Taller de busqueda (searching)

Algoritmos y Estructuras de Datos I

Departamento de Computación, FCEyN, Universidad de Buenos Aires.

Problemas “difíciles”

Algoritmos 1 nos enfrenta a problemas comunes en computación: **búsqueda** (searching) y **ordenamiento** (sorting)
Podemos considerar que la cantidad de formas para resolver este tipo de problemas puede ser prácticamente infinita. O al menos, si en aula hay 45 alumnos, podrían proponerse 45 soluciones correctas.

Problemas “difíciles”

Algoritmos 1 nos enfrenta a problemas comunes en computación: **búsqueda** (searching) y **ordenamiento** (sorting)
Podemos considerar que la cantidad de formas para resolver este tipo de problemas puede ser prácticamente infinita. O al menos, si en aula hay 45 alumnos, podrían proponerse 45 soluciones correctas.

Lo difícil es pensar algoritmos “eficientes”

Algoritmos “eficientes”

Precisamos una manera de comparar los algoritmos y establecer cuál puede ser más eficiente con respecto a otros.

Algoritmos “eficientes”

Precisamos una manera de comparar los algoritmos y establecer cuál puede ser más eficiente con respecto a otros.
Un algoritmo más eficiente es el que resuelve el problema con la menor cantidad de operaciones (comparado con los demás).

Algoritmos “eficientes”

Precisamos una manera de comparar los algoritmos y establecer cuál puede ser más eficiente con respecto a otros.
Un algoritmo más eficiente es el que resuelve el problema con la menor cantidad de operaciones (comparado con los demás).
La cantidad de memoria utilizada también puede tomarse en cuenta para considerar la eficiencia de un algoritmo.

Algoritmos "eficientes"

Precisamos una manera de comparar los algoritmos y establecer cuál puede ser más eficiente con respecto a otros.
Un algoritmo más eficiente es el que resuelve el problema con la menor cantidad de operaciones (comparado con los demás).
La cantidad de memoria utilizada también puede tomarse en cuenta para considerar la eficiencia de un algoritmo.

La cantidad de operaciones depende principalmente de la forma de resolver el problema, e incluye las comparaciones realizadas, operaciones matemáticas, etc., que es preciso aplicar a una entrada de tamaño n.

Algoritmos "eficientes"

La cantidad de operaciones depende principalmente de la forma de resolver el problema, e incluye las comparaciones realizadas, operaciones matemáticas, etc., que es preciso aplicar a una entrada de tamaño n.

Ejemplo:

- Un algoritmo con una entrada correspondiente a un vector de tamaño n

$$\underbrace{k_1\, n \log(n) + k_3}_{\substack{\text{Cantidad exactas de} \\ \text{operaciones peor caso}}} = \underbrace{O(n \log(n))}_{\substack{\text{Complejidad} \\ \text{peor caso}}}$$

Algoritmos “eficientes”

La cantidad de operaciones depende principalmente de la forma de resolver el problema, e incluye las comparaciones realizadas, operaciones matemáticas, etc., que es preciso aplicar a una entrada de tamaño n.

Ejemplo:

• Un algoritmo con una entrada correspondiente a un vector de tamaño n

$$\underbrace{k_1\, n \log(n) + k_3}_{\text{Cantidad exactas de operaciones peor caso}} = \underbrace{O(n \log(n))}_{\text{Complejidad peor caso}}$$

• La notación de Landau ($BigO$) se queda con el mayor orden de magnitud eliminando los términos de menor orden para definir la complejidad del peor caso.

Principales órdenes de magnitud

Orden	Nombre
O(1)	constante
O(log n)	logarítmica
O(n)	lineal
O(n log n)	cuasi lineal
O(n^2)	cuadrática
O(n^3)	cúbica
O(a^n)	exponencial

Problema de búsqueda

Muchas veces necesitamos saber si un elemento está en una lista:

$$
\begin{array}{l}
\text{proc } \mathit{buscar}(\text{in } s : \mathit{seq}\langle \mathbb{Z} \rangle, \text{in } x : \mathbb{Z}, \text{out } \mathit{res} : \text{Bool})\{ \\
\quad \text{Pre } \{\mathit{True}\} \\
\quad \text{Post } \{\mathit{res} = \mathit{true} \leftrightarrow (\exists i : \mathbb{Z})(0 \leq i < |s| \wedge_L s[i] = x)\} \\
\}
\end{array}
$$

