Búsqueda y ordenamiento

Programación de Estructuras de Datos y Algoritmos Fundamentales (TC1031)

M.C. Xavier Sánchez Díaz
mail@tec.mx

Outline

Ordenando números

Introducción

El problema de ordenar una secuencia de números es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: tras un número determinado de pasos, la secuencia de números estará ordenada.

2) **Es fácil de verificar**: podemos revisar *rápidamente* que la secuencia ya está en el orden correcto.

Revisaremos distintos enfoques para hacerlo en estas *slides*.

Ordenando números

Introducción

El problema de ordenar una secuencia de números es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: tras un número determinado de pasos, la secuencia de números estará ordenada.

2) **Es fácil de verificar**: podemos revisar *rápidamente* que la secuencia ya está en el orden correcto.

Revisaremos distintos enfoques para hacerlo en estas *slides*.

Ordenando números

Introducción

El problema de ordenar una secuencia de números es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: tras un número determinado de pasos, la secuencia de números estará ordenada.

2) **Es fácil de verificar**: podemos revisar *rápidamente* que la secuencia ya está en el orden correcto.

Revisaremos distintos enfoques para hacerlo en estas *slides*.

Ordenando números

Introducción

El problema de ordenar una secuencia de números es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: tras un número determinado de pasos, la secuencia de números estará ordenada.

2) **Es fácil de verificar**: podemos revisar *rápidamente* que la secuencia ya está en el orden correcto.

Revisaremos distintos enfoques para hacerlo en estas *slides*.

Antes de comenzar...

Introducción

El problema de **ordenamiento** ha sido ampliamente estudiado en ciencias computacionales, pues es más sencillo buscar en una secuencia que **ya está ordenada** que en una lista sin ordenar (por ejemplo en un directorio telefónico o un diccionario).

El problema de ordenamiento se ha resuelto de varias maneras y con múltiples algoritmos; es de los *problemas favoritos* en entrevistas de trabajo de *pizarrón*.[1]

Existe un *límite inferior* de complejidad del problema probado teóricamente: $\mathcal{O}(n \log n)$

[1] Generalmente mal vista por la ansiedad que genera en algunas personas.

Antes de comenzar...

Introducción

El problema de **ordenamiento** ha sido ampliamente estudiado en ciencias computacionales, pues es más sencillo buscar en una secuencia que **ya está ordenada** que en una lista sin ordenar (por ejemplo en un directorio telefónico o un diccionario).

El problema de ordenamiento se ha resuelto de varias maneras y con múltiples algoritmos; es de los *problemas favoritos* en entrevistas de trabajo de *pizarrón*.[1]

Existe un *límite inferior* de complejidad del problema probado teóricamente: $\mathcal{O}(n \log n)$

[1] Generalmente mal vista por la ansiedad que genera en algunas personas.

Antes de comenzar...

Introducción

El problema de **ordenamiento** ha sido ampliamente estudiado en ciencias computacionales, pues es más sencillo buscar en una secuencia que **ya está ordenada** que en una lista sin ordenar (por ejemplo en un directorio telefónico o un diccionario).

El problema de ordenamiento se ha resuelto de varias maneras y con múltiples algoritmos; es de los *problemas favoritos* en entrevistas de trabajo de *pizarrón*.[1]

Existe un *límite inferior* de complejidad del problema probado teóricamente: $\mathcal{O}(n \log n)$

[1] Generalmente mal vista por la ansiedad que genera en algunas personas.

Límite inferior (de un problema)

Introducción

El límite inferior de complejidad de **un problema** se refiere a la **cantidad mínima de operaciones necesarias** para poder resolverlo.

AL MENOS debes hacer esto para poder resolverlo.

Esto es el **límite inferior** de **un problema**.

Límite inferior (de un problema)

Introducción

El límite inferior de complejidad de **un problema** se refiere a la **cantidad mínima de operaciones necesarias** para poder resolverlo.

AL MENOS debes hacer esto para poder resolverlo.

Esto es el **límite inferior** de **un problema**.

Límite inferior (de un problema)

Introducción

El límite inferior de complejidad de **un problema** se refiere a la **cantidad mínima de operaciones necesarias** para poder resolverlo.

AL MENOS debes hacer esto para poder resolverlo.

Esto es el **límite inferior** de **un problema**.

Límite superior (de un algoritmo)

Introducción

El límite superior de tiempo de ejecución de **un algoritmo** se refiere a la **cantidad máxima de operaciones hechas** por un algoritmo al resolver un problema.

