Análisis de la eficiencia de un algoritmo

IIC2283

Decisión versus computación

En el capítulo anterior definimos Máquinas de Turing que aceptan lenguajes.

- Una MT M con alfabeto de entrada Σ define el lenguaje $L(M) = \{w \in \Sigma^* \mid M \text{ acepta } w\}$

Este tipo de Máquinas de Turing sirven para decidir la pertenencia de un elemento a un lenguaje.

- Hablamos entonces de problemas de decisión

Pero también nos interesa computar valores.

- Ordenar una lista, multiplicar dos números, ...

Decisión versus computación

En un problema de computación queremos un algoritmo para calcular una función $f : \Sigma^* \to \Sigma^*$

- ¿Cómo puede ser representado el problema de ordenar una lista de enteros como el problema de calcular una función? ¿Qué alfabeto usaría?

Podemos modificar las Máquinas de Turing para que calculen funciones.

Ejercicio

Indique cómo hacer esta modificación para que una MT M calcule una función $f : \Sigma^* \to \Sigma^*$

- Sólo se debe llegar a una convención de como representar la salida, para lo cual puede ser útil tener más de una cinta.
- ¿Son útiles los estados finales en este caso?

Unificando los modelos

Vamos a pensar en una algoritmo $\mathcal{A}$ como una función $\mathcal{A} : \Sigma^* \to \Sigma^*$

Esta es una representación general que incluye tanto a problemas de decisión como a los de computación.

- ¿Cómo se representa un problema de decisión?

Pero hay que tener cuidado porque el modelo es demasiado general.

- Hay funciones $\mathcal{A} : \Sigma^* \to \Sigma^*$ que no pueden ser calculadas por una MT. ¿Por qué?

La noción de algoritmo en este curso

Supuesto

Vamos a consider funciones $\mathcal{A} : \Sigma^* \to \Sigma^*$ para las cuales existen Máquinas de Turing que las calculan.

Pero no estamos interesados en construir estas Máquinas de Turing, vamos a calcular estas funciones en otros modelos de computación.

Tiempo de ejecución de un algoritmo

A cada algoritmo $\mathcal{A}$ asociamos una función $\text{tiempo}_{\mathcal{A}} : \Sigma^* \to \mathbb{N}$ tal que:

$\text{tiempo}_{\mathcal{A}}(w)$: número de pasos realizados por $\mathcal{A}$ con entrada $w \in \Sigma^*$

Para definir esta función tenemos que definir qué operaciones vamos a contar, y qué costo les asignamos.

Tiempo de ejecución de un algoritmo en el peor caso

El primer tipo de análisis que vamos a realizar de la complejidad de un algoritmo va a estar basado en el peor caso.

Para cada algoritmo $\mathcal{A}$ asociamos una función $t_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ tal que

$$t_{\mathcal{A}}(n) \quad = \quad \text{máx}\{\text{tiempo}_{\mathcal{A}}(w) \mid w \in \Sigma^* \text{ y } |w| = n\},$$

donde $|w|$ es el largo de w

Notación asintótica

En muchos casos, nos interesa conocer el *orden* de un algoritmo en lugar de su complejidad exacta.

- Queremos decir que un algoritmo es lineal o cuadrático, en lugar de decir que su complejidad es $3n^2 + 17n + 22$

Vamos a desarrollar notación para hablar del orden de un algoritmo.

Notación asintótica

Supuesto

La complejidad de un algoritmo va a ser medida en términos de funciones de la forma $f : \mathbb{N} \to \mathbb{R}_0^+$, donde $\mathbb{R}^+ = \{r \in \mathbb{R} \mid r > 0\}$ y $\mathbb{R}_0^+ = \mathbb{R}^+ \cup \{0\}$

Estas funciones incluyen a las funciones definidas en las transparencias anteriores, y también sirven para modelar el tiempo de ejecución de un algoritmo

La notación $O(f)$

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$

Definición

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}_0^+ \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}) \\ (\forall n \geq n_0)\,(g(n) \leq c \cdot f(n))\}$$

Decimos entonces que $g \in O(f)$

- También usamos la notación g es $O(f)$, lo cual es formalizado como $g \in O(f)$

Ejercicio

Demuestre que $3n^2 + 17n + 22 \in O(n^2)$

Las notaciones $\Omega(f)$ y $\Theta(f)$

Definición

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}_0^+ \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}) \\ (\forall n \geq n_0)\,(c \cdot f(n) \leq g(n))\}$$

$$\Theta(f) = O(f) \cap \Omega(f)$$

Ejercicios

1. Demuestre que $3n^2 + 17n + 22 \in \Theta(n^2)$
2. Demuestre que $g \in \Theta(f)$ si y sólo si existen $c, d \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tal que para todo $n \geq n_0$: $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$

Ejercicios

1. Sea $p(n)$ un polinomio de grado $k \geq 0$ con coeficientes en los números enteros. Demuestre que $p(n) \in O(n^k)$.

2. ¿Cuáles de las siguientes afirmaciones son ciertas?
   - $n^2 \in O(n)$
   - Si $f(n) \in O(n)$, entonces $f(n)^2 \in O(n^2)$
   - Si $f(n) \in O(n)$, entonces $2^{f(n)} \in O(2^n)$

3. Suponga que $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existe y es igual a ℓ. Demuestre lo siguiente:
   - Si $\ell = 0$, entonces $f \in O(g)$ y $g \notin O(f)$
   - Si $\ell = \infty$, entonces $g \in O(f)$ y $f \notin O(g)$
   - Si $\ell \in \mathbb{R}^+$, entonces $f \in \Theta(g)$

4. Encuentre funciones f y g tales que $f \in \Theta(g)$ y $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no existe

Búsqueda binaria

Suponga que tiene una lista ordenada (de menor a mayor) $L[1 \ldots n]$ de números enteros con $n \geq 1$

¿Cómo podemos verificar si un número a está en L?

Búsqueda binaria

BúsquedaBinaria(a, L, i, j)
    if $i > j$ **then return** no
    else if $i = j$ **then**
        if $L[i] = a$ **then return** i
        else return no
    else
        $p := \lfloor \frac{i+j}{2} \rfloor$
        if $L[p] < a$ **then return BúsquedaBinaria**(a, L, $p+1$, j)
        else if $L[p] > a$ **then return BúsquedaBinaria**(a, L, i, $p-1$)
        else return p

Llamada inicial al algoritmo: **BúsquedaBinaria**(a, L, 1, n)

Tiempo de ejecución de búsqueda binaria

¿Cuál es la complejidad del algoritmo?

- ¿Qué operaciones vamos a considerar?
- ¿Cuál es el peor caso?

Si contamos sólo las comparaciones, entonces la siguiente expresión define la complejidad del algoritmo:

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + d & n > 1 \end{cases}$$

donde $c \in \mathbb{N}$ y $d \in \mathbb{N}$ son constantes tales que $c \geq 1$ y $d \geq 1$

Esta es una ecuación de recurrencia

Solucionando una ecuación de recurrencia

¿Cómo podemos solucionar una ecuación de recurrencia?

- Técnica básica: sustitución de variables

Para la ecuación anterior usamos la sustitución $n = 2^k$

- Vamos a resolver la ecuación suponiendo que n es una potencia de 2
- Vamos a utilizar inducción para demostrar que la solución obtenida nos da el orden del algoritmo

Ecuaciones de recurrencia: sustitución de variables

Si realizamos la sustitución $n = 2^k$ en la ecuación:

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + d & n > 1 \end{cases}$$

obtenemos:

$$T(2^k) = \begin{cases} c & k = 0 \\ T(2^{k-1}) + d & k > 0 \end{cases}$$

Ecuaciones de recurrencia: sustitución de variables

Extendiendo la expresión anterior obtenemos:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + d \\ &= (T(2^{k-2}) + d) + d \\ &= T(2^{k-2}) + 2d \\ &= (T(2^{k-3}) + d) + 2d \\ &= T(2^{k-3}) + 3d \\ &= \dots \end{aligned}$$

Deducimos la expresión general para $k - i \geq 0$:

$$T(2^k) = T(2^{k-i}) + i \cdot d$$

Ecuaciones de recurrencia: sustitución de variables

Considerando $i = k$ obtenemos:

$$\begin{aligned} T(2^k) &= T(1) + k \cdot d \\ &= c + k \cdot d \end{aligned}$$

Dado que $k = \log_2(n)$, obtenemos que $T(n) = c + d \cdot \log_2(n)$ para n potencia de 2