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[$|s|-1$]
$\neq x$	$= x?$					
	$\uparrow$					
	i					

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[$\|s\| - 1$]
$\neq x$	$\neq x$	$= x$?				
		$\uparrow$				
		i				

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

$$
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{s[0]} & \text{s[1]} & \text{s[2]} & \text{s[3]} & \text{s[4]} & \ldots & \text{s}[|s|-1] \\
\hline
\multicolumn{1}{c}{\neq x} & \multicolumn{1}{c}{\neq x} & \multicolumn{1}{c}{\neq x} & \multicolumn{1}{c}{= x?} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\
\multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\uparrow} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\
\multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{i} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{}
\end{array}
$$

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[$\|s\| - 1$]
$\neq x$	$\neq x$	$\neq x$	$\neq x$			$= x$?
						$\uparrow$
						i

Listas desordenadas

- Problema: buscar un elemento en una lista no ordenada.
- Algoritmo: búsqueda lineal. Recorre en orden la lista.

s[0]	s[1]	s[2]	s[3]	s[4]	...	s[$\|s\|-1$]
$\neq x$	$\neq x$	$\neq x$	$\neq x$			$\neq x$

Complejidad $O(|s|)$

La cantidad de operaciones depende de la longitud de la lista.

Búsqueda lineal

```
bool busquedaLineal(vector<int> lista, int x){
  bool res = false;
  for(int i=0; i < lista.size() ; i++){
    if(lista[i] == x){
      res = true;
    }
  }
  return res;
}
```

Listas ordenadas

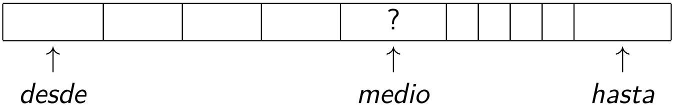
- Problema: buscar un elemento en una lista ordenada.

Listas ordenadas

- Problema: buscar un elemento en una lista ordenada.
- Algoritmo: Búsqueda binaria. Mirar el del medio. Si es mayor, seguir por la derecha, sino izquierda.

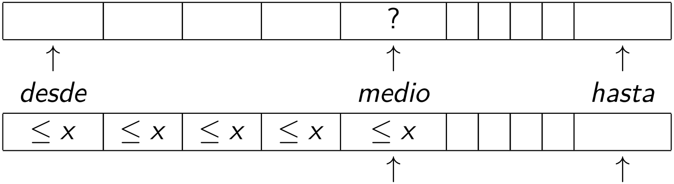
Listas ordenadas

- Problema: buscar un elemento en una lista ordenada.
- Algoritmo: Búsqueda binaria. Mirar el del medio. Si es mayor, seguir por la derecha, sino izquierda.



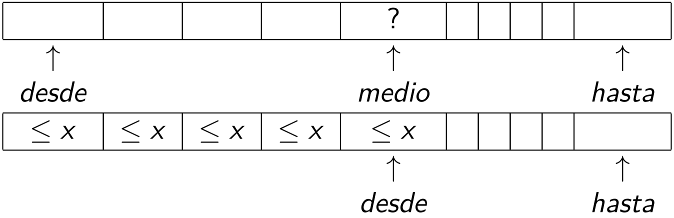
Listas ordenadas

- Problema: buscar un elemento en una lista ordenada.
- Algoritmo: Búsqueda binaria. Mirar el del medio. Si es mayor, seguir por la derecha, sino izquierda.



Listas ordenadas

- Problema: buscar un elemento en una lista ordenada.
- Algoritmo: Búsqueda binaria. Mirar el del medio. Si es mayor, seguir por la derecha, sino izquierda.



