# Cota Mínima

M. Andrea Rodríguez-Tastets
Ayudante: Erick Elejalde

Universidad de Concepción,Chile
www.inf.udec.cl\ ∼andrea
andrea@udec.cl

I Semestre - 2014

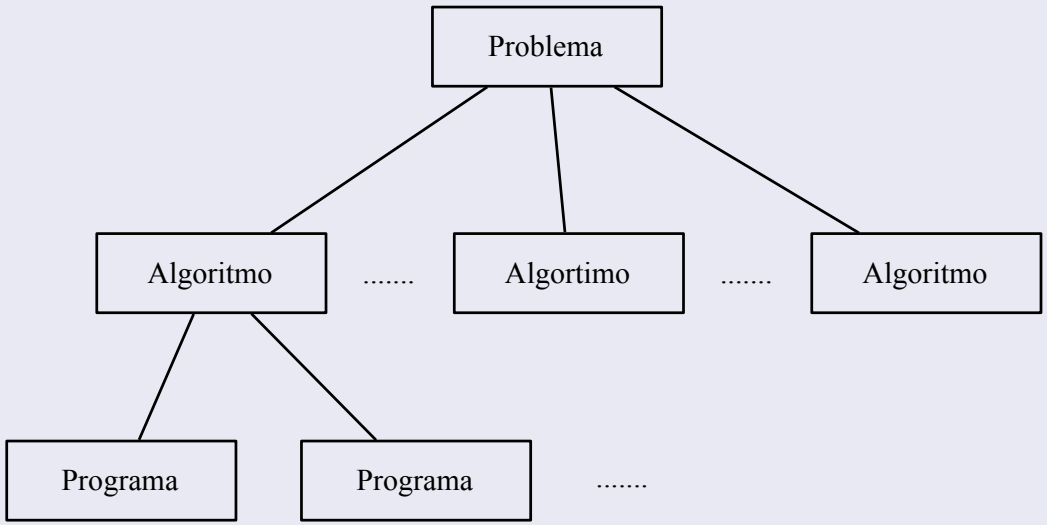
## Introducción

Al terminar esta sesión del curso los alumnos deberán ser capaces de

- Comprender el concepto de cota mínima o lower bound de un problema
- Conocer y aplicar estrategias para determinar la cota mínima de un problema simple.

## Problema-Algoritmo-Programa

La relación entre un problema, algoritmos y programas puede ser vista como una jerarquía.



## Problema-Algoritmo

- Para los algoritmos, usamos la notación de order para indicar a qué razón crece el costo de un algoritmo en función del tamaño de la entrada.
- Pero para un problema dado, ¿Cómo saber si existe un algoritmo mejor?

## Complejidad de un problema versus la de un algoritmo

- La complejidad de un algoritmos se define por el análisis del algoritmo.
- La complejidad de un problema se define por un upper bound, definido por un algoritmo, y un lower bound definido por una demostración.

## Upper and lower bound

- Upper bound: es el mejor algoritmo que ha sido encontrado para un problema. Se achica el upper bound al encontrar un algoritmo con una complejidad menor.
- Lower bound: es la mejor solución que es teóricamente posible. Se eleva el lower bound al encontrar una mejor demostración.
- Para problemas cerrados (closed problems), el upper y lower bound son los mismos. En los casos de problemas cerrados solo se pueden mejorar las constantes escondidas en la notación asintótica.

## Problema-Algoritmo: clasificación

- Problemas polinomiales son tratables.
- Algoritmos polinomiales son razonables.
- Problemas exponenciales son intratables
- Algoritmos exponenciales son no razonables.

## Tiempo de Complejidad de un Problema

- Upper bound (cota máxima): El problem $P$ es resuelto en tiempo $T_{upper}(n)$ si existe un algoritmo $A$ que entrega el resultado correcto en ese tiempo. Sea $P(I)$ la salida del problema $P$ para instancia $I$ y $A(I)$ la salida del algoritmo $A$ para instancia $I$.

$$\exists A, \forall I, A(I) = P(I) \wedge Time(A, I) \leq T_{upper}(|I|)$$

- Lower bound (cota mínima): Es el tiempo mínimo por un algoritmo para resolver un problema. $T_{lower}(n)$ es la cota mínima si no existe algoritmo que lo pueda resolve más rápido.

$$\forall A, \exists I, A(I) \neq P(I) \vee Time(A, I) \geq T_{lower}(|I|)$$

Lower bounds son difíciles de demostrar, porque se debe considerar cada posible algoritmo.

