DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

# CC4102 Diseño y Análisis de Algoritmos


Prof. Benjamin Bustos

*Departamento de Ciencias de la Computación*
*Facultad de Ciencias Físicas y Matemáticas*
*Universidad de Chile*

# Capítulo 1

Conceptos básicos y complejidad

# Proceso de diseño y análisis

- Algoritmo
  - Procedimiento computacional bien definido
  - Toma como entrada uno o varios valores
  - Produce como salida uno o más valores
  - El algoritmo es la serie de pasos que se realizan para transformar la entrada en la salida
- Estructura de datos
  - Formas de organizar la información en la memoria del computador

# Proceso de diseño y análisis

- Un algoritmo es correcto si para cada entrada posible termina con la salida correcta
  - Algunos algoritmos “incorrectos” pueden ser útiles (algoritmos aproximados)
- Técnicas de diseño de un algoritmo
  - Iterativos
  - Recursivos
  - Dividir para reinar
  - Programación dinámica

# Proceso de diseño y análisis

- Principal foco de este curso: eficiencia de los algoritmos
  - Para el mismo problema, pueden haber distintos algoritmos con distinta eficiencia
  - Eficiencia significa principalmente velocidad, pero en general se refiere a cuántos recursos utiliza para generar la salida
    - Tiempo
    - Espacio

# Proceso de diseño y análisis

- Tamaño máximo de un problema (f(n) ms)

| | 1 seg | 1 min | 1 hora | 1 día | 1 mes | 1 año | 1 siglo |
|---|---|---|---|---|---|---|---|
| log n | | | | | | | |
| n^1/2 | | | | | | | |
| n | 1000 | 6 x 10^4 | 3.6 x 10^6 | | | | |
| n log n | 140 | 4893 | 2.0 x 10^5 | | | | |
| n^2 | 31 | 244 | 1897 | | | | |
| n^3 | 10 | 39 | 153 | | | | |
| 2^n | 9 | 15 | 21 | | | | |
| n! | | | | | | | |

# Proceso de diseño y análisis

- ¿Cómo se mide el desempeño de un algoritmo?
  - Cantidad de recursos que consume
    - Tiempo de CPU
      - Número de instrucciones proporcionales
    - Número de accesos a disco (secuencial/aleatorio)
    - Cantidad de comunicación
    - Cantidad de memoria

# Proceso de diseño y análisis

- ¿Cómo se mide el desempeño de un algoritmo?
  - Ejemplo 1: Cálculo del mínimo

```
Minimum(A) // Tamaño de A == n
min <- A[1]
for i = 2 to n
     if A[i] < min  // instrucción representativa
         min <- A[i]
```

# Proceso de diseño y análisis

- ¿Cómo se mide el desempeño de un algoritmo?
  - Ejemplo 2: Bubblesort

```
T({1,2,3}) = 0
T({1,3,2}) = 1
T({3,2,1}) = 3
...
```

# Proceso de diseño y análisis

- Definiciones:

$$T_A(x) = \text{tiempo del algoritmo } A \text{ para la entrada } x.$$

$$T_A(n) = \max_{|x|=n} T_A(x) \text{ (PEOR CASO).}$$

$$T_A(x) = \sum_{|x|=n} p(x) T_A(x) \text{ (CASO PROMEDIO).}$$

  - Para caso promedio
    - Es necesario definir función de probabilidad p(x)
    - No es confiable
    - Es más complicado de calcular

# Proceso de diseño y análisis

- Notación O

Definición: $f(n)$ es $O(g(n))$ si $\exists n_0, c > 0$ tal que $\forall n > n_0,\ f(n) \leq c \cdot g(n)$.

- Notas:
  - Puede que para ciertos valores de n no sea cierta la desigualdad
  - f(n) es proporcional a g(n)

# Proceso de diseño y análisis

- Ejemplos:

$$
\begin{aligned}
n^2 - 3n + 12\log n - 7 &= O(n^2) \\
3n^2 + 17n + 3 &= O(n^2) \\
&= O(n^3) \\
\log^d n = O(n^\alpha)\ \forall d > 0, \alpha > 0 & \qquad (\log n \text{ es } O(\sqrt{n})) \\
2^{2n} = O(2^n)? &
\end{aligned}
$$

# Proceso de diseño y análisis

- Otras definiciones

$$f(n) \text{ es } \Omega(g(n)) \text{ si } \exists n_0, c > 0 \text{ tal que } \forall n > n_0, f(n) \geq c \cdot g(n).$$

  - Ejemplo: “Ordenar es Ω(n log n)”. Se refiere al problema, no al algoritmo.

$$f(n) \text{ es } o(g(n)) \Leftrightarrow f(n) \text{ no es } \Omega(g(n)).$$

$$f(n) \text{ es } w(g(n)) \Leftrightarrow f(n) \text{ no es } o(g(n)).$$

$$f(n) \text{ es } \Theta(g(n)) \text{ si } f(n) \text{ es } O(g(n)) \text{ y es } \Omega(g(n)).$$

# Proceso de diseño y análisis

- Algunas fórmulas útiles

Aproximación de Stirling:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

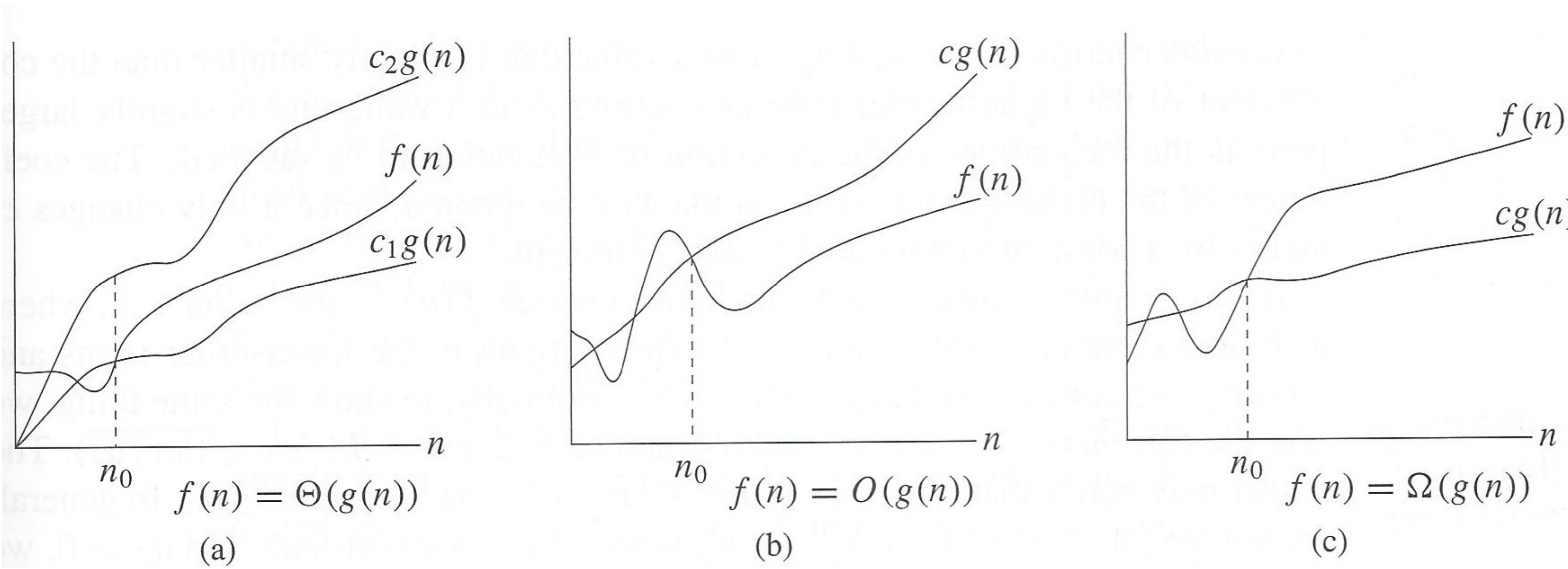
$$n! = o(n^n)$$

$$n! = \omega(2^n)$$

$$\log(n!) = \Theta(n\log n)$$

# Proceso de diseño y análisis

- Ejemplos notación O



# Proceso de diseño y análisis

- Resumen

$$
\begin{aligned}
O &: \quad f \preccurlyeq g \\
\Omega &: \quad f \succcurlyeq g \\
o &: \quad f \prec g \\
\omega &: \quad f \succ g \\
\Theta &: \quad f \approx g
\end{aligned}
$$

- Ejemplo: “Ordenar es $\Theta$(n log n)” (problema resuelto)

# Técnicas para demostrar cotas inferiores

- Técnica: Estrategia del adversario
  - Adversario: va construyendo el peor caso posible
  - Algoritmo: “funciona” lo mejor posible
- Ejemplo 1: calcular el mínimo de un arreglo
  - Algoritmo básico: n-1 comparaciones
  - Esto es óptimo
    - Lo mismo para calcular el máximo

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo
  - Mínimo: n-1 comparaciones
  - Máximo: n-2, se puede obviar el mínimo
  - Total: 2n-3 (cota superior)
  - ¿Cota inferior mejor?

