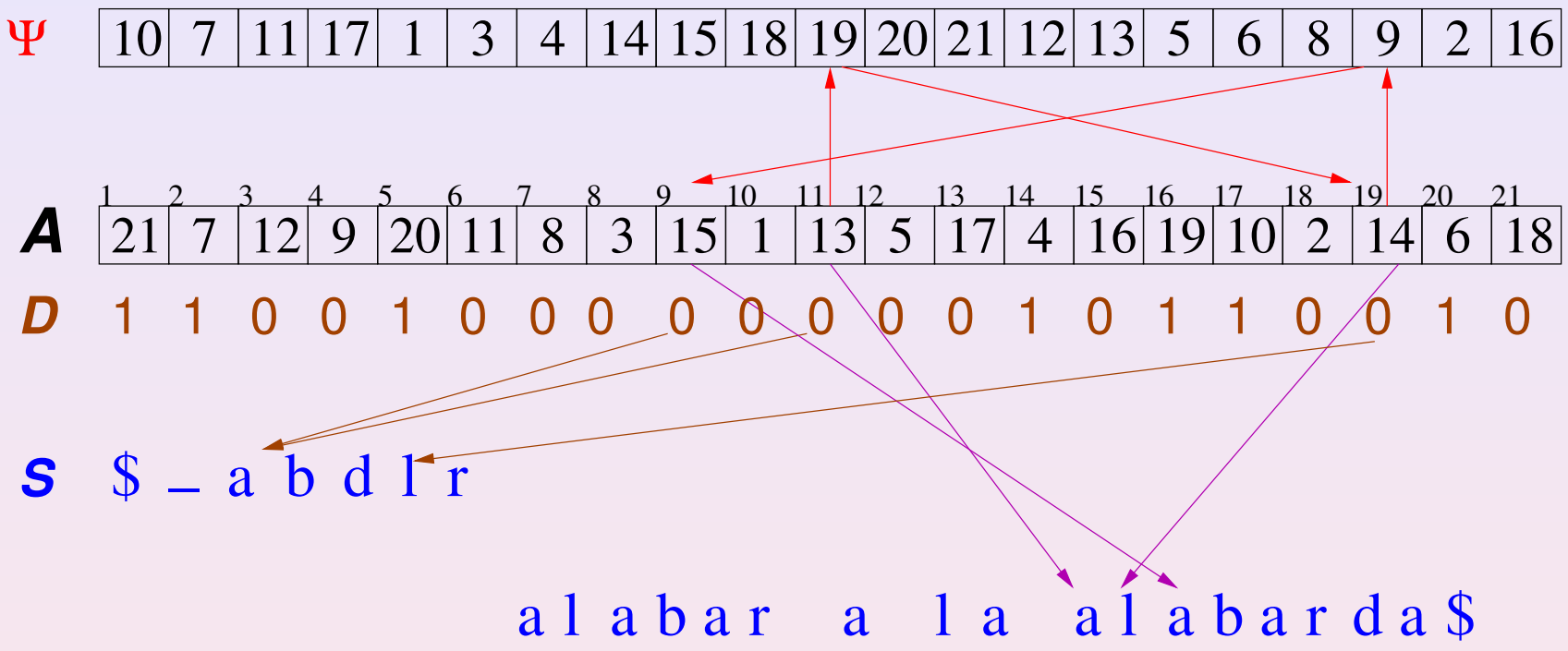
$$\Psi(i) \quad = \quad A^{-1}[A[i]+1]$$

(con el caso especial $\Psi(i) = A^{-1}[1]$ si $A[i] = n$).

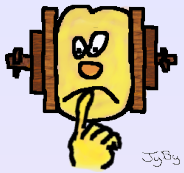
- Dicho de otro modo,

$$A[\Psi(i)] \quad = \quad A[i]+1.$$

- Dada una celda $A[i]$ que apunta al sufijo $T[A[i]\ldots]$, Ψ dice dónde está en A el puntero al siguiente sufijo.


Ψ
10 7 11 17 1 3 4 14 15 18 19 20 21 12 13 5 6 8 9 2 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
A
21 7 12 9 20 11 8 3 15 1 13 5 17 4 16 19 10 2 14 6 18
D
1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0
S
$ _ a b d l r
a l a b a r a l a a l a b a r d a $

El Arreglo de Sufijos Comprimido



- Definamos la función $C(1..\sigma)$ como

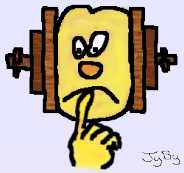
$$C(c) \quad = \quad \text{cantidad de ocurrencias de símbolos } < c \text{ en } T$$

- Implementemos C con
  - Un bitmap $D[1, n]$, con $D[1] = 1$ y $D[C(c) + 1] = 1$ para todo $c \in \Sigma$.
  - Una lista $S[1, \sigma]$ de los distintos caracteres que aparecen en T, en orden alfabético.
- D y S se pueden representar con $O(\sigma \log n)$ bits.
- La operación que realmente me interesará es

$$c \quad = \quad S[rank(D, i)]$$

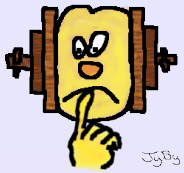
es decir, con qué caracter comienza $T[A[i]]$.

El Arreglo de Sufijos Comprimido



- Veamos cómo extraer $T[A[i]\ldots]$ sin A ni T...
- ... sino con D, S y Ψ.
  - $T[A[i]]$ es $S[rank(D, i)]$.
  - $T[A[i]+1] = T[A[\Psi(i)]$ es $S[rank(D, \Psi(i))]$.
  - $T[A[i]+2] = T[A[\Psi^2(i)]$ es $S[rank(D, \Psi^2(i))]$.
  - ...
- Podemos implementar la búsqueda binaria en el mismo tiempo.
- Podemos contar en tiempo $O(m \log n)$.
- Pero... Ψ es tan grande como A!

El Arreglo de Sufijos Comprimido

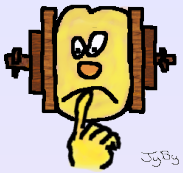


- Ψ es compresible cuando T es compresible.
- En la zona de A donde los sufijos comienzan con la misma letra, Ψ es creciente.
- Por lo tanto Ψ consiste de σ listas crecientes.
- Con bitmaps comprimidos se pueden representar usando

$$\sum_{c \in \Sigma} n_c \log \frac{n}{n_c} + O(n) + o(\sigma n) \;=\; nH_0(T) + O(n) + o(\sigma n) \text{ bits}$$

- Con D tenemos acceso en tiempo constante a Ψ.
- Además debemos sumar los $O(\sigma \log n)$ bits de D y S, pero eso normalmente puede ignorarse.

El Arreglo de Sufijos Comprimido

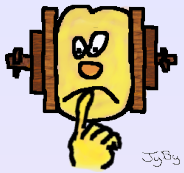


- El problema es el espacio extra $o(\sigma n)$.
- Lo podemos convertir en $O(n \log\log \sigma)$ usando codificación delta.
- Comencemos con la codificación gama de un número x:
  - Codificamos $|x|$ en unario.
  - Luego codificamos x en binario
  - Por ejemplo $x = 20 = 10100$ se codifica como

$$10000\ 10100$$

  - El código necesita $2\lceil \log(x+1) \rceil$ bits.

El Arreglo de Sufijos Comprimido

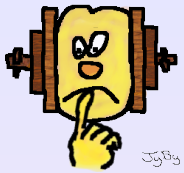


- El código delta funciona así:
  - Codificamos $|x|$ con código gama.
  - Luego codificamos x en binario
  - Por ejemplo $x = 20 = 10100$ ($|x| = 5 = 101$) se codifica como

$$100\ 101\ 10100$$

  - El código necesita $\lceil \log(x+1) \rceil + 2\lceil \log(\lceil \log(x+1) \rceil + 1) \rceil$ bits.

El Arreglo de Sufijos Comprimido

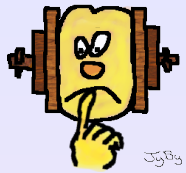


- Si codificamos $\Psi(i) - \Psi(i-1)$ en las zonas crecientes usando códigos delta, obtenemos

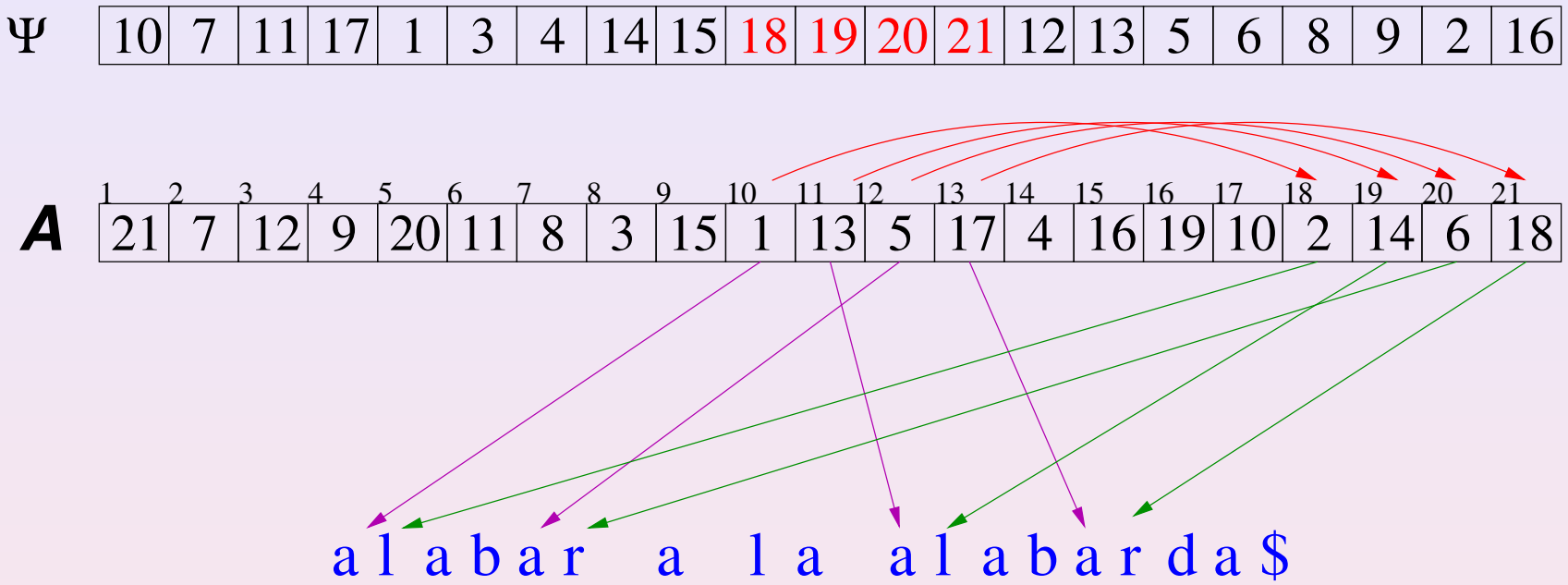
$$nH_0(T) + O(n \log\log \sigma) \quad \text{bits.}$$

- Agregamos valores absolutos cada $O(\log n)$ bits (agrega $O(n)$ bits).
- Y con eso podemos decodificar cualquier $\Psi(i)$ en tiempo constante.
- Notar que:
  - No necesitamos A ni T.
  - Aún podemos contar en tiempo $O(m \log n)$.
- Pero Ψ se puede comprimir más aún...

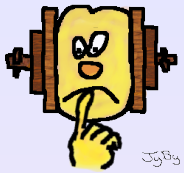
El Arreglo de Sufijos Comprimido



- Si una cadena *abcd* se repite frecuentemente en T...
- habrá una zona en A con los punteros que apuntan a las *a* y otra con los punteros que apuntan a las *b*...
- ... que serán los mismos de las *a* desplazados en 1.
- Estas seudo-copias en A se llaman runs.
- Un run se ve en Ψ como una secuencia de valores consecutivos.
- Codificando los runs en forma especial, nos acercamos a la entropía de orden superior.
- Todo está muy bien, pero
  - ¿Podemos ubicar las ocurrencias, si no tenemos A?
  - ¿Podemos mostrar una parte del texto, si no tenemos T?