Usando inducción vamos a extender esta solución y vamos a demostrar que $T(n) \in O(\log_2(n))$

Inducción constructiva

Sea $T(n)$ definida como:

$$T(n) = \begin{cases} c & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + d & n > 1 \end{cases}$$

Queremos demostrar que $T(n) \in O(\log_2(n))$

- Vale decir, queremos demostrar que existen $e \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $T(n) \leq e \cdot \log_2(n)$ para todo $n \geq n_0$

Inducción nos va servir tanto para demostrar la propiedad y como para determinar valores adecuados para e y n_0

- Por esto usamos el término inducción constructiva

Inducción constructiva

Dado que $T(1) = c$ y $\log_2(1) = 0$ no es posible encontrar una valor para e tal que $T(1) \leq e \cdot \log_2(1)$

Dado que $T(2) = (c + d)$, si consideramos $e = (c + d)$ tenemos que $T(2) \leq e \cdot \log_2(2)$

- Definimos entonces $e = (c + d)$ y $n_0 = 2$

Tenemos entonces que demostrar lo siguiente:

$$(\forall n \geq 2)(\, T(n) \leq e \cdot \log_2(n))$$

Inducción constructiva y fuerte

¿Cuál es el principio de inducción adecuado para el problema anterior?

- Tenemos n_0 como punto de partida
- n_0 es un caso base, pero podemos tener otros
- Dado $n > n_0$ tal que n no es un caso base, suponemos que la propiedad se cumple para todo $k \in \{n_0, \ldots, n-1\}$

Inducción constructiva y fuerte

Queremos demostrar que $(\forall n \geq 2)\,(T(n) \leq e \cdot \log_2(n))$

- 2 es el punto de partida y el primer caso base
- También 3 es un caso base ya que $T(3) = T(1) + d$ y para $T(1)$ no se cumple la propiedad
- Para $n \geq 4$ tenemos que $T(n) = T(\lfloor \frac{n}{2} \rfloor) + d$ y $\lfloor \frac{n}{2} \rfloor \geq 2$, por lo que resolvemos este caso de manera inductiva
  - Suponemos que la propiedad se cumple para todo $k \in \{2, \ldots, n-1\}$

La demostración por inducción fuerte

Casos base:

$$\begin{array}{lllll} T(2) & = & c+d & = & e \cdot \log_2(2) \\ T(3) & = & c+d & < & e \cdot \log_2(3) \end{array}$$

Caso inductivo:

Suponemos que $n \geq 4$ y para todo $k \in \{2, \ldots, n-1\}$ se tiene que $T(k) \leq e \cdot \log_2(k)$

La demostración por inducción fuerte

Usando la definición de $T(n)$ y la hipótesis de inducción concluimos que:

$$\begin{aligned}
T(n) &= T(\lfloor \frac{n}{2} \rfloor) + d \\
&\leq e \cdot \log_2(\lfloor \frac{n}{2} \rfloor) + d \\
&\leq e \cdot \log_2(\frac{n}{2}) + d \\
&= e \cdot \log_2(n) - e \cdot \log_2(2) + d \\
&= e \cdot \log_2(n) - (c + d) + d \\
&= e \cdot \log_2(n) - c \\
&< e \cdot \log_2(n)
\end{aligned}$$

Un segundo ejemplo de inducción constructiva

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 0 & n = 0 \\ n^2 + n \cdot T(n-1) & n > 0 \end{cases}$$

Queremos determinar una función $f(n)$ para la cual se tiene que $T(n) \in O(f(n))$

- ¿Alguna conjetura sobre quién podría ser $f(n)$?

Una posible solución para la ecuación de recurrencia

Dada la forma de la ecuación de recurrencia, podríamos intentar primero con $f(n) = n!$

Tenemos entonces que determinar $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $T(n) \leq c \cdot n!$ para todo $n \geq n_0$

- Pero nos vamos a encontrar con un problema al tratar de usar la hipótesis de inducción

Una posible solución para la ecuación de recurrencia

Supongamos que la propiedad se cumple para n:

$$T(n) \leq c \cdot n!$$

Tenemos que:

$$\begin{aligned} T(n+1) &= (n+1)^2 + (n+1) \cdot T(n) \\ &\leq (n+1)^2 + (n+1) \cdot (c \cdot n!) \\ &= (n+1)^2 + c \cdot (n+1)! \end{aligned}$$

Pero no existe una constante c para la cual $(n+1)^2 + c \cdot (n+1)! \leq c \cdot (n+1)!$

- Dado que $n \in \mathbb{N}$

¿Cómo solucionamos el problema con la demostración?

Una demostración por inducción puede hacerse más simple considerando una propiedad más fuerte.

- Dado que la hipótesis de inducción se va a volver más fuerte

Vamos a seguir tratando de demostrar que $T(n) \in O(n!)$ pero ahora considerando una propiedad más fuerte.

Vamos a demostrar lo siguiente:

$$(\exists c \in \mathbb{R}^+)(\exists d \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(T(n) \leq c \cdot n! - d \cdot n)$$

Inducción constructiva sobre una propiedad más fuerte

Para tener una mejor idea de los posible valores para c, d y n_0 vamos a considerar primero el paso inductivo en la demostración.

Supongamos que la propiedad se cumple para n:

$$T(n) \leq c \cdot n! - d \cdot n$$

Tenemos que:

$$\begin{aligned} T(n+1) &= (n+1)^2 + (n+1) \cdot T(n) \\ &\leq (n+1)^2 + (n+1) \cdot (c \cdot n! - d \cdot n) \\ &= c \cdot (n+1)! + (n+1)^2 - d \cdot n \cdot (n+1) \\ &= c \cdot (n+1)! + ((n+1) - d \cdot n) \cdot (n+1) \end{aligned}$$

Inducción constructiva sobre una propiedad más fuerte

Para poder demostrar que la propiedad se cumple para $n+1$ necesitamos que lo siguiente sea cierto:

$$(n+1) - d \cdot n \quad \leq \quad -d$$

De lo cual concluimos la siguiente restricción para d:

$$\frac{(n+1)}{(n-1)} \quad \leq \quad d$$

Si consideramos $n \geq 2$ concluimos que $d \geq 3$

- Consideramos entonces $n_0 = 2$ y $d = 3$

Inducción constructiva sobre una propiedad más fuerte

Para concluir la demostración debemos considerar el caso base $n_0 = 2$

Tenemos que:

$$\begin{array}{lcll} T(0) & = & 0 & \\ T(1) & = & 1^2 + 1 \cdot T(0) & = & 1 \\ T(2) & = & 2^2 + 2 \cdot T(1) & = & 6 \end{array}$$

Entonces se debe cumplir que $T(2) \leq c \cdot 2! - 3 \cdot 2$, vale decir,

$$6 \quad \leq \quad c \cdot 2 - 6$$

Concluimos que $c \geq 6$, por lo que consideramos $c = 6$

- Tenemos entonces que $(\forall n \geq 2)(T(n) \leq 6 \cdot n! - 3 \cdot n)$, de lo cual concluimos que $T(n) \in O(n!)$

El Teorema Maestro

Muchas de las ecuaciones de recurrencia que vamos a usar en este curso tienen la siguiente forma:

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n) & n \geq 1 \end{cases}$$

donde a, b y c son constantes, y $f(n)$ es una función arbitraria.

El Teorema Maestro nos dice cuál es el orden de $T(n)$ dependiendo de ciertas condiciones sobre a, b y $f(n)$

El Teorema Maestro

El Teorema Maestro también se puede utilizar cuando $\lfloor \frac{n}{b} \rfloor$ es reemplazado por $\lceil \frac{n}{b} \rceil$

Antes de dar el enunciado del Teorema Maestro necesitamos definir una condición de regularidad sobre la función $f(n)$

Una condición de regularidad sobre funciones

Dado: función $f : \mathbb{N} \to \mathbb{R}_0^+$ y constantes $a, b \in \mathbb{R}$ tales que $a \geq 1$ y $b > 1$

Definición

f es (a, b)-regular si existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$(\forall n \geq n_0)(a \cdot f(\lfloor \frac{n}{b} \rfloor) \leq c \cdot f(n))$$

Ejercicio

1. Demuestre que las funciones n, n^2 y 2^n son (a, b)-regulares si $a < b$.
2. Demuestre que la función $\log_2(n)$ no es (1,2)-regular.