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo
  - Sean las siguientes variables:
    - O: los elementos todavía no comparados
    - G: los elementos que ganaron todas sus comparaciones hasta ahora
    - P: los elementos que perdieron todas sus comparaciones hasta ahora
    - E: los valores eliminados (que perdieron al menos una comparación y ganaron al menos una comparación)

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo
  - Observaciones
    - El vector (o, g, p, e) describe el estado de cualquier algoritmo
    - Siempre se tiene que o + g + p + e = n
    - Al principio la tupla es (o,g,p,e) = (n,0,0,0)
    - Un algoritmo correcto debe terminar con la tupla (o,g,p,e) = (0,1,1,n-2)

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo
  - Después una comparación a?b en cualquier algoritmo del modelo de comparación (o, g, p, e), el vector cambia en función del resultado de la manera siguiente:

| | $a \in O$ | $a \in G$ | $a \in P$ | $a \in E$ |
|---|---|---|---|---|
| $b \in O$ | $o-2, g+1, p+1, e$ | $o-1, p, e+1$<br>$o-1, g, p+1, e$ | $o-1, g, p, e+1$<br>$o-1, g+1, p, e$ | $o-1, g+1, p, e$<br>$o-1, g, p+1, e$ |
| $b \in G$ | | $o, g-1, p, e+1$ | $o, g, p, e$<br>$o, g-1, p-1, e+2$ | $o, g, p, e$<br>$o, g-1, p, e+1$ |
| $b \in P$ | | | $o, g, p-1, e+1$ | $o, g, p, e$<br>$o, g, p-1, e+1$ |
| $b \in E$ | | | | $o, g, p, e$ |

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo
  - En algunas configuraciones, el cambio del vector estado depende del resultado de la comparación: un adversario puede maximizar la complejidad del algoritmo eligiendo el resultado de cada comparación. En la siguiente tabla se marcan las opciones que maximizan la complejidad del algoritmo

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: calcular el mínimo y el máximo

| | $a \in O$ | $a \in G$ | $a \in P$ | $a \in E$ |
|---|---|---|---|---|
| $b \in O$ | $o-2, g+1, p+1, e$ | $o-1, p, e+1$<br>$\mathbf{o-1, g, p+1, e}$ | $o-1, g, p, e+1$<br>$\mathbf{o-1, g+1, p, e}$ | $o-1, g+1, p, e$<br>$o-1, g, p+1, e$ |
| $b \in G$ | | $o, g-1, p, e+1$ | $\mathbf{o, g, p, e}$<br>$o, g-1, p-1, e+2$ | $\mathbf{o, g, p, e}$<br>$o, g-1, p, e+1$ |
| $b \in P$ | | | $o, g, p-1, e+1$ | $\mathbf{o, g, p, e}$<br>$o, g, p-1, e+1$ |
| $b \in E$ | | | | $o, g, p, e$ |

  - floor(n/2) transiciones de O a (G U P), y
  - n-2 transiciones de (G U P) a E
  - Complejidad en el peor caso:

$$\lceil 3n/2 \rceil - 2 = 3n/2 + O(1) \text{ comparaciones}$$

# Técnicas para demostrar cotas inferiores

```
Algoritmo MinMax(A)
1 Dividir A en n/2 pares (si n impar, elemento x extra).
2. Comparar los dos elementos de cada par.
3. Poner los elementos superiores en el grupo S, y los
elementos inferiores en el grupo I.
4. Calcular el mínimo m del grupo I con el algoritmo
básico, que realiza n/2 – 1 comparaciones
5. Calcular el máximo M del grupo S, misma complejidad.
6. Si n es par, m y M son respectivamente el mínimo y el
máximo de A.
7. Sino, si x < m, x y M son respectivamente el mínimo y
el máximo de A.
8. Sino, si x > M, m y x son respectivamente el mínimo y
el máximo de A.
9. Sino, m y M son respectivamente el mínimo y el máximo
de A.
```

# Técnicas para demostrar cotas inferiores

- Complejidad del algoritmo MinMax
  - Si n es par: $3n/2 - 2$
  - Si n es impar: $3n/2 + ½$
  - En ambos casos el algoritmo es $3n/2 + O(1)$

# Técnicas para demostrar cotas inferiores

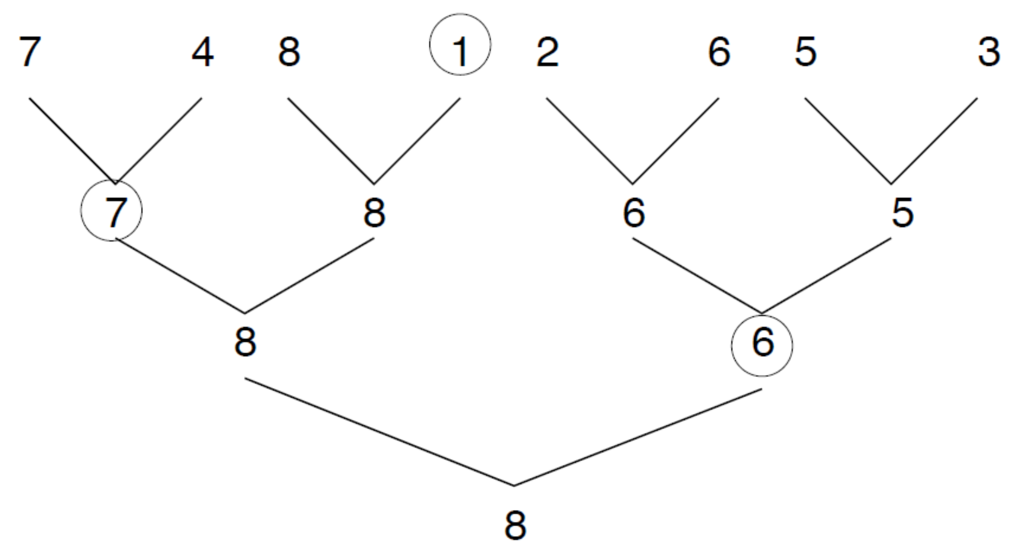
- Ejemplo 3: máximo y el segundo máximo
  - Usando algoritmo básico:
    - Encontrar el máximo: n-1 comparaciones
    - Encontrar segundo máximo: n-2 comparaciones
    - Total: 2n-3 comparaciones
    - ¿Se puede hacer mejor?

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: máximo y el segundo máximo
  - Observación: durante la fase de obtención del máximo, se obtuvo información que puede ser utilizada para encontrar el segundo máximo
    - El segundo máximo tuvo que haber "perdido" su comparación contra el máximo, independiente del algoritmo de búsqueda utilizado
    - Idea: buscar el segundo máximo sólo entre aquellos elementos que perdieron contra el máximo

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: máximo y el segundo máximo
  - Algoritmo del torneo:




  - Elementos marcados perdieron con el máximo

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: máximo y el segundo máximo
  - Algoritmo del torneo:
    - Número de elementos que perdieron contra el máximo es la altura del árbol, log(n)
    - Usando algoritmo básico entre estos elementos usa log(n)-1 comparaciones
    - Costo total: n+log(n)-2 comparaciones
    - ¿Es esto óptimo?

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: máximo y el segundo máximo
  - Cota inferior para el problema usando estrategia del adversario
    - Idea básica: ajustar los valores de los elementos (SIN modificar las decisiones tomadas con anterioridad) de forma de forzar que al menos hayan log(n) perdedores con el máximo

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: máximo y el segundo máximo
  - El adversario (B) mantiene “pesos” por cada elemento
  - El algoritmo (A) usa los pesos para comparar. Los pesos son información auxiliar usada sólo por B y no son parte de los datos de A
  - Los pesos son modificados por B mientras A se está ejecutando
  - Inicialmente B fija todos los pesos en 1, por lo que su suma es n. B mantiene esta suma como invariante durante toda la ejecución del algoritmo

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: calcular el máximo y el segundo máximo
  - Si A compara x con y, B ajusta los pesos y entrega una respuesta:
    - (i) Si W(x) > W(y), B responde "x > y" y cambia los pesos a W'(x) = W(x) + W(y), y W'(y) = 0
    - (ii) Si W(x) == W(y) > 0, B hace como en (i)
    - (iii) Sino, W(x) == W(y) == 0, y B responde algo que no entre en conflicto con respuestas pasadas y no cambia los pesos

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: calcular el máximo y el segundo máximo
  - Se tiene que:
    - (a) $W(x) = 0$ ssi x perdió en una comparación
    - (b) Si $W(x) > 0$, x no ha perdido aún y podría ser el máximo
    - (c) La suma de los pesos es siempre n
  - Además, A sólo termina en forma correcta cuando hay un único x tal que $W(x) > 0$, con $W(x) = n$
  - ¿Cuántos incrementos $W_1(x), W_2(x), \ldots, W_k(x)$ ha tenido este único x (el máximo) desde su peso inicial de 1?