## Estrategias de Demostración

- Cota mínima trivial
- Juego del adversario
- Árboles de decisión
- Reducción

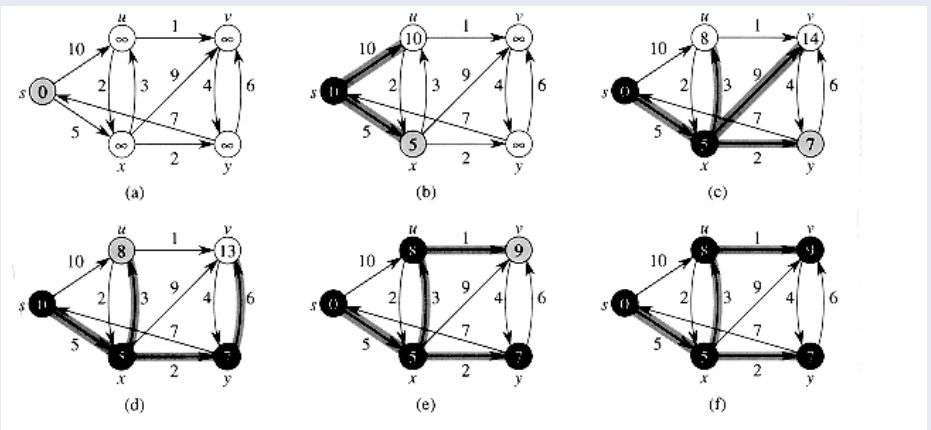
## Cota Mínima Trivial

- El método consiste en contar el número de entradas que deben ser examinadas y el número de salidas que deben ser producidas, y notar que el algoritmo debe al menos leer las entradas y escribir las salidas.

# Cota Mínima Trivial: ejemplo I

Dado un grafo $G$ dirigido con arcos con peso no negativos, y un vértice $v$, encontrar el camino más corto (con mínimo peso) desde $v$ a cada uno de los otros vértices en $G$. El algoritmo de Dijkstra da solución a este problema, donde todos los pesos son conocidos y no negativos.

```
DIJKSTRA(G,w,s)
1  INITIALIZE-SINGLE-SOURCE (G,s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8                do RELAX (u,v,w)
```




Este algoritmo corre normalmente en $O(|V|^2 + |E|)$, pero si consideramos una implementación de una cola de prioridad con un heap binario, tenemos que el costo es de $((|E| + |V|)log|V|)$. Se puede llegar a $O(|V|log|V| + E)$ con Fibonacci heap.

## Cota Mínima Trivial: ejemplo I (cont.)

Una variación al problema permite pesos negativos pero no permite ciclos con peso negativo. El algoritmo debe poder detectar ciclos con peso negativos. Sea $|V|$ el número de vértices, entonces podrían haber $|V|(|V|-1)$ arcos en $G$, y cualquier algoritmo debería inspeccionar cada uno de ellos para resolver la variación del problema. Si el algoritmo ignora alguno de los arcos, entonces se podría cambiar el peso haciendo que el camino más corto o un ciclo con peso negativo se perdiera, forzando al algoritmo a dar una respuesta errada. Entonces $\Omega(|V|^2)$, $(|V|(|V|-1))$ es una cota mínima para el peor caso de cualquier algoritmo que encuentra el problema del camino más corto con origen único.

## Cota Mínima Trivial: ejemplo 2

Otro ejemplo es la multiplicación de matrices de $n \times n$, lo que requiere que $n^2$ salidas sean producidas. Esto no dice nada acerca del número de multiplicaciones requeridas para resolver el problema, pero indica que la operación de salida debe realizarse al menos $n^2$ veces, y por lo tanto, la operación dominante debe realizarse al menos ese número de veces.

