Técnicas para establecer cotas inferiores

IIC2283

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.

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

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 dos técnicas para demostrar cotas inferiores:

- Mejor estrategia del adversario
- Arboles de decisión

Calculando el máximo de una lista

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

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

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?

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.

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 $\mathcal{C}$

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.

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

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?

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

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 $\mathcal{A}$

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 $u + g + p = n$

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)

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

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 $\mathcal{A}$:

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:

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

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

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.

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

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?

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

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

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 $\mathcal{A}$

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 $u + g + p + e = n$

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.

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.

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

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:

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

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 $(0, 1, 1, n - 2)$

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.

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.

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 $\mathcal{A}$

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

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

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 $\mathcal{A}$

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

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?
- ¿Qué representa el largo (número de arcos) de este camino?

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?

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?

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 $\mathcal{T}_{\mathcal{A},n}$

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 $\Omega(f(n))$

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($\mathcal{T}_{\mathcal{A},n}$)?

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 $\mathcal{T}_{\mathcal{A},n}$

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

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

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 $\text{profundidad}(\mathcal{T}_{\mathcal{A},n})$

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 $\text{profundidad}(\mathcal{T}_{\mathcal{A},n})$

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

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$(\mathcal{T}_{\mathcal{A},n})$

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.

Búsqueda binaria resuelve el problema realizando $\Theta(\log_2(n))$ comparaciones

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

Búsqueda binaria resuelve el problema realizando $\Theta(\log_2(n))$ comparaciones

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 $\log_2(n)$?

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?

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

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

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

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

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 $g(n) \in o(\log_2(n))$?

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

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

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

Encontrando una cota inferior

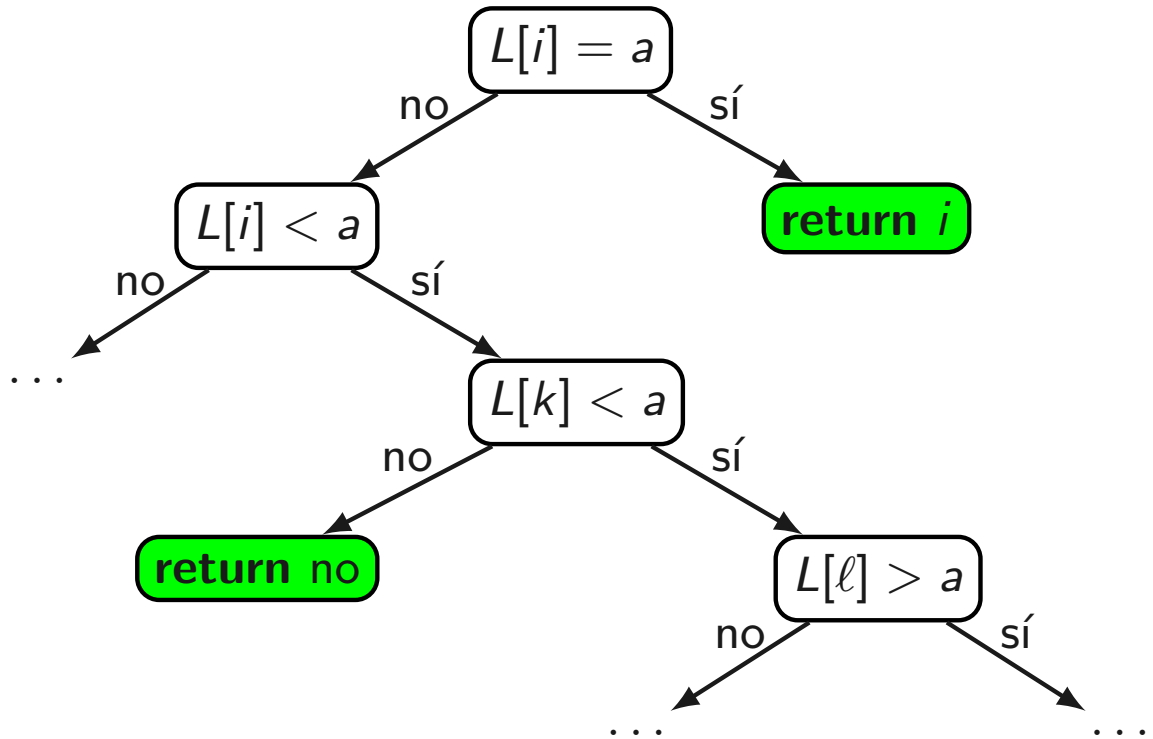
Sea $\mathcal{A}$ un algoritmo que resuelve el problema de encontrar un elemento en una lista ordenada