# Técnicas para demostrar cotas inferiores

- Ejemplo 3: calcular el máximo y el segundo máximo
  - Se sigue que k es al menos log(n), y cada incremento se debe a ganarle a un potencial segundo máximo
  - Por lo tanto, al menos hay log(n) perdedores contra el máximo
- Cota inferior es $\Omega(n + O(\log n))$
  - Esto implica que algoritmo del torneo es óptimo

# Técnicas para demostrar cotas inferiores

- Técnica: Teoría de la información
  - Árbol de decisión:
    - Árbol en donde cada nodo interno está etiquetado con una consulta (pregunta sobre los datos de entrada)
    - Las aristas que salen de un nodo corresponden a las distintas respuestas posibles a la pregunta
    - Cada hoja del árbol se etiqueta con una salida (resultado)

# Técnicas para demostrar cotas inferiores

- Para calcular con un árbol de decisión:
  - Se comienza en la raíz del árbol
  - Dependiendo de la respuesta en cada nodo interno visitado, se continúa por la rama respectiva
  - Cuando se llega una hoja, se retorna su etiqueta como resultado
- Tiempo de ejecución del algoritmo en el árbol de decisión es el número de consultas realizadas desde la raíz hasta llegar a la hoja

# Técnicas para demostrar cotas inferiores

- El número de decisiones realizadas es una cota inferior del tiempo total que requerirá el algoritmo
  - Cota inferior: altura mínima del árbol de decisión
- Las cotas basadas en árboles de decisión se fundamentan en la siguiente idea:
  - “Las respuestas a las consultas deben entregar información suficiente para especificar cualquier resultado posible”

# Técnicas para demostrar cotas inferiores

- Lema: sea D un árbol binario de altura h. D tiene a lo más 2^h hojas.
  - Demostración: por inducción
    - Si h = 0, el árbol tiene un solo nodo que necesariamente es una hoja (caso base)
    - En el caso general, se tiene una raíz, que no puede ser una hoja, que posee un subárbol izquierdo y derecho, cada uno con una altura máxima de h-1. Por hipótesis de inducción, los subárboles pueden tener a lo más 2^(h-1) hojas, dando un total de a lo más 2^h hojas entre ambos subárboles. QED

# Técnicas para demostrar cotas inferiores

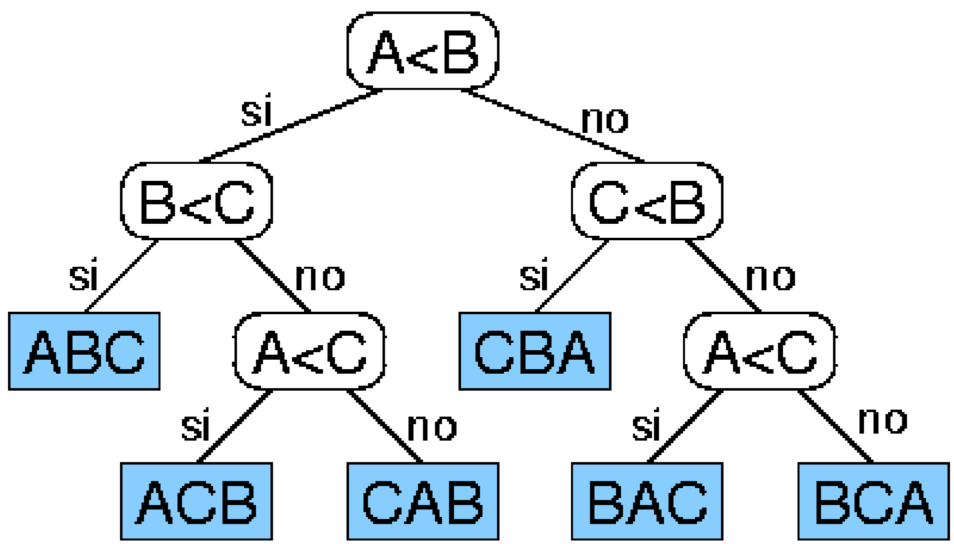
- Lema: un árbol binario con H hojas debe tener una profundidad de al menos log(H)
  - Demostración: directo del lema anterior
- Si un problema tiene n resultados distintos, su árbol de decisión tiene al menos n hojas
- Si cada pregunta tiene dos respuestas posibles, entonces la altura del árbol de decisión debe ser al menos $\log(n) = \Omega \log(n)$

# Técnicas para demostrar cotas inferiores

- Ejemplo 1: Ordenamiento basado en comparaciones
  - Inserción, Selección, Burbuja (cota superior O(n^2) )
  - Mergesort (cota superior O(n log n) )
  - Heapsort (cota superior O(n log n) )
  - Quicksort (cota superior O(n^2) )

# Técnicas para demostrar cotas inferiores

- Ejemplo 1: Ordenamiento basado en comparaciones
  - Árbol de decisión:



# Técnicas para demostrar cotas inferiores

- Ejemplo 1: Ordenamiento basado en comparaciones
  - Número de hojas del árbol de decisión: n!
  - Altura del árbol de decisión >= log(n!)
  - Usando aproximación de Stirling:

$$\lceil \log(n!) \rceil > \lceil \log \left(\frac{n}{e}\right)^n \rceil = \lceil n \log(n) - n \log(e) \rceil = \Omega(n \log n)$$

  - Cota inferior $\Omega$(n log n) implica que Mergesort y Heapsort son óptimos bajo este modelo

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: Búsqueda en arreglo ordenado
  - Búsqueda secuencial
    - Peor caso: n comparaciones
    - Caso promedio: O(n) comparaciones

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: Búsqueda en arreglo ordenado
  - Búsqueda binaria
    - Arreglo A de tamaño n, en donde se tiene almacenado el conjunto de elementos ordenados de menor a mayor.
    - Para buscar un elemento x en A:
      - Buscar el índice m de la posición media del arreglo. Inicialmente, $m = n/2$.
      - Si $a[m] = x$ se encontró el elemento (fin de la búsqueda),
      - En caso contrario, se sigue buscando en el lado derecho o izquierdo del arreglo dependiendo si $a[m] < x$ o $a[m] > x$ respectivamente

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: Búsqueda en arreglo ordenado
  - Costo de la búsqueda binaria:
    - $T(n) = 1 + T(n/2)$ (aproximadamente)
    - $T(n) = 2 + T(n/4)$
    - $T(n) = 3 + T(n/8)$
    - ...
    - $T(n) = k + T(n/2^k)$ para todo $k>=0$
    - Eligiendo $k = \log n \Rightarrow T(n) = \log n + T(1) = 1 + \log n = O(\log n)$.

# Técnicas para demostrar cotas inferiores

- Ejemplo 2: Búsqueda en arreglo ordenado
  - Cota inferior usando árboles de decisión:
    - Modelo: comparaciones entre elementos del arreglo y valor buscado
    - Número de hojas del árbol de decisión: n+1
    - Altura del árbol de decisión >= log(n+1)
    - Cota inferior $\Omega(\log n)$
  - Esto implica que la búsqueda binaria es óptima

# Técnicas para demostrar cotas inferiores

- Técnica: Reducción
  - Se tienen dos problemas, A y B
  - Si se puede mostrar que:
    - Un algoritmo para A se puede modificar para resolver B, y
    - no se añade “demasiado” al tiempo de ejecución de dicho algoritmo
  - En este caso, una cota inferior para el problema B es válida también para el problema A

# Técnicas para demostrar cotas inferiores

- Ejemplo: Multiplicación de dos matrices
  - Sean dos matrices simétricas. ¿Es posible multiplicarlas más rápido que dos matrices arbitrarias?
    - Respuesta: No, y lo vamos a demostrar usando reducción
  - Sea ArbM el problema de calcular el producto de dos matrices arbitrarias (problema B)
  - Sea SymM el problema de calcular el producto de dos matrices simétricas (problema A)

# Técnicas para demostrar cotas inferiores

- Ejemplo: Multiplicación de dos matrices
  - Es obvio que SymM no es más difícil que ArbM (dado que SymM es un caso particular de ArbM)
  - Supuesto: se dispone de un algoritmo para resolver SymM
  - Reducción: hay que mostrar que se puede ocupar dicho algoritmo como una caja negra (black-box) para resolver el problema general ArbM