Lower bound trivial son en general de menos interés que cotas ajustadas que requieren de métodos más sofisticados. Sin embargo, en algunos casos, son los únicos lower bounds disponibles. Ya que son más fáciles de obtener, es recomendable intentar obtenerlos primeros.

## Juego del Adversario: Upper Bound

La idea es hacer jugar al algoritmo en contra del adversario para lograr la demostración.

$$\exists A, \forall I, A(I) = P(I) \wedge Time(A, I) \leq T_{upper}(|I|)$$

| **Algoritmo** | **Adversario** |
|---|---|
| Yo funciono y soy rapido | |
| | Oh sí , yo tengo una entrada $I$ para la cual no es así |
| Yo gano si dado $I$ (1) doy un salida correcta y (2) en el tiempo esperado. | |

Esto fue esencialmente lo que se hizo para las cotas máximas en los algoritmos de estructuras de datos.

## Juego del Adversario: lower bound (preguntas si y no)

$$\forall A, \exists I, A(I) \neq P(I) \vee Time(A, I) \geq T_{lower}(|I|)$$

| **Algoritmo** | **Adversario** |
|---|---|
| Hace preguntas con k posibles respuesta para obtener la mayor cantidad de información | Escoje una entrada<br><br>Respondo entregando la menor cantidad de información |
| Encuentra solución | |

El tiempo es el número de preguntas en el peor caso.

## Prueba del Juego del Adversario: si y no (cont.)

(i) El algoritmo quiere obtener la mayor cantidad de información para alcanzar mayor eficiencia.
(ii) En cambio el adversario quiere darle la menor información posible de manera de obtener el peor caso.
(iii) La regla del juego es la consistencia. El adversario puede hacer trampa pero no causar inconsistencia en el información dada.

(iv) Se puede ver como un forma de construir el peor caso para un algoritmo desconocido.

## Prueba del Juego del Adversario: ejemplo 1

**Problema:** Encontrar el mínimo y máximo en un arreglo desordenado $E$ usando índices de 1 a $n$.
**Enfoque**:
(i) Asuma el peor adversario: entrega la menor información.
(ii) Escoja una pregunta lo más balanceada posible, ejemplo, para comparaciones $x > y$, *true* si $x > y$ y *false* si $x \leq y$.
(iii) El problema es equivalente a responder la pregunta: Si el arreglo estuviera ordenado en orden creciente, el elemento más grande estaría en $E[n]$, el menor en $E[1]$ y la mediana en $E[\lceil n/2 \rceil]$.

## Prueba del Juego del Adversario: ejemplo 1 (cont.)

**Teorema:** Cualquier algoritmo que encuentre el mínimo y máximo de $n$ clave por comparación de claves debe hacer al menos $3n/2 - 2$ comparaciones de clave en el caso peor.

**Demostración**:

(i) Para conocer que una clave $v$ es la mayor, el algoritmo debe conocer que cada otra clave ha perdido su comparación (*false*). A la inversa, para saber que una clave $v$ es la menor, el algoritmo debe saber que cada otra clave gana la comparación (*true*).

(ii) Si contamos los *true* y los *false* como una unidad de información, entonces el algoritmo debe tener al menos $2(n-1)$ unidades de información para encontrar el mínimo y máximo.

(iii) Necesitamos determina cuántas comparaciones son requeridas (en el peor caso) para obtener las $2(n-1)$ unidades de información.

(iv) El adversario para obtener el peor caso, nos dará la menor cantidad de información posible.

## Prueba del Juego del Adversario: ejemplo 1 (cont.)

**Demostración: Argumento del adversario**: El adversario marca $+$ a cada elemento que puede ser el máximo y $-$ a cada elemento que puede ser el mínimo. Inicialmente, el adversario inicializa a todos los elementos con $+$ y $-$. Si el algoritmo compara dos elementos marcados con $+$ y $-$, entonces el adversario declara uno más pequeño, remueve el $+$ del más pequeño y el $-$ del más grande. En otros casos, el adversario responde de manera que a lo más una marca es removida. Por ejemplo, si el algoritmo compara un elemento no visto antes (con marcas $+$ y $-$) contra un elemento con marca $-$, entonces, el adversario le dice al algoritmo que el elemento marcado $-$ es menor que el marcado $+$ y $-$, removiendo entonces de este último el $-$.