Una solución al segundo problema

Por contradicción, supongamos que $\log_2(n)$ es (1,2)-regular.

Entonces existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$(\forall n \geq n_0)(\log_2(\lfloor \frac{n}{2} \rfloor) \leq c \cdot \log_2(n))$$

De esto concluimos que:

$$(\forall k \geq n_0)(\log_2(\lfloor \frac{2 \cdot k}{2} \rfloor) \leq c \cdot \log_2(2 \cdot k))$$

Una solución al segundo problema

Vale decir:

$$(\forall k \geq n_0)(\log_2(k) \leq c \cdot (\log_2(k) + 1))$$

Dado que $0 < c < 1$, concluimos que:

$$(\forall k \geq n_0)(\log_2(k) \leq \frac{c}{1-c})$$

Lo cual nos lleva a una contradicción. □

El enunciado del Teorema Maestro

Teorema

Sea $f : \mathbb{N} \to \mathbb{R}_0^+$, $a, b, c \in \mathbb{R}_0^+$ *tales que* $a \geq 1$ *y* $b > 1$, *y* $T(n)$ *una función definida por la siguiente ecuación de recurrencia:*

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T(\lfloor \frac{n}{b} \rfloor) + f(n) & n \geq 1 \end{cases}$$

Se tiene que:

1. *Si* $f(n) \in O(n^{\log_b(a)-\varepsilon})$ *para* $\varepsilon > 0$, *entonces* $T(n) \in \Theta(n^{\log_b(a)})$
2. *Si* $f(n) \in \Theta(n^{\log_b(a)})$, *entonces* $T(n) \in \Theta(n^{\log_b(a)} \cdot \log_2(n))$
3. *Si* $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ *para* $\varepsilon > 0$ *y* f *es* (a, b)*-regular, entonces* $T(n) \in \Theta(f(n))$

Usando el Teorema Maestro

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 0 \\ 3 \cdot T(\lfloor \frac{n}{2} \rfloor) + c \cdot n & n \geq 1 \end{cases}$$

Dado que $\log_2(3) > 1.5$, tenemos que $\log_2(3) - 0.5 > 1$

Deducimos que $c \cdot n \in O(n^{\log_2(3)-0.5})$, por lo que usando el Teorema Maestro concluimos que $T(n) \in \Theta(n^{\log_2(3)})$

El Teorema Maestro y la función $\lceil x \rceil$

Suponga que cambiamos $\lfloor \frac{n}{b} \rfloor$ por $\lceil \frac{n}{b} \rceil$ en la definición de (a, b)-regularidad.

Entonces el Teorema Maestro sigue siendo válido pero con $T(\lfloor \frac{n}{b} \rfloor) + f(n)$ reemplazado por $T(\lceil \frac{n}{b} \rceil) + f(n)$

Analizando la complejidad de un algoritmo

Sea $\mathcal{A} : \Sigma^* \to \Sigma^*$ un algoritmo

- Recuerde que $t_{\mathcal{A}}(n)$ es el mayor número de pasos realizados por $\mathcal{A}$ sobre las entradas $w \in \Sigma^*$ de largo n

Definición (Complejidad en el peor caso)

Decimos que $\mathcal{A}$ en el peor caso es $O(f(n))$ si $t_{\mathcal{A}}(n) \in O(f(n))$

Analizando la complejidad de un algoritmo

Hasta ahora sólo hemos analizado la complejidad de un algoritmo considerando el peor caso.

También tiene sentido estudiar la complejidad del algoritmo $\mathcal{A}$ en el caso promedio, el cual está dado por la siguiente expresión:

$$\frac{1}{k^n} \cdot \left(\sum_{w \in \Sigma^n} \text{tiempo}_{\mathcal{A}}(w) \right)$$

donde $k = |\Sigma|$ y $\Sigma^n = \{w \in \Sigma^* \mid |w| = n\}$

El caso promedio de un algoritmo

La definición anterior asume que todas las entradas son igualmente probables.

- Esto podría no reflejar la distribución de las entradas en la práctica

Para solucionar este problema podemos usar otras distribuciones de probabilidad.

El caso promedio de un algoritmo

Suponemos que para cada $n \in \mathbb{N}$ hay una distribución de probabilidades:

$\Pr_n(w)$ es la probabilidad de que $w \in \Sigma^n$ aparezca como entrada de $\mathcal{A}$

Nótese que $\sum_{w \in \Sigma^n} \Pr_n(w) = 1$

Ejemplo

Para la definición de caso promedio en las transparencias anteriores tenemos que $\Pr_n(w) = \frac{1}{k^n}$ con $k = |\Sigma|$

El caso promedio de un algoritmo

Para definir el caso promedio, para cada $n \in \mathbb{N}$ usamos una variable aleatoria X_n

- Para cada $w \in \Sigma^n$ se tiene que $X_n(w) = \mathit{tiempo}_{\mathcal{A}}(w)$

Para las entradas de largo n, el número de pasos de $\mathcal{A}$ en el caso promedio es el valor esperado de la variable aleatoria X_n:

$$E(X_n) \quad = \quad \sum_{w \in \Sigma^n} X_n(w) \cdot \Pr_n(w)$$

Definición (Complejidad en el caso promedio)

Decimos que $\mathcal{A}$ en el caso promedio es $O(f(n))$ si $E(X_n) \in O(f(n))$

Sobre las definiciones de peor caso y caso promedio

Notación

Las definiciones de peor caso y caso promedio pueden ser modificadas para considerar las notaciones Θ y Ω

- Simplemente reemplazando $O(f(n))$ por $\Theta(f(n))$ u $\Omega(f(n))$, respectivamente

Por ejemplo, decimos que $\mathcal{A}$ en el caso promedio es $\Theta(f(n))$ si $E(X_n) \in \Theta(f(n))$

Un ejemplo: el algoritmo de ordenación Quicksort

Quicksort es un algoritmo de ordenación muy utilizado en la práctica.

La función clave para la definición de Quicksort:

Partición(L, m, n)
    $pivote := L[m]$
    $i := m$
    for $j := m + 1$ **to** n **do**
        if $L[j] \leq pivote$ **then**
            $i := i + 1$
            intercambiar $L[i]$ con $L[j]$
    intercambiar $L[m]$ con $L[i]$
    return i

Nótese que en la definición de **Partición** la lista L es pasado por referencia

- Las listas (o arreglos) siempre son pasados por referencia en este curso

Un ejemplo: el algoritmo de ordenación Quicksort

La definición de Quicksort:

Quicksort$(L,\ m,\ n)$
    if $m < n$ **then**
        $\ell :=$ **Partición**$(L,\ m,\ n)$
        Quicksort$(L,\ m,\ \ell - 1)$
        Quicksort$(L,\ \ell + 1,\ n)$

¿Cuál es la complejidad de Quicksort?

Vamos a contar el número de comparaciones al medir la complejidad de **Quicksort**.

- ¿Es ésta una buen medida de complejidad?

Sí **Partición** siempre divide a una lista en dos listas del mismo tamaño, entonces la complejidad de **Quicksort** estaría dada por la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 0 \\ 2 \cdot T(\lfloor \frac{n}{2} \rfloor) + c \cdot n & n \geq 1 \end{cases}$$

¿Cuál es la complejidad de Quicksort?

Dado que $c \cdot n \in \Theta(n^{\log_2(2)})$, concluimos usando el Teorema Maestro que $T(n) \in \Theta(n \cdot \log_2(n))$

¿Pero qué nos asegura que **Partición** divide a una lista en dos listas del mismo tamaño?

Ejercicio

1. Dada una lista con n elementos, ¿cuál es el peor caso para **Quicksort**?
2. Demuestre que **Quicksort** en el peor caso es $\Theta(n^2)$

¿Por qué usamos Quicksort entonces?

Vamos a demostrar que **Quicksort** en el caso promedio es $\Theta(n \cdot \log_2(n))$

- Suponiendo una distribución uniforme en las entradas

Vamos a considerar listas sin elementos repetidos.