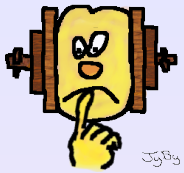

Ψ
10 7 11 17 1 3 4 14 15 18 19 20 21 12 13 5 6 8 9 2 16
A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
21 7 12 9 20 11 8 3 15 1 13 5 17 4 16 19 10 2 14 6 18
a l a b a r a l a a l a b a r d a $

El Arreglo de Sufijos Comprimido

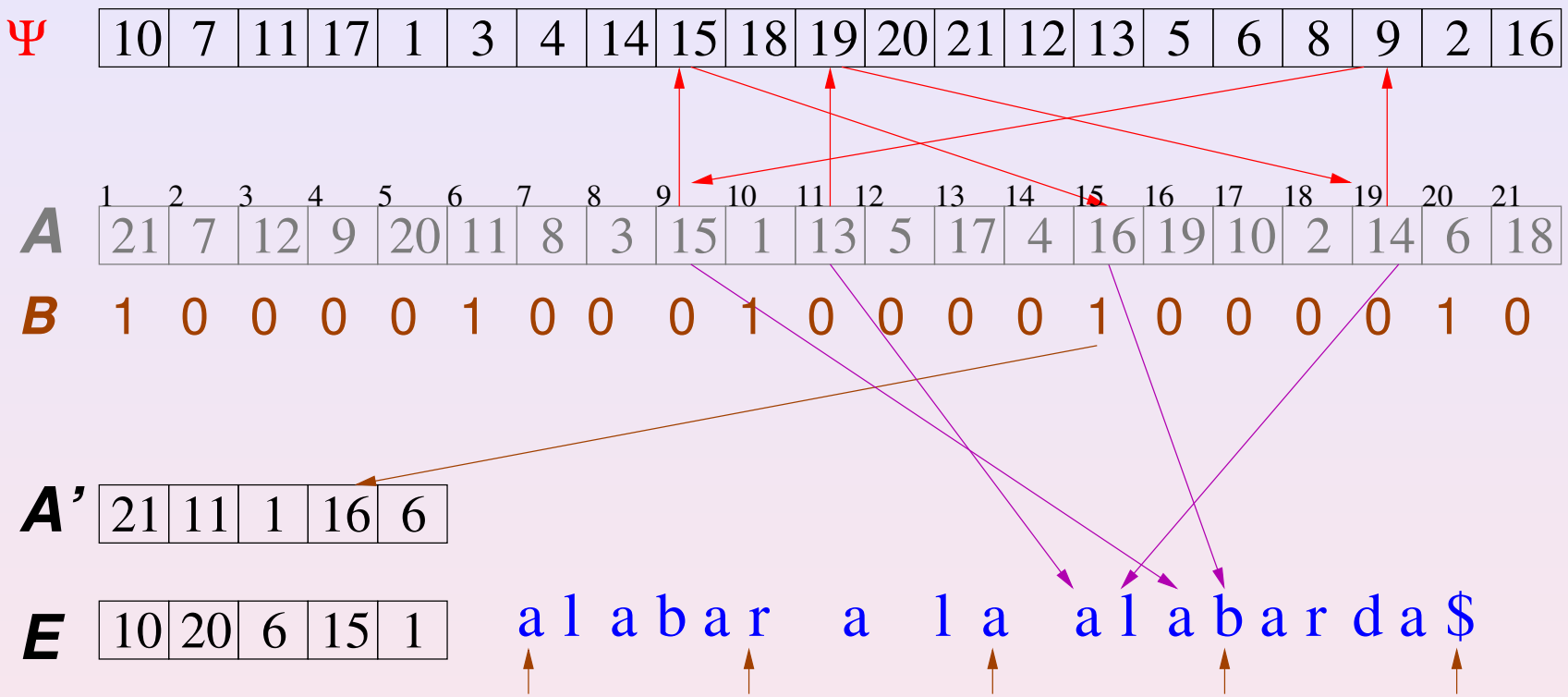


- Haremos un sampling de celdas de A y las almacenaremos.
- Tomaremos los $A[i]$ que apuntan a posiciones de T múltiplos de b.
- Los almacenaremos en orden creciente de i en forma contigua, en un arreglo $A'[1, n/b]$.
- ... y tendremos un bitmap $B[1, n]$ marcando los i sampleados.
- Las posiciones y el bitmap ocuparán $O(\frac{n}{b} \log n)$ bits.

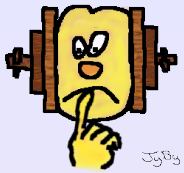
El Arreglo de Sufijos Comprimido



- Supongamos que queremos conocer $A[i]$ (pero no tenemos A).
  - Si $B[i] = 1$, $A[i] = A'[rank(B, i)]$.
  - Sino, si $B[\Psi(i)] = 1$, $A[\Psi(i)] = A'[rank(B, \Psi(i))]$, $A[i] = A[\Psi(i)] - 1$.
  - Sino, ...
  - Si $B[\Psi^t(i)] = 1$, $A[i] = A'[rank(B, \Psi^t(i))] - t$.
- Esto debe terminar en a lo sumo b pasos.
- Por lo tanto podemos conocer $A[i]$ en tiempo $O(b)$.
- Por ejemplo, en tiempo $O(\log n)$, gastando $O(n)$ bits extra.
- Ahora sí hemos reemplazado al suffix array!

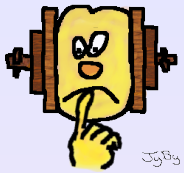

Ψ
10 7 11 17 1 3 4 14 15 18 19 20 21 12 13 5 6 8 9 2 16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
A
21 7 12 9 20 11 8 3 15 1 13 5 17 4 16 19 10 2 14 6 18
B
1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
A'
21 11 1 16 6
E
10 20 6 15 1
a l a b a r a l a a l a b a r d a $

El Arreglo de Sufijos Comprimido



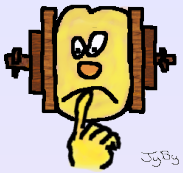
- Ahora almacenaremos el mismo sampling de otra forma.
- Almacenaremos los valores i en orden creciente de T, en $E[1, n/b]$.
- Supongamos que queremos conocer $T[l, r]$ (pero no tenemos T).
  - La última posición sampleada es $l' = 1 + \lfloor l/b \rfloor \cdot b$.
  - Y es apuntada desde $A[i] = A[E[1 + \lfloor l/b \rfloor]]$.
  - Sabemos que $T[l'] = S[rank(D, i)]$.
  - Sabemos que $T[l' + 1] = S[rank(D, \Psi(i))]$.
  - ...
  - Sabemos que $T[r] = S[rank(D, \Psi^{r-l'+1}(i))]$.
- El costo total es $O(b + r - l)$.
- Por ejemplo, en tiempo $O(r - l + \log n)$, gastando $O(n)$ bits extra.
- Ahora sí hemos reemplazado al texto!

El Arreglo de Sufijos Comprimido

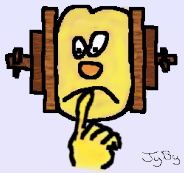


- Todas las estructuras suman $nH_0(T) + O(n \log\log \sigma)$ bits.
- Se puede probar que realmente suman $nH_k(T) + O(n \log\log \sigma)$ bits para k moderado.
- Permiten contar en tiempo $O(m \log n)$.
- Permiten ubicar cada ocurrencia en tiempo $O(\log n)$.
- Permiten mostrar $T[l, r]$ en tiempo $O(r - l + \log n)$.
- Estas estructuras reemplazan el suffix array pero también reemplazan el texto!
- Cuando logramos esto, tenemos un auto-índice comprimido.
  - Representa el texto.
  - Permite búsqueda indexada.
  - Ocupa espacio cercano al del texto comprimido.

Parte II: Otras Estructuras

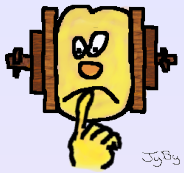


La Transformación de Burrows-Wheeler (BWT)



- Es una permutación reversible de T.
- Se usa como paso previo a algoritmos de compresión como bzip2.
- Pone juntos caracteres que tienen el mismo contexto.
- Basta comprimir esos "caracteres juntos" a orden cero para obtener $nH_k(T)$.
- Veremos que además deriva en un índice comprimido para texto.

La Transformación de Burrows-Wheeler (BWT)



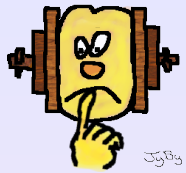
- Tomamos todos los shifts cíclicos de T.
- Es decir, $t_i t_{i+1} \ldots t_{n-1} \$ t_1 t_2 \ldots t_{i-1}$.
- El resultado es una matriz M de $n \times n$ caracteres.
- Ordenamos las filas lexicográficamente.
- M es esencialmente la lista de los sufijos de T en orden:

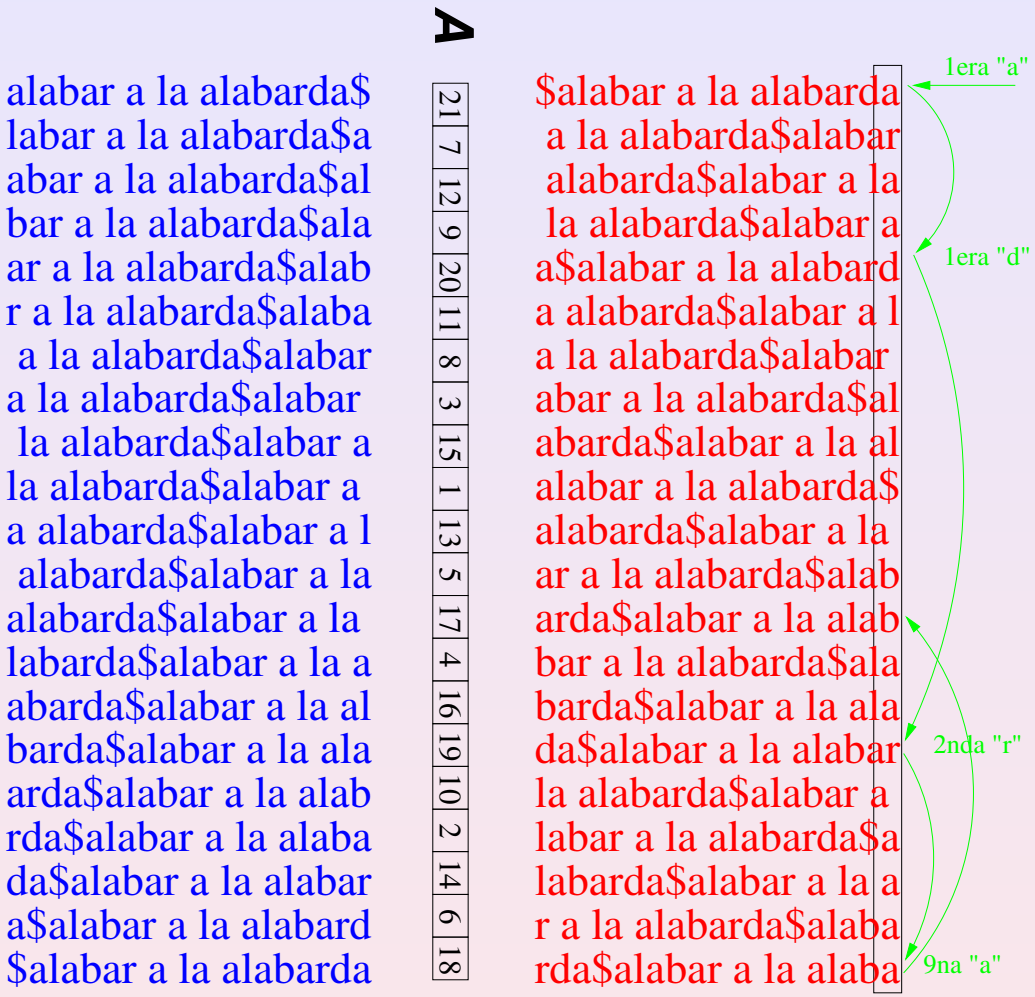
$$M[i] = T[A[i] \ldots n] \cdot T[1 \ldots A[i] - 1]$$

- La primera columna, F, tiene los primeros caracteres de los sufijos: $F[i] = S[rank(D, i)]$.
- La última columna, L, es la BWT de T, $T^{bwt} = L$.
- Es la secuencia de los caracteres que preceden a los sufijos $T[A[i]..]$.

$$L[i] \;=\; T^{bwt}[i] \;=\; T[A[i] - 1]$$

(excepto si $A[i] = 1$, donde $T^{bwt} = T[n] = \$$).

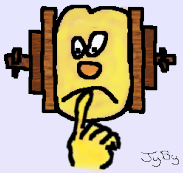




T alabar a la alabarda$

T^{bwt} araadl ll$ bbaar aaaa

La Transformación de Burrows-Wheeler (BWT)

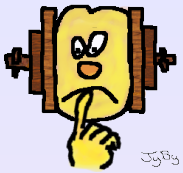


- ¿Cómo invertir la BWT?
- Para todo i, $T = \ldots L[i]F[i]\ldots$.
- Sabemos que $L[1] = T[n-1]$, pues $F[1] = \$ = T[n]$.
- ¿Dónde está $c = L[1]$ en F?
- Todas las ocurrencias de c aparecen en el mismo orden en F y L:
  - Es el orden dado por el sufijo que sigue a c en T.
- Sea entonces $j = rank_c(L, i)$.
- Ese c está en $F[C(c) + j]$.
- Se llama LF-mapping, pues lleva de la columna L a la F:

$$LF(i) = C(L[i]) + rank_{L[i]}(L, i)$$

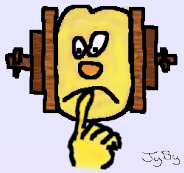
- Entonces $T[n-2] = F[LF(1)] = S[rank(D, LF(1))]$.
- Y $T[n-3] = F[LF(LF(1))]$, etc.

El FM-Index



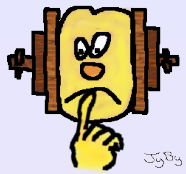
- Usaremos el paralelo entre el arreglo de sufijos y la BWT.
- Reemplazaremos la búsqueda binaria por la búsqueda hacia atrás, más eficiente.
- Para buscar $P[1, m]$ comenzaremos por p_m.
- El segmento de A que le corresponde es $A[C(p_m) + 1, C(p_m + 1)]$.
- En general, sabremos que las ocurrencias de $P[i + 1, m]$ comienzan en $A[sp_{i+1}, ep_{i+1}]$...
- ... y usaremos algo similar al LF-mapping para obtener la zona $A[sp_i, ep_i]$ correspondiente a $P[i, m]$.
- Terminaremos con sp_1 y ep_1.