Búsqueda binaria

```cpp
bool busquedaBin(vector<int> lista, int desde, int hasta, int x){
  bool res = false;
  while (desde <= hasta){
    int medio = desde + (hasta-desde)/2;
    if (lista[medio] == x){
      res = true; //encontre el elemento
    }
    if (lista[medio] < x){
      desde = medio + 1; //esta en la mitad derecha
    }else{
      hasta = medio - 1; //esta en la mitad izquierda
    }
  }
  return res;
}
```

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia $hasta - desde$
0	$\|s\| - 1$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia $hasta - desde$
0	$\|s\| - 1$
1	$\cong (\|s\| - 1)/2$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia $hasta - desde$
0	$\lvert s\rvert - 1$
1	$\cong (\lvert s\rvert - 1)/2$
2	$\cong (\lvert s\rvert - 1)/4$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia $hasta - desde$
0	$\|s\| - 1$
1	$\cong (\|s\| - 1)/2$
2	$\cong (\|s\| - 1)/4$
3	$\cong (\|s\| - 1)/8$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia *hasta* − *desde*
0	$\|s\| - 1$
1	$\cong (\|s\| - 1)/2$
2	$\cong (\|s\| - 1)/4$
3	$\cong (\|s\| - 1)/8$
$\vdots$	$\vdots$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia *hasta* − *desde*
0	$\|s\| - 1$
1	$\cong (\|s\| - 1)/2$
2	$\cong (\|s\| - 1)/4$
3	$\cong (\|s\| - 1)/8$
$\vdots$	$\vdots$
t	$\cong (\|s\| - 1)/2^t$

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia *hasta* − *desde*
0	$\lvert s\rvert - 1$
1	$\cong (\lvert s\rvert - 1)/2$
2	$\cong (\lvert s\rvert - 1)/4$
3	$\cong (\lvert s\rvert - 1)/8$
$\vdots$	$\vdots$
t	$\cong (\lvert s\rvert - 1)/2^t$

- Sea t la cantidad de iteraciones necesarias para llegar a $high - low = 1$.

Complejidad busqueda binaria

- ¿Cuántas iteraciones realiza el ciclo (en peor caso)?

Número de iteración	Subsecuencia *hasta − desde*
0	$\lvert s\rvert - 1$
1	$\cong (\lvert s\rvert - 1)/2$
2	$\cong (\lvert s\rvert - 1)/4$
3	$\cong (\lvert s\rvert - 1)/8$
$\vdots$	$\vdots$
t	$\cong (\lvert s\rvert - 1)/2^t$

• Sea t la cantidad de iteraciones necesarias para llegar a $high - low = 1$.

$$\boxed{1 \cong (|s| - 1)/2^t \quad \Longrightarrow \quad 2^t \cong |s| - 1 \quad \Longrightarrow \quad t \cong \log_2(|s| - 1)}$$

¿Si ir para atras fuera muuuy costoso?

Jump Search

Descripción:

- Empezamos con saltos de tamaño fijo, m.
- Cuando nos pasamos, hacemos búsqueda lineal.

Jump Search

Descripción:

- Empezamos con saltos de tamaño fijo, m.
- Cuando nos pasamos, hacemos búsqueda lineal.

Ejemplo

Buscar el elemento x=10 de a pasos de tamaño $m = 3$

0	1	2	3	4	5	6
1	4	7	9	10	12	14

$\uparrow$

i

$v[i] < x$

Jump Search

Descripción:

- Empezamos con saltos de tamaño fijo, m.
- Cuando nos pasamos, hacemos búsqueda lineal.

Ejemplo

Buscar el elemento x=10 de a pasos de tamaño $m = 3$

0	1	2	3	4	5	6
1	4	7	9	10	12	14

$\uparrow$

i

$v[i] < x$

👍

Jump Search

Descripción:

- Empezamos con saltos de tamaño fijo, m.
- Cuando nos pasamos, hacemos búsqueda lineal.

Ejemplo

Buscar el elemento x=10 de a pasos de tamaño $m = 3$

0	1	2	3	4	5	6
1	4	7	9	10	12	14
						$\uparrow$
						i
						$v[i] < x$

Jump Search

Descripción:

- Empezamos con saltos de tamaño fijo, m.
- Cuando nos pasamos, hacemos búsqueda lineal.