## Prueba del Juego del Adversario: ejemplo 1 (cont.)

**Demostración (cont.)**:
(i) El algoritmo hace a lo más $n/2$ comparaciones de las claves no vistas anteriormente. Asumamos que $n$ es par. Entonces, se obtienen $n$ nuevas unidades de información con las $n/2$ comparaciones.
(ii) El algoritmo necesita $2(n-1) = 2n-2$, entonces se necesitan $n-2$ unidades de información adicionales. Por cada comparación se gana al menos una unidad de información, entonces necesitamos al menos $n-2$ comparaciones.
(iii) En total, el algoritmo requiere de al menos $n/2 + n - 2$ comparaciones. Si $n$ es impar, $\lceil 3n/2 - 3/2 \rceil$ comparaciones son necesarias.

## Prueba del Juego del Adversario: Ejemplo 2

En la búsqueda en un tabla ordenada, las preguntas son del tipo: ¿ Es la clave de búsqueda menos que este elemento? La respuesta obvia del adversario es siempre dar una respuesta de manera que la clave esté en la porción más grande de la lista. De esta manera a lo más la mitad de la tabla será eliminada en cada comparación, y al menos $\lceil logn \rceil$ comparaciones deberán ser hechas en el peor caso.

## Prueba del Juego del Adversario: Ejemplo 3

Considere una forma de comprobar el lower bound para el ordenamiento de un arreglo usando a un adversario. Para ello, note que existen $n!$ posibles permutaciones del arreglo de entrada. El adversario mantiene un lista $L$ de todas las permutaciones que son consistentes con las comparaciones que el algoritmo ha hecho hasta el momento. Al inicio $L$ contienen $n!$ permutaciones. La estrategia del adversario para responder a la pregunta ¿ Es $A[i] \leq A[j]$ verdadero? es la siguiente:

- Sea $L_{si}$ las permutaciones en $L$ donde $A[i] \leq A[j]$ y sea $L_{no}$ las permutaciones en $L$ donde $A[i] > A[j]$ ($L = L_{si} \cup L_{no}$).
- El adversario responde *si* cuando $|L_{si}| \geq |L_{no}|$. Posteriormente, el adversario actualiza $L$.
- Esta estrategia asegura que en cada paso, al menos la mitad de los elementos de $L$ permanece. Es decir, al menos la mitad de los elementos es removido. El algoritmo sigue hasta que $L = 1$, y el número de comparaciones posibles es al menos $\lceil log(n!) \rceil = \Theta(nlogn)$ en el peor caso.

## Prueba del Juego del Adversario: Ejemplo 4

Considere el problema de saber si el grafo es conectado.

- El adversario mantiene 2 grafos $Y$ y $M$. $Y$ es el grafo, inicialmente vacío, que contiene los arcos conocidos por el algoritmo que están en el grafo de entrada. $M$ es el grafo, inicialmente completo, que contiene los arcos que el algoritmo piensa que pueden estar en el grafo de entrada.

  Cuando el algoritmo pregunta acerca de un arco $e$, el cual está en $M$ y no en $Y$, el adversario usa el siguiente argumento: Si $M \setminus \{e\}$ está conectado entonces remover $e$ de $M$ y retornar 0, sino agregar $e$ a $Y$ y retornar 1. Note que ambos $Y$ y $M$ son consistentes con el adversario.

## Prueba del Juego del Adversario: Ejemplo 4 (cont.)

El adversario mantiene las siguiente invariantes: (1) $Y \subseteq M$ (2), $M$ está conectado, (3) Si $M$ tiene un ciclo, ninguno de sus arcos está en $Y$, (4) Y es acíclico y (5) Si $Y \neq M$, entonces $Y$ está desconectado.

Se puede pensar en la estrategia del adversario en términos del máximo spanning tree. La idea es considerar arcos uno a uno en order creciente de largo, si remover un arco desconecta el grafo entonces forma parte del spanning tree (se agrega al $Y$), sino elimínelo (remueva de $M$). Si el algoritmo examina todos los arcos, entonces $Y$ y $M$ son ambos iguales al máximo spanning tree del grafo completo, donde el peso de un arco es el tiempo cuando el algoritmo pregunta por él.