- Usted va a tener que pensar cómo obtener la misma complejidad para listas con elementos repetidos

La complejidad de Quicksort en el caso promedio

Sea $n \in \mathbb{N}$ tal que $n \geq 2$, y sea $\mathcal{E}_n$ el conjunto de listas L con n elementos distintos sacados desde el conjunto $\{1, \ldots, n\}$

- Tenemos que $|\mathcal{E}_n| = n!$

Para cada $L \in \mathcal{E}_n$ tenemos lo siguiente:

- $\Pr_n(L) = \frac{1}{n!}$
- $X_n(L)$ es el número de comparaciones realizadas por la llamada **Quicksort**$(L, 1, n)$

La complejidad de Quicksort en el caso promedio

La complejidad de **Quicksort** en el caso promedio está dada por $E(X_n)$

¿Por qué nos restringimos a las listas con elementos sacados desde el conjunto $\{1, \ldots, n\}$?

- ¿En qué sentido estamos considerando todas las listas posibles con n elementos (sin repeticiones)?
- ¿Cómo refleja esto el hecho de que estamos considerando las entradas de un cierto largo?

Calculando el valor esperado de X_n

Sea $L \in \mathcal{E}_n$

Para cada $i, j \in \{1, \ldots, n\}$ con $i \leq j$ defina la siguiente variable aleatoria:

$Y_{i,j}(L)$: número de veces que i es comparado con j en la llamada **Quicksort**$(L, 1, n)$

Entonces tenemos lo siguiente:

$$X_n(L) = \sum_{i=1}^{n} \sum_{j=i}^{n} Y_{i,j}(L)$$

Calculando el valor esperado de X_n

Dado que el valor esperado de una variable aleatoria es una función lineal, concluimos que:

$$E(X_n) = \sum_{i=1}^{n} \sum_{j=i}^{n} E(Y_{i,j})$$

Para calcular $E(X_n)$ basta entonces calcular $E(Y_{i,j})$ para cada $i, j \in \{1, \ldots, n\}$ con $i \leq j$

Calculando el valor esperado de $Y_{i,j}$

Por la definición de **Partición** no es posible comparar un elemento consigo mismo, por lo que $Y_{i,i}(L) = 0$ para cada $i \in \{1, \ldots, n\}$ y $L \in \mathcal{E}_n$

- Tenemos entonces que $E(Y_{i,i}) = 0$

Consideremos ahora el caso $i = 1$ y $j = n$

Los elementos 1 y n sólo pueden ser comparados en la llamada **Partición**$(L, 1, n)$

- Estos elementos son comparados si $L[1] = 1$ o $L[1] = n$
- Se realiza a lo más una comparación entre ellos

Tenemos entonces que $Y_{1,n}$ es igual a 0 ó 1

Calculando el valor esperado de $Y_{i,j}$

Además, $\Pr(Y_{1,n} = 1)$ es igual a la probabilidad de que $L[1] = 1$ o $L[1] = n$ dado que L es escogido al azar y con distribución uniforme desde el conjunto $\mathcal{E}_n$

Tenemos entonces que:

$$\Pr(Y_{1,n} = 1) \;=\; \frac{2 \cdot (n-1)!}{n!} \;=\; \frac{2}{n}$$

Nótese que esta probabilidad puede cambiar si consideramos otra distribución de probabilidades sobre las listas en $\mathcal{E}_n$

Concluimos que:

$$E(Y_{1,n}) \;=\; 0 \cdot \Pr(Y_{1,n} = 0) + 1 \cdot \Pr(Y_{1,n} = 1) \;=\; \frac{2}{n}$$

Calculando el valor esperado de $Y_{i,j}$

Consideramos ahora el caso general $1 \leq i < j \leq n$

Mientras las llamadas a **Partición** escojan un valor para *pivote* tal que $pivote < i$ o $pivote > j$, los elementos i y j no son comparados y están en una parte de la lista que va a ser ordenada por **Quicksort**.

- i y j pueden ser comparados en las siguientes llamadas a **Partición**

Calculando el valor esperado de $Y_{i,j}$

Si **Partición** escoge un valor para *pivote* tal que $i < pivote < j$, entonces i no es comparado con j en la ejecución completa de **Quicksort**.

La única forma en que **Quicksort** puede comparar i con j es que el primer elemento que escoja **Partición** desde el conjunto $\{i, i+1, \ldots, j\}$ sea i ó j

- Se realiza a lo más una comparación entre i y j en la ejecución completa de **Quicksort**

Calculando el valor esperado de $Y_{i,j}$

Tenemos entonces que $Y_{i,j}$ es igual a 0 ó 1, y además que:

$$\Pr(Y_{i,j} = 1) \quad = \quad \frac{2}{j - i + 1}$$

Concluimos que:

$$E(Y_{i,j}) \;=\; 0 \cdot \Pr(Y_{i,j} = 0) + 1 \cdot \Pr(Y_{i,j} = 1) \;=\; \frac{2}{j - i + 1}$$

El cálculo final

Concluimos que:

$$
\begin{aligned}
E(X_n) &= \sum_{i=1}^{n} \sum_{j=i}^{n} E(Y_{i,j}) \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(Y_{i,j}) \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{k=2}^{n} (n+1-k) \cdot \frac{2}{k} \\
&= 2 \cdot (n+1) \cdot \left(\sum_{k=2}^{n} \frac{1}{k} \right) - 2 \cdot (n-1) \\
&= 2 \cdot (n+1) \cdot \left(\sum_{k=1}^{n} \frac{1}{k} \right) - 4 \cdot n
\end{aligned}
$$

La sumatoria armónica

Para terminar el calculo de $E(X_n)$ tenemos que acotar la sumatoria armónica $\sum_{k=1}^{n} \frac{1}{k}$

Tenemos que:

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{1}{x}\, dx \leq \sum_{k=1}^{n} \frac{1}{k}$$

Dado que $\int_{1}^{n} \frac{1}{x}\, dx = \ln(n) - \ln(1) = \ln(n)$, concluimos que:

$$\ln(n) \leq \sum_{k=1}^{n} \frac{1}{k} \leq \ln(n) + 1$$

La sumatoria armónica

Por lo tanto:

$$2 \cdot (n+1) \cdot \ln(n) - 4 \cdot n \ \leq \ E(X_n) \ \leq \ 2 \cdot (n+1) \cdot (\ln(n)+1) - 4 \cdot n$$

De lo cual concluimos que $E(X_n) \in \Theta(n \cdot \log_2(n))$

- Vale decir, **Quicksort** en el caso promedio es $\Theta(n \cdot \log_2(n))$

Técnicas para demostrar cotas inferiores

Queremos establecer una cota inferior para el número de operaciones realizadas por una clase de algoritmos que resuelven un problema específico.

- Por ejemplo, una cota inferior para los algoritmos que ordenan una lista de números enteros y cuya operación básica es la comparación

Vamos a estudiar tres técnicas para demostrar cotas inferiores:

- Mejor estrategia del adversario
- Arboles de decisión
- Reducciones

Calculando el máximo de una lista

Sea $L[1 \ldots n]$ una lista de números enteros sin repeticiones.

Queremos estudiar el número de comparaciones que debe realizar un algoritmo para encontrar el máximo elemento de L

- Consideramos algoritmos cuya operación básica es la comparación

Ejercicio

Construya un algoritmo que encuentre el máximo elemento de $L[1 \ldots n]$ realizando $n - 1$ comparaciones de elementos de la lista.

Calculando el máximo de una lista

¿Es posible calcular el máximo elemento de L con menos de $n-1$ comparaciones de elementos de L?

Vamos a demostrar que no es posible.

- Vamos a utilizar una técnica basada en buscar la mejor estrategia del adversario

Una técnica basada en la mejor estrategia del adversario

Recuerde que vamos a utilizar esta técnica para establecer una cota inferior para el número de operaciones realizadas por una clase de algoritmos $\mathcal{C}$ que resuelven un problema específico.

Lo primero que necesitamos entonces es una caracterización general de los algoritmos $\mathcal{A}$ que pertenecen a $\mathcal{C}$

Esta caracterización debe servir para identificar qué puede decidir $\mathcal{A}$ y qué depende del medio externo.