# Técnicas para demostrar cotas inferiores

- Ejemplo: Multiplicación de dos matrices
  - Sean M y N matrices arbitrarias de tamaño n x n
  - Considere la expresión (matrices 2n x 2n):

$$\begin{pmatrix} 0 & M \\ M^T & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & N^T \\ N & 0 \end{pmatrix} = \begin{pmatrix} MN & 0 \\ 0 & M^T N^T \end{pmatrix}$$

  - La reducción sigue del hecho que las dos matrices a la izquierda son simétricas
    - Se usa algoritmo para SymM para calcular el producto
    - La esquina superior izquierda contiene el resultado para ArbM

# Técnicas para demostrar cotas inferiores

- Ejemplo: Multiplicación de dos matrices
  - Teorema: Si hay un algoritmo que calcula el producto de dos matrices simétricas n x n en tiempo O(T(n)), tal que T(2n)=O(T(n)), entonces hay un algoritmo para calcular el producto de dos matrices arbitrarias n x n en tiempo O(T(n) + n^2)
  - Dem.: Usamos el algoritmo para SymM para calcular la multiplicación como se mostró previamente. Toma tiempo O(n^2) calcular las transpuestas de M y N, y toma T(2n) multiplicarlas

# Técnicas para demostrar cotas inferiores

- Ejemplo: Multiplicación de dos matrices
  - Nota: T(2n)=O(T(n)) no es muy restrictivo, por ejemplo cualquier polinomio lo satisface
  - Por reducción: una cota inferior para T(n) es la cota inferior para multiplicar dos matrices arbitrarias (mejor cota conocida es $\Omega$(n^2))
    - Esto implica que es imposible utilizar las propiedades simétricas de una matriz para obtener un algoritmo asintóticamente mejor para multiplicar matrices

# Ecuaciones de recurrencia

- Cuando un algoritmo contiene una llamada recursiva, su tiempo de ejecución se puede describir con una ecuación de recurrencia
- Ejemplo: Mergesort

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),\ T(1) = \Theta(1)$$

- Consideraciones:
  - T(n) sólo definido para valores de n entero
  - Condiciones de borde: T(constante) = $\Theta(1)$

# Ecuaciones de recurrencia

- Recurrencia telescópica

$$\begin{aligned} X_{n+1} &= X_n + a_n \\ &= X_0 + \sum_{i=0}^{n} a_i \end{aligned}$$

  - Ejemplo:

$$\begin{aligned} X_{n+1} &= n + X_n, \;\; X_0 = 0 \\ X_{n+1} &= 0 + \sum_{i=0}^{n} i \\ X_{n+1} &= \frac{n(n+1)}{2} \\ \therefore X_n &= \frac{n(n-1)}{2} \end{aligned}$$

# Ecuaciones de recurrencia

- Ejemplo:

$$
\begin{aligned}
X_{n+1} &= b_n X_n + a_n \\
\frac{X_{n+1}}{\prod_{i=0}^{n} b_i} &= \frac{X_n}{\prod_{i=0}^{n-1} b_i} + \frac{a_n}{\prod_{i=0}^{n} b_i} \\
Y_{n+1} &= \frac{X_{n+1}}{\prod_{i=0}^{n} b_i}, \quad c_n = \frac{a_n}{\prod_{i=0}^{n} b_i} \\
\Rightarrow Y_{n+1} &= Y_n + c_n \\
Y_{n+1} &= Y_0 + \sum_{i=0}^{n} c_i
\end{aligned}
$$

# Ecuaciones de recurrencia

- Ejemplo:

Caso particular $b_i = b$ (constante)

$$\begin{aligned} Y_{n+1} &= Y_0 + \sum_{i=0}^{n} c_i \\ \Rightarrow X_{n+1} &= b^{n+1} X_0 + b^{n+1} \sum_{i=0}^{n} \frac{a_i}{b^{i+1}} \\ &= b^{n+1} X_0 + \sum_{i=0}^{n} b^{n-i} a^i \end{aligned}$$

# Ecuaciones de recurrencia

- Ejemplo:

Caso Torres de Hanoi:

$$\begin{aligned} h_n &= h_{n-1} + 1 + h_{n-1},\ h_0 = 0 \\ h_{n+1} &= 2h_n + 1 \\ \Rightarrow h_{n+1} &= 2^{n+1}h_0 + \sum_{i=0}^{n} 2^{n-i} \\ \therefore h_{n+1} &= 2^{n+1} - 1,\ h_n = 2^n - 1 \end{aligned}$$

# Ecuaciones de recurrencia

- Recurrencias lineales homogéneas
  - Son ecuaciones de la forma

$$a_k X_{n+k} + a_{k-1} X_{n+k-1} + \ldots + a_1 X_{n+1} + a_0 X_n = 0$$

  - Las soluciones de este tipo de recurrencias son combinaciones lineales de la forma $X_n=\lambda^n$

$$a_k \lambda^{n+k} + a_{k-1} \lambda^{n+k-1} + \ldots + a_1 \lambda^{n+1} + a_0 \lambda^n = 0$$

$$\lambda^n \left( a_k \lambda^k + a_{k-1} \lambda^{k-1} + \ldots + a_1 \lambda + a_0 \right) = 0$$

# Ecuaciones de recurrencia

- Recurrencias lineales homogéneas
  - Polinomio característico y ecuación característica:

$$a_k\lambda^k + a_{k-1}\lambda^{k-1} + \ldots + a_1\lambda + a_0 = 0$$

  - Se resuelve el polinomio característico y se obtienen k raíces
  - La solución es de la forma

$$X_n = c_1\lambda_1^n + c_2\lambda_2^n + \ldots + c_k\lambda_k^n$$

# Ecuaciones de recurrencia

- Recurrencias lineales homogéneas
  - Para encontrar las constantes debe resolverse el siguiente sistema lineal:

$$
\begin{aligned}
X_0 &= c_1\lambda_1^0 + c_2\lambda_2^0 + \ldots + c_k\lambda_k^0 \\
X_1 &= c_1\lambda_1^1 + c_2\lambda_2^1 + \ldots + c_k\lambda_1^0 \\
&\vdots \\
X_{k-1} &= c_1\lambda_1^{k-1} + c_2\lambda_2^{k-1} + \ldots + c_k\lambda_1^{k-1}
\end{aligned}
$$

# Ecuaciones de recurrencia

- Recurrencias lineales homogéneas
  - Ejemplo: Fibonacci

$$f_{n+2} = f_{n+1} + f_n, \; f_0 = 0, \; f_1 = 1$$

$$\begin{aligned} f_{n+2} - f_{n+1} - f_n &= 0 \\ \Rightarrow \lambda^2 - \lambda - 1 &= 0 \end{aligned}$$

$$\begin{aligned} \lambda_1 &= \frac{1+\sqrt{5}}{2} \approx 1.618 = \phi \\ \lambda_2 &= \frac{1-\sqrt{5}}{2} \approx -0.618 = \hat{\phi} \end{aligned}$$

# Ecuaciones de recurrencia

- Recurrencias lineales homogéneas
  - Ejemplo: Fibonacci

$$f_n = c_1 \phi^n + c_2 \hat{\phi}^n$$

$$f_0 = 0 = c_1 + c_2 \Rightarrow c_2 = -c_1$$

$$f_n = c_1 \left( \phi^n - \hat{\phi}^n \right)$$

$$f_1 = 1 = c_1 \left( \phi - \hat{\phi} \right) \Leftarrow c_1 = \frac{1}{\sqrt{5}}$$

$$f_n = \frac{1}{\sqrt{5}} \left( \phi^n - \hat{\phi}^n \right) = O\left( \phi^n \right)$$

# Ecuaciones de recurrencia

- Teorema Maestro

La recurrencia $T(n) = kn + pT(\frac{n}{q})$ tiene las siguientes soluciones:

$$\begin{aligned} O(n^{\log_p q}) &\quad \text{si} \quad p > q \\ O(n \log n) &\quad \text{si} \quad p = q \\ O(n) &\quad \text{si} \quad p < q \end{aligned}$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - Desenrollando la ecuación

$$T(n) = kn + p \cdot T\left(\frac{n}{q}\right) = kn + p \cdot \left(k \cdot \frac{n}{q} + p \cdot T\left(\frac{n}{q^2}\right)\right)$$

$$T(n) = kn \cdot \left(1 + \frac{p}{q}\right) + p^2 \cdot T\left(\frac{n}{q^2}\right)$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - En general se tiene que (*)

$$T(n) = kn \cdot \left( 1 + \frac{p}{q} + \left(\frac{p}{q}\right)^2 + \ldots + \left(\frac{p}{q}\right)^{j-1} \right) + p^j T\left(\frac{n}{q^j}\right)$$

  - Si p>q

$$T(n) = kn \cdot \frac{\left(\frac{p}{q}\right)^j - 1}{\frac{p}{q} - 1} + p^j T\left(\frac{n}{q^j}\right)$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - Escoger j tal que $q^j=n$ (o sea, $j=\log_q n$):

$$T(n) = kn \cdot \frac{\left(\frac{p}{q}\right)^{\log_q n} - 1}{\frac{p}{q} - 1} + p^{\log_q n} T(1)$$

