Capítulo 2

Computación con recursos limitados

2.1 Problemas e instancias

Para formalizar los conceptos de complejidad, es necesario antes formalizar el concepto de "problema".

> **Problema.** Cuestión que se trata de aclarar; Proposición dudosa. || *Mat.* Proposición dirigida a averiguar el modo de obtener un resultado cuando ciertos datos son conocidos.

Pero hay problemas y problemas, particulares y generales. Los que aquí nos conciernen son problemas genéricos, aquellos que dependen de parámetros, y para los cuales queremos soluciones uniformes, es decir, que sirvan para todos los parámetros de entrada. Así, no nos interesarán problemas del tipo: "encontrar la factorización del número 125.675.441", o "encontrar el grafo dual del grafo de Petersen". Nuestro objeto de estudio serán problemas genéricos del tipo: "encontrar la factorización de un número", "dado un sistema de ecuaciones lineales, determinar si tiene solución". Obsérvese que estos dos últimos problemas no son *un* problema, sino un conjunto infinito de problemas, uno por cada número natural en el primer caso, y uno por cada conjunto finito de números reales en el segundo (los coeficientes de las ecuaciones).

Luego para nosotros un *problema* será una *proposición lógica* junto a ciertos *datos conocidos.* Nos interesará sobre todo ser muy precisos en el *planteamiento* del problema (el cual generalmente identificaremos con el problema mismo). Este planteamiento constará de:

1. *Entrada* (o instancia). El tipo de dato que necesitamos ingresar.
2. *Respuesta* (o salida). El tipo de dato que queremos obtener como respuesta.

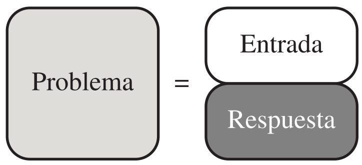

Figura 2.1: Un problema consiste en una entrada y una respuesta

Veamos algunos ejemplos, cuyo plantemiento nos servirá de modelo para el resto de estos apuntes. Partamos por el conocido problema que consiste en ordenar una lista de números:

Problema 1 *(Ordenar una lista)*
ENTRADA. *Una lista finita de números enteros.*

RESPUESTA. *La lista ordenada de menor a mayor.*

El segundo es un conocido problema de aritmética:

Problema 2 *(Adición)*
ENTRADA. *Un par a, b de números naturales.*
RESPUESTA. *Un número natural c tal que $c = a + b$.*

Problema 3 *(Primalidad)*
ENTRADA. *Un número natural.*
RESPUESTA. *Sí o No, dependiendo si el número tiene factores no triviales.*

Nótese que para todos los efectos prácticos, vía una adecuada codificación cada entrada y cada salida, puede codificarse por palabras de un alfabeto Σ_0. Consideremos entonces un alfabeto finito (que incluya el símbolo en blanco $*$) y definamos $\Sigma_0 = \Sigma \setminus \{*\}$. En este sentido, definiremos un *problema* como un conjunto P de pares ordenados (I, A) de palabras en Σ_0^* donde I se llama la *instancia*, A se llama una *respuesta* para esa instancia, y cada palabra de Σ_0^* aparece como primera componente en *al menos* un par. (Esta última condición es necesaria por razones técnicas que veremos más adelante.)

Esta definición es suficientemente general para abarcar la mayoría de las necesidades. Para enfatizar esta generalidad, muchas veces estos problemas son llamados *problemas de búsqueda.* Sin embargo, para estudiar la complejidad, es conveniente restringirse a problemas cuya respuesta (segunda componente del par) es SI o NO (o "1" o "0", según la codificación). Estos problemas son llamados problemas de *decisión.* Un ejemplo es *Primalidad* que vimos más arriba. Así, un *problema de decisión* puede considerarse *un* lenguaje: el conjunto de palabras x para las cuales (x, SI) están en el problema. En este sentido hablaremos indistintamente de *problema* (de decisión) o de *lenguaje.*

Nota 2 *La asimetría en la definición de los* problema de decisión *(el lenguaje está constituído por las palabras x cuya segunda componente es SI; ¿porqué no elegimos NO?) jugará un importante rol en el futuro.*