Si el algoritmos termina antes de examinar todos los ascos, existe al menos un arco en $M$ que no está en $Y$. Ya que el algoritmo no puede distinguir entre $M$ y $Y$, el algoritmo no puede dar la respuesta correcta. Entonces, se deben examinar todos los arcos.

## Prueba del Juego del Adversario: Ejemplo 5

Considere el problema de dado un arreglo $A$ de $n$ bits, encontrar si el substring 01 aparece en el string. ¿Se puede responder esta pregunta sin mirar todos los bits?

- Para $n$ impar, no se necesita mirar todos los bits. Lo primero que hay que hacer es mirar todos las posiciones que son pares: $A[2], A[4], \ldots A[n-1]$. Si $A[i] = 0$ y $A[j] = 1$ para algún $i < j$, entonces el patrón 01 está en el arreglo. Si se ven solo 1s seguidos por 0s no hay que seguir evaluando. Si cada elemento par es 0, no hay que mirar el $A[1]$ y si cada elemento par es 1 no hay que mirar $A[n]$. Entonces, en el peor caso, el algoritmo debe mirar $n-1$ elementos de $n$ bits.
- Para el caso de un número par de elementos en $A$, usamos la siguiente estrategia del adversario para demostrar que debe se deben mirar todos los bits.

## Prueba del Juego del Adversario: Ejemplo 5 (cont.)

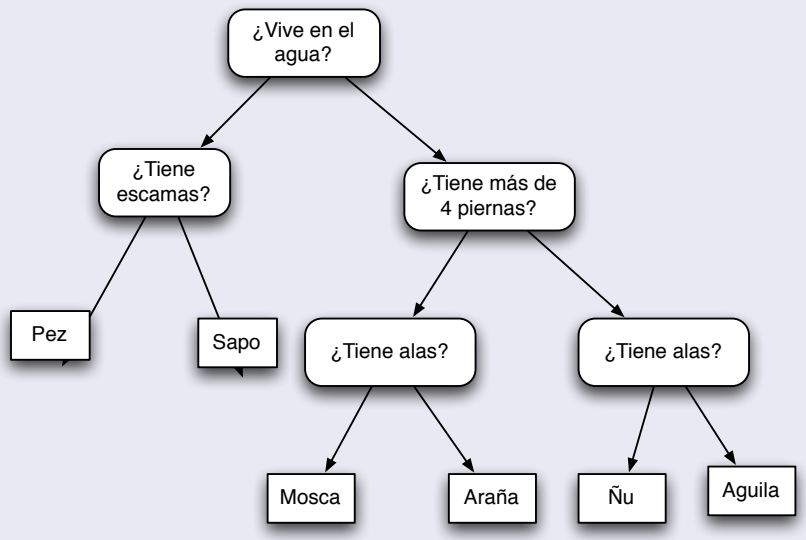
El adversario trata de producir una entrada $A$ sin el substring 01, los cuales tienen la forma de 111110000 . . . 0. El adversario mantiene dos índices $l$ y $r$ y pretende que el prefijo $A[1 \ldots l]$ contiene solo 1s y el sufijo $A[r \ldots n]$ solo 0s. Inicialmente $l = 0$ y $r = n$. Lo que está entre $l$ y $r$ representa para el adversario los bits desconocidos.
El adversario mantiene el invariante que $r - l$ es par. Cuando el algoritmo mira un bits entre $l$ y $r$, el adversario escoge cualquier valor que preserve la paridad de la porción intermedia y mueve $l$ o $r$. En particular, si se chequea por $i$ e $i \leq l$, entonces $A[i] = 1$, sino si $i \geq r$ entonces $A[i] = 0$, sino si $i - l$ es par entonces $A[i] = 0, r = i$ sino $A[i] = 1, l = i$.

Si el algoritmo no evalúa todos los bits a la derecha de $r$, el adversario puede reemplazar algún bit no examinado por 1. Similarmente, si no se examinan todos los que están a la izquierda de $l$, el adversario puede reemplazar algún bit no examinado por 0. Si no se examinan todos los que están entre $l$y $r$, deben existir al menos dos bits ($r - l$ es par) y el adversario puede colocar un 01.