El FM-Index



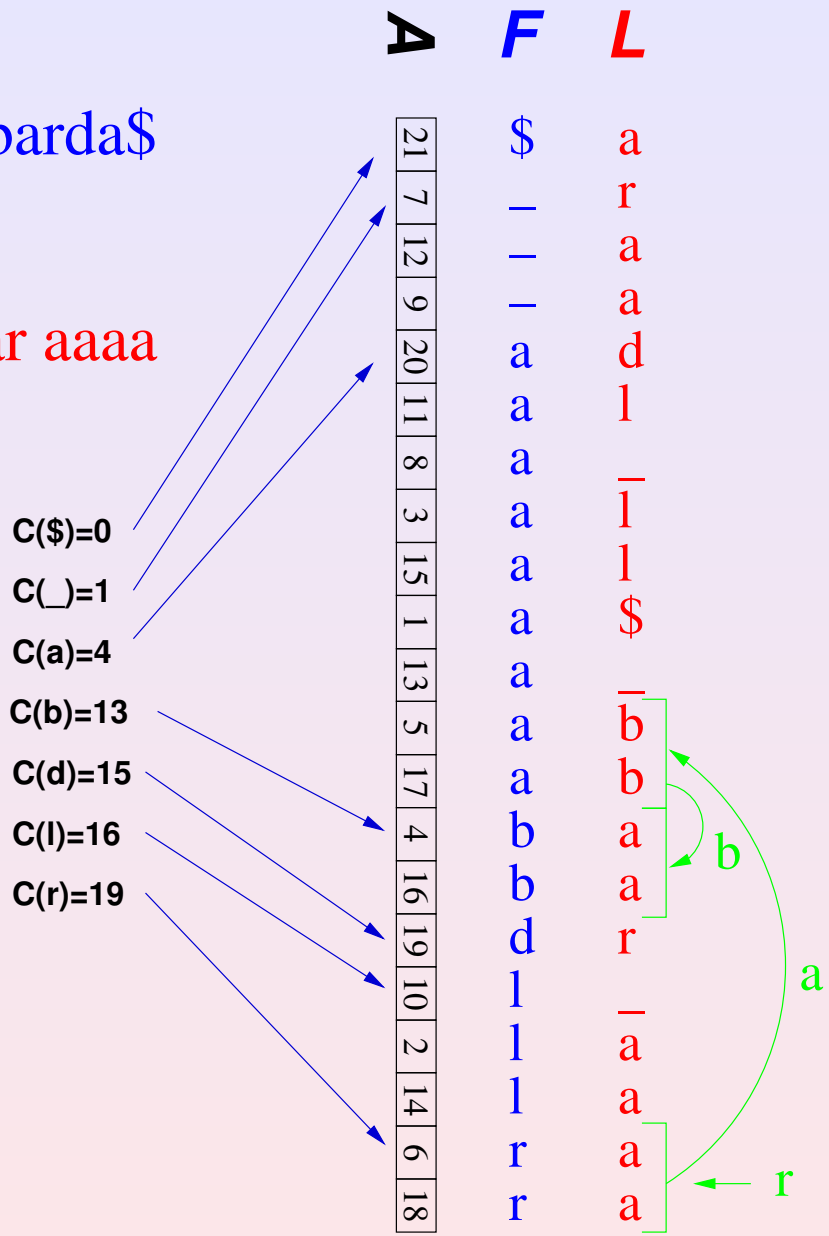
- Supongamos que conocemos el segmento $A[sp_{i+1}, ep_{i+1}]$ de las ocurrencias de $P[i+1, m]$.
- ¿Cómo lo actualizamos a las ocurrencias de $P[i, m]$?
  - Para un $sp_{i+1} \leq j \leq ep_{i+1}$, $M[j][1..m-i] = P[i+1, m]$.
  - Las ocurrencias de $P[i, m]$ en T aparecen como $L[j] = p_i$ en esa área, pues $T = \ldots L[j] \cdot M[j][1..m-i] \ldots$
  - Por lo tanto la respuesta es el rango de los $LF(j)$ donde $L[j] = p_i$.
- Eso se calcula simplemente como

$$\begin{aligned} sp_i &= C(p_i) + rank_{p_i}(L, sp_{i+1} - 1) + 1 \\ ep_i &= C(p_i) + rank_{p_i}(L, ep_{i+1}) \end{aligned}$$

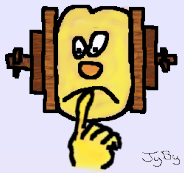


T alabar a la alabarda$

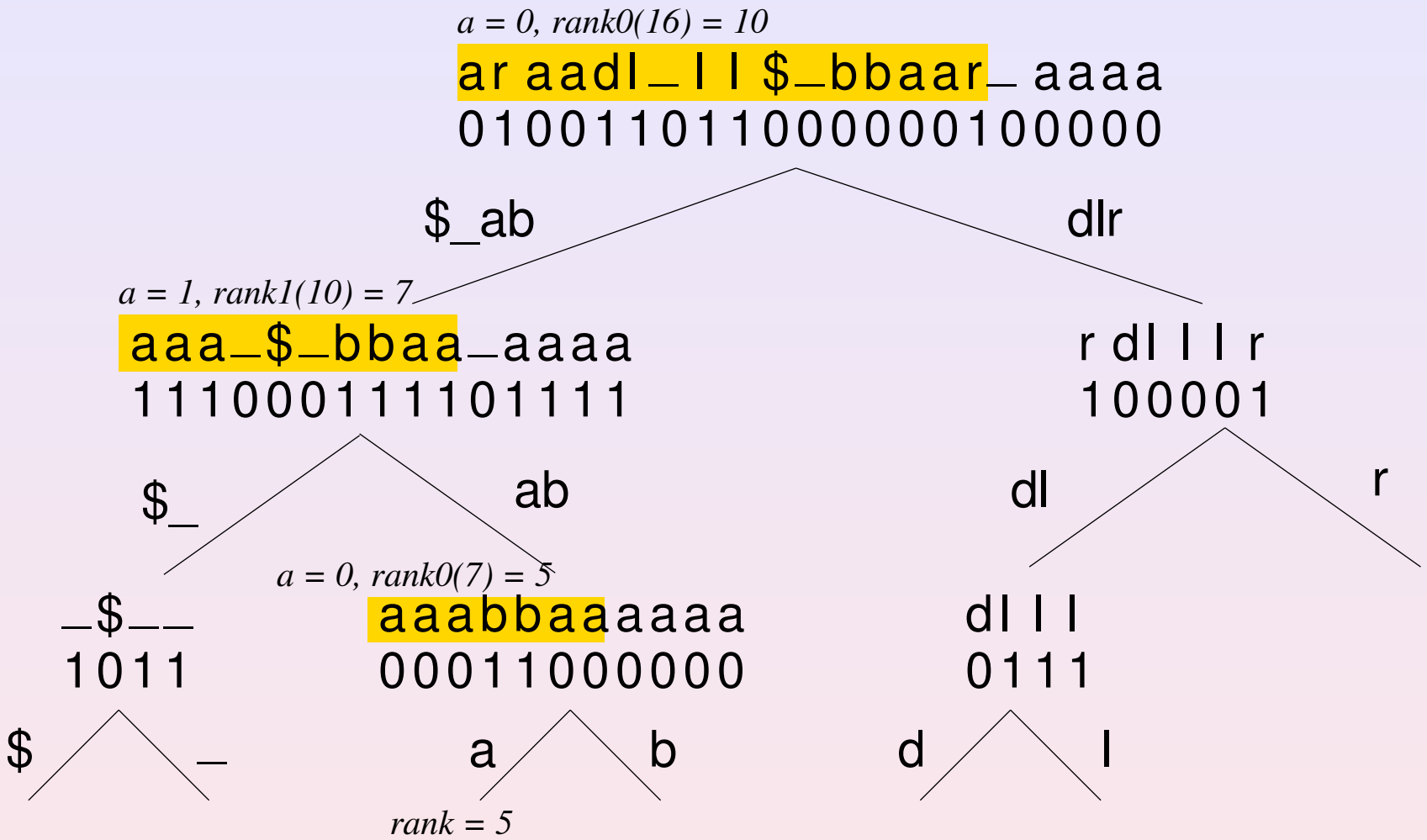
T^{bwt} araadl ll$ bbaar aaaa



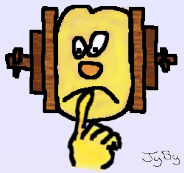
El FM-Index



- Luego de m iteraciones, tenemos el intervalo de $P[1, m]$.
- El costo son $2m$ invocaciones de $rank_c$.
- Usando un wavelet tree sobre $T^{bwt} = L$,
  - Obtenemos espacio $nH_0(T) + o(n \log \sigma)$.
  - Contamos en tiempo $O(m(1 + \frac{\log \sigma}{\log \log n}))$.
- El espacio $nH_0(T)$ se puede mejorar a $nH_k(T)$:
  - Particionando L en bloques que comparten el mismo prefijo $M[j][1, k]$.
  - Particionando L en forma óptima (vale para todo k).
  - Usando bitmaps comprimidos en el wavelet tree, lo que da $nH_k(T)$ automáticamente para todo k.

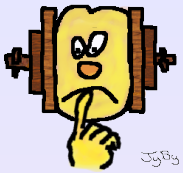

a = 0, rank0(16) = 10
ar aadl_ l l $_bbaar_ aaaa
0100110110000000100000
$_ab
dlr
a = 1, rank1(10) = 7
aaa_$_bbaa_aaaa
1110001111101111
r dl l l r
100001
$_
ab
dl
r
a = 0, rank0(7) = 5
_$__
1011
aaabbaaaaaa
00011000000
dl l l
0111
$
_
a
b
d
l
rank = 5

El FM-Index

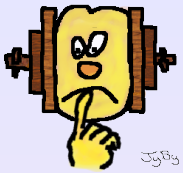


- Para obtener $A[i]$ y $T[l, r]$ usamos el mismo mecanismo de sampling.
- Para conservar $o(n \log \sigma)$ espacio extra,
  - Sampleamos cada $b = \log_\sigma n \log\log n$.
  - Ubicamos cada ocurrencia en tiempo $O(\log n \log\log n)$.
  - Mostramos texto en $O((b + r - l)(1 + \frac{\log \sigma}{\log\log n}))$.
- Es uno de los mejores índices en la teoría.
- En la práctica también, pero el suffix array comprimido es muy competitivo también.

Parte II: Otras Estructuras

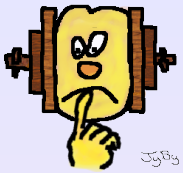


Indices tipo Lempel-Ziv



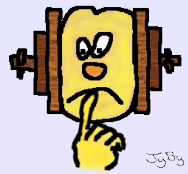
- Lempel-Ziv es un tipo de compresión basado en sustituir cadenas de T por punteros a sus apariciones previas.
- Cuanto más repetitivo es un texto, más se puede reemplazar y mejor se comprime.
- Los compresores zip, gzip, pkzip, arj, etc. son de este tipo.
- Nos interesará un tipo particular llamado LZ78.
- Veremos que tiene propiedades que permiten indexar el texto comprimido.

Indices tipo Lempel-Ziv

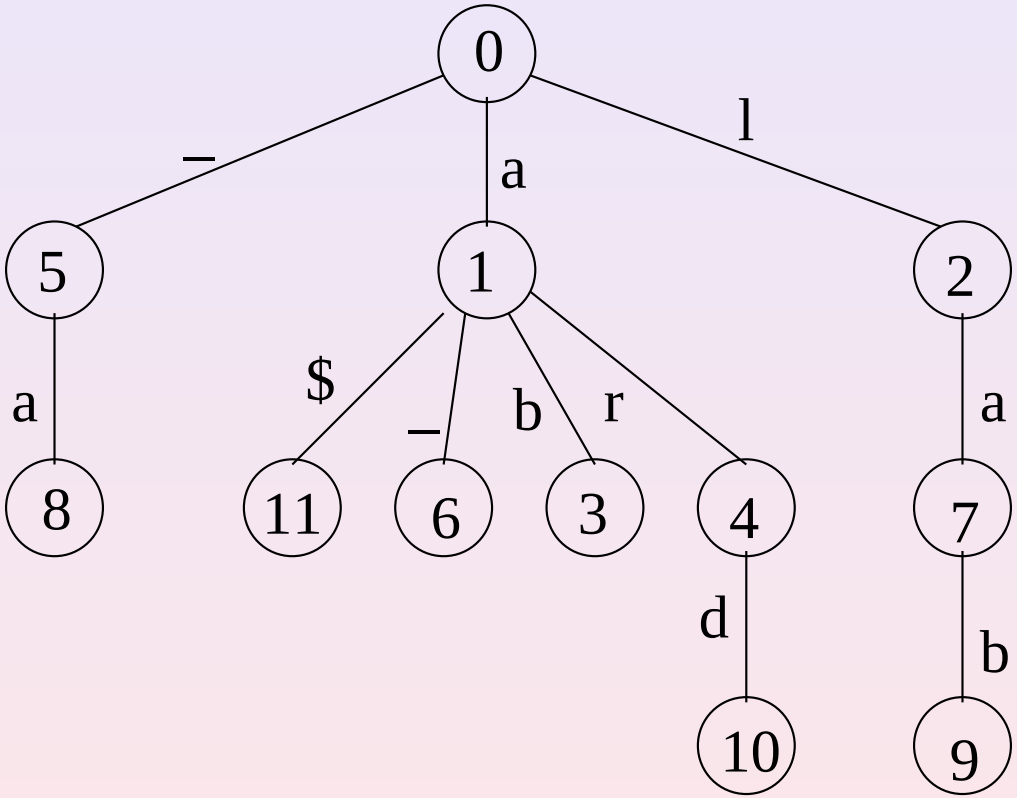


Compresión LZ78

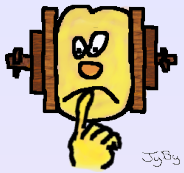
- Cortamos el texto en frases.
- Cada frase consiste en la frase previa más larga posible y una letra adicional.
- Todas las frases son distintas.
- Se almacenan todas las frases en un árbol digital, donde cada nodo es una frase.
- En el peor caso se forman $n' \leq n/\log_\sigma n$ frases en T.
- Representando cada frase con $\log n$ bits el total queda $nH_k(T) + o(n \log \sigma)$ para $k = o(\log_\sigma n)$.



a.l.ab.ar. .a .la. a.lab.ard.a$

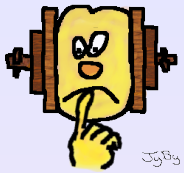


Indices tipo Lempel-Ziv



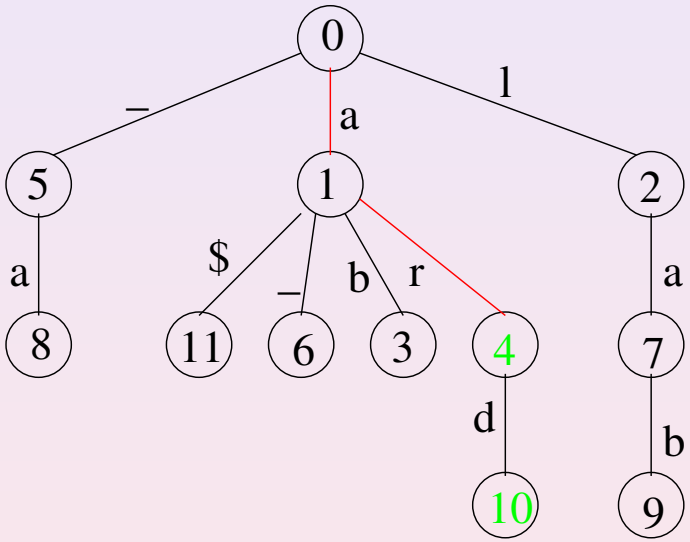
El LZ-Index

- El LZ-Index almacena varias componentes:
  - *LZTrie*: el árbol de las frases, como $2n'$ paréntesis más los números de frases en preorden (*lds*).
  - *RevTrie*: el árbol de las frases reversas, como $4n'$ paréntesis más los números de frase en preorden (*RIds*).
  - *Node*: el mapeo de número de frase a nodos de *LZTrie*.
  - *RNode*: el mapeo de número de frase a nodos de *RevTrie*.
  - *Range*: puntos en dos dimensiones: $(preorden_{Rev}(j), preorden_{LZ}(j+1))$ para cada frase j.
- Con esas estructuras puede ubicar las ocurrencias en tiempo $O(m^2 \log m + (m + occ) \log n)$.
- En la práctica el conteo es muy lento, pero ubicar las ocurrencias es rápido.