2.2 Solubilidad algorítmica versus solubilidad práctica

En estas notas asumimos que el lector conoce la diferencia que existe para un problema entre ser *soluble* y *no ser soluble* algorítmicamente, es decir, por medio de computadores. Informalmente, un problema (de decisión) es *soluble algorítmicamente*[1] si existe un método, procedimiento, algoritmo, por el cual pudiéramos responder afirmativa o negativamente a cada una de las instancias del problema. En este sentido, la primera gran sorpresa de la computación fué descubrir que existían problemas que *no* se pueden resolver por computadores. (Para aquellos que quieran refrescar estos conceptos, sugerimos leer el Apéndice). Por supuesto que nosotros nos restringimos a aquellos problemas *solubles* algorítmicamente.

Pero la *solubilidad algorítmica* de un problema puede estar muy lejos de su solubilidad *práctica.* Hay problemas solubles algorítmicamente que no pueden ser resueltos, para una entrada de tamaño dado, en menos de un número exponencial o doble exponencial de pasos. La *teoría complejidad computacional* investiga la solubilidad de problemas particulares bajo ciertas restricciones

[1] Cuando el problema es de decisión, en vez de "soluble algorítmicamente" generalmente se dice *decidible.*

de recursos. Los recursos considerados más importantes son tiempo y espacio (almacenamiento).

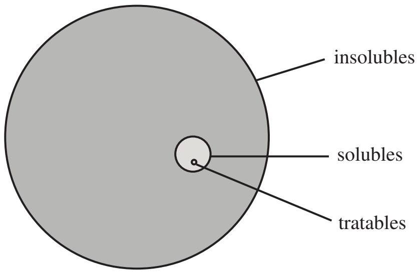

Figura 2.2: Los problemas (a) pueden ser divididos en aquellos que son solubles algorítmicamente (b) y en aquellos que son solubles prácticamente (c).

Definiremos estas nociones en términos del modelo de computación de la máquina de Turing. Esta definición es adecuada para el estudio teórico; para describir algoritmos el modelo RAM es más conveniente y es una mejor aproximación a la realidad. Veremos, sin embargo, que desde el punto de vista de las restricciones de recursos más importantes (por ejemplo tiempo y espacio limitados por una función polinomial) no importa que modelo de máquina es utilizado en la definición.

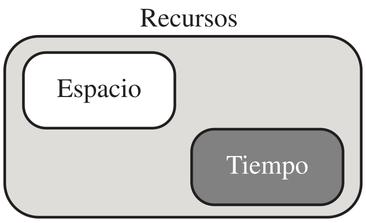

Figura 2.3: Los recursos más importantes son espacio y tiempo

Esto nos lleva a la definición de varias *clases de complejidad*: clases de problemas solubles en determinado tiempo fijo, parametrizado por el largo de la entrada. Toda función positiva que dependa de la entrada define una clase, pero algunas de ellas son particularmente importantes. La clase de complejidad más importante es la de *tiempo polinomial*. Muchos algoritmos importantes en la práctica corren en un tiempo limitado por un una expresión polinomial en el tamaño de la entrada (en breve, "son polinomiales").

Los algoritmos polinomiales son a menudo muy interesantes matemáticamente, debido a que son construidos con una profunda visión de de la estructura matemática de los problemas, y a menudo usan potentes herramientas matemáticas.

Restringiremos las computaciones a problemas de decisión (es decir, con respuesta sí o no). Esta no es una restricción muy limitante y simplifica bastante la representación. Notemos que la tarea de computar cualquier salida puede ser reducida a computar sus bits en alguna representación binaria razonable.

2.3 Tiempo y espacio

En este capítulo, cuando una máquina de Turing es utilizada en alguna computación, asumiremos que tiene una cinta de entrada, una de salida y $k \geq 1$ cintas disponibles. Al inicio, hay una palabra de Σ_0^* escrita en la cinta de entrada.

La *demanda de tiempo* de una máquina de Turing T es una función $\mathrm{time}_T(n)$ definida como el máximo número de pasos que toma T para todas las posibles entradas de largo n. Asumiremos que $\mathrm{time}_T(n) \geq n$, es decir que la máquina debe leer la entrada; esto no tiene por que ser así necesariamente, pero con este supuesto excluímos sólo casos triviales. Podría suceder que

$\mathrm{time}_T(n) = \infty$.