A LO MUCHO mi método hace esto para resolver el problema.

Esto es el **límite superior** de un **algoritmo**.

Límite superior (de un algoritmo)

Introducción

El límite superior de tiempo de ejecución de **un algoritmo** se refiere a la **cantidad máxima de operaciones hechas** por un algoritmo al resolver un problema.

A LO MUCHO mi método hace esto para resolver el problema.

Esto es el **límite superior** de un **algoritmo**.

Límite superior (de un algoritmo)

Introducción

El límite superior de tiempo de ejecución de **un algoritmo** se refiere a la **cantidad máxima de operaciones hechas** por un algoritmo al resolver un problema.

A LO MUCHO mi método hace esto para resolver el problema.

Esto es el **límite superior** de un **algoritmo**.

Eficiencia

Introducción

Un **algoritmo** es óptimo si el **límite superior de su tiempo de ejecución** es igual al **límite inferior de la complejidad del problema** que resuelve.

Para resolver mi problema, A LO MENOS debo hacer esto.
Mi algoritmo hace, A LO MUCHO eso mínimo que debo hacer.
Eso significa que mi algoritmo es eficiente.

El límite inferior de la **complejidad del problema de ordenar** es

$$\mathcal{O}(n \log n)$$

Eficiencia

Introducción

Un **algoritmo** es óptimo si el **límite superior de su tiempo de ejecución** es igual al **límite inferior de la complejidad del problema** que resuelve.

Para resolver mi problema, A LO MENOS debo hacer esto.

Mi algoritmo hace, A LO MUCHO eso mínimo que debo hacer.

Eso significa que mi algoritmo es eficiente.

El límite inferior de la **complejidad del problema de ordenar** es

$$\mathcal{O}(n \log n)$$

Eficiencia

Introducción

Un **algoritmo** es óptimo si el **límite superior de su tiempo de ejecución** es igual al **límite inferior de la complejidad del problema** que resuelve.

Para resolver mi problema, A LO MENOS debo hacer esto.
Mi algoritmo hace, A LO MUCHO eso mínimo que debo hacer.

Eso significa que mi algoritmo es eficiente.

El límite inferior de la **complejidad del problema de ordenar** es

$$\mathcal{O}(n \log n)$$

Eficiencia

Introducción

Un **algoritmo** es óptimo si el **límite superior de su tiempo de ejecución** es igual al **límite inferior de la complejidad del problema** que resuelve.

Para resolver mi problema, A LO MENOS debo hacer esto.
Mi algoritmo hace, A LO MUCHO eso mínimo que debo hacer.
Eso significa que mi algoritmo es eficiente.

El límite inferior de la **complejidad del problema de ordenar** es

$$\mathcal{O}(n \log n)$$

Eficiencia

Introducción

Un **algoritmo** es óptimo si el **límite superior de su tiempo de ejecución** es igual al **límite inferior de la complejidad del problema** que resuelve.

Para resolver mi problema, A LO MENOS debo hacer esto.
Mi algoritmo hace, A LO MUCHO eso mínimo que debo hacer.
Eso significa que mi algoritmo es eficiente.

El límite inferior de la **complejidad del problema de ordenar** es

$$\mathcal{O}(n \log n)$$

¿Por qué estudiarlo?

Ordenamiento (ineficiente horrible)

Si es ineficiente, ¿entonces por qué estudiarlo?

1) Por razones históricas

2) Para reforzar la programación

3) Para reforzar el análisis del tiempo de ejecución

¿Por qué estudiarlo?

Ordenamiento (ineficiente horrible)

Si es ineficiente, ¿entonces por qué estudiarlo?

1) Por razones históricas

2) Para reforzar la programación

3) Para reforzar el análisis del tiempo de ejecución

¿Por qué estudiarlo?

Ordenamiento (ineficiente horrible)

Si es ineficiente, ¿entonces por qué estudiarlo?

1) Por razones históricas

2) Para reforzar la programación

3) Para reforzar el análisis del tiempo de ejecución

¿Por qué estudiarlo?

Ordenamiento (ineficiente horrible)

Si es ineficiente, ¿entonces por qué estudiarlo?