  - Observar que:

$$\left(\frac{p}{q}\right)^{\log_q n} = \frac{p^{\log_q n}}{n} = \frac{\left(q^{\log_q p}\right)^{\log_q n}}{n} = \frac{n^{\log_q p}}{n}$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - Por lo tanto, si p>q se tiene que

$$T(n) = O(n^{\log_p q})$$

  - Si p=q, de (*) se obtiene que ($j=\log_q n$):

$$T(n) = knj + q^j T\left(\frac{n}{q^j}\right)$$

$$T(n) = k(n \log n) + O(n) = O(n \log n)$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - Caso p<q:

$$T(n) \leq kn\frac{1}{1-\frac{p}{q}} + p^jT\left(\frac{n}{q^j}\right)$$

$$T(n) = k'n + o(n) = O(n)$$

# Ecuaciones de recurrencia

- Teorema Maestro
  - Ejemplo: Mergesort

$$T(n) = n + 2T\left(\frac{n}{2}\right)$$

$$\Rightarrow T(n) = O(n \log n)$$

# Ecuaciones de recurrencia

- Ejercicio: Analizar el siguiente código

```
void algoritmo(int[] A, int i, int j)
{
   if (j-i < 3)
   {
      if (A[i] > A[j-1])
      {
         intercambiar A[i] y A[j-1];
      }
   }
   else
   {
      k1 = i + Math.ceil(2/3*(j-i));
      k2 = i + Math.floor(1/3*(j-i));
      algoritmo(A, i, k1);
      algoritmo(A, k2, j);
      algoritmo(A, i, k1);
   }
}
```

# Ecuaciones de recurrencia

- Método de substitución
  - Deducir la forma de la solución de la ecuación de recurrencia
  - Usar inducción para encontrar las constantes y mostrar que la solución es válida
- El método sirve cuando es posible o es fácil “adivinar” la forma de la solución

# Ecuaciones de recurrencia

- Método de substitución
  - Ejemplo: cota superior para

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

  - Suponer que la solución es T(n) = O(n log n)
    - Hay que demostrar que, para algún c>0 se tiene que

$$T(n) \leq cn \log n$$

# Ecuaciones de recurrencia

- Método de substitución
  - Suponemos que la cota es válida para floor(n/2)

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

  - El último paso es válido para c >= 1
    - Faltaría mostrar que las condiciones de borde son válidas

# Ecuaciones de recurrencia

- Método de substitución
  - Para que el método funcione, es necesario demostrar la misma cota supuesta
  - Ejemplo: para la ecuación

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

  - Suponemos cota O(n):

$$T(n) \leq cn$$

# Ecuaciones de recurrencia

- Método de substitución
  - Substituyendo el supuesto en la ecuación se obtiene

$$
\begin{aligned}
T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\
&= cn + 1
\end{aligned}
$$

  - Esto no implica el supuesto para ningún c

# Ecuaciones de recurrencia

- Método de substitución
  - Cambiando supuesto (b >= 0 constante)

$$T(n) \leq cn - b$$

  - Aplicando inducción

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

# Ecuaciones de recurrencia

- Ejemplo: Selección ($k$-ésimo)
  - Problema: dado un arreglo desordenado encontrar el $k$-ésimo del conjunto
- Determinar mínimo o máximo: $\Theta(n)$ (cota inferior y superior)
- Recordando algoritmo del torneo:
  - Supongamos que $x$ es el primero (máximo)
  - El segundo puede ser cualquiera de los que perdieron directamente con $x$

# Ecuaciones de recurrencia

- Luego, para calcular segundo, tercero, …, toma tiempo:
  - Segundo: $n+\log_2 n$
  - Tercero: $n+2\log_2 n$
  - …
  - $k$: $n+(k-1)\log_2 n$
- Esto está bien para $k$ constante, pero para un $k$ genérico (como la mediana)
  - $k=n/2$: $O(n \log n)$

# Ecuaciones de recurrencia

- Quickselect
  - Se basa en el tipo de operaciones de quicksort
  - Algoritmo
    - Se escoge pivote al azar
    - Se hace una partición el arreglo de acuerdo al pivote escogido
    - Si el pivote cae más allá de la posición *k*, sólo se sigue buscando en la parte izquierda
    - Si el pivote estaba en la posición *k*, lo encontramos de inmediato

# Ecuaciones de recurrencia

- Seudocódigo

```
Quickselect(S,k)
{
  Sea p en S
  S1 = {x en S, x < p}
  S2 = {x en S, x > p}
  Si k <= |S1|
    return Quickselect(S1,k)
  Si k = |S1|+1 return p
  return Quickselect(S2, k-|S1|-1)
}
```

# Ecuaciones de recurrencia

- Peor caso: $O(n^2)$ (mala elección del pivote)
- Caso promedio: $O(n)$
- En la práctica este algoritmo es muy rápido, pero su peor caso es pésimo
- Uno quisiera asegurar una garantía de orden lineal para encontrar el $k$-ésimo
- Idea: buscar un pivote tal que deje fuera por lo menos una fracción fija del total de elementos

# Ecuaciones de recurrencia

- Método de selección lineal
  - Dividir $S$ en $|S/5|$ conjuntos (cada $S_i$ contiene 5 elementos)
  - Obtener las medianas $m_1$, $m_2$, …
  - Obtener $p$=Select($\{m_i\}$, $(|S|/5)/2$) (mediana de las medianas)

# Ecuaciones de recurrencia

- Características de *p*
  - Mayor que la mitad de las medianas
  - Menor que la otra mitad de las medianas
  - De los grupos con medianas menores (que fueron obtenidas de entre 5 elementos)
    - Al menos 3 elementos son menores que *p*
  - De los grupos con medianas mayores
    - Al menos 3 elementos son mayores que *p*
  - Esto implica que 3/10 elementos son menores que *p* y que 3/10 son mayores que *p*

# Ecuaciones de recurrencia

- El pivote *p* debe ser mayor que el 3/10 menor y menor que el 3/10 mayor de *S*
  - En el peor caso habrá que buscar recursivamente en un grupo con 7/10 de los elementos

$$T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$$

  - Cálculo de $m_i$ y particiones + cálculo de mediana de medianas + recursión sobre (7/10)*n* restantes

# Ecuaciones de recurrencia

- Análisis usando substitución: suponiendo solución $O(n)$

$$T(n) \leq dn \Rightarrow T(n) \leq n + \frac{dn}{5} + \frac{7}{10}dn \leq dn$$

$$d \geq 10 \Rightarrow T(n) = O(n)$$

# Ecuaciones de recurrencia

- La elección de 5 elementos para los grupos $S_i$ se debe a que:
  - Este número debe ser impar para obtener mediana exacta
  - Debe ser mayor o igual a 5 para asegurar linealidad del algoritmo
- Se escoge 5 porque:
  - Mediana de medianas queda muy a la mitad
  - Para números muy grandes de elementos calcular las medianas toma tiempo mayor

# Técnicas básicas de diseño de algoritmos

- Dividir y reinar
- Programación dinámica
- Inducción
- Búsqueda exhaustiva
- Algoritmos avaros (greedy)

# Técnicas básicas de diseño de algoritmos

- Subsecuencia de suma máxima
  - Dados enteros $A_1$, …, $A_n$ (posiblemente negativos), encontrar el maximo valor de

$$\sum_{k=i}^{j} A_k$$

  - Si todos los números son negativos, la subsecuencia de suma máxima es 0

# Técnicas básicas de diseño de algoritmos

- Ejemplo:
  - Secuencia: -2,11,-4,13,-5,-2
  - Respuesta: 20
- Veremos cuatro soluciones distintas para este problema

# Técnicas básicas de diseño de algoritmos

- Primera solución (Búsqueda exhaustiva):
  - Calcular la suma de todas las subsecuencias
  - Quedarse con la suma mayor

# Técnicas básicas de diseño de algoritmos

- Solución 1: Búsqueda exhaustiva

```
int maxSum = 0;
for( i=0; i<a.length; i++)
{
  for( j=i; j<a.length; j++)
  {
    int thisSum = 0;
    for (k=i; k<=j; k++)
      thisSum += a[k];
    if (thisSum > maxSum)
      maxSum = thisSum;
  }
}
```

# Técnicas básicas de diseño de algoritmos

- Número de sumas realizadas:

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\frac{n^3+3n^2+2n}{6}$$

- Complejidad temporal $O(n^3)$

# Técnicas básicas de diseño de algoritmos

- Segunda solución (mejora a Solución 1)
  - Notar que

$$\sum_{k=i}^{j} A_k = A_j + \sum_{k=i}^{j-1} A_k$$

  - Por lo tanto, el tercer ciclo **for** se puede eliminar

# Técnicas básicas de diseño de algoritmos

- Solución 2: Mejora a Solución 1

```
int maxSum = 0;
for( i=0; i<a.length; i++)
{
  int thisSum = 0;
  for (j=i; j<=a.length; j++)
  {
    thisSum += a[j];
    if (thisSum > maxSum)
      maxSum = thisSum;
  }
}
```

*Tiempo: $O(n^2)$*

# Técnicas básicas de diseño de algoritmos

- Solución 3: Usando "dividir para reinar"
  - Idea: dividir el problema en dos subproblemas del mismo tamaño
  - Resolver recursivamente
  - Mezclar las soluciones
  - Obtener solución final

# Técnicas básicas de diseño de algoritmos

- Dividiendo el problema
  - Subsecuencia de suma máxima puede estar en tres partes:
    - Primera mitad
    - Segunda mitad
    - Cruza por el medio ambas mitades

# Técnicas básicas de diseño de algoritmos

- Dividiendo el problema
  - Ejemplo:

| Primera mitad | Segunda mitad |
|---|---|
| 4 -3 5 -2 | -1 2 6 -2 |

# Técnicas básicas de diseño de algoritmos

- Dividiendo el problema
  - Ejemplo:

| Primera mitad | Segunda mitad |
|---|---|
| 4 -3 5 -2 | -1 2 6 -2 |

  - Suma máxima primera mitad: 6

# Técnicas básicas de diseño de algoritmos

- Dividiendo el problema
  - Ejemplo:

| Primera mitad | Segunda mitad |
|---|---|
| 4 -3 5 -2 | -1 2 6 -2 |

  - Suma máxima segunda mitad: 8

# Técnicas básicas de diseño de algoritmos

- Dividiendo el problema
  - Ejemplo:

| Primera mitad | Segunda mitad |
|---|---|
| 4 -3 5 -2 | -1 2 6 -2 |

  - Suma máxima incluyendo último primera mitad: 4
  - Idem primer elemento segunda mitad: 7
  - Total: 11 (mayor que máximo en ambas mitades)

# Técnicas básicas de diseño de algoritmos

- Algoritmo:
  - Dividir secuencia en dos (izquierda, derecha)
  - Resolver recursivamente las mitades
    - Caso base: secuencia de largo 1
  - Calcular suma máxima centro (borde izquierdo + borde derecho)
  - Retornar max{izquierda, derecha, centro}

# Técnicas básicas de diseño de algoritmos

- Complejidad del algoritmo:
  - Dos llamadas recursivas de tamaño *n*/2
  - Suma máxima centro: $O(n)$
  - Ecuación de recurrencia:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

*Tiempo: O(n log(n)) (Teorema Maestro, caso p=q)*

# Técnicas básicas de diseño de algoritmos

- Solución 4: Inducción
  - Observaciones:
    - No es necesario conocer donde esta la mejor subsecuencia
    - La mejor subsecuencia no puede comenzar en un número negativo
      - Corolario: cualquier subsecuencia negativa no puede ser prefijo de la subsecuencia óptima

# Técnicas básicas de diseño de algoritmos

- Solución 4: Inducción
  - Inducción (reforzada)
    - Se conoce la mejor subsecuencia entre 1 y *j*
    - Se conoce la mejor subsecuencia que termina en *j*
  - Algoritmo
    - Se almacenan ambos valores (inicialmente 0)
    - Se incrementa *j* en 1
    - Se actualiza mejor subsecuencia si es necesario
    - Si subsecuencia que termina en *j* es < 0 se puede descartar, volver su valor a 0

# Técnicas básicas de diseño de algoritmos

- Seudocódigo

```
int maxSum = 0, thisSum = 0;
for( j=0; j<a.length; j++)
{
  thisSum += a[j];
  if (thisSum > maxSum)
    maxSum = thisSum;
  else if (thisSum < 0)
    thisSum = 0;
}
```

*Tiempo: O(n)*

# Técnicas básicas de diseño de algoritmos

- Comparación entre las distintas soluciones

| $n$ | $O(n^3)$ | $O(n^2)$ | $O(n \log n)$ | $O(n)$ |
|---|---|---|---|---|
| 10 | 0,00103 | 0,00045 | 0,00066 | 0,00034 |
| 100 | 0,47015 | 0,01112 | 0,00486 | 0,00063 |
| 1.000 | 448,7 | 1,1233 | 0,05843 | 0,00333 |
| 10.000 | NA | 111,13 | 0,68631 | 0,03042 |
| 100.000 | NA | NA | 8,0113 | 0,29832 |

# Técnicas básicas de diseño de algoritmos

- Problema: comparar dos secuencias de ADN
  - ADN: secuencia de moléculas llamadas bases
  - Se puede representar como un string (A, C, G, T)
- Cómo determinar si dos secuencias son similares
  - Una es *substring* de la otra
  - Costo de transformar una en otra (distancia edición)
  - Encontrar una tercera que se parezca a ambas

# Técnicas básicas de diseño de algoritmos

- Definiciones
  - Subsecuencia: la secuencia con cero o más elementos dejados fuera
  - Formalmente:

$$X = \langle x_1, \ldots, x_m \rangle, \ Z = \langle z_1, \ldots, z_k \rangle$$

*Z* es subsecuencia de *X* si existe secuencia de índices creciente de *X* tal que

$$\langle i_1, \ldots, i_k \rangle, \ \forall j = 1, \ldots, k \quad x_{i_j} = z_j$$

# Técnicas básicas de diseño de algoritmos

- Definiciones
  - *Z* es subsecuencia común de *X* e *Y* si es subsecuencia de *X* y de *Y*
  - Ejemplos:

$$X = \langle A, B, C, B, D, A, B \rangle, \;\; Y = \langle B, D, C, A, B, A \rangle$$

$$Z = \langle B, C, A \rangle, \;\; Z' = \langle B, C, B, A \rangle$$

- Problema: encontrar subsecuencia común más larga (LCS) de *X* e *Y*

# Técnicas básicas de diseño de algoritmos

- Solución por búsqueda exhaustiva:
  - Enumerar todas las subsecuencias de $X$
  - Chequear si cada una es también subsecuencia de $Y$
  - Guardar la subsecuencia común más larga
- Tiempo:
  - $X$ tiene $2^m$ subsecuencias
  - Este método requiere tiempo exponencial

# Técnicas básicas de diseño de algoritmos

- Idea: intentar dividir el problema
- Definición: *i*-ésimo prefijo de *X*

$$X = \langle x_1, \ldots, x_m \rangle$$

$$X_i = \langle x_1, \ldots, x_i \rangle, \; i = 1, \ldots, m$$

- Subproblemas de LCS: prefijos de *X* e *Y*

# Técnicas básicas de diseño de algoritmos

- Propiedad de subestructura óptima
  - Un problema exhibe una subestructura óptima si una solución óptima al problema contiene soluciones óptimas a subproblemas
  - Si un problema exhibe una subestructura óptima, es un buen indicio que se podría utilizar programación dinámica para resolverlo (o una estrategia avara)

# Técnicas básicas de diseño de algoritmos

- Teorema: Subestructura óptima de una LCS
  - *X* (*m*) e *Y* (*n*) secuencias, *Z* (*k*) una LCS de *X* e *Y*

1. $x_m = y_n \Rightarrow z_k = x_m = y_n \;\wedge$
   $Z_{k-1}$ es LCS de $X_{m-1}$ e $Y_{n-1}$
2. $x_m \neq y_n \Rightarrow z_k \neq x_m$ implica que
   $Z$ es LCS de $X_{m-1}$ e $Y$
3. $x_m \neq y_n \Rightarrow z_k \neq y_n$ implica que
   $Z$ es LCS de $X$ e $Y_{n-1}$

# Técnicas básicas de diseño de algoritmos

- Teorema implica revisar uno o dos subproblemas
- La solución del subproblema es parte de la solución final (óptima)
- Nota: Encontrar LCS de casos (2) y (3) del Teorema implica calcular LCS de $X_{m-1}$ e $Y_{n-1}$
  - Muchos subproblemas comparten otros subproblemas
  - Total subproblemas distintos: $m*n$

# Técnicas básicas de diseño de algoritmos

- Solución: Programación dinámica
- Definición: Matriz *C* de *m* x *n*

$$c[i,j] = \begin{cases} 0 & \text{si} \quad i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{si} \quad i, j > 0 \text{ y } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{si} \quad i, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