Similarmente, la función $\mathrm{space}_T(n)$ está definida como el número máximo, sobre todas las entradas de largo n, de celdas sobre las que la máquina escribe. De esta manera, las celdas ocupadas por la entrada no se cuentan dentro de los requerimientos de tiempo. Asumiremos que $\mathrm{space}_T(n) \geq 1$, es decir, la máquina debe entregar alguna salida.

Una máquina de Turing será llamada *de tiempo polinomial*, si hay una función polinomial $f(n)$ para la cual $\mathrm{time}_T(n) \in \mathcal{O}(f(n))$. Esto es equivalente a decir que hay una constante c para la cual la demanda de tiempo de T está en $\mathcal{O}(n^c)$. En otras palabras, una máquina de Turing T es polinomial si para toda entrada de tamaño n, entrega la respuesta en un número de pasos no superior a $f(n)$, donde f es algún polinomio. Similarmente, podemos definir una máquina de Turing *exponencial* (en este caso la demanda de tiempo es $\mathcal{O}(2^{n^c})$ para algún $c \geq 0$). Análogamente, también podemos definir máquinas de Turing que trabajan en *espacio* polinomial y en *espacio* exponencial.

Recordemos que una máquina de Turing T *decide* un lenguaje $\mathcal{L} \subseteq \Sigma^*$ si para cada entrada $x \in \Sigma^*$ se detiene y responde 1 si $x \in \mathcal{L}$ y 0 si $x \notin \mathcal{L}$. Ahora estamos en condiciones de definir la complejidad de un problema. Diremos que un lenguaje $\mathcal{L} \subseteq \Sigma_0^*$ tiene *complejidad en tiempo a lo más* $f(n)$, si existe una máquina de Turing con demanda de tiempo a lo más $f(n)$ que lo decide. Esto es, si existe una máquina de Turing que para cada palabra x, responde en tiempo no mayor a $f(|x|)$ con 1 si $x \in \mathcal{L}$ o con 0 si $x \notin \mathcal{L}$.

Gran parte de la teoría de complejidad computacional consiste en "clasificar" los lenguajes (i.e. los problemas) de acuerdo a su complejidad (ya sea en tiempo, espacio, etc.). En este sentido, es conveniente agrupar todos los lenguajes "similares" desde este punto de vista en clases. Denotaremos por DTIME($f(n)$) la clase de los lenguajes que tienen complejidad en tiempo a lo más $f(n)$. (La letra "D" indica que sólo consideramos algoritmos determinísticos; más adelante consideraremos algoritmos que son "no-determinísticos" o que usan aleatoriedad). Denotaremos por PTIME, o simplemente por P, la clase de lenguajes decidibles por una máquina de Turing de tiempo polinomial. Similarmente cuando un lenguaje tiene complejidad en espacio a lo más $f(n)$, diremos que pertenece a la clase DSPACE($f(n)$). Así por ejemplo, PSPACE es la clase de los lenguajes de espacio polinomial.

L	Espacio Logarítmico
P	Tiempo polinomial
PSPACE	Espacio polinomial
EXPTIME	Tiempo exponencial

Tabla 2.1: Algunas clases determinísticas

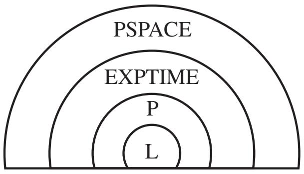

Figura 2.4: Jerarquía de algunas clases de complejidad

Nota 3 *Podría ser tentador definir la complejidad en tiempo de un lenguaje $\mathcal{L}$ como el tiempo óptimo que toma una máquina de Turing T que decide el lenguaje. Note que fuimos más cuidadosos anteriormente y solamente definimos cuando la complejidad en tiempo era a lo más $f(n)$. La razón es que podría no haber un mejor* *algoritmo (máquina de Turing) que resuelva un problema dado: algunos algoritmos funcionarán mejor para instancias pequeñas, otros para largas, u otros para las todavía más largas etc. La sección* **??** *contiene un teorema que nos presenta tales ejemplos.*