- El medio externo es el adversario

Una técnica basada en la mejor estrategia del adversario

En la medida que $\mathcal{A}$ procesa una entrada:

- El adversario toma decisiones que ponen $\mathcal{A}$ en el peor caso
- $\mathcal{A}$ responde de la mejor manera posible

A partir de esta interacción establecemos una cota inferior para el número de operaciones realizadas por $\mathcal{A}$

- Esta es una cota inferior para el número de operaciones realizadas por los algoritmos en $\mathcal{C}$, ya que $\mathcal{A}$ es un elemento arbitrario de $\mathcal{C}$

Calculando el máximo: mejor estrategia del adversario

Sea $\mathcal{A}$ un algoritmo para calcular el máximo elemento de una lista de números enteros sin repeticiones y cuya operación básica es la comparación.

- Suponemos que $\mathcal{A}$ no compara un elemento consigo mismo. ¿Es razonable suponer esto? ¿Cómo se puede implementar esto?

Caracterizamos $\mathcal{A}$ como un torneo donde el máximo es el ganador.

- Si b es comparado con c por $\mathcal{A}$, decimos que b gana la comparación si $b > c$

Calculando el máximo: mejor estrategia del adversario

Suponga que $L[1 \ldots n]$ es una lista de números enteros que es dada como entrada a $\mathcal{A}$

- Recuerde L no tiene elementos repetidos

Para describir el funcionamiento de $\mathcal{A}$ utilizamos los siguientes conjuntos que muestran el estado del algoritmo en cada instance de tiempo:

U : elementos de L que todavía no han sido comparados

G : elementos de L que han ganado todas sus comparaciones (y al menos han participado en una comparación)

P : elementos de L que perdieron al menos una comparación

Calculando el máximo: mejor estrategia del adversario

Sean $u = |U|$, $g = |G|$ y $p = |P|$

En cada instante de tiempo el vector (u, g, p) describe el estado de $\mathcal{A}$

- En cada instante se tiene que $u + g + p = n$
- $\mathcal{A}$ encontró el máximo elemento de L si $(u, g, p) = (0, 1, n - 1)$

Calculando el máximo: mejor estrategia del adversario

La siguiente tabla describe el cambio del vector (u, g, p) dependiendo de la comparación $b > c$:

	$c \in U$	$c \in G$	$c \in P$
$b \in U$	$(u-2, g+1, p+1)$	$(u-1, g, p+1)$	$b > c$: $(u-1, g+1, p)$ $b < c$: $(u-1, g, p+1)$
$b \in G$		$(u, g-1, p+1)$	$b > c$: (u, g, p) $b < c$: $(u, g-1, p+1)$
$b \in P$			(u, g, p)

Las preguntas fundamentales: ¿Qué elige el algoritmo? ¿Qué elige el adversario?

Las decisiones del algoritmo y el adversario

$\mathcal{A}$ escoge los elementos b y c a comparar

- En particular, elige los conjuntos desde los cuales sacar estos elementos

El adversario elige el peor caso según la decisión tomada por $\mathcal{A}$:

	$c \in U$	$c \in G$	$c \in P$
$b \in U$	$(u-2, g+1, p+1)$	$(u-1, g, p+1)$	$b > c$: $(u-1, g+1, p)$ $b < c$: $(u-1, g, p+1)$
$b \in G$		$(u, g-1, p+1)$	$b > c$: (u, g, p) $b < c$: $(u, g-1, p+1)$
$b \in P$			(u, g, p)

Una cota inferior para el cálculo del máximo

Dado un vector (u, g, p), una comparación de $\mathcal{A}$ da como resultado un vector (u', g', p') tal que:

$$p' = p \quad \text{o} \quad p' = p + 1$$

Entonces para que $\mathcal{A}$ llegue al vector $(0, 1, n-1)$ debe realizar al menos $n-1$ comparaciones

Conclusión

En el peor caso, un algoritmo debe realizar al menos $n-1$ comparaciones para encontrar el máximo elemento de una lista con n elementos no repetidos, suponiendo que el algoritmo no compara un elemento consigo mismo.

Calculando el máximo y el mínimo de una lista

Sea $L[1 \ldots n]$ una lista de número enteros sin elementos repetidos.

Queremos estudiar el número de comparaciones que debe realizar un algoritmo para encontrar tanto el máximo como el mínimo elemento de L

- Nuevamente consideramos algoritmos cuya operación básica es la comparación

Ejercicio

Construya un algoritmo que encuentre el máximo y el mínimo elemento de $L[1 \ldots n]$ realizando $2 \cdot n - 3$ comparaciones de elementos de la lista.

Calculando el máximo y el mínimo de una lista

¿Es posible calcular el máximo y el mínimo elemento de L con menos de $2 \cdot n - 3$ comparaciones de elementos de L?

Vamos a utilizar la técnica basada en la mejor estrategia del adversario para encontrar una cota inferior para este problema.

- Esto nos va a dar una idea de cómo construir un algoritmo más eficiente

Máximo y mínimo: mejor estrategia del adversario

Sea $\mathcal{A}$ un algoritmo para calcular el máximo y el mínimo elemento de una lista de números enteros sin repeticiones y cuya operación básica es la comparación.

- Suponemos que $\mathcal{A}$ no compara un elemento consigo mismo

Al igual que para el caso del cálculo del máximo, caracterizamos $\mathcal{A}$ como un torneo donde el máximo y el mínimo son los ganadores.

Máximo y mínimo: mejor estrategia del adversario

Suponga que $L[1 \ldots n]$ es una lista de números enteros sin repeticiones que es dada como entrada a $\mathcal{A}$, donde $n \geq 2$

Para describir el funcionamiento de $\mathcal{A}$ utilizamos los siguientes conjuntos que muestran el estado del algoritmo en cada instance de tiempo:

- U : elementos de L que todavía no han sido comparados
- G : elementos de L que han ganado todas sus comparaciones (y al menos han participado en una comparación)
- P : elementos de L que han perdido todas sus comparaciones (y al menos han participado en una comparación)
- E : elementos de L que ganaron al menos una comparación y perdieron al menos una comparación

¿Por qué en este caso utilizamos los conjuntos P y E?

Máximo y mínimo: mejor estrategia del adversario

Sean $u = |U|$, $g = |G|$, $p = |P|$ y $e = |E|$

En cada instante de tiempo el vector (u, g, p, e) describe el estado de $\mathcal{A}$

- En cada instante se tiene que $u + g + p + e = n$
- $\mathcal{A}$ encontró el máximo y el mínimo elemento de L si $(u, g, p, e) = (0, 1, 1, n - 2)$

Máximo y mínimo: mejor estrategia del adversario

Al igual que para el cálculo del máximo, la siguiente tabla describe el cambio del vector (u, g, p, e) dependiendo de la comparación $b > c$:

	$c \in U$	$c \in G$	$c \in P$	$c \in E$
$b \in U$	$(u-2, g+1, p+1, e)$	$b > c$: $(u-1, g, p, e+1)$ $b < c$: $(u-1, g, p+1, e)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p, e+1)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p+1, e)$
$b \in G$		$(u, g-1, p, e+1)$	$b > c$: (u, g, p, e) $b < c$: $(u, g-1, p-1, e+2)$	$b > c$: (u, g, p, e) $b < c$: $(u, g-1, p, e+1)$
$b \in P$			$(u, g, p-1, e+1)$	$b > c$: $(u, g, p-1, e+1)$ $b < c$: (u, g, p, e)
$b \in E$				(u, g, p, e)

Máximo y mínimo: las decisiones del adversario

Usando la tabla podemos ver cuáles son las decisiones que va a tomar el adversario, las cuales corresponden a los peores casos para $\mathcal{A}$:

	$c \in U$	$c \in G$	$c \in P$	$c \in E$
$b \in U$	$(u-2, g+1, p+1, e)$	$b > c$: $(u-1, g, p, e+1)$ $b < c$: $(u-1, g, p+1, e)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p, e+1)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p+1, e)$
$b \in G$		$(u, g-1, p, e+1)$	$b > c$: (u, g, p, e) $b < c$: $(u, g-1, p-1, e+2)$	$b > c$: (u, g, p, e) $b < c$: $(u, g-1, p, e+1)$
$b \in P$			$(u, g, p-1, e+1)$	$b > c$: $(u, g, p-1, e+1)$ $b < c$: (u, g, p, e)
$b \in E$				(u, g, p, e)

Máximo y mínimo: las decisiones del algoritmo

Hay posible decisiones de $\mathcal{A}$ que no cambian el vector (u, g, p, e) dada la estrategia del adversario.