Ejemplo

Buscar el elemento x=10 de a pasos de tamaño $m = 3$

0	1	2	3	4	5	6
1	4	7	9	10	12	14

Búsqueda lineal

Código

Ustedes!

Iteraciones

¿Cuántas iteraciones hace en el peor caso?

Iteraciones

¿Cuántas iteraciones hace en el peor caso?

- En el peor caso, hacemos $\frac{|s|}{m}$ saltos
- Al final, tenemos $m - 1$ pasos más por la búsqueda lineal.

Iteraciones

¿Cuántas iteraciones hace en el peor caso?

- En el peor caso, hacemos $\frac{|s|}{m}$ saltos
- Al final, tenemos $m - 1$ pasos más por la búsqueda lineal.

$$\boxed{\text{iteraciones} \approx \frac{|s|}{m} + m - 1}$$

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

$$O(\frac{|s|}{\sqrt{|s|}} + \sqrt{|s|} - 1)$$

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

$$O(\frac{|s|}{\sqrt{|s|}} + \sqrt{|s|} - 1) = O(\frac{\sqrt{|s|}}{\cancel{\sqrt{|s|}}}\frac{\cancel{|s|}}{\cancel{\sqrt{|s|}}} + \sqrt{|s|} - 1)$$

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

$$
\begin{aligned}
O(\frac{|s|}{\sqrt{|s|}} + \sqrt{|s|} - 1) &= O(\frac{\sqrt{|s|}}{\cancel{\sqrt{|s|}}}\frac{\cancel{|s|}}{\cancel{\sqrt{|s|}}} + \sqrt{|s|} - 1) \\
&= O(\sqrt{|s|} + \sqrt{|s|} - 1)
\end{aligned}
$$

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

$$
\begin{aligned}
O(\frac{|s|}{\sqrt{|s|}} + \sqrt{|s|} - 1) &= O(\frac{\sqrt{|s|}}{\cancel{\sqrt{|s|}}} \frac{\cancel{|s|}}{\cancel{\sqrt{|s|}}} + \sqrt{|s|} - 1) \\
&= O(\sqrt{|s|} + \sqrt{|s|} - 1) \\
&= O(2\sqrt{|s|} - 1)
\end{aligned}
$$

Complejidad

Sabiendo que el bloque óptimo es $m = \sqrt{|s|}$,

¿Cuál es la complejidad?

$$
\begin{aligned}
O(\frac{|s|}{\sqrt{|s|}} + \sqrt{|s|} - 1) &= O(\frac{\sqrt{|s|}}{\cancel{\sqrt{|s|}}}\frac{\cancel{|s|}}{\cancel{\sqrt{|s|}}} + \sqrt{|s|} - 1) \\
&= O(\sqrt{|s|} + \sqrt{|s|} - 1) \\
&= O(2\sqrt{|s|} - 1) \\
&= O(\sqrt{|s|})
\end{aligned}
$$

- ¿Cómo medimos el tiempo que tarda en ejecutar un algoritmo?

- ¿Cómo medimos el tiempo que tarda en ejecutar un algoritmo?
- Vamos a utilizar:
  - **clock()**: tiempo aproximado de CPU que transcurrió desde que nuestro programa fue iniciado, expresado en ticks de reloj.

- ¿Cómo medimos el tiempo que tarda en ejecutar un algoritmo?
- Vamos a utilizar:
    - **clock()**: tiempo aproximado de CPU que transcurrió desde que nuestro programa fue iniciado, expresado en ticks de reloj.
    - *CLOCKS_PER_SEC*: representa el número de ticks de reloj por segundo.

Ejemplo

Queremos saber cuanto tiempo tarda en ejecutar la siguiente función

```
int indicePrimeraAparicion(vector<int>& v, int elem){
    int res = -1;
    for(int i = 0; i < v.size(); i++){
        if(v[i] == elem){
            res = i;
        }
    }
    return res;
}
```

¿Cómo podemos hacer?

Medición de tiempo con clock

```
vector<int> v = {1, 2, 3, 4, 5, 6}

double t0 = clock();
int indice = indicePrimeraAparicion(v, 1);
double t1 = clock();

double tiempo = (double(t1-t0)/CLOCKS_PER_SEC);
```