Nota 4 *Cuando decimos que la multiplicación entre dos números de largo n puede obtenerse en tiempo n^2, entonces hemos encontrado una cota superior a la complejidad de una* función *(multiplicación de dos números representados por las cadenas de entrada) en lugar de un* lenguaje. *Las clases* DTIME($f(n)$), DSPACE($f(n)$), *etc. están definidas como clases de* lenguajes*; las clases correspondientes para* funciones *también pueden ser definidas.*

A veces es trivial dar cotas inferiores a la complejidad de una función. Consideremos por ejemplo la función $x \times y$ donde x e y son números en notación binaria. Su computación requiere a lo menos $|x| + |y|$ pasos, debido a que este es el largo de la salida. Cotas inferiores sobre la complejidad de lenguajes casi nunca son así de triviales, puesto que la salida de un problema de decisión es un solo bit.

2.3.1 Tiempo versus espacio

En los capítulos que siguen demostraremos teoremas generales respecto a las medidas de complejidad y relaciones entre las clases. Mostraremos que hay lenguajes que requieren una gran cantidad de tiempo para decidirlos. Teoremas análogos pueden probarse para complejidad de espacio. Es natural preguntarse acerca de la relación de estas dos medidas de complejidad.

Hay una variedad de preguntas naturales e interesantes acerca del compromiso entre espacio y tiempo. Mencionemos primero que todos los problemas prácticos bien conocidos que corren en la mayoría de los computadores pueden ser acelerados significativamente agregando memoria. La relación aquí no es realmente entre complejidad de almacenamiento y tiempo de computación, sino sólo entre memoria más lenta y más rápida. Posiblemente, entre "memoria RAM" versus memoria en disco, la que es cercana al acceso en serie del modelo de las máquinas de Turing.

Pero existen algunos ejemplos reales de compromiso espacio-tiempo en la práctica. Por ejemplo, supongamos que durante una computación los valores de una función Booleana pequeña pero compleja sean usados repetidamente; entonces, en una máquina de acceso aleatorio vale la pena calcular estos valores una vez al comienzo para todas los entradas y utilizar posteriormente una tabla de accesos. Similarmente, si cierto campo de nuestros registros en una base de datos es utilizado frecuentemente, entonces vale la pena computar una tabla facilitando este tipo de búsqueda. Estos ejemplos caen en la siguiente categoría. Conocemos algún problema P y un algoritmo A que lo resuelve, y otro algoritmo A' que resuelve P en menos tiempo y mayor espacio que A. Pero, generalmente, no tenemos ninguna demostración de que al usar menos tiempo efectivamente mayor espacio sea necesario para resolver P. Además, cuando se conoce una cota inferior para complejidad tiempo de alguna función, generalmente no tenemos una estimación de la complejidad de espacio más que la trivial mencionada arriba (y vice versa).

Sin embargo, hay resultados generales que se pueden demostrar, como los siguientes. Es trivial que el almacenamiento demandado por una máquina de Turing de k cintas es a lo más k veces su demanda de tiempo (ya que en un paso no se pueden escribir más de k celdas). Por lo tanto, si tenemos un lenguaje $\mathcal{L} \in$ DTIME(f),

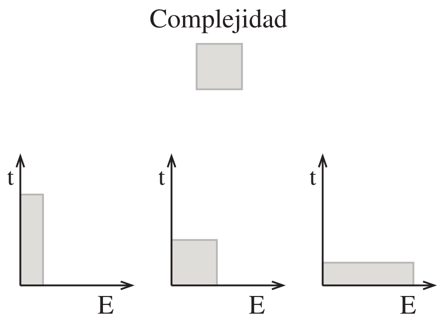

Figura 2.5: Existe un compromiso entre el espacio y el tiempo que requiere un problema

entonces existe una constante c (que depende del lenguaje) tal que $\mathcal{L} \in \mathrm{DSPACE}(cf)$, y luego, si se permite extender el alfabeto y $f(n) < n$ entonces $\mathrm{DSPACE}(cf) = \mathrm{DSPACE}(f)$, y por lo tanto