Estas posibilidades no deben ser consideradas por $\mathcal{A}$:

	$c \in U$	$c \in G$	$c \in P$	$c \in E$
$b \in U$	$(u-2, g+1, p+1, e)$	$b > c$: $(u-1, g, p, e+1)$ $b < c$: $(u-1, g, p+1, e)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p, e+1)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p+1, e)$
$b \in G$		$(u, g-1, p, e+1)$		
$b \in P$			$(u, g, p-1, e+1)$	

Una cota inferior para el cálculo del máximo y el mínimo

Cada vez que un elemento es colocado en E puede dejar de ser comparado.

El adversario entonces tiene que evitar que un elemento sea colocado en E, lo cual es representado por las elecciones en rojo:

	$c \in U$	$c \in G$	$c \in P$
$b \in U$	$(u-2, g+1, p+1, e)$	$b > c$: $(u-1, g, p, e+1)$ $b < c$: $(u-1, g, p+1, e)$	$b > c$: $(u-1, g+1, p, e)$ $b < c$: $(u-1, g, p, e+1)$
$b \in G$		$(u, g-1, p, e+1)$	
$b \in P$			$(u, g, p-1, e+1)$

Una cota inferior para el cálculo del máximo y el mínimo

Los elementos que son agregados a E vienen entonces de G o P

El algoritmo $\mathcal{A}$ debe entonces tratar de que los conjuntos G y P crezcan lo más rápido posible.

- Puede realizar $\lfloor \frac{n}{2} \rfloor$ comparaciones entre elementos de U, después de lo cual se va a tener que:

$$g = p = \lfloor \frac{n}{2} \rfloor$$

Si n es impar se debe realizar una comparación adicional para que todos los elementos en algún momento estén en G o P

- Si n es par esta propiedad se tiene después de realizar las comparaciones entre elementos de U

Una cota inferior para el cálculo del máximo y el mínimo

Se debe realizar comparaciones entre elementos de G y entre elementos de P para lograr que $e = n - 2$

- Cada una de estas comparaciones incrementa el valor de e en 1
- Recuerde que $\mathcal{A}$ encontró el máximo y el mínimo cuando llega al vector $(0, 1, 1, n-2)$

Concluimos entonces que $\mathcal{A}$ llega al vector $(0, 1, 1, n-2)$ después de realizar el siguiente número de comparaciones:

n par: $\frac{n}{2} + (n-2) = \frac{3}{2} \cdot n - 2$

n impar: $\lfloor \frac{n}{2} \rfloor + 1 + (n-2) = \lceil \frac{n}{2} \rceil + (n-2) = \lceil \frac{3}{2} \cdot n \rceil - 2$

Una cota inferior para el cálculo del máximo y el mínimo

Conclusión

En el peor caso, un algoritmo debe realizar al menos $\lceil \frac{3}{2} \cdot n \rceil - 2$ comparaciones para encontrar el máximo y el mínimo elemento de una lista con n elementos no repetidos, suponiendo que el algoritmo no compara un elemento consigo mismo.

Ejercicio

Encuentre un algoritmo que logre la cota inferior.

Demostrando cotas inferiores: Arboles de decisión

De la misma forma que la técnica basada en la mejor estrategia del adversario, vamos a utilizar los árboles de decisión para establecer una cota inferior para el número de operaciones realizadas por una clase de algoritmos $\mathcal{C}$ que resuelven un problema específico.

En particular, vamos a utilizar esta técnica para algoritmos cuya operación básica es la comparación.

- Por ejemplo, buscar un elemento en una lista u ordenar una lista

Arboles de decisión para un algoritmo

Sea $\mathcal{A}$ un algoritmo en una clase $\mathcal{C}$ de algoritmos

- Recuerde que medimos la complejidad de $\mathcal{A}$ contando el número de comparaciones realizadas por $\mathcal{A}$

Para cada $n \in \mathbb{N}$ definimos un árbol de decisión $\mathcal{T}_{\mathcal{A},n}$ que describe el funcionamiento de $\mathcal{A}$ con las entradas de largo n

- Usamos el mismo árbol para todas las entradas de largo n
  - El árbol de decisión tiene un registro de las comparaciones hechas por el algoritmo $\mathcal{A}$
- Para distintos valores de n podemos tener distintos árboles de decisión

Las etiquetas en un árbol de decisión

$\mathcal{T}_{\mathcal{A},n}$ es un árbol con etiquetas en los nodos.

La etiqueta de un nodo u de $\mathcal{T}_{\mathcal{A},n}$:

- es una comparación si u es un nodo interno
- es una instrucción de la forma **return** v si u es una hoja, donde v es un valor retornado por $\mathcal{A}$

Los caminos en un árbol de decisión

Si v es un posible valor retornado por $\mathcal{A}$, entonces debe existir al menos una hoja de $\mathcal{T}_{\mathcal{A},n}$ con etiqueta **return** v

- Más de una hoja en $\mathcal{T}_{\mathcal{A},n}$ puede tener etiqueta **return** v

Dado un camino en $\mathcal{T}_{\mathcal{A},n}$ desde la raíz hasta una hoja con etiqueta **return** v:

- ¿Qué representa este camino? Una ejecución de $\mathcal{A}$ que retorna v
- ¿Qué representa el largo (número de arcos) de este camino? El número de comparaciones realizadas por $\mathcal{A}$ en una ejecución que retorna v

La profundidad de un árbol de decisión

¿Qué representa el camino más largo en $\mathcal{T}_{\mathcal{A},n}$ desde la raíz hasta las hojas?

- El peor caso para el algoritmo $\mathcal{A}$ para las entradas de largo n

Notación

profundidad($\mathcal{T}_{\mathcal{A},n}$): largo del camino de mayor longitud entre la raíz y las hojas de $\mathcal{T}_{\mathcal{A},n}$

Tenemos que $\mathcal{A}$ en el peor caso es $\Omega(\text{profundidad}(\mathcal{T}_{\mathcal{A},n}))$

La profundidad de un árbol de decisión

Si demostramos que profundidad($\mathcal{T}_{\mathcal{A},n}$) es $\Omega(f(n))$, entonces $\mathcal{A}$ en el peor caso es $\Omega(f(n))$

- Como $\mathcal{A}$ es un algoritmo arbitrario en $\mathcal{C}$, concluimos que todo algoritmos en $\mathcal{C}$ en el peor caso es $\Omega(f(n))$

¿Cómo podemos deducir una cota inferior para profundidad($\mathcal{T}_{\mathcal{A},n}$)?

- Una manera sencilla es utilizando el número de hojas del árbol

Las hojas de un árbol de decisión

Notación

hojas($\mathcal{T}_{\mathcal{A},n}$): número de hojas de $\mathcal{T}_{\mathcal{A},n}$

Dado que $\mathcal{T}_{\mathcal{A},n}$ es un árbol binario, tenemos la siguiente relación:

Lema

$hojas(\mathcal{T}_{\mathcal{A},n}) \leq 2^{profundidad(\mathcal{T}_{\mathcal{A},n})}$

Ejercicio

Demuestre el lema.

Las hojas de un árbol de decisión

Concluimos que $\log_2(\text{hojas}(\mathcal{T}_{\mathcal{A},n})) \leq \text{profundidad}(\mathcal{T}_{\mathcal{A},n})$

- Tenemos entonces una cota inferior para profundidad$(\mathcal{T}_{\mathcal{A},n})$

¿Pero cómo obtenemos una cota inferior para hojas$(\mathcal{T}_{\mathcal{A},n})$?

Sabemos que hojas$(\mathcal{T}_{\mathcal{A},n})$ debe ser mayor o igual al número de posibles valores retornados por $\mathcal{A}$ para las entradas de largo n

- En general esta cota inferior es suficiente para obtener una buena cota inferior para profundidad$(\mathcal{T}_{\mathcal{A},n})$

Vamos a ver dos aplicaciones de esta técnica ...

Encontrando un elemento en una lista ordenada

Sea $L[1 \ldots n]$ una lista de números enteros ordenada de menor a mayor, y sea a un número entero.