- Algoritmo: llenar tabla en forma *bottom-up*

# Técnicas básicas de diseño de algoritmos

- Implementación:

```
m=X.length-1; n=Y.length-1; // indices 1 a m,n
for(i=1; i<=m; i++) c[i,0]=0;
for(j=0; j<=n; j++) c[0,j]=0;

for(i=1; i<=m; i++)
  for(j=1; j<=n; j++)
    if (X[i]==Y[j]){
      c[i,j]=c[i-1,j-1]+1; b[i,j]="\";}
    else if (c[i-1,j]>=c[i,j-1]){
      c[i,j]=c[i-1,j]; b[i,j]="|"}
    else{
      c[i,j]=c[i-1,j]; b[i,j]="-"}

return {c,b};
```

# Técnicas básicas de diseño de algoritmos

- Ejemplo:
  - Para imprimir LCS

```
void LCS(b,X,i,j){
   if (i==0 || j==0)
     return;
   if (b[i,j]==“\”){
     LCS(b,X,i-1,j-1);
     print(X[i]);}
   else if (b[i,j]==“|”)
     LCS(b,X,i-1,j);
   else \\ “-”
     LCS(b,X,i,j-1);
}
```

| $j$ | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| $i$ | | $y_j$ | $B$ | $D$ | $C$ | $A$ | $B$ | $A$ |
| 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $A$ | 0 | ↑0 | ↑0 | ↑0 | ↖1 | ←1 | ↖1 |
| 2 | $B$ | 0 | ↖1 | ←1 | ←1 | ↑1 | ↖2 | ←2 |
| 3 | $C$ | 0 | ↑1 | ↑1 | ↖2 | ←2 | ↑2 | ↑2 |
| 4 | $B$ | 0 | ↖1 | ↑1 | ↑2 | ↑2 | ↖3 | ←3 |
| 5 | $D$ | 0 | ↑1 | ↖2 | ↑2 | ↑2 | ↑3 | ↑3 |
| 6 | $A$ | 0 | ↑1 | ↑2 | ↑2 | ↖3 | ↑3 | ↖4 |
| 7 | $B$ | 0 | ↖1 | ↑2 | ↑2 | ↑3 | ↖4 | ↑4 |

# Técnicas básicas de diseño de algoritmos

- Algoritmos avaros (greedy)
  - Resuelven un problema en etapas, realizando lo que parece ser lo mejor en cada etapa
  - No siempre garantizan encontrar la solución óptima
- Ejemplo: Algoritmo de Dijkstra para encontrar distancias mínimas en un grafo dirigido G(V,E)

# Técnicas básicas de diseño de algoritmos

- Distancias mínimas en un grafo dirigido
  - En este problema los rótulos de los arcos se interpretan como distancias o pesos w
    - La distancia (o largo) de un camino es la suma de los largos o pesos de los arcos que lo componen
  - El problema de encontrar los caminos más cortos corresponde a encontrar los n caminos más cortos desde un nodo dado s hasta todos los nodos del grafo

# Técnicas básicas de diseño de algoritmos

- Algoritmo de Dijkstra
  - La idea del algoritmo es mantener un conjunto S de nodos “alcanzables” desde el nodo origen s e ir extendiendo este conjunto en cada iteración
  - Los nodos alcanzables son aquellos para los cuales ya se ha encontrado su camino óptimo desde el nodo origen
    - Para esos nodos su distancia óptima al origen es conocida

# Técnicas básicas de diseño de algoritmos

- Algoritmo de Dijkstra
  - Para los nodos fuera de S se conoce el camino óptimo desde s que pasa sólo por nodos de S
    - Este es un camino óptimo tentativo
  - En cada iteración, el algoritmo encuentra el nodo que no está en S y cuyo camino óptimo tentativo tiene largo mínimo
    - Este nodo se agrega a S y su camino óptimo tentativo se convierte en su camino óptimo
    - Luego, se actualizan los caminos óptimos tentativos para los demás nodos

# Técnicas básicas de diseño de algoritmos

- Algoritmo de Dijkstra
  - Pseudocódigo (w(u,v)>=0 por cada arista (u,v))

```
Dijkstra(G,w,s)
Initialize-Single-Source(G,s)
S <- φ
Q <- V[G] // Q contiene los nodos fuera de S
while Q != φ
    u <- Extract-Min(Q)  // primera vez u = s
    S <- S U {u}
    for each vertex v in Adj[u]
        Relax(u,v,w)
```

# Técnicas básicas de diseño de algoritmos

- Algoritmo de Dijkstra
  - Pseudocódigo (w(u,v)>=0 por cada arista (u,v))

```
Initialize-Single-Source(G,s)
for each vertex v in V[G]
      d[v] <- infinito
      p[v] <- 0 // no tiene previo
d[s] <- 0
```

```
Relax(u,v,w)
if d[v] > d[u] + w(u,v)
      d[v] <- d[u] + w(u,v)
      p[v] <- u
```

# Técnicas básicas de diseño de algoritmos

- Algoritmo de Dijkstra
  - Ejemplo (en la pizarra)

# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Notación:
    - $\delta(s,u)$: camino más corto (distancia) entre s y u
    - d[u]: estimación del camino más corto al nodo u
  - Importante:
    - Todos los arcos deben tener pesos no negativos

# Técnicas básicas de diseño de algoritmos

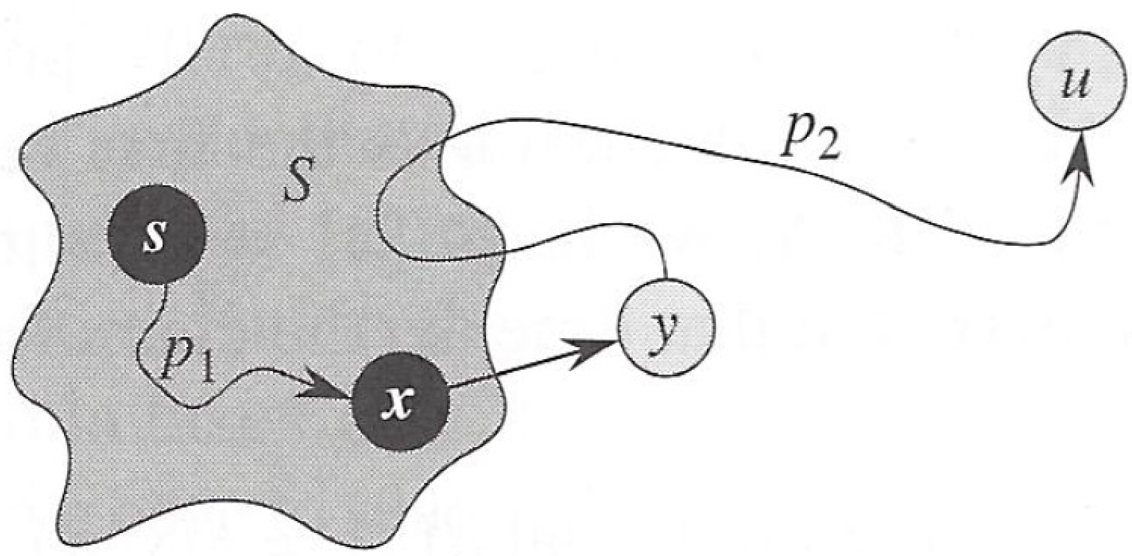
- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Sea u el primer vértice para el cual d[u] != $\delta$(s,u) cuando u se agrega a S
    - u != s ya que d[u] = $\delta$(s,s) = 0
    - Lo anterior implica que S no es vacío justo antes de agregar u a S
    - Tiene que haber al menos un camino de s a u (sino d[u] = $\delta$(s,u) = infinito), por lo que debe haber un camino de costo mínimo

# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Considerar caminos $p_1$ y $p_2$ de la figura en la siguiente slide (podrían no tener arcos)
    - Sea y el primer arco en el camino en V-S
    - Sea x el predecesor de y (x en S)

# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima



# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Observar que d[y] = δ(s,y) cuando se añade u a S
  - Dado que y está antes que u en el camino más corto se tiene δ(s,y) <= δ(s,u) y por lo tanto

$$\begin{aligned} d[y] &= \delta(s,y) \\ &\leq \delta(s,u) \\ &\leq d[u] \end{aligned}$$

# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Pero dado que ambos vértices u e y estaban en V-S cuando u fue escogido para agregarlo a S, se tiene que d[u] <= d[y], por lo que las desigualdades resultan ser igualdades

$$
\begin{aligned}
d[y] &= \delta(s,y) \\
&= \delta(s,u) \\
&= d[u]
\end{aligned}
$$

# Técnicas básicas de diseño de algoritmos

- Teorema: el algoritmo de Dijkstra encuentra la solución óptima
  - Lo anterior implica que $d[u] = \delta(s,u)$, pero esto contradice la elección de u
    - $d[u] = \delta(s,u)$ cuando u se agrega a S
  - Corolario: el algoritmo Dijkstra encuentra la solución óptima