$$\mathrm{DTIME}(f) \subseteq \mathrm{DSPACE}(f).$$

Por otro lado, el tiempo requerido no es mayor que una función exponencial del espacio requerido por una razón muy simple: la misma configuración, tomando en cuenta también las posiciones de los cabezales y el estado de la unidad de control, no puede ocurrir más de una vez durante una computación sin entrar en un ciclo. De hecho, calculando más precisamente, y suponiendo que la máquina usa no más de $f(n)$ celdas de memoria, el número de configuraciones de memoria diferentes es a lo más $c \cdot f(n)^d m^{f(n)}$, donde m es el tamaño del alfabeto y c, d son constantes. Puesto que como hemos visto, en las cotas de complejidad las constantes no son relevantes, y en esta cota superior los números c, d, m son constantes, se sigue que si $f(n) > \log n$ y $\mathcal{L} \in \mathrm{DSPACE}(f(n))$ entonces $\mathcal{L} \in \mathrm{DTIME}((m+1)^{f(n)})$, esto es para alguna constante c,

$$\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(c^{f(n)}).$$

2.3.2 Medidas para RAM

¿Como definir la demanda de tiempo en una máquina RAM? El número de pasos de la Máquina de Acceso Aleatorio no es la mejor medida de "el tiempo que toma trabajando". Uno podría (mal) usar el hecho de que las instrucciones operan sobre números naturales de tamaño arbitrario, y utilizar trucos computacionales que funcionen en ese modelo computacional, pero que utilicen números tan grandes, que se transformen en computaciones prácticamente imposibles. Por ejemplo, podemos simular suma de vectores sumando dos números naturales muy grandes.

Por lo tanto, preferimos caracterizar el tiempo de trabajo de algoritmos en RAM como dos números, y decimos que "la máquina toma a lo más n pasos con números de a lo más k bits". Similarmente, los requerimientos de espacio se caracterizan diciendo "La máquina almacena a lo más n números de a lo más k bits".

Si quisiéramos un solo número para caracterizar el tiempo de trabajo de un algoritmo RAM, podemos considerar como unidad de tiempo, no cada paso sino que el número de bits que ocurren en cada paso. (tanto registros de direcciones como datos contenidos en ellos). Como el número de bits de un número es esencialmente el logaritmo en base dos de su valor absoluto, usualmente se llama a este modelo *costo RAM logarítmico.*

En algoritmos aritméticos y algebraicos, frecuentemente conviene contar las operaciones aritméticas; en una Máquina de Acceso Aleatorio, esto corresponde a extender el set de operaciones básicas del lenguaje de programación, incluyendo la resta, multiplicación, división (con resto) y comparación de enteros. Y contar el número de operaciones (en términos del largo de la entrada) sobre números con a lo más un polinomial número de dígitos, por lo tanto, nuestro algoritmo sería polinomial en el modelo de costo logarítmico.

2.3.3 Simulación revisitada

Dados dos lenguajes $\mathcal{L}_1$ y $\mathcal{L}_2$, decimos que $\mathcal{L}_1$ se *reduce* a $\mathcal{L}_2$ con complejidad C, simbólicamente $\mathcal{L}_1 \leq \mathcal{L}_2$, si hay una función computable de complejidad C tal que

$$x \in \mathcal{L}_1 \text{ si y sólo si } f(x) \in \mathcal{L}_2.$$

Esta es una versión de reducibilidad "con recursos limitados". Lo que nos interesará en el futuro es reducir ciertos problemas a otros atendiendo a los recursos usados en la reducción. Esto se hace para evitar considerar reducciones que pueden costar más que resolver el problema mismo. En general la clase de complejidad C será mucho menor en la jerarquía de complejidad que la de los lenguajes L_1, L_2.

En general en estos apuntes usaremos reducciones *polinomiales*, i.e. funciones f calculables en tiempo polinomial. Nótese sin embargo, que de esta forma, cualquier par de lenguajes en P son reducibles uno a otro. Si queremos hilar más fino al interior de la clase P, generalmente se usa reducciones en la clase L.