Queremos estudiar el número de comparaciones que debe realizar un algoritmo para determinar una posición i tal que $a = L[i]$ o retornar no en caso que a no esté en L

- Consideramos algoritmos cuya operación básica es la comparación

Ejercicio

Construya un algoritmo que resuelva el problema realizando $f(n)$ comparaciones, donde $f(n) \in \Theta(\log_2(n))$

Encontrando un elemento en una lista ordenada

¿Existe un algoritmo para encontrar un elemento en una lista ordenada que realice $g(n)$ comparaciones, donde $g(n)$ es de orden menor que $\log_2(n)$?

- ¿Qué significa que una función g sea de orden menor que una función h?

Definición

$$o(h) \quad = \quad \{f : \mathbb{N} \to \mathbb{R}_0^+ \mid (\forall c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\,(f(n) \leq c \cdot h(n))\}$$

Ejercicio

Demuestre que si $f \in o(h)$ y se cumple que $(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(h(n) > 0)$, entonces $f \in O(h)$ y $h \notin O(f)$

Encontrando un elemento en una lista ordenada

La pregunta reformulada: ¿Existe un algoritmo para encontrar un elemento en una lista ordenada que realice $g(n)$ comparaciones, donde $g(n) \in o(\log_2(n))$?

Vamos a demostrar que no existe tal algoritmo.

- Vamos a utilizar una técnica basada en árboles de decisión

Encontrando una cota inferior

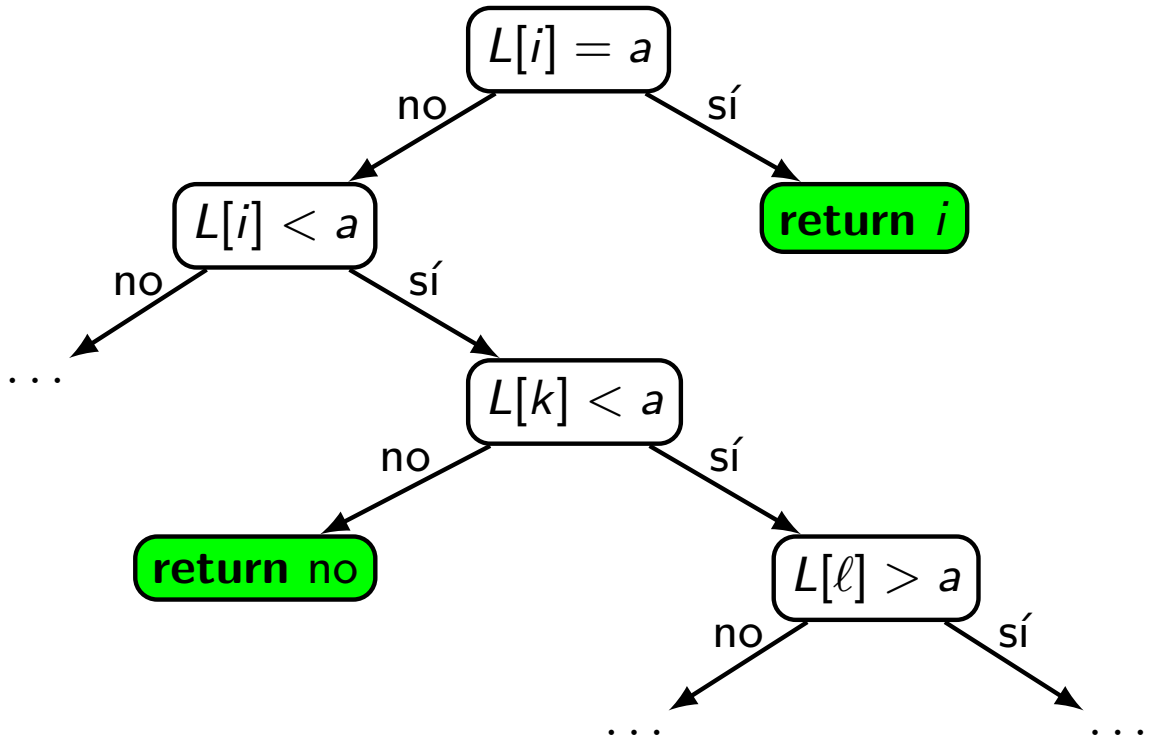
Sea $\mathcal{A}$ un algoritmo que resuelve el problema de encontrar un elemento en una lista ordenada

- Dada una lista ordenada de números enteros $L[1 \ldots n]$ y un número entero a, el algoritmo $\mathcal{A}$ determina una posición i tal que $a = L[i]$ o retorna no en caso que a no esté en L

Utilizamos un árbol de decisión $\mathcal{T}_{\mathcal{A},n}$ para describir el funcionamiento de $\mathcal{A}$ con las entradas (L, a) tales que la lista L tiene n elementos

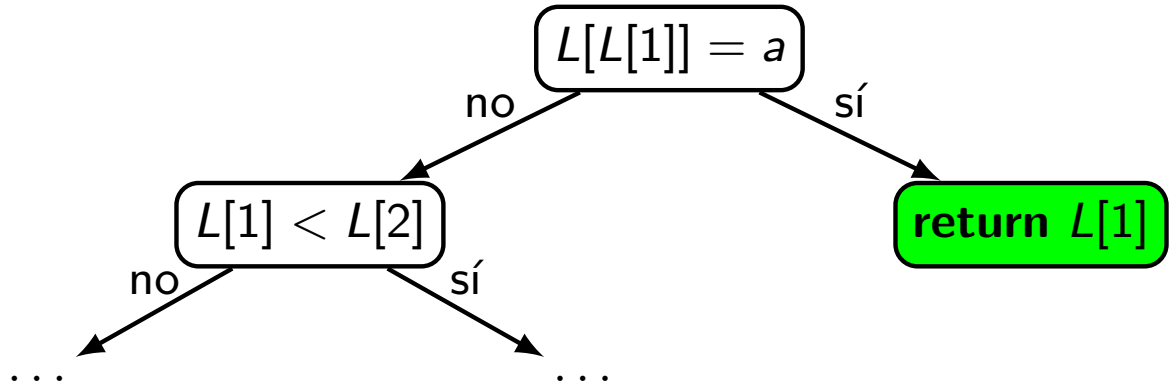
- Recuerde que debemos usar el mismo árbol para todas las entradas donde L tiene n elementos
- Para distintos valores de n podemos tener distintos árboles de decisión

El árbol de decisión $\mathcal{T}_{\mathcal{A},n}$



Algunas características de $\mathcal{T}_{\mathcal{A},n}$

Todas las comparaciones hechas por $\mathcal{A}$ son almacenadas en los nodos internos de $\mathcal{T}_{\mathcal{A},n}$, en las hojas se almacenan los valores retornados:



La etiqueta de un nodo u de $\mathcal{T}_{\mathcal{A},n}$:

- es una comparación si u es un nodo interno
- es una instrucción de la forma **return** no o **return** i, donde $i \in \{1, \ldots, n\}$, si u es una hoja

Una cota inferior para la búsqueda en una lista ordenada

En este caso tenemos que $n + 1 \leq \text{hojas}(\mathcal{T}_{\mathcal{A},n})$

Deducimos que $\log_2(n+1) \leq \text{profundidad}(\mathcal{T}_{\mathcal{A},n})$

Conclusión

En el peor caso, un algoritmo debe realizar al menos $\lceil \log_2(n+1) \rceil$ comparaciones para determinar, dado un entero a y una lista ordenada de enteros $L[1 \ldots n]$, una posición i tal que $a = L[i]$ o retornar no en caso que a no esté en L

- Todo algoritmo para encontrar un elemento en una lista ordenada y que esté basado en comparaciones en el peor caso es $\Omega(\log_2(n))$

Ordenando una lista

Sea $L[1 \ldots n]$ una lista de números enteros.

Queremos estudiar el número de comparaciones que debe realizar un algoritmo para ordenar L de menor a mayor.

- Al igual que en los casos anteriores consideramos algoritmos cuya operación básica es la comparación

Ejercicio

Construya un algoritmo que resuelva el problema realizando $f(n)$ comparaciones, donde $f(n) \in \Theta(n \cdot \log_2(n))$

Ordenando una lista

¿Existe un algoritmo para ordenar una lista que realice $g(n)$ comparaciones, donde $g(n) \in o(n \cdot \log_2(n))$?

Vamos a demostrar que no existe tal algoritmo.