# Análisis caso promedio de Quicksort

- Quicksort se basa en el paradigma dividir-para-reinar
- Algoritmo para ordenar un subarreglo A[p,r]
  - Realizar una partición del arreglo A[p,r] en dos subarreglos A[p,q-1] y A[q+1,r] (pueden estar vacíos), tal que cada elemento de A[p,q-1] es menor o igual que A[q], y A[q] es menor o igual que cada elemento de A[q+1,r]
  - Ordenar ambos subarreglos en forma recursiva
  - No se requiere trabajo extra para ordenar A[p,r]

# Análisis caso promedio de Quicksort

- Pseudocódigo:

```
Quicksort(A,p,r)
if p < r
      q <- Partition(A,p,r)
      Quicksort(A,p,q-1)
      Quicksort(A,q+1,r)
```

- Llamada inicial: Quicksort(A,1,length(A))

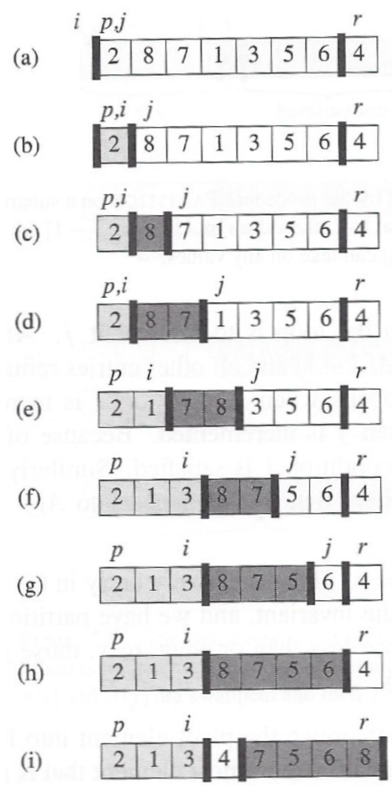
# Análisis caso promedio de Quicksort

- Realizando la partición del arreglo
  - Se realiza in-place

```
Partition(A,p,r)
x <- A[r]
i <- p-1
for j <- p to r-1
    if A[j] <= x
        i <- i+1
        swap(A[i],A[j])
swap(A[i+1],A[r])
return i+1
```

# Análisis caso promedio de Quicksort

- Realizando la partición del arreglo



# Análisis caso promedio de Quicksort

- Versión aleatorizada de Quicksort
  - Servirá después para el análisis del caso promedio

```
Randomized-Partition(A,p,r)
i <- Random(p,r)
swap(A[r],A[i])
return Partition(A,p,r)

Randomized-Quicksort(A,p,r)
if p < r
    q <- Randomized-Partition(A,p,r)
    Randomized-Quicksort(A,p,q-1)
    Randomized-Quicksort(A,q+1,r)
```

# Análisis caso promedio de Quicksort

- Análisis del peor caso de Quicksort
  - Usando el método de substitución: sea T(n) el costo del peor caso de Quicksort para una entrada de tamaño n. Se tiene que

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

  - q está en el rango [0, n-1] porque Partition produce dos subproblemas con tamaño total n-1

# Análisis caso promedio de Quicksort

- Análisis del peor caso de Quicksort
  - Suponiendo T(n) <= c n^2

$$\begin{aligned} T(n) &\leq \max_{0\leq q\leq n-1}(cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0\leq q\leq n-1}(q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

  - La expresión q^2+(n-q-1)^2 alcanza el máximo para q=0 o q=n-1. Con esto se obtiene la cota

$$\max_{0\leq q\leq n-1}(q^2 + (n-q-1)^2) \leq (n-1)^2 = n^2 - 2n + 1$$

# Análisis caso promedio de Quicksort

- Análisis del peor caso de Quicksort
  - Finalmente se obtiene que

$$\begin{aligned} T(n) &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

  - Escogiendo una constante c lo suficientemente grande para que el término c(2n-1) domine el término $\Theta$(n)
  - Se demuestra que T(n) = O(n^2)

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - El tiempo de ejecución de Quicksort se concentra en el tiempo gastado en Partition
  - Cada vez que se invoca Partition se escoge un pivote, y este elemento nunca es incluido en algún llamado recursivo futuro
    - A lo más pueden haber n invocaciones a Partition durante la ejecución de Quicksort

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Una invocación a Partition: O(1) más un tiempo proporcional al número de iteraciones del ciclo for
    - Toda iteración del ciclo for realiza una comparación (linea 4)
    - Si fuera posible contar cuántas comparaciones se realizan, se podría acotar el tiempo total gastado por el ciclo for en la ejecución completa de Quicksort
  - Lema: Sea X el número de comparaciones realizadas en la línea 4. La complejidad de Quicksort es O(n+X)

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Nuestro objetivo ahora es calcular X
    - No se intentará hacerlo por cada invocación a Partition, sino que se derivará una cota para el número total de comparaciones
    - Para facilitar el análisis, se renombrarán los elementos de A como $z_1,z_2,\ldots,z_n$, con $z_i$ el i-ésimo elemento menor
    - $Z_{ij} = \{z_i,z_{i+1},\ldots,z_j\}$

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - ¿Cuando compara el algoritmo $z_i$ con $z_j$?
    - Observación: cada par de elementos se compara a lo más una vez
    - Esto es porque los elementos se comparan contra los pivotes, y después que cada llamada a Partition termina, el pivote utilizado en dicha llamado no se compara nunca más contra otro elemento
  - Se defina variable indicadora (0 o 1)

$$X_{ij} = I\{z_i \text{ se compara con } z_j\}$$

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Dado que cada par se compara a lo más una vez, el número total de comparaciones realizadas es

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

  - Calcular caso promedio ⇔ calcular esperanza (esperanza es lineal y esperanza de una variable indicadora es su probabilidad)

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ se compara con } z_j\}$$

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Falta calcular la probabilidad de comparar $z_i$ con $z_j$
  - Este análisis supone que cada pivote es elegido aleatoria e independientemente (Randomized-Quicksort), y que los elementos son todos distintos

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - ¿Cuándo NO se comparan dos elementos?
    - Sea A = {números del 1 al 10, cualquier orden}
    - Suponer que el primer pivote escogido es 7
    - Partition separa el arreglo en
      - {1, 2, 3, 4, 5, 6}
      - {8, 9, 10}
      - El pivote 7 se comparó contra todos estos elementos
      - Notar que ningún número del primer subarreglo se comparará posteriormente con algún elemento del segundo subarreglo

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - En general, cuando se escoge un pivote x y se tiene que $z_i <= x <= z_j$, se sabe que $z_i$ y $z_j$ no serán comparados posteriormente
  - Por otra parte
    - Si $z_i$ se escoge como pivote antes que cualquier otro elemento en $Z_{ij}$, $z_i$ será comparado con cada elemento en $Z_{ij}$ (excepto él mismo)
    - Lo mismo vale para $z_j$
      - En el ejemplo anterior, 7 y 9 se comparan porque 7 es el primer elemento en $Z_{7,9}$ en ser escogido pivote

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Calculando la probabilidad que esto ocurra
    - Antes que se escoja un pivote en $Z_{ij}$, el subarreglo $Z_{ij}$ está junto en la misma partición
    - Cualquier elemento en $Z_{ij}$ tiene la misma probabilidad de ser escogido como pivote
    - Dado que $Z_{ij}$ tiene j-i+1 elementos, y dado que los pivotes se escogen aleatoria e independientemente, la probabilidad de cada uno de ser el escogido es 1/(j-1+1)

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Con todo lo anterior se tiene que

$$
\begin{aligned}
Pr\{z_i \text{ es comparado con } z_j\} &= Pr\{z_i \text{ o } z_j \text{ es el primer pivote escogido de } Z_{ij}\} \\
&= Pr\{z_i \text{ es primer pivote escogido de } Z_{ij}\} + Pr\{z_j \text{ es primer pivote escogido de } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}
$$

  - Segunda igualdad es válida porque son eventos mutuamente excluyentes

# Análisis caso promedio de Quicksort

- Análisis del caso promedio de Quicksort
  - Combinando ecuación E[X] y probabilidad calculada (k = j-i)

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\log n) \\
&= O(n \log n)
\end{aligned}
$$

# Análisis caso promedio de Quicksort

- En conclusión, el caso promedio de Randomized-Quicksort es O(n log n)

# Metodología de Experimentación

- Consideraciones generales
  - Etapas:
    - Diseño del experimento (incluye definir hipótesis de trabajo)
    - Elección de las medidas
    - Ejecución
    - Interpretación de los resultados
    - Volver al diseño del experimento
  - Presentación sigue la misma estructura, pero sólo excepcionalmente describe más de una iteración (la mejor, no necesariamente la última) del ciclo

# Metodología de Experimentación

- Lecturas en Material Docente:
  - “A Theoretician's Guide to the Experimental Analysis of Algorithms”, David S. Johnson, 2001
  - “Presenting Data from Experiments in Algorithmics”, Peter Sanders, 2002