Ejemplo. Consideremos el problema de decidir si es posible colorear un grafo con sólo 2 colores (sin que dos nodos adyacentes tengan el mismo color). Este problema es reducible a 2-SAT de la siguiente forma: asociemos a cada variable Booleana x_i un nodo n_i del grafo, y codifiquemos las condiciones para una correcta coloración del grafo de la siguiente forma. A cada color le asociamos los valores de verdad 1 y 0 respectivamente, y para cada eje (i, j) del grafo, tenemos la fórmula $(\neg x_i \vee \neg x_j) \wedge (x_i \vee x_j)$, que es satisfacible si y sólo si los nodos vecinos n_i y n_j tienen colores distintos. Luego haciendo la conjunción de todas estas fórmulas tenemos una fórmula de entrada para 2-SAT que es satisfacible si y sólo si hay una coloración con dos colores del grafo en cuestión.

¿Cuánto tiempo toma el proceso de reducción? No más que polinomial. Luego 2-COLOR $\leq$ 2-SAT en tiempo polinomial.

2.4 Consideraciones técnicas sobre las medidas

Por que la notación asintótica. El lector habrá notado que desde los comienzos, hemos preferido hablar de cotas *asintóticas* (notación $\mathcal{O}$, etc.) en vez de cotas precisas. Esto tiene sus razones. De hecho, no parece posible hacer una teoría elegante si consideramos las constantes multiplicativas y aditivas cuando hablamos de los recursos de tiempo y espacio. Veremos también que *no cualquier función* sirve como cota.

El siguiente teorema muestra porque las constantes no jugarán rol en nuestra teoría.

Teorema 7 (Aceleramiento lineal) *Para toda máquina de Turing T y $c > 0$ existe una máquina de Turing S sobre el mismo alfabeto que decide el mismo lenguaje y para la cual* $\text{time}_S(n) \leqslant c \cdot \text{time}_T(n) + n$.

DEMOSTRACIÓN. Por simplicidad, asumamos que T tiene sólo una cinta de trabajo (la demostración sería similar para k cintas) y que $c = 1/p$, donde p es un entero.

Sea S la máquina de Turing con una cinta de entrada, $2p - 1$ cintas de "partida" y $2p - 1$ cintas de trabajo. Numeremos estas cintas desde $1-p$ hasta $p-1$. Definamos el *índice* de la celda j de la cinta (de partida o de trabajo) i como el número $j(2p-1)+i$. La celda de partida (resp. de trabajo) con índice t va a corresponder a la celda t en la cinta de entrada (resp. de trabajo) de la máquina T. La máquina S tendrá además una cinta de salida.

La máquina S comienza su trabajo copiando cada letra de la entrada x desde su cinta de entrada a la celda con el índice correspondiente en sus cintas de partida, luego mueve cada cabezal de vuelta a la celda 0. De ahí en adelante ignora la cinta de entrada "real".

Cada paso sucesivo de la máquina S va a corresponder a p pasos consecutivos en la máquina T. Luego de pk pasos de la máquina T, los cabezales de la cinta de entrada y la cinta de trabajo estarán en las celdas t y s respectivamente. Vamos a planificar la máquina S de manera tal que en este caso, cada celda de cada cinta de partida (resp. de trabajo) de S mantenga el mismo símbolo que la correspondiente celda de la correspondiente cinta de T, y los cabezales descansan en las celdas de la cinta de partida con índices $t-p+1,\ldots,t+p-1$ y las celdas de la cinta de trabajo con índices $s-p+1,\ldots,s+p-1$. Asumimos que la unidad de control de la máquina S "sabe" también qué cabezal explora la celda corresponde a t (resp s). Conoce además cual es el estado de la unidad de control de T.

Debido a que la unidad de control de S ve no sólo lo que es leído por la unidad de control de T en el momento actual en su cintas de entrada y de trabajo, sino que además las celdas a una distancia a lo más $p-1$ de éstas, puede computar hacia dónde se moverán y qué escribirán los cabezales de T en los siguientes p pasos. Luego de p pasos los cabezales de T estarán en las posiciones, digamos $t+i$ y $s+j$ (donde $i,j>0$). Obviamente, $i,j<p$. Notemos que mientras tanto el cabezal de "trabajo" puede cambiar los símbolos escritos en la cinta de trabajo sólo en el intervalo $[s-p+1,s+p-1]$.