# Árbol de Decisión

Los árboles de decisión son un modelo de computación donde cada nodo interno en el árbol en etiquetado con una consulta, la cual es una consulta acerca de la entrada. Los arcos de un nodo representan posibles respuestas. Cada hoja es entonces un salida. Entonces, la computación con un árbol de decisión parte de la raíz y termina en la hoja.

**Ejemplo 1:** Considere el caso de un juego de adivinanza, donde una persona piensa un animal y la otra trata con una serie de preguntar descubrir el animal.



## Árbol de Decisión (cont.)

**Ejemplo 2:** La búsqueda en un arreglo de enteros puede ser resuelta considerando que se ordena el arreglo (recordando la posición original del elemento en el arreglo) y se hace una búsqueda binaria. El árbol de búsqueda binaria en sí es un árbol de decisión.

(i) El tiempo de cómputo de un árbol de decisión de una entrada dada es el número de consultas realizadas desde la raíz hasta la hoja. Por lo que en el peor caso, el tiempo de computación está dado por la profundidad o altura del árbol. Esta decisión ignora otro tipo de operaciones que el algoritmo pueda llevar acabo y que no tengan nada que ver con la consulta. Pero en todo caso es un lower bound.
(ii) Los nodos del árbol puede no ser siempre binarios sino de un grado mayor, si es que las preguntas tienen más de 2 posibles respuestas. Entonces un árbol de decisión $k$-ario es un árbol que puede tener preguntas con $k$ respuestas y ese $k$ es constante.

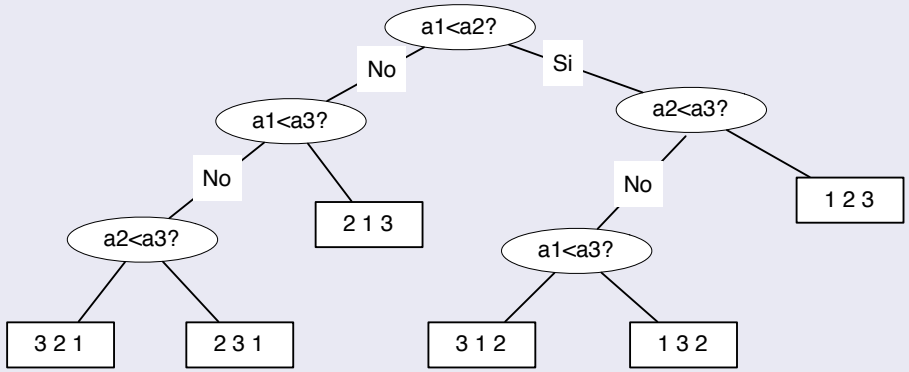
## Árbol de Decisión (cont.)

Una observación importante es que este modelo solo sirve para representar la computación basada en preguntas con respuestas de orden constante. De lo contrario alguien podría discutir que la búqueda no tiene cota mínima *logn* ya que podemos aplicar una función hash. El problema es que aplicar una función hash (*¿Cuál es el valor hash del valor* x*?*) no restringe el grado del árbol de decisión. Esto se aplica no solo a funciones hash sino a preguntas del tipo *¿Cuál(es) es el índice* i *(si existe alguno) para el cual* A[i] = x?

*Escoger el modelo correcto de computación es absolutamente esencial para demostrar lower bound*. Si el modelo es muy poderoso, entonces el problema puede tener un algoritmo trivial. Por el contrario, si se considera un modelo muy restrictivo, el problema puede no tener solución y el lower bound es irrelevante

## Ordenación: lower bound

Algoritmos de ordenamiento pueden ser vistos como árboles. Asuma un arreglo de 3 elementos. Entonces el siguiente árbol muestra el proceso de ordenamiento.




La profundidad del árbol representa el número de comparaciones desde la raíz hasta la hoja. Cada hoja representa una posible permutación.