- Vamos a utilizar una técnica basada en árboles de decisión

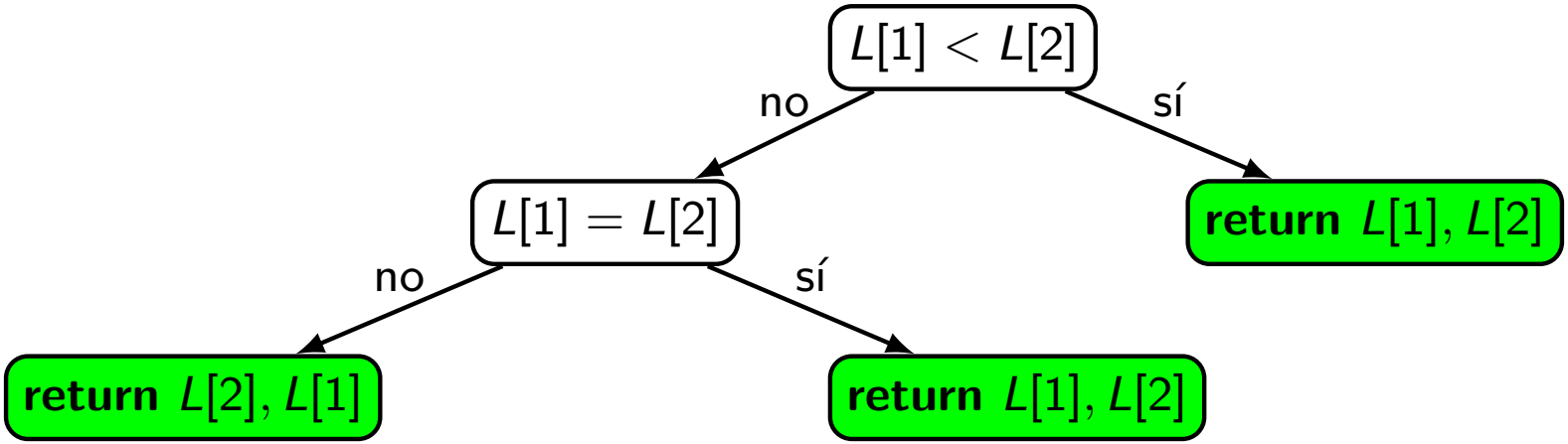
Encontrando una cota inferior

Sea $\mathcal{B}$ un algoritmo que resuelve el problema de ordenar una lista

- Dada una lista de números enteros $L[1 \ldots n]$, el algoritmo $\mathcal{B}$ retorna la lista L ordenada de menor a mayor

Utilizamos un árbol de decisión $\mathcal{T}_{\mathcal{B},n}$ para describir el funcionamiento de $\mathcal{B}$ con las entradas $L[1 \ldots n]$

El árbol de decisión $\mathcal{T}_{\mathcal{B},2}$



Una cota inferior para la ordenación

En este caso tenemos que $n! \leq \text{hojas}(\mathcal{T}_{\mathcal{B},n})$

- ¿Por qué?

Deducimos que $\log_2(n!) \leq \text{profundidad}(\mathcal{T}_{\mathcal{B},n})$

Lema

Para todo $n \in \mathbb{N}$ *se tiene que* $n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$

Una demostración del lema

El lema se cumple para $n = 0$ y $n = 1$

- Suponemos entonces que $n \geq 2$

Si $n = 2 \cdot k$ tenemos que:

$$
\begin{aligned}
n! &= (2 \cdot k)! \\
&= (2 \cdot k) \cdot (2 \cdot k - 1) \cdot \ldots \cdot (k + 1) \cdot k \cdot (k - 1) \cdot \ldots \cdot 1 \\
&\geq (2 \cdot k) \cdot (2 \cdot k - 1) \cdot \ldots \cdot (k + 1) \\
&> \underbrace{k \cdot k \cdot \ldots \cdot k}_{k \text{ veces}} \\
&= k^k \\
&= \left(\frac{n}{2}\right)^{\frac{n}{2}}
\end{aligned}
$$

Una demostración del lema

Finalmente, si $n = 2 \cdot k + 1$ tenemos que:

$$
\begin{aligned}
n! &= (2 \cdot k + 1)! \\
&= (2 \cdot k + 1) \cdot (2 \cdot k) \cdot (2 \cdot k - 1) \cdot \ldots \cdot (k + 1) \cdot k \cdot (k - 1) \cdot \ldots \cdot 1 \\
&\geq (2 \cdot k + 1) \cdot (2 \cdot k) \cdot (2 \cdot k - 1) \cdot \ldots \cdot (k + 1) \\
&> \underbrace{(k + 1) \cdot (k + 1) \cdot \ldots \cdot (k + 1)}_{(k+1) \text{ veces}} \\
&= (k + 1)^{(k+1)} \\
&> \left(\frac{n}{2}\right)^{\frac{n}{2}}
\end{aligned}
$$

□

Una cota inferior para la ordenación

Del lema deducimos que $\frac{n}{2} \cdot \log_2 \left(\frac{n}{2}\right) \leq \log_2(n!) \leq \text{profundidad}(\mathcal{T}_{\mathcal{B},n})$

Conclusión

En el peor caso, un algoritmo debe realizar al menos $\lceil \frac{n}{2} \cdot \log_2(\frac{n}{2}) \rceil$ comparaciones para ordenar una lista de números enteros.

- Todo algoritmo de ordenación basado en comparaciones en el peor caso es $\Omega(n \cdot \log_2 n)$

Demostrando cotas inferiores: reducciones

Hemos visto dos técnicas para obtener cotas inferiores para el número de operaciones realizadas por una clase de algoritmos: mejor estrategia del adversario y árboles de decisión

Vamos a introducir una tercera técnica para obtener cotas inferiores que está basada en la idea de reducir un problema F a otro problema G

- De esta forma podemos usar una cota inferior para F para deducir una cota inferior para G

Una noción de reducción

Sean $F : \Sigma^* \to \Sigma^*$ y $G : \Sigma^* \to \Sigma^*$ dos funciones.

Una función $H : \Sigma^* \to \Sigma^*$ es una reducción de F a G si para cada $w \in \Sigma^*$ tenemos que $F(w) = G(H(w))$

- H transforma una entrada w de F en una entrada $H(w)$ de G con la cual se tiene la misma salida

Una cota inferior a partir de la reducción

Suponga lo siguiente:

- Todo algoritmo que calcula F debe realizar al menos $f(n)$ operaciones para las entradas de largo n
- Existe un algoritmo que calcula H y que realiza $h(n)$ operaciones para las entradas de largo n
- Para cada $w \in \Sigma^*$ se tiene que $|H(w)| = \ell(|w|)$

Una cota inferior a partir de la reducción

Sea $\mathcal{A}$ un algoritmo que calcula G

- Recuerde que la función $t_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ nos da el número de operaciones realizadas por $\mathcal{A}$ en el peor caso

Dado que H es una reducción de F en G, concluimos por los supuestos anteriores que $f(n) \leq h(n) + t_{\mathcal{A}}(\ell(n))$

La ecuación $f(n) \leq h(n) + t_{\mathcal{A}}(\ell(n))$ es usada para obtener una cota inferior para $t_{\mathcal{A}}(n)$

- Si $\mathcal{A}$ es un algoritmo arbitrario en una clase $\mathcal{C}$ de algoritmos, entonces obtenemos una cota inferior para el número de operaciones que debe realizar cualquier algoritmo en $\mathcal{C}$ en el peor caso

Algunas consideraciones importantes

- Los supuestos iniciales pueden ser relajados
  - Por ejemplo, podemos consider una cota inferior para F en el peor caso o podemos dejar de exigir que $|H(w)| = \ell(|w|)$
- Otras nociones de reducción pueden ser utilizadas
  - Por ejemplo, podemos pedir que existan funciones H_1 y H_2 tales que $F(w) = H_2(G(H_1(w)))$
- Otros supuestos pueden ser considerados
  - Por ejemplo, la función f es no decreciente o estrictamente creciente

Algunas consideraciones importantes

Bajo estas nuevas condiciones se deduce otra ecuación que nos permite dar una cota inferior para el número de operaciones realizadas por un algoritmo que calcula G

Vamos a ver un ejemplo de esta técnica en el próximo capítulo