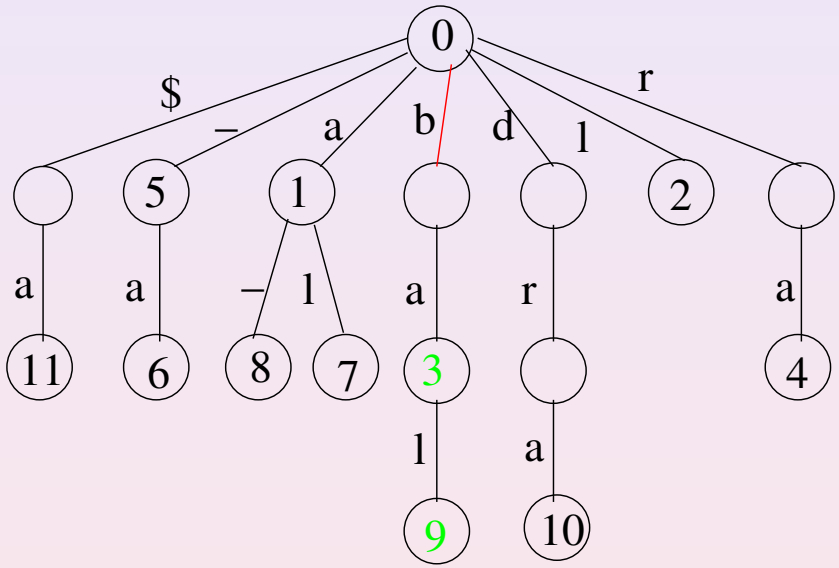


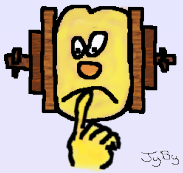
a.l.ab.ar. .a .la. a.lab.ard.a$

LZTrie



RevTrie

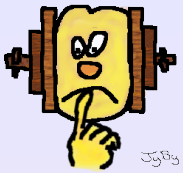




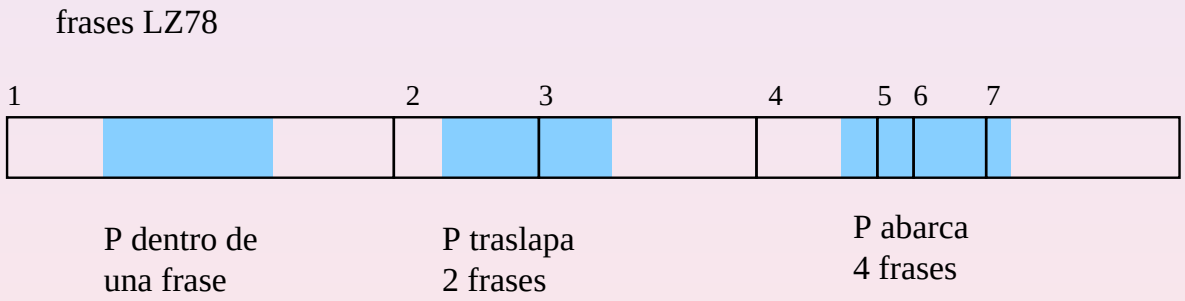
a.l.ab.ar. .a .la. a.lab.ard.a$

	$a	_	_a	a	a_	al	ba	bal	dra	l	ra
_											4
_a						7					
a											
a$									10		
a_		5									
ab										2	
ar							3				
ard								9			
l				1							
la			6								
lab					8						

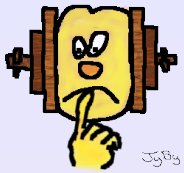
Indices tipo Lempel-Ziv



- Tres tipos de ocurrencias a encontrar:
  - Tipo 1: *P* está totalmente contenido en una frase.
  - Tipo 2: *P* traslapa con dos frases consecutivas.
  - Tipo 3: *P* abarca tres frases o más.



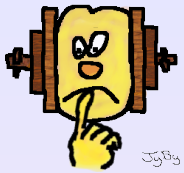
Indices tipo Lempel-Ziv



Ocurrencias tipo 1

- Si una frase B contiene P pero no termina en P...
- ... y dado que $B = B' \cdot c$ para otra frase B' y carácter c...
- ... se deduce que P está contenido en B' también.
- Buscamos P^R en *RevTrie* para hallar todas las frases que terminan con P.
- Para cada una de esas posiciones en preorden j de *RevTrie*, $Node(Rids(j))$ es la posición en el *LZTrie*.
- Todos los descendientes de ese nodo son ocurrencias.
- Recorremos el subárbol de *LZTrie* reportando los *Ids*.
- Tardamos $O(m)$ más $O(1)$ por cada ocurrencia.

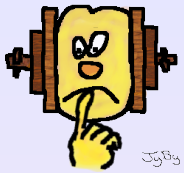
Indices tipo Lempel-Ziv



Ocurrencias tipo 2

- Si P comienza en la frase B_j y termina en B_{j+1}...
- ... entonces B_j termina con $P[1,i]$ y B_{j+1} empieza con $P[i+1,m]$ para algún i.
- Buscamos cada $P[i+1,m]$ en *LZTrie*, obteniendo el rango de preórdenes $[x_i, x'_i]$.
- Buscamos cada $P[1,i]^R$ en *RevTrie*, obteniendo el rango de preórdenes $[y_i, y'_i]$.
- Reportamos todos los puntos de *Range* en $[x_i, x'_i] \times [y_i, y'_i]$.
- El tiempo es $O(m^2 + m \log n)$ más $O(\log n)$ por ocurrencia.

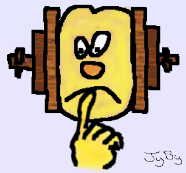
Indices tipo Lempel-Ziv



Ocurrencias tipo 3

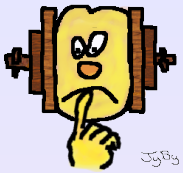
- Si P abarca más de 2 frases, entonces contiene completamente una frase.
- Como todas las frases del parsing LZ78 son distintas...
- ... existen a lo sumo $O(m^2)$ ocurrencias que verificar en T.
- Se pueden verificar en tiempo $O(m^2 \log m)$ usando *Node* y *RNode*.

Indices tipo Lempel-Ziv

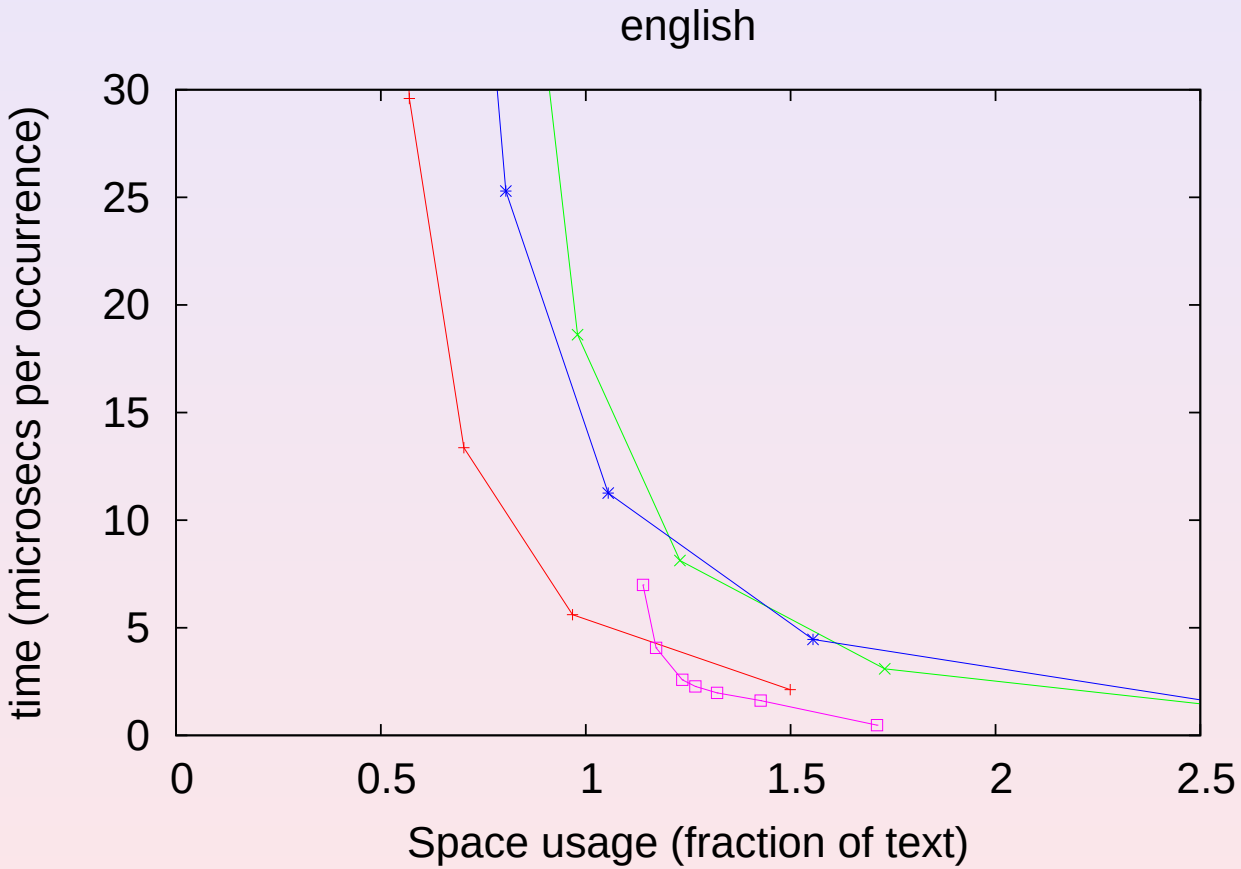


- *Ids* y *Node* son permutaciones inversas.
- *RIds* y *RNode* son permutaciones inversas.
- Por lo tanto, cada par ocupa $(1+\epsilon)n\log n$ bits.
- La estructura de Range ocupa $n\log n$ bits.
- En total el espacio es $(3+\epsilon)nH_k(T)+o(n\log\sigma)$, para cualquier constante ϵ.
- Se puede eliminar *Range* para reducir el espacio a $(2+\epsilon)nH_k(T)+o(n\log\sigma)$...
- ... y los tiempos son mejores en la práctica.
- Para eliminar *Range* se verifican todos los candidatos de *LZTrie* en *RevTrie* (usando *Ids* y *RNode*) o al revés (usando *RIds* y *Node*).
- El espacio se puede reducir incluso a $(1+\epsilon)nH_k(T)+o(n\log\sigma)$.