- 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

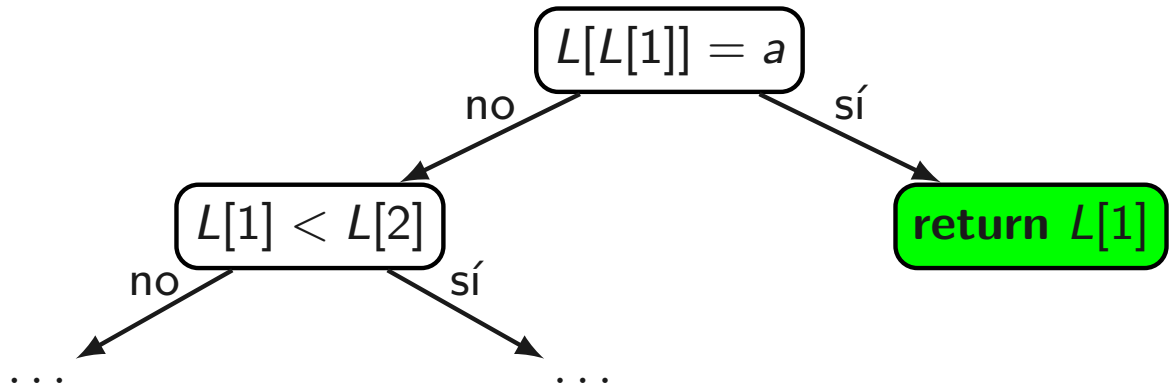
- Recuerde que debemos usar el mismo árbol para todas las entradas donde L tiene n elementos
- Para distintos valores de n podemos tener distintos árboles de decisión

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



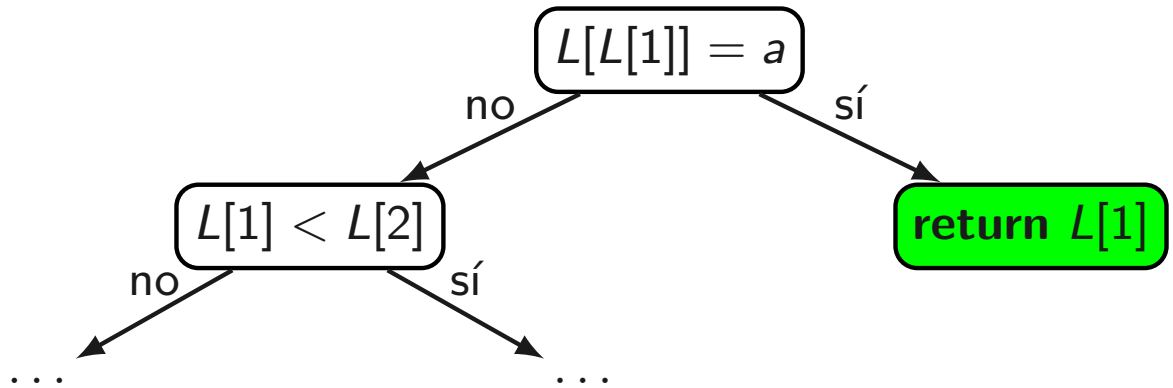
Algunas características de $\mathcal{T}_{\mathcal{A},n}$

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:



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 $n + 1 \leq \text{hojas}(\mathcal{T}_{\mathcal{A},n})$

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

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.

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

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 $g(n) \in o(n \cdot \log_2(n))$?

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

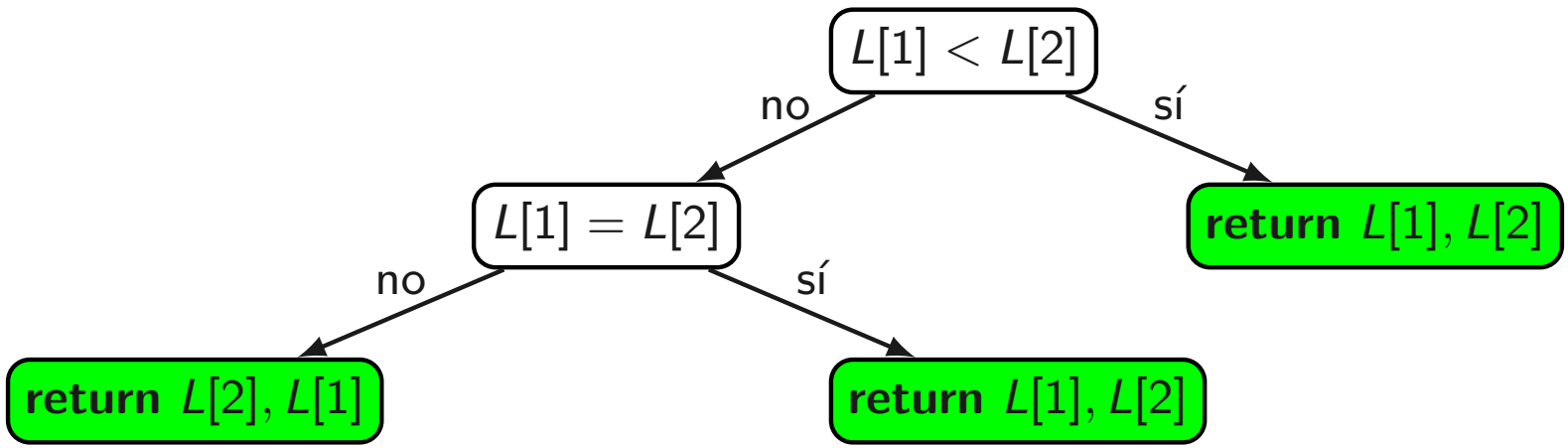
Encontrando una cota inferior

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?

Una cota inferior para la ordenación

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

- ¿Por qué?

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

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

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 $\Omega(n \cdot \log_2 n)$