## Ordenación: lower bound (cont)

**Teorema:** Cualquier árbol de decisión que ordene $n$ elementos debe tener una altura mayor a ($nlogn$).

**Demostración:** El árbol debe contener $\geq n!$ hojas, ya que hay $n!$ permutaciones posibles. Un árbol binario de altura $h$ tiene $\leq 2^h$ hojas. Así $n! \leq 2^h$. Entonces:

$$\begin{aligned} h &\geq log(n!) \\ &\geq log((n/e)^n)) \\ &= nlogn - nloge \\ &= \Omega(nlogn) \text{ Nota: log es monótona creciente} \end{aligned}$$

## Ordenación: lower bound (cont)

- ¿Existe algo más rápido para ordenar?

## Ordenación: lower bound (cont)

- ¿Existe algo más rápido para ordenar?
- No para el caso de ordenamiento basado en comparaciones.

## Ordenación: lower bound (cont)

- ¿Existe algo más rápido para ordenar?
- No para el caso de ordenamiento basado en comparaciones.
- ¿Y para otro modelo de computación?

## Ordenación: lower bound (cont)

- ¿Existe algo más rápido para ordenar?
- No para el caso de ordenamiento basado en comparaciones.
- ¿Y para otro modelo de computación?
- Considere el caso de un ordenamiento por conteo donde no se hacen comparaciones entre elementos:

  (i) Entrada: $A[1 \ldots n]$, donde $A[j] \in \{1, 2, 3, \ldots, k\}$

  (ii) Salida: $B[1 \ldots n]$ ordenado

  (iii) Almacenamiento auxiliar: $C[1 \ldots k]$

## Ordenación: lower bound (cont)

*CountingSort*$(A, n)$

```
begin
 for i ← 1 to k do
  C[i] ← 0
 enddo
 for j ← 1 to n do
  C[A[j]] ← C[A[j]] + 1
 enddo
 for i ← 2 to k do
  C[i] ← C[i] + C[i − 1]
 enddo
 for j ← n downto 1 do
  B[C[A[j]]] ← A[j]
  C[A[j]] ← C[A[j]] − 1
 enddo
end
```

## Ordenación: lower bound (cont)

| | *CountingSort*$(A, n)$ |
|---|---|
| | **begin** |
| $\Theta(k)$ | **for** $i \leftarrow 1$ **to** $k$ **do** |
| $\Theta(k)$ | $C[i] \leftarrow 0$ |
| | **enddo** |
| $\Theta(n)$ | **for** $j \leftarrow 1$ **to** $n$ **do** |
| $\Theta(n)$ | $C[A[j]] \leftarrow C[A[j]] + 1$ |
| | **enddo** |
| $\Theta(k)$ | **for** $i \leftarrow 2$ **to** $k$ **do** |
| $\Theta(k)$ | $C[i] \leftarrow C[i] + C[i-1]$ |
| | **enddo** |
| $\Theta(n)$ | **for** $j \leftarrow n$ **downto** 1 **do** |
| $\Theta(n)$ | $B[C[A[j]]] \leftarrow A[j]$ |
| $\Theta(n)$ | $C[A[j]] \leftarrow C[A[j]] - 1$ |
| | **enddo** |
| | **end** |
| $\Theta(k+n)$ | |

## Ordenación: lower bound (cont)

- Si $k = O(n)$, entonces el ordenamiento por conteo es de orden $\Theta(n)$

## Ordenación: lower bound (cont)

- Si $k = O(n)$, entonces el ordenamiento por conteo es de orden $\Theta(n)$
- Pero si necesitaba al menos $\Omega(nlogn)$ comparaciones. ¿Dónde está la falacia?

## Ordenación: lower bound (cont)

- Si $k = O(n)$, entonces el ordenamiento por conteo es de orden $\Theta(n)$
- Pero si necesitaba al menos $\Omega(nlogn)$ comparaciones. ¿Dónde está la falacia?
- Ordenamiento por conteo no es un ordenamiento por comparación. De hecho, no se hace ninguna comparación.