Comparando Indices para Texto



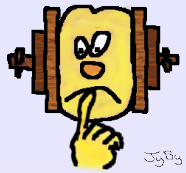
Lenguaje Natural



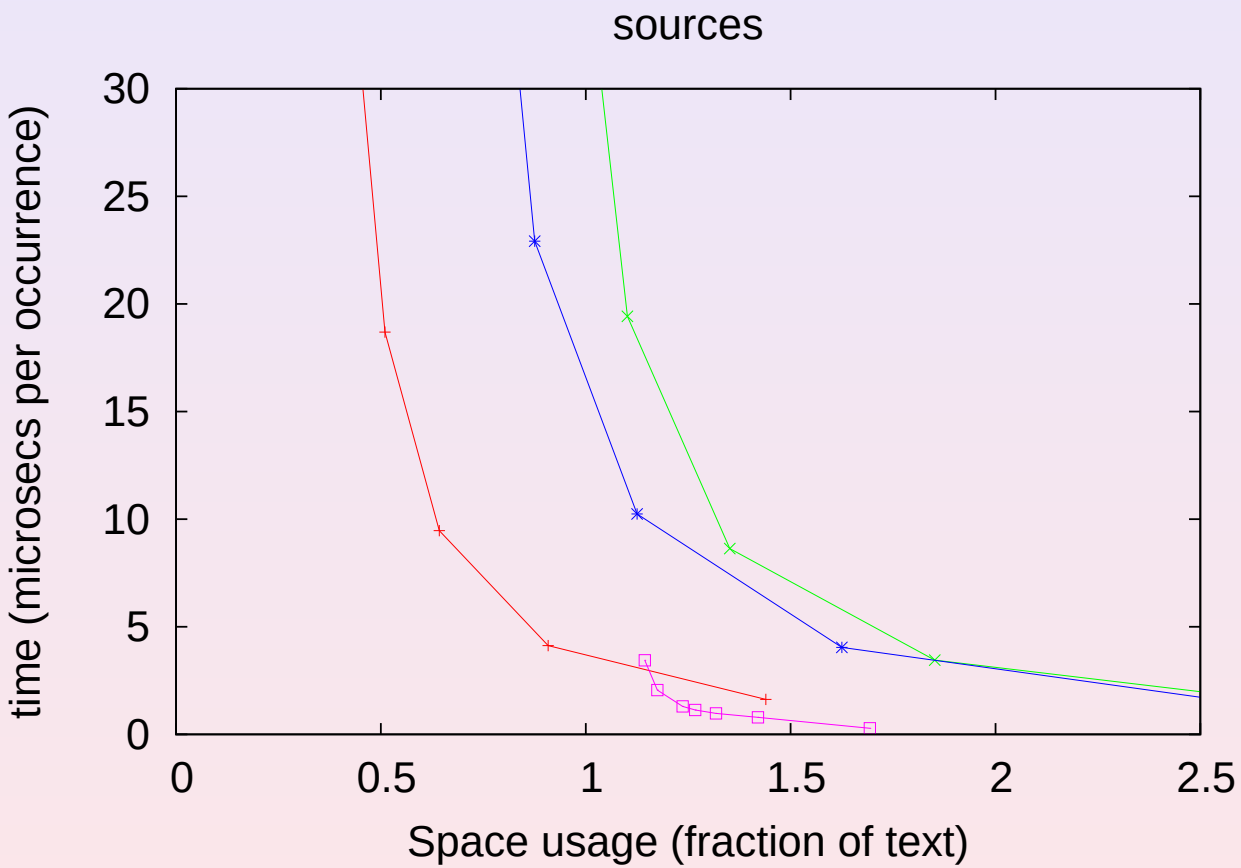
Arreglo de Sufijos Comprimido
FM-Index con Wavelet Tree
FM-Index particionado
LZ-Index

Datos obtenidos por Rodrigo González y Rossano Venturini con los archivos del sitio http://pizzachili.dcc.uchile.cl

Comparando Indices para Texto



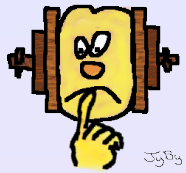
Código Fuente



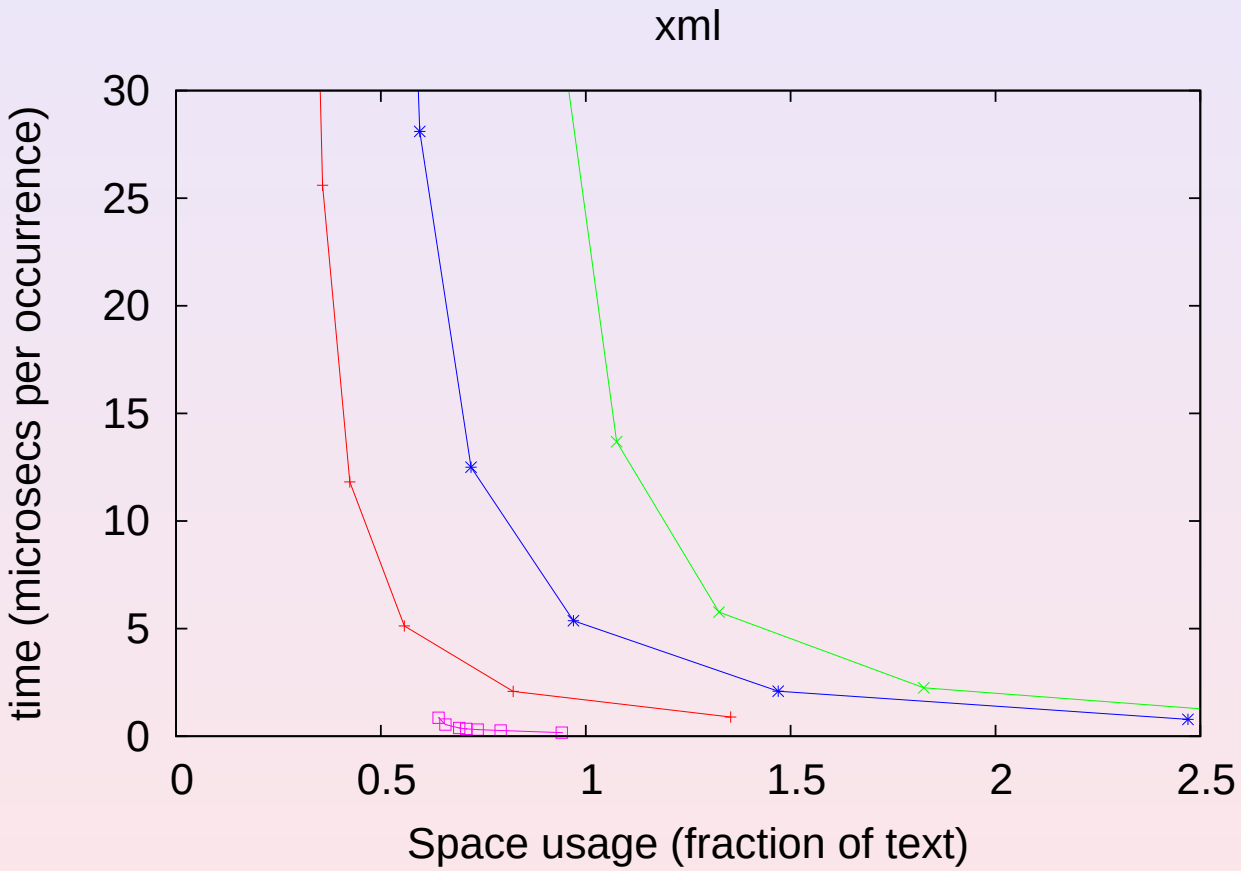
Arreglo de Sufijos Comprimido
FM-Index con Wavelet Tree
FM-Index particionado
LZ-Index

Datos obtenidos por Rodrigo González y Rossano Venturini con los archivos del sitio http://pizzachili.dcc.uchile.cl

Comparando Indices para Texto



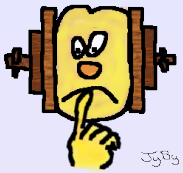
Texto Semiestructurado



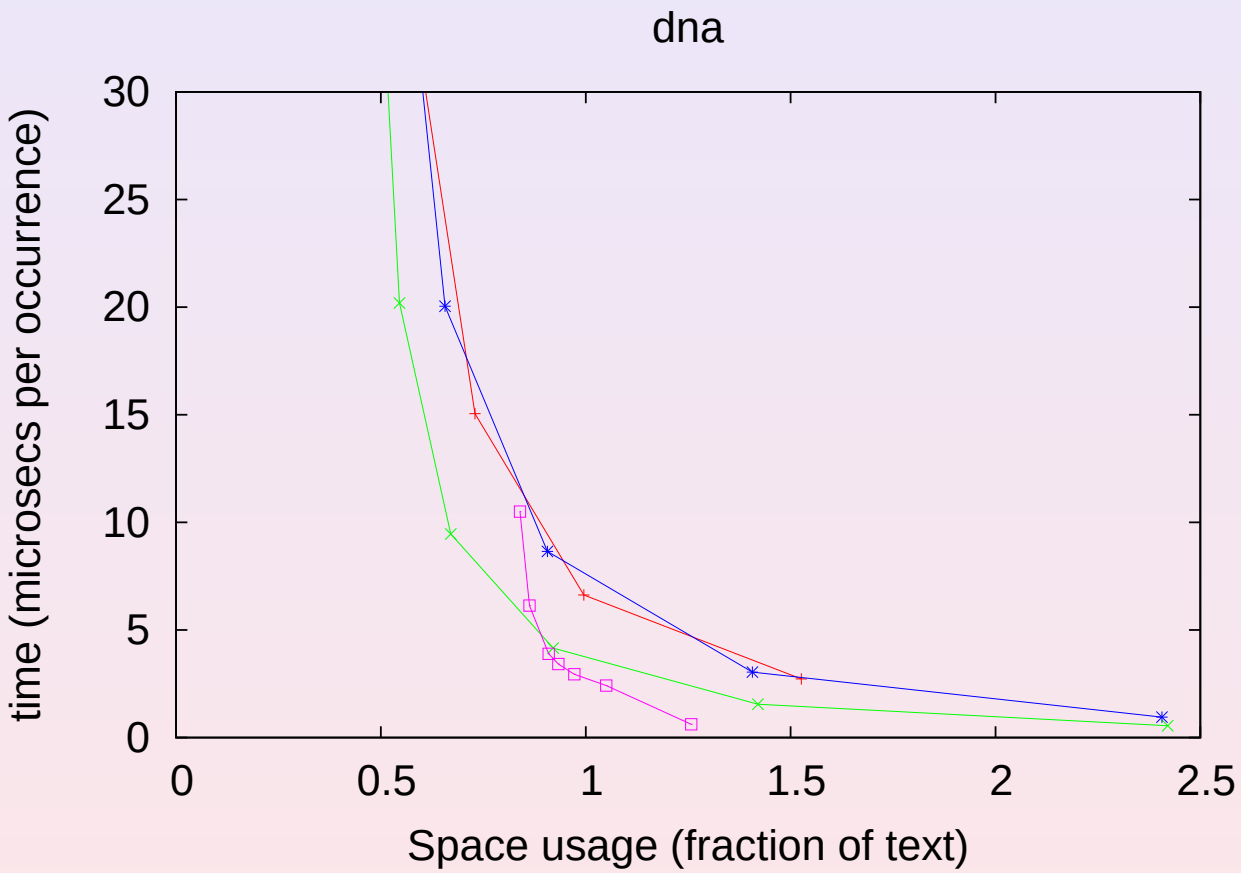
Arreglo de Sufijos Comprimido
FM-Index con Wavelet Tree
FM-Index particionado
LZ-Index

Datos obtenidos por Rodrigo González y Rossano Venturini con los archivos del sitio http://pizzachili.dcc.uchile.cl

Comparando Indices para Texto



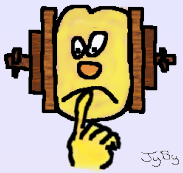
ADN



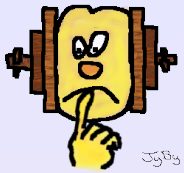
Arreglo de Sufijos Comprimido
FM-Index con Wavelet Tree
FM-Index particionado
LZ-Index

Datos obtenidos por Rodrigo González y Rossano Venturini con los archivos del sitio http://pizzachili.dcc.uchile.cl

Parte II: Otras Estructuras

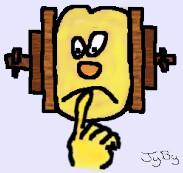


Grafos



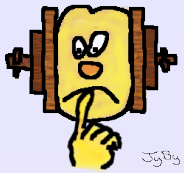
- Un conjunto de n nodos y e aristas.
- Las aristas son flechas que van de un nodo a otro.
- Los vecinos de un nodo son los alcanzables desde él por una flecha.
- Los vecinos reversos son los que llegan a él por una flecha.
- Nos interesa navegar de un nodo a sus vecinos o vecinos reversos.
- Y otras preguntas como
  - ¿Existe una arista de un cierto nodo a cierto otro nodo?
  - Grado interior: ¿cuántas aristas llegan a un nodo?
  - Grado exterior: ¿cuántas aristas salen de un nodo?

Grafos

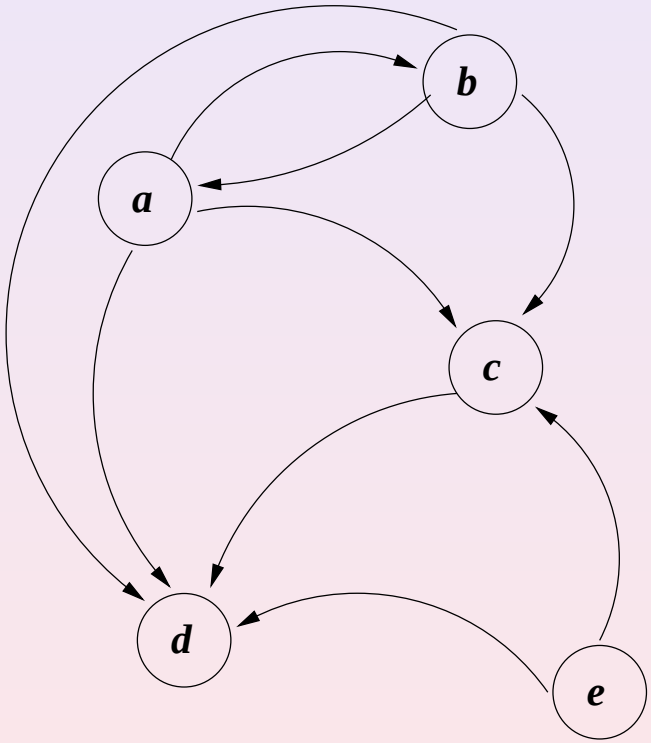


- Nos interesaremos especialmente en los grafos de la Web.
- Se necesita correr algoritmos sobre grandes subconjuntos de la Web.
  - Para calcular PageRank (Google).
  - Para descubrir comunidades (Yahoo!).
  - Para análisis de redes sociales.
  - Y muchas otras aplicaciones.
- Esos algoritmos no corren bien en memoria secundaria.
- Una representación compacta navegable permitirá analizar grafos más grandes en RAM.