Hagamos ahora que la unidad de control de S realice lo siguiente: que compute y recuerde cuál será el estado de la unidad de control de T p pasos después. Que recuerde qué cabezales descansan en las celdas correspondientes a las posiciones $(t{+}i)$ y $(s{+}j)$. Que reescriba los símbolos en la cinta de trabajo de acuerdo a la configuración p pasos después (esto es posible debido a que existe un cabezal en cada celda de trabajo con índices en el intervalo $[s-p+1,s+p-1]$). Finalmente, que mueva el cabezal de partida con índices en el intervalo $[t-p+1,t-p+i]$ y los cabezales de trabajo con índices en el intervalo $[s-p+1,s-p+j]$ un paso a la derecha; de esta manera, los índices utilizados por ellos llenarán el intervalo $[t+p,t+p+i-1]$ (resp. $[s+p,s+p+i-1]$), los cuales, junto con los cabezales que permanecieron en su lugar, entregan el intervalo $[t+i-p,t+i+p-1]$ (resp. $[s+j-p+1,s+j+p-1]$).

Si durante los p pasos considerados T escribe en la cinta de salida (ya sea 0 ó 1) y se detiene, entonces hagamos que S también lo haga. Asi entonces construimos una máquina de Turing S que (aparte del copiado inicial) realiza sólo p veces el número de pasos de T y decide el mismo lenguaje. ⊙

Es posible probar un teorema similar para medidas de espacio.

No cualquier función sirve de cota. La teoría que hemos venido delineando se comporta bien para las funciones usuales: logaritmos, polinomios, exponenciales, combinaciones de ellas. Desgraciadamente hay funciones anómalas que pueden destruir todo el edificio. La buena noticia es que son funciones muy raras.

Llamamos a una función $f: Z_+ \to Z_+$ *constructible completamente en tiempo* si existe una máquina de Turing de múltiples cintas tal que para cada entrada de largo n se detiene usando exactamente $f(n)$ pasos. El significado de esta extraña definición es que con funciones constructibles completamente en tiempo es fácil acotar el tiempo de ejecución de máquina de Turing: si existe una máquina de Turing que realiza exactamente $f(n)$ pasos en cada entrada de largo n, entonces podemos incluir ésta en cualquier otra máquina de Turing como un reloj: sus cintas, excepto las cintas de trabajo, son distintas, y la máquina de Turing combinada entrega en cada paso el trabajo de ambas cintas.

Obviamente, toda función constructible completamente en tiempo es recursiva. Por otro lado, es fácil ver que $n^2, 2^n, n!$ y toda función "razonable" es constructible completamente en tiempo. El lema que sigue garantiza la existencia de muchas funciones completamente constructibles en tiempo.

Llamemos una función $f: Z_+ \to Z_+$ *buena* si existe una máquina de Turing que compute $f(n)$ en tiempo $\mathcal{O}(f(n))$. (Aquí escribimos n y $f(n)$ en notacion unaria: el número n es dado por una secuencia $1\cdots1$ de largo n y queremos como salida una secuencia $1\cdots1$ de largo $f(n)$. El resultado no variaría, sin embargo, si n y $f(n)$ fuesen representadas en notación binaria).

Ahora el siguiente lema es fácil de demostrar:

Lema 2
1. *Para toda función buena $f(n)$ existe una función completamente constructible en tiempo $g(n)$ tal que $f(n) \le g(n) \le cf(n)$ para alguna constante c.*
2. *Para toda función completamente constructible en tiempo $g(n)$ existe una función buena $f(n)$ con $g(n) \le f(n) \le cg(n)$ para alguna constante c.*
3. *Para toda función recursiva $f(n)$ existe una función completamente constructible en tiempo $g(n)$ con $f(n) \le g(n)$.*

Este lema nos permite utilizar indistintamente, en la mayoría de los casos, funciones completamente constructibles en tiempo y funciones buenas.

Teorema 8 *Si $f(n)$ es completamente constructible en tiempo y $g(n) \log f(n) = o(f(n))$, entonces existe un lenguaje en* DTIME(f) *que no pertenece a* DTIME(g).