## Problema de Reducción

**Fundamento**: Si un problema $Q$ puede ser "reducido" a un problema $P$, entonces $Q$ es al menos tan fácil como $P$, o lo que es equivalente, $P$ es al menos tan difícil como $Q$.
**Reduccón de $Q$ a $P$**: Diseñe un algoritmo $Q$ usando un algoritmo $P$ como una subrutina.
**Idea:** Si el problema $P$ es al menos tan difícil como el problema $Q$, entonces el lower bound de $Q$ es también el lower bound de $P$. Consecuentemente, encontrar el problema $Q$ con un lower bound conocido que puede ser reducido al problema $P$ en cuestión.

## Problema de Reducción (cont.)

**Ejemplo 1:** Suponga que se quiere probar el lower bound para el problema del convex hull de un conjunto de $n$ puntos en el plano. Para eso, se puede usar la reducción de ordenamiento a convex hull.

Para ordenar la lista de $n$ números $\{a, b, c, \dots\}$, transforme el conjunto a un conjunto de $n$ puntos $\{(a, a^2), (b, b^2), (c, c^2), \dots\}$. Se puede pensar en los número originales como un conjunto de puntos en una línea de número reales horizontales, y la transformación como el levantamiento de estos puntos a una parárabola $y = x^2$. Entonces se calcula el cómputo del convex hull de los puntos en la parábola. Finalmente se obtiene la lista ordenada, se obtienen las primeras coordenadas en cada vértice del convex, comenzando desde el vértice más a la izquierda y recorriendo el convex en sentido contrario al reloj. Entonces

$$\begin{aligned} T_{sort}(n) &\leq T_{convex\_hull}(n) + O(n) \\ T_{convex\_hull}(n) &\geq T_{sort}(n) - O(n) \end{aligned}$$

Ahora como $T_{sort}(n)$ toma $\Theta(nlogn)$, entonces cualquier algoritmo que resuelva el $T_{convex\_hull}(n)$ toma al menos $\Omega(nlogn)$ en el peor caso.

## Problema de Reducción (cont.)

**Ejemplo 2:** Para conocer el lower bound para encontrar la *cobertura mínima Euclediana para n puntos en el plano* (STP), use el problema del *elemento único* (UE). Esto siginifica se usa STP para resolver EU cuyos algoritmos toman al menos $\Omega(nlogn)$. Por lo tanto, cualquier algoritmo STP toma $\Omega(nlogn)$.

Suponga que uno quiere saber si dos pares de números en un arreglo $x_1, x_2, \ldots, x_n$ son iguales. Esto se pude resolver por cualquier algoritm STP con los puntos $(x_1, 0), (x_2, 0), \ldots, (x_n, 0)$. Los dos puntos más cercanos se saben que quedan unidos por uno de los $n - 1$ arcos del STP, así uno puede recorrer en tiempo linear estar arcos y determinar si el arco tiene largo 0. Ese arcos existe si y solo si hay dos valores iguales. Entonces si un algoritmo del STP corre en menos que $\Omega(nlgn)$ en el peor caso, entonces un algoritmo del UE debería también poderse resolver en un tiempo menor para el peor caso.

## Ejercicio

Asuma que tenemos un problema $P_1$ que tiene un lower bound $\Omega(n)$ y otro problema $P_2$ con lower bound $\Omega(nlogn)$. También se sabe que para reducir $P_1$ a $Q$ requiero $O(nlogn)$ y que para reducir $P_2$ a $Q$ requiero $O(n)$. ¿Qué puede decir del lower bound de $Q$?

La reducción respecto a $P_1$ queda:

$$T_{P_1}(n) \leq T_Q(n) + O(nlogn)$$
$$O(n) \leq T_Q(n) + O(nlogn)$$
$$O(n) - O(nlogn) \leq T_Q(n)$$

Esta reducción entonces no se puede aplicar ya que tiene un costo mayor que el costo de resolver el problema.
Por otro lado, respecto a $P_2$:

$$T_{P_2}(n) \leq T_Q(n) + O(n)$$
$$O(nlogn) \leq T_Q(n) + O(n)$$
$$O(nlogn) - O(n) \leq T_Q(n)$$

Luego, el lower bound de $Q$ queda expresado por $O(nlogn)$.

## Ejercicio (cont.)

Pruebe que para verificar si el patrón 11 se encuentra en una secuencia de $n$ bits se debe chequear todos los bits cuando $n \ mod \ 3 = 1$.