Representación Clásica de Grafos

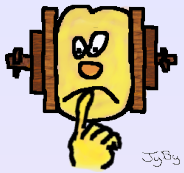


Matriz de Incidencia



	a	b	c	d	e
a		●	●	●	
b	●		●	●	
c				●	
d					
e			●	●	

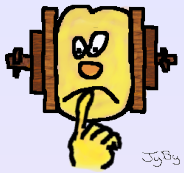
Representación Clásica de Grafos



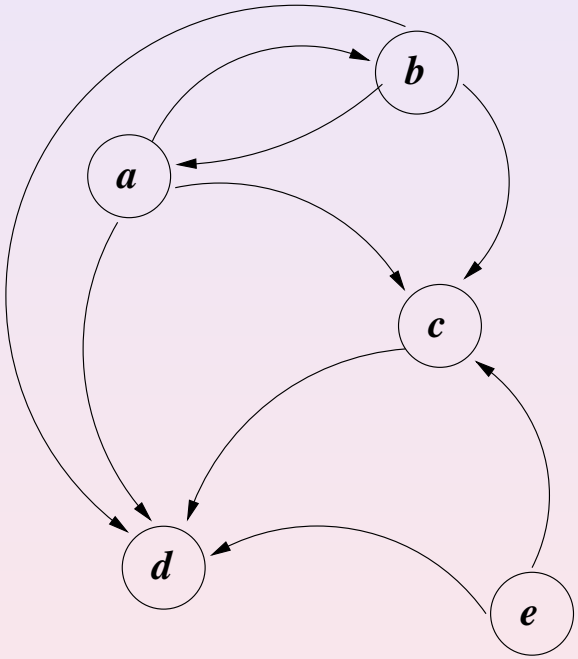
Matriz de Incidencia

- Con n nodos y e aristas, ocupa n^2 bits.
- Responde existencia en tiempo $O(1)$.
- Encuentra todos los vecinos directos/reversos en $O(n)$.
- Calcula grado interior/exterior en $O(n)$.

Representación Clásica de Grafos

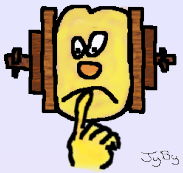


Lista de Adyacencia



a: *d,c,b*	*a:* *b*
b: *d,c,a*	*b:* *a*
c: *d*	*c:* *a,b,e*
d:	*d:* *a,b,c,e*
e: *d,c*	*e:*

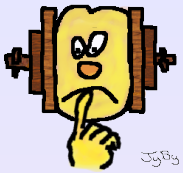
Representación Clásica de Grafos



Lista de Adyacencia

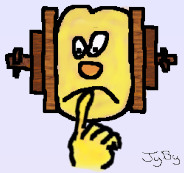
- Con n nodos y e aristas, ocupa $n \log e + e \log n$ bits.
- Encuentra cada vecino en tiempo $O(1)$.
- Calcula grado exterior en tiempo $O(1)$.
- Responde existencia en tiempo $O(n)$.
- Necesita otro tanto para los reversos.

Compresión de Grafos



- Viéndolo como una relación binaria entre nodos...
- ... se puede conseguir $e \log(n^2/e)\ (1 + o(1))$ bits...
- ... y responder todo en tiempo $O(\log \log n)$.
- Obtiene lo mejor de los dos mundos...
- ... pero aún no es demasiado bueno para la Web.

Representación de Grafos

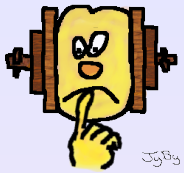


Algunos ejemplos sobre grafos Web

Medidos en *bpe* (bits por arista).

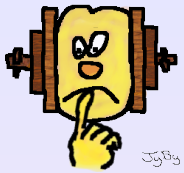
Grafo	*n*	*e*	Matriz	Lista	2 x Lista	Rel. Bin
UK	18.5M	298M	1.15M	25.89	51.78	20.13
EU	860K	19M	39K	20.81	41.62	15.25
Arabic	23M	640M	827K	25.51	51.02	19.66
Indochina	7M	194M	253K	23.73	47.46	17.95

Compresión de Grafos



Grafos Planares y Variantes

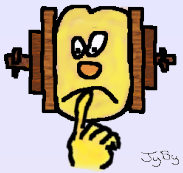
- Con n nodos, tienen $O(n)$ aristas.
- Distintas técnicas para comprimirlos a $O(n)$ bits.
- La mayoría consiste en dividirlos en varios árboles y representar estos árboles como paréntesis.
- Algunas permiten acceso directo.
- Hay algunos resultados para tipos especiales de grafos.
- Es improbable que tengan algún impacto en grafos Web.



Separadores de Grafos

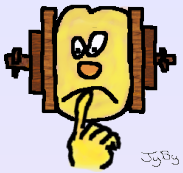
- Encontrar zonas que se pueden desconectar cortando unas pocas aristas.
- Renumerar los nodos y comprimirlos separadamente.
- Pueden obtener 13–16 bpe y ser más rápidos que la versión descomprimida (por efectos de caché).
- Para vecinos reversos necesitan el grafo traspuesto.

Compresión de Grafos Web



- Distribución sesgada de grados interior/exterior (power laws).
  - Una lista de adyacencia tiene baja entropía.
- Localidad de referencia: la mayoría de los links apuntan al mismo site.
  - Listar los nodos en orden lexicográfico de URL.
  - Usar técnicas de codificación de gaps para comprimir las listas.
- Modelo de copia: los links que salen se parecen a los de alguna otra página.
  - Encontrar una página similar y codificar diferencialmente.

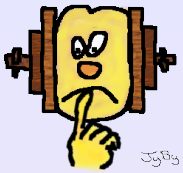
Compresión de Grafos Web



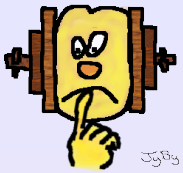
Un muy buen exponente: WebGraph

- 3 bpe para compresión pura.
- 6 bpe para recuperar cada vecino directo o reverso dentro del microsegundo.
- Esto considera el grafo y su traspuesto.

Parte II: Otras Estructuras

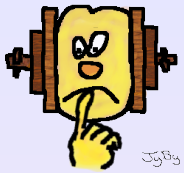


Comprimiendo Grafos como Texto

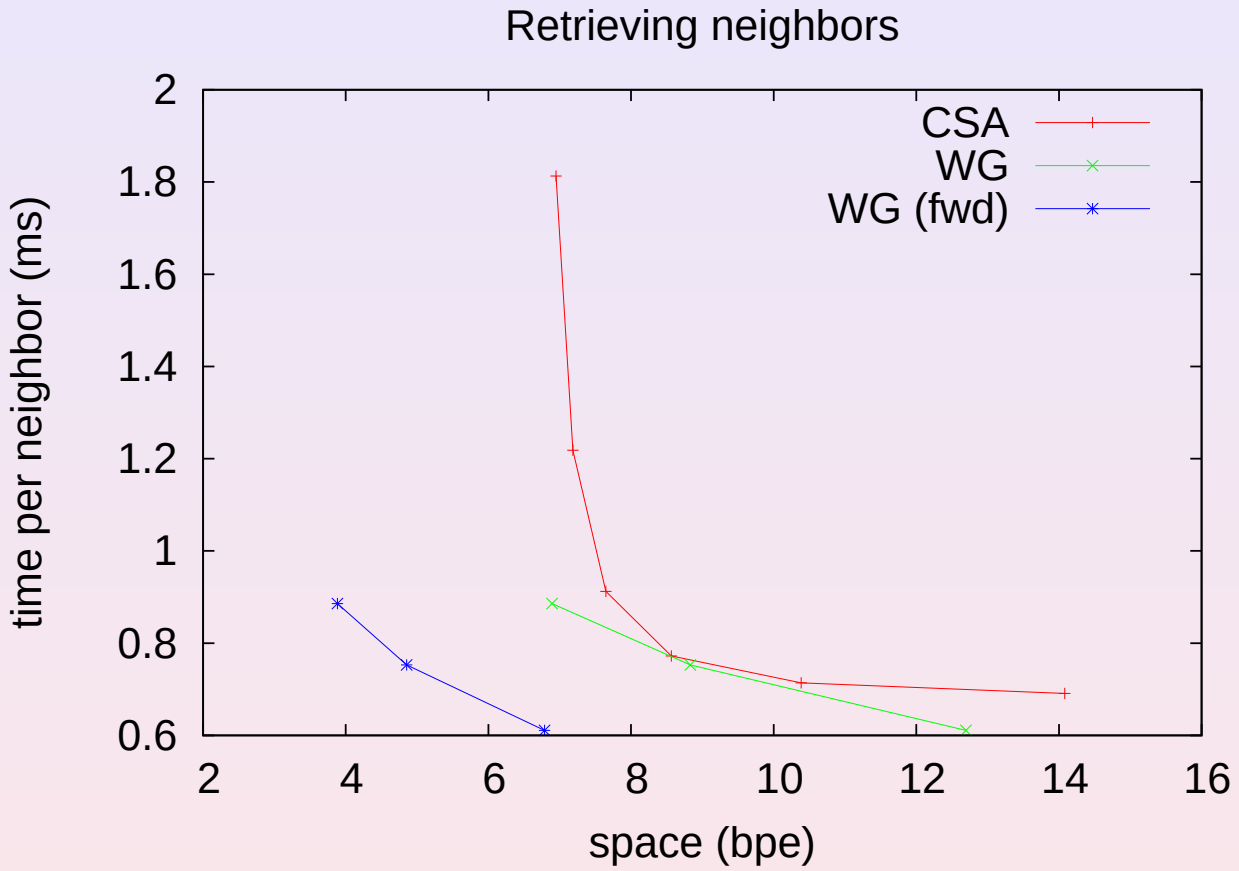


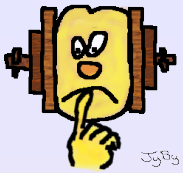
- Concatenar las listas de adyacencia en un texto.
- Construir un autoíndice comprimido sobre ese texto.
- Mostrar: vecinos de un nodo
- Ubicar: vecinos reversos de un nodo
- Contar: grado interior de un nodo
- La entropía de orden k de este texto captura el modelo de copia.
- La entropía de orden cero captura la distribución sesgada de grado interior.

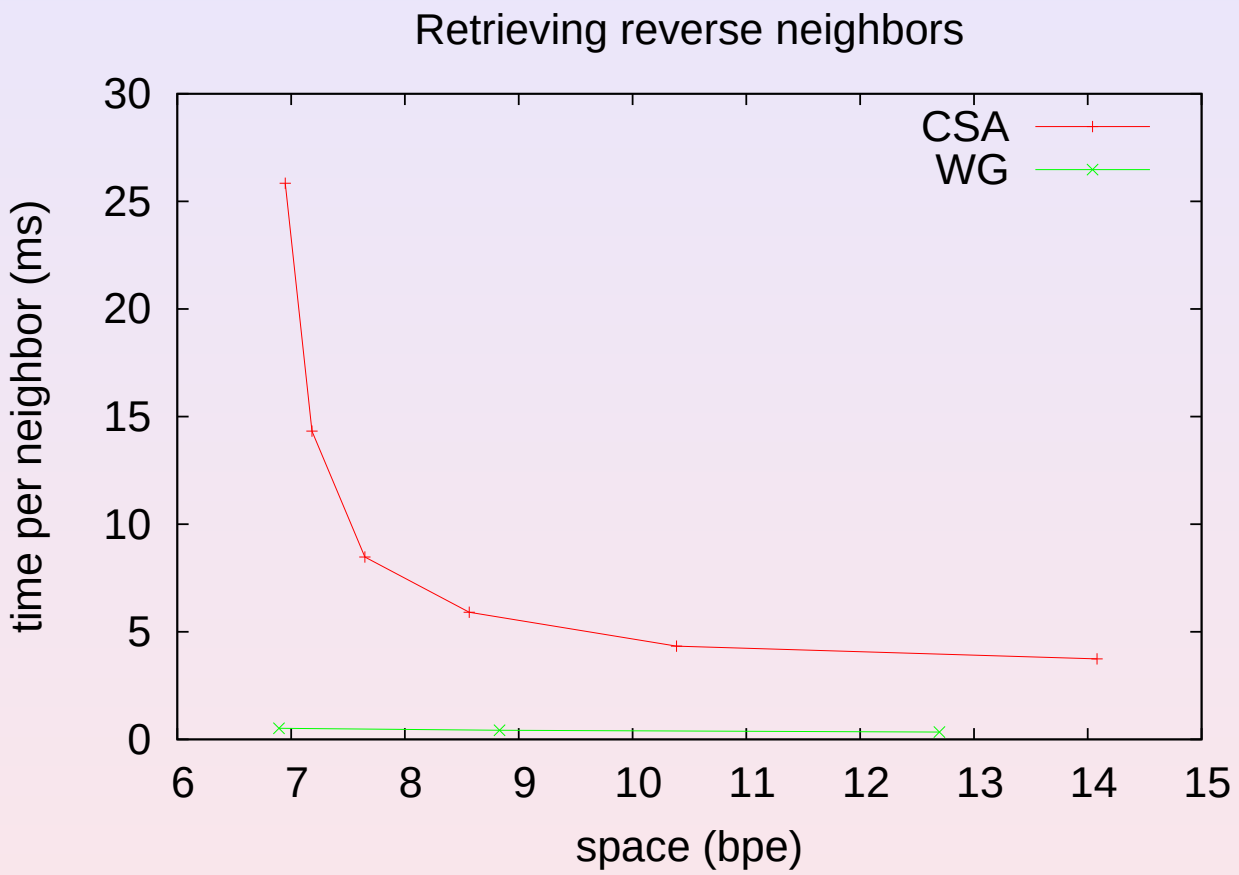
Comparación Experimental



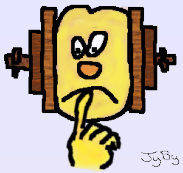
- Usando el Arreglo de Sufijos Comprimido como autoíndice.
- Contra los resultados de WebGraph.
- Sobre el mismo crawl UK, 18.5 Mnodos, 292 Mlinks.
- Vecinos: resultados comparables.
- Reversos: WebGraph el 10 veces mejor.