Este resultado dice que las complejidades en tiempo de los lenguajes recursivos son "suficientemente densas". Análogamente, pero más simple, el resultado se mantiene para complejidades de espacio. La constructibilidad completamente en tiempo de la función f juega un rol muy importante en el último teorema. Si no la consideramos, es posible encontrar funciones tales que $\mathrm{DTIME}(2^{2^{f(n)}}) = \mathrm{DTIME}(f(n))$. Su construcción está dada por el llamado teorema de brecha cuyo enunciado es el siguiente:

Teorema 9 (de la Brecha) *Para toda función recursiva $\varphi(n) \geq n$ existe una función recursiva $f(n)$ tal que:*

$$\mathrm{DTIME}(\varphi \circ f) = \mathrm{DTIME}(f).$$

Luego, existe una función recursiva f con $\mathrm{DTIME}(f^2) = \mathrm{DTIME}(f)$ *e incluso existe una tal que* $\mathrm{DTIME}(2^{2^{f(n)}}) = \mathrm{DTIME}(f(n))$.

Otro resultado interesante, el llamado *teorema de aceleramiento*, muestra que existen problemas (lenguajes) arbitrariamente "acelerables", o de otra forma, que para un problema dado, generalmente no hay un "mejor" algoritmo:

Teorema 10 (de aceleramiento) *Para toda función recursiva $g(n)$ existe un lenguaje recursivo $\mathcal{L}$ con la siguiente propiedad: para toda máquina de Turing T que decide $\mathcal{L}$ existe una máquina de Turing S que decide $\mathcal{L}$ con $g(\mathrm{time}_S(n)) < \mathrm{time}_T(n)$.*

El teorema de aceleramiento lineal se aplica a todo lenguaje; este teorema expresa sólo la existencia de un lenguaje arbitrariamente "acelerable". En general, para un lenguaje arbitrario, no puede esperarse algo mejor que aceleramiento lineal. La esencia de la demostración del teorema es que permitimos máquinas más complicadas para almacenar más información en la unidad de control. Luego, la máquina necesita trabajar sólo con entradas mas largas que "su propio mérito"

La conclusión más importante que se desprende del teorema de aceleramiento es que que aun cuando es conveniente hablar de complejidad computacional de un cierto lenguaje $\mathcal{L}$, las afirmaciones rigurosas referentes a la complejidad generalmente no se refieren a una única función $t(n)$, sino sólo dan *cotas superiores* $t'(n)$ (a través de la construcción de una máquina de Turing que decida el lenguage en tiempo $t'(n)$)) o *cotas inferiores* $t''(n)$ (mostrando que ninguna máquina de Turing puede resolver el problema en tiempo $t''(n)$ para todo n).

2.5 Complejidad Intrínseca

Resumen de complejidad descriptiva.

2.6 Ejercicios

1. Ordenar en una lista $\mathcal{O}$-creciente la siguiente lista de funciones (para $n \geq 1$): n^2, n^3, $n^2 \lg n$, $5n^2 + 100$, 2^n, $n^{\lg n}$, $n!$, 2^{2^n}, n^n, n^2 si n es par, 2^n de lo contrario.

2. Para toda máquina de Turing T y $c < 0$, se puede encontrar una máquina de Turing S con el mismo número de cintas que decide el mismo lenguaje y para la cual $\mathrm{time}_S(n) \leqslant c \cdot \mathrm{time}_T(n) + n$ (aquí permitimos la extensión del alfabeto).

3. Formule y demuestre el análogo al problema anterior para espacio en vez de tiempo.

4. Probar el Teorema 8 y la siguiente sentencia relacionada. Sea $t\prime(n) \log t\prime(n) = o(t(n))$. Entonces el problema de la parada con cota t no puede decidirse en una máquina de Turing de dos cintas en tiempo $t\prime(n)$.

5. Muestre que si $s(n)$ es cualquier función y $s\prime(n) = o(s(n))$ entonces el problema de la parada para espacio con cota $s(n)$ no puede ser resuelto en tiempo $s\prime(n)$.