Retrieving neighbors
time per neighbor (ms)
space (bpe)
CSA
WG
WG (fwd)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
2
4
6
8
10
12
14
16

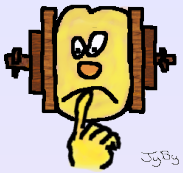



Retrieving reverse neighbors
time per neighbor (ms)
space (bpe)
CSA
WG
0
5
10
15
20
25
30
6
7
8
9
10
11
12
13
14
15

Parte II: Otras Estructuras

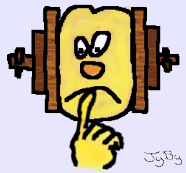


Comprimiendo con Re-Pair



- Hay una forma elegante y efectiva de comprimir T?
- Re-Pair: encontrar el par más repetido en T y reemplazarlo por un nuevo símbolo, hasta que todos los pares sean únicos.

Re-Pair



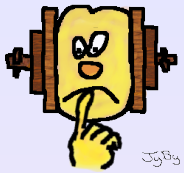
$a\ a\ a\ b\ c\ a\ a\ b\ a\ a\ a\ b\ c\ a\ b\ d\ a\ b\ d$

a	a	4
a	b	5
b	c	2
c	a	2
b	d	2
d	a	1

$A \longrightarrow a\ b$

$a\ a\ A\ c\ a\ A\ a\ a\ A\ c\ A\ d\ A\ d$

Re-Pair



$a\ a\ A\ c\ a\ A\ a\ a\ A\ c\ A\ d\ A\ d$

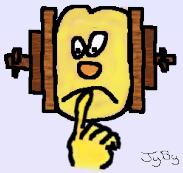
$a\ a$	2
$a\ A$	3
$A\ c$	2
$c\ a$	1
$A\ d$	2
$c\ A$	1
$d\ A$	1

$A \longrightarrow a\ b$

$B \longrightarrow a\ A$

$a\ B\ c\ B\ a\ B\ c\ A\ d\ A\ d$

Re-Pair



$a\ B\ c\ B\ a\ B\ c\ A\ d\ A\ d$

$a\ B$	2
$B\ c$	2
$c\ B$	1
$B\ a$	1
$c\ A$	1
$A\ d$	2
$d\ A$	1

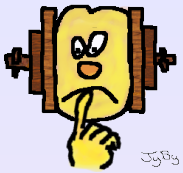
$A \longrightarrow a\ b$

$B \longrightarrow a\ A$

$C \longrightarrow A\ d$

$a\ B\ c\ B\ a\ B\ c\ C\ C$

Re-Pair



$a\ B\ c\ B\ a\ B\ c\ C\ C$

$a\ B$	2
$B\ c$	2
$c\ B$	1
$B\ a$	1
$c\ C$	1
$C\ C$	1

$A \longrightarrow a\ b$

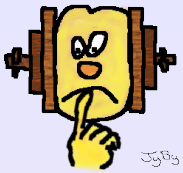
$B \longrightarrow a\ A$

$C \longrightarrow A\ d$

$D \longrightarrow B\ c$

$a\ D\ B\ a\ D\ C\ C$

Re-Pair



$a\ D\ B\ a\ D\ C\ C$

$a\ D$	2
$D\ B$	1
$B\ a$	1
$D\ C$	1
$C\ C$	1

$A \rightarrow a\ b$

$B \rightarrow a\ A$

$C \rightarrow A\ d$

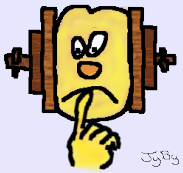
$D \rightarrow B\ c$

$E \rightarrow a\ D$

diccionario

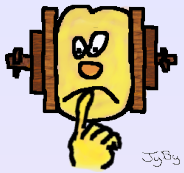
$E\ B\ E\ C\ C$ ← *secuencia comprimida*

Re-Pair



- Comprime bien, descomprime rápido.
- Aprovecha la propiedad de copia.
- Comprime mejor si se codifica T diferencialmente.
- Se comporta bien en memoria secundaria
  - Siempre que el diccionario quepa en RAM.
- Se puede mejorar Re-Pair mismo:
  - Representar el diccionario con estructuras de datos compactas.
  - Ganamos hasta un 50% en el espacio del diccionario.
  - Esto es importante sobre todo en memoria secundaria.

Mejorando Re-Pair



Rules

A	→	a	b
B	→	a	A
C	→	A	d
D	→	B	c
E	→	a	D

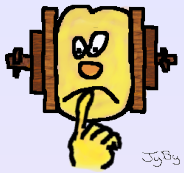
Rules

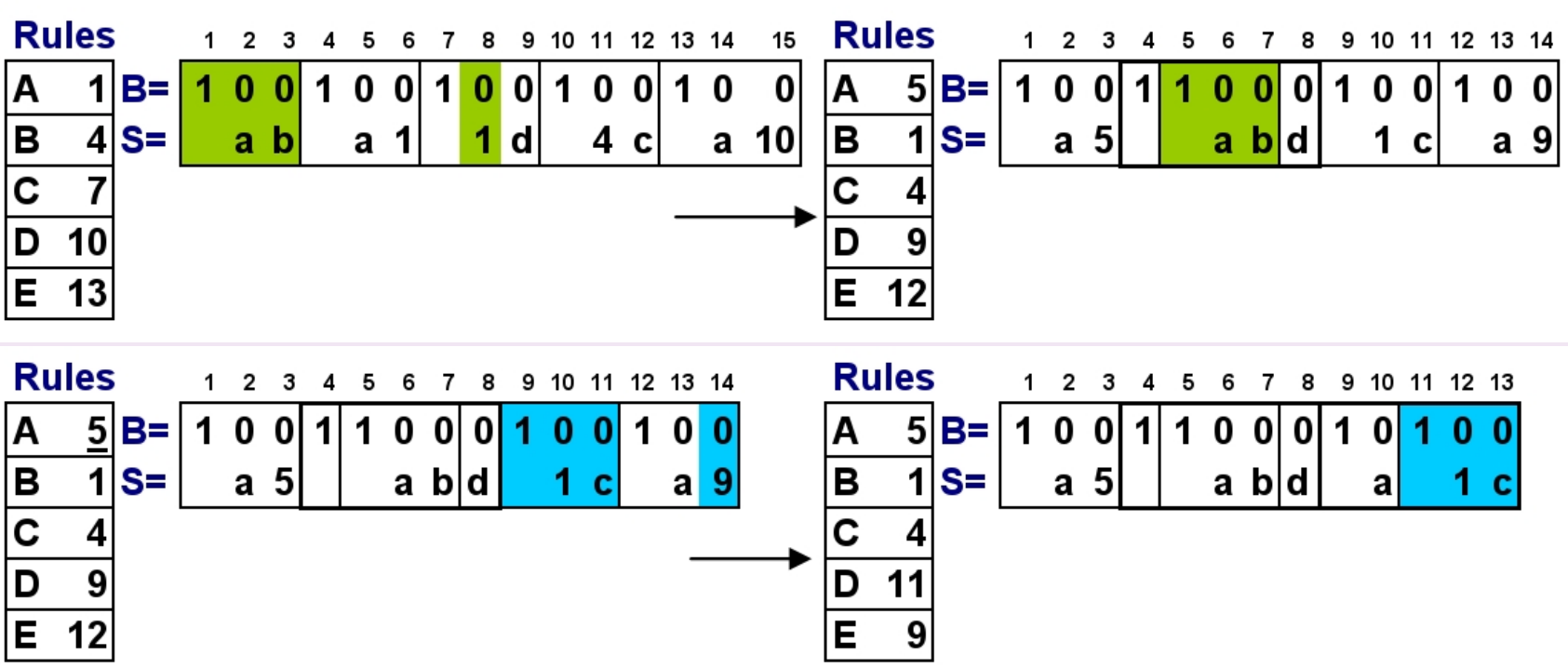
A	1
B	4
C	7
D	10
E	13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B=	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
S=		a	b		a	1		1	d		4	c		a	10

Figuras hechas por mi alumno Rodrigo González.

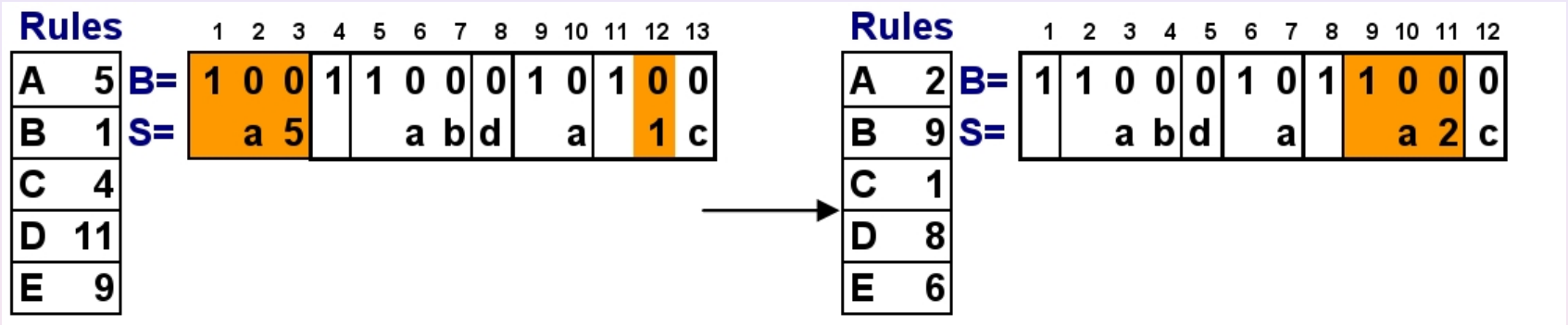
Mejorando Re-Pair





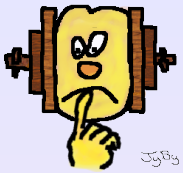
Figuras hechas por mi alumno Rodrigo González.

Mejorando Re-Pair

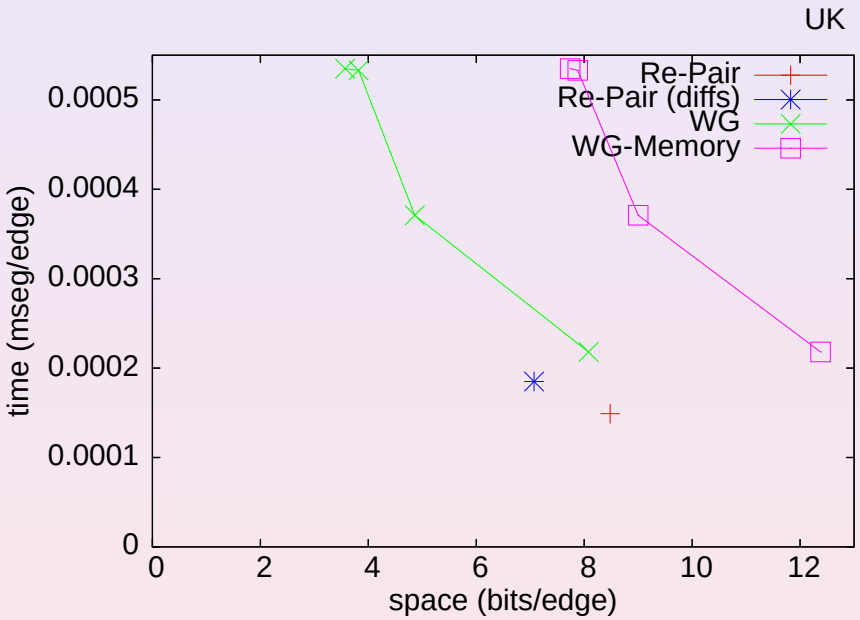


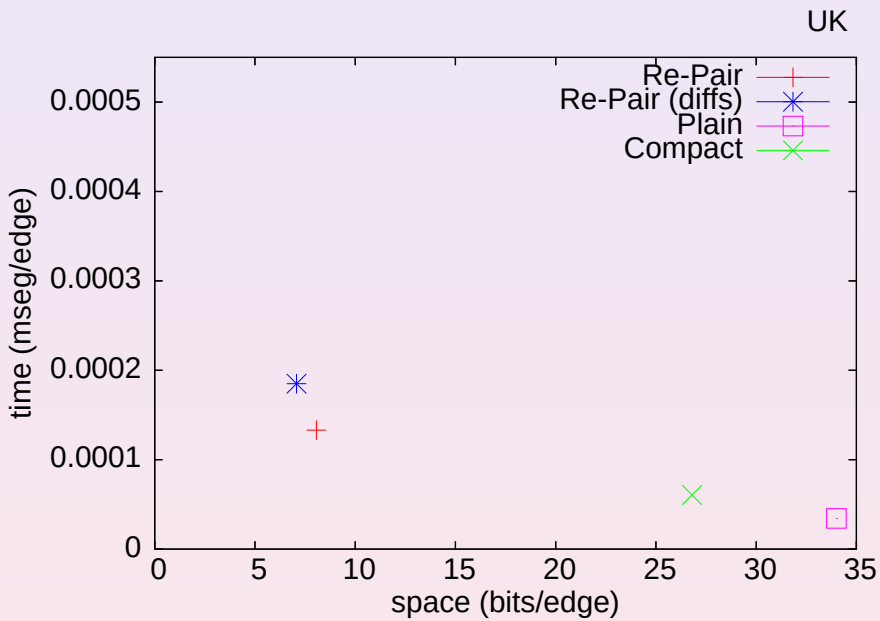
Figuras hechas por mi alumno Rodrigo González.

Resultados Experimentales



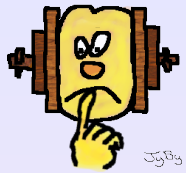
UK: 18.5M nodos, 298M aristas



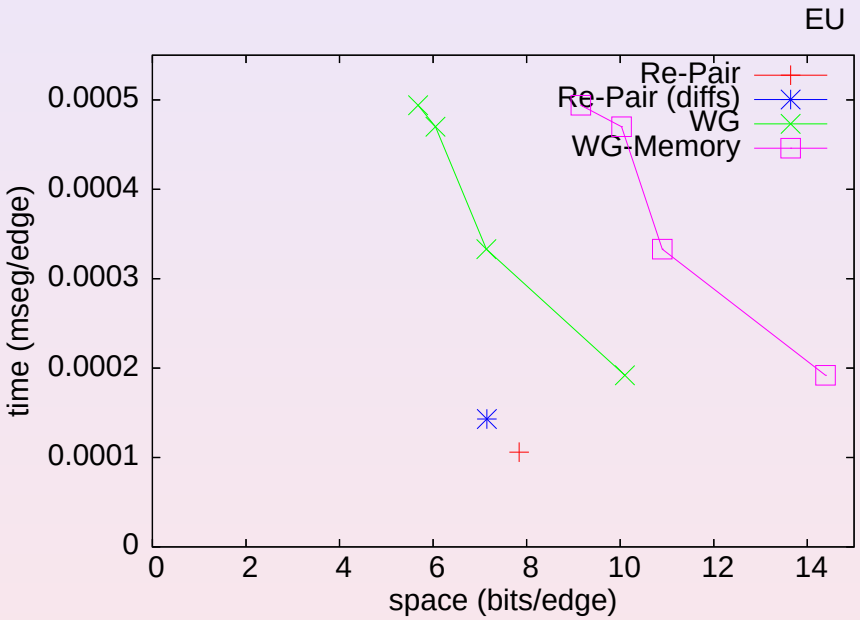


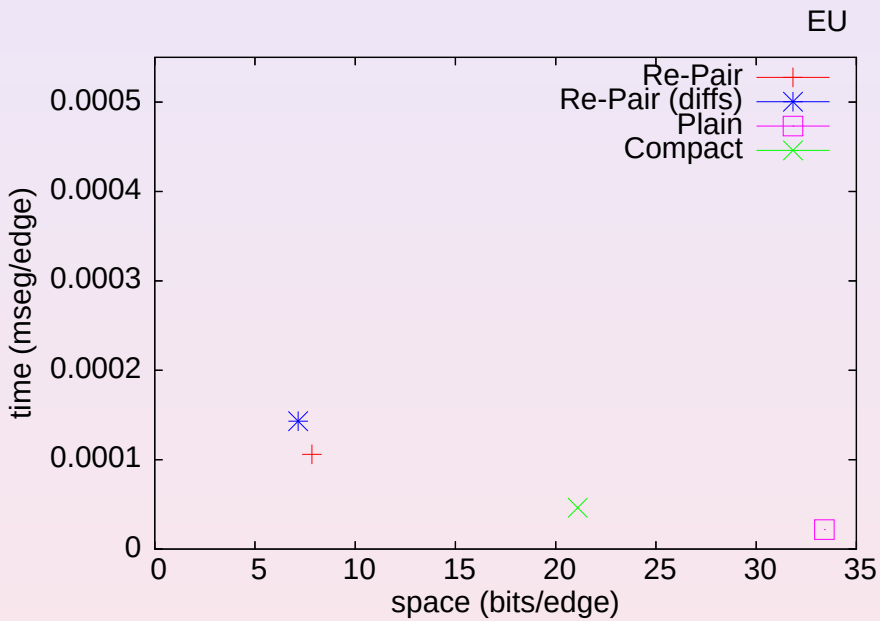
Resultados obtenidos por mi alumno Francisco Claude.

Resultados Experimentales



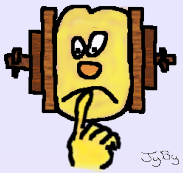
EU: 860K nodos, 19M aristas



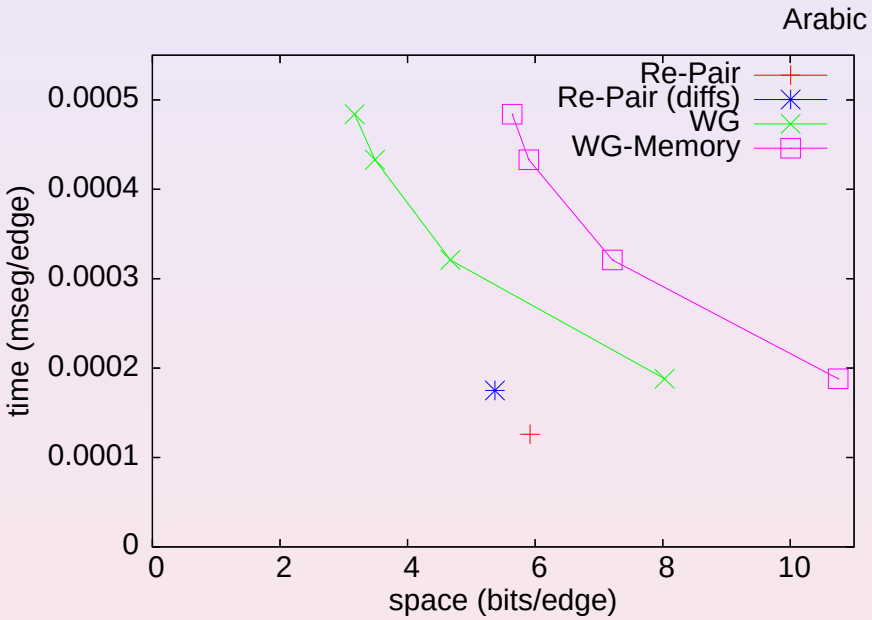


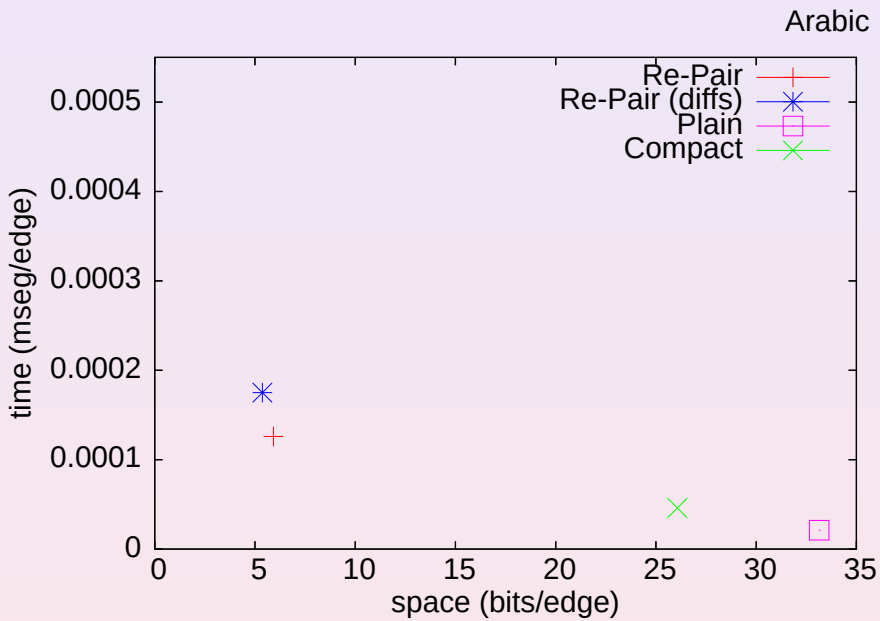
Resultados obtenidos por mi alumno Francisco Claude.

Resultados Experimentales



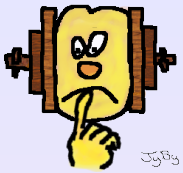
Arabic: 23M nodos, 640M aristas



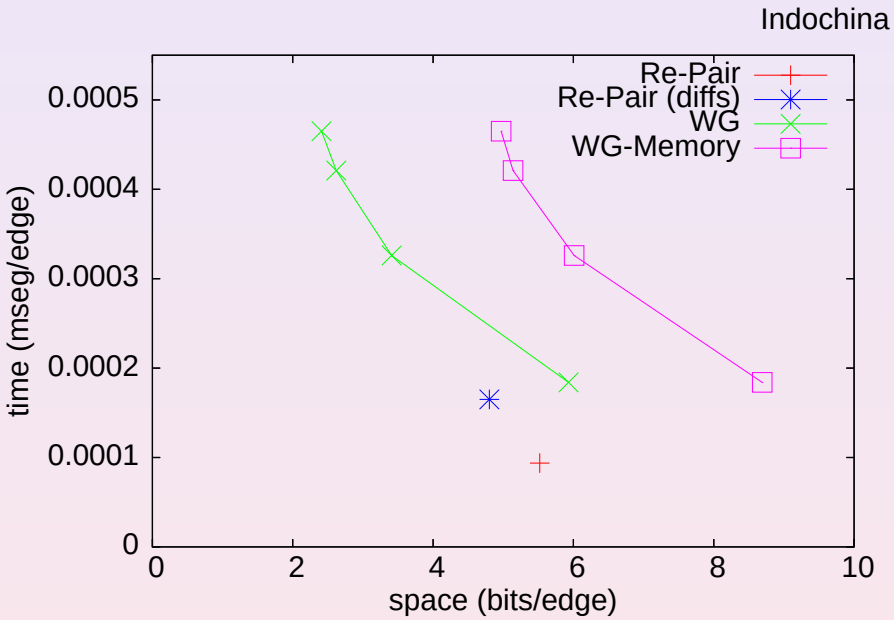


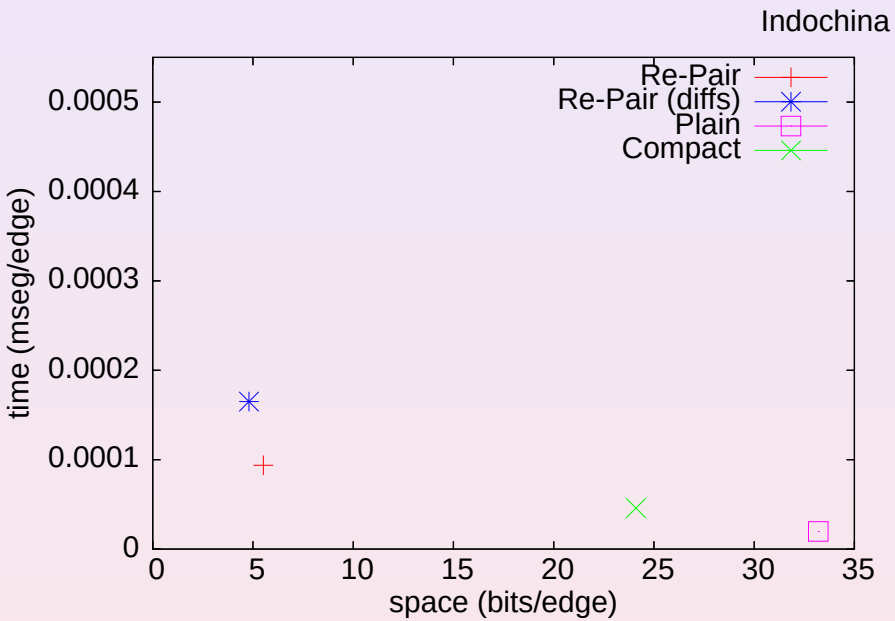
Resultados obtenidos por mi alumno Francisco Claude.

Resultados Experimentales



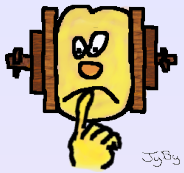
Indochina: 7M nodos, 194M aristas





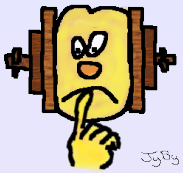
Resultados obtenidos por mi alumno Francisco Claude.

Desafío Actual

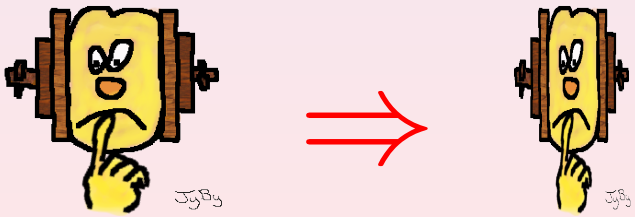


- Re-Pair obtiene 5–7 bpe y mejor compromiso tiempo/espacio que WebGraph.
- Podríamos tener operaciones reversas eficientes con 10–14 bpe.
- Y si en vez tuviéramos la relación binaria sobre Re-Pair?
  - Cada símbolo aparece de muchas formas distintas.
  - Hay que partir por encontrar qué símbolos me representan en el diccionario (*rank* y *select* sobre S!).
  - Vecinos y vecinos reversos: ok ($O(1)$ por vecino).
  - Grado exterior/interior eficientes con 1 bpe extra cada uno. (pero son compresibles a 0.17–0.25 bpe c/u).
  - Existe: Muy caro.
- Tema de investigación abierto.

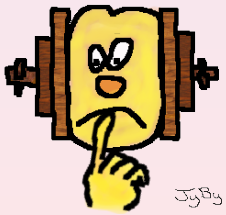
Epílogo



- Las estructuras de datos compactas son un tema de interés práctico para obtener implementaciones eficientes en los computadores modernos.
- Son relevantes por las grandes cantidades de información a manejar y por la jerarquía de memoria.
- Combinan conceptos de algoritmos y estructuras de datos con conceptos de compresión y teoría de la información.
- Son un campo sumamente activo de investigación.
- Y están esperando la contribución de jóvenes entusiastas y con talento!



Copyright



El logo usado en esta presentación es cortesía de Jeremy Barbay, www.cs.uwaterloo.ca/~jbarbay.