1) Por razones históricas

2) Para reforzar la programación

3) Para reforzar el análisis del tiempo de ejecución

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Intercambio (Swap Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por intercambio o *swap* se basa en leer el vector a ordenar de manera secuencial:

a) Compara el primer elemento con cada uno de los demás

b) Si encuentra uno más pequeño, entonces los cambia de lugar (**swap**)

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ vueltas
2. En cada una de esas vueltas, hará a lo mucho $n-1$ **swaps**

es decir

$$\mathcal{O}(n^2)$$

Swap Sort

Ordenamiento (ineficiente horrible)

```
2  #include <vector>
3
4  using namespace std;
5
6  template <typename T>
7  void swapSort(vector<T> &A){
8      T aux;
9      int i, j;
10     int n = A.size();
11
12     for (i = 0; i <= n-2; i++){
13         for (j = i+1; j <= n-1; j++){
14             if(A[i] > A[j]){
15                 aux = A[i];
16                 A[i] = A[j];
17                 A[j] = aux;
18             }
19         }
```

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Burbuja (Bubble Sort)

El *sorting* por burbuja o *bubble* se basa en comparar pares de elementos de manera secuencial:

a) Compara los primeros dos y los pone en el orden correcto
b) Después compara los siguientes dos, y los pone en el orden correcto...
c) Al terminar, el más grande estará al final del vector
d) Ahora hay que repetir el proceso hasta que ya no haya intercambios

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **swaps** para el primer elemento
2. Esto podría repetirse para los $n-1$ elementos que no se han acomodado (si la lista viene en el orden inverso)

es decir

$$\mathcal{O}(n^2)$$

Bubble Sort

Ordenamiento (ineficiente horrible)

```
23 template <typename T>
24 void bubbleSort(vector<T> &A){
25     T aux;
26     bool sentinel = true;
27     int n = A.size();
28     int i;
29 
30     while (sentinel){
31         for(i=0; i < n-1; i++){
32             if(A[i] > A[i+1]){
33                 aux = A[i];
34                 A[i] = A[i+1];
35                 A[i+1] = aux;
36             }
37         }
38         sentinel = false;
39 
40         for(i=0; i < n-1; i++){
41             if(A[i] > A[i+1]){
42                 // not sorted
43                 sentinel = true;
44                 break;
45             }
46         }
47     }
48 }
```

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
. . .

Es decir que si tengo un vector de n elementos:

1. Hará $n - 1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n - 1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico

b) Pongo el más chico al principio

c) Repito lo mismo con todos, menos con el primero (que ya acomodé)

. . .

Es decir que si tengo un vector de n elementos:

1. Hará $n - 1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n - 1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
   ...

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n-1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
   ...

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n-1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
   ...

Es decir que si tengo un vector de n elementos:

1. Hará $n - 1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n - 1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
   ...

Es decir que si tengo un vector de n elementos:

1. Hará $n - 1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n - 1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selección Directa (Selection Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por selección directa va llenando posiciones buscando de un extremo a otro:

a) Leo todos los números y me quedo con el más chico
b) Pongo el más chico al principio
c) Repito lo mismo con todos, menos con el primero (que ya acomodé)
...

Es decir que si tengo un vector de n elementos:

1. Hará $n-1$ comparaciones mientras busca el menor
2. Y SIEMPRE buscará $n-1$ veces

es decir

$$\mathcal{O}(n^2)$$

Selection Sort

Ordenamiento (ineficiente horrible)

```
50 template <typename T>
51 void selectSort(vector<T> &A){
52     int aux, pos_smallest, i, j;
53     int n = A.size();
54 
55     for(i = 0; i < n-1; i++){
56         pos_smallest = i;
57         for(j = i+1; j < n; j++){
58             if(A[pos_smallest] > A[j]){
59                 pos_smallest = j;
60             }
61         } // you now have the smallest
62         if (i != pos_smallest)
63         {
64             aux = A[i];
65             A[i] = A[pos_smallest];
66             A[pos_smallest] = aux;
67         }
68     }
69 }
```

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? . . .
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? . . .
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? ...
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? ...
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? ...
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? ...
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Inserción (Insertion Sort)

Ordenamiento (ineficiente horrible)

El *sorting* por inserción o *insertion* se basa en insertar los elementos del vector de la parte *no ordenada* a la parte ya *ordenada*. Similar a como ordenamos alfabéticamente:

a) Ponle que el primer elemento es parte de lo ordenado
b) El segundo, ¿va antes o después del primero? ...
c) Y luego para cada elemento i se va insertando en el lugar correcto $A[i]$

Es decir que si tengo un vector de n elementos:

1. Hará en el peor de los casos $n-1$ **inserciones**
2. Y cada inserción revisa, a lo mucho, $n-2$ posiciones posibles

es decir

$$\mathcal{O}(n^2)$$

Insertion Sort

Ordenamiento (ineficiente horrible)

```
71 template <typename T>
72 void insertionSort(vector<T> &A){
73     int aux, i, j;
74     int n = A.size();
75 
76     for(i = 1; i < n; i++){ // start at 1
77         j = i;
78         aux = A[i];
79         while(j > 0 && aux < A[j-1]){
80             A[j] = A[j-1];
81             j--;
82         }
83         A[j] = aux;
84     }
85 }
```

Búsqueda

Búsquedas

El problema de búsqueda (en contenedor unidimensional) es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: en el peor de los casos, buscamos todo el contenedor hasta el final $(\mathcal{O}(n))$[2]

2) **Es fácil de verificar**: si nos dan la posición de lo que estamos buscando, podemos revisar *rápidamente* si realmente contiene lo que buscamos.

[2] Hay maneras más eficientes de hacerlo

Búsqueda

Búsquedas

El problema de búsqueda (en contenedor unidimensional) es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: en el peor de los casos, buscamos todo el contenedor hasta el final ($\mathcal{O}(n)$)[2]

2) **Es fácil de verificar**: si nos dan la posición de lo que estamos buscando, podemos revisar *rápidamente* si realmente contiene lo que buscamos.

[2] Hay maneras más eficientes de hacerlo

Búsqueda

Búsquedas

El problema de búsqueda (en contenedor unidimensional) es un problema $\mathcal{P}$ ya que...

1) **Es fácil de resolver**: en el peor de los casos, buscamos todo el contenedor hasta el final $(\mathcal{O}(n))$[2]

2) **Es fácil de verificar**: si nos dan la posición de lo que estamos buscando, podemos revisar *rápidamente* si realmente contiene lo que buscamos.

[2] Hay maneras más eficientes de hacerlo

Búsqueda secuencial

Búsquedas

En la búsqueda secuencial iremos de uno por uno en el contenedor hasta encontrar el elemento que estamos buscando (si existe). Si lo encontramos, informaremos de su posición.

- En términos técnicos, suele hacerse la comparación de buscar *una aguja en un pajar*
- La aguja (*needle*) es el **elemento** a buscar
- El pajar (*haystack*) es el **contenedor** en el que se realiza la búsqueda

No hay manera de acortar la búsqueda ya que no sabemos si el elemento existe en la parte no revisada hasta llegar al final. Por eso en el peor de los casos exploramos todo el contenedor, es decir que su complejidad es lineal:

$$\mathcal{O}(n)$$

Búsqueda secuencial

Búsquedas

En la búsqueda secuencial iremos de uno por uno en el contenedor hasta encontrar el elemento que estamos buscando (si existe). Si lo encontramos, informaremos de su posición.

- En términos técnicos, suele hacerse la comparación de buscar *una aguja en un pajar*
- La aguja (*needle*) es el **elemento** a buscar
- El pajar (*haystack*) es el **contenedor** en el que se realiza la búsqueda

No hay manera de acortar la búsqueda ya que no sabemos si el elemento existe en la parte no revisada hasta llegar al final. Por eso en el peor de los casos exploramos todo el contenedor, es decir que su complejidad es lineal:

$$\mathcal{O}(n)$$

Búsqueda secuencial

Búsquedas

En la búsqueda secuencial iremos de uno por uno en el contenedor hasta encontrar el elemento que estamos buscando (si existe). Si lo encontramos, informaremos de su posición.

- En términos técnicos, suele hacerse la comparación de buscar *una aguja en un pajar*
- La aguja (*needle*) es el **elemento** a buscar
- El pajar (*haystack*) es el **contenedor** en el que se realiza la búsqueda

No hay manera de acortar la búsqueda ya que no sabemos si el elemento existe en la parte no revisada hasta llegar al final. Por eso en el peor de los casos exploramos todo el contenedor, es decir que su complejidad es lineal:

$$\mathcal{O}(n)$$

Búsqueda secuencial

Búsquedas

En la búsqueda secuencial iremos de uno por uno en el contenedor hasta encontrar el elemento que estamos buscando (si existe). Si lo encontramos, informaremos de su posición.

- En términos técnicos, suele hacerse la comparación de buscar *una aguja en un pajar*
- La aguja (*needle*) es el **elemento** a buscar
- El pajar (*haystack*) es el **contenedor** en el que se realiza la búsqueda

No hay manera de acortar la búsqueda ya que no sabemos si el elemento existe en la parte no revisada hasta llegar al final. Por eso en el peor de los casos exploramos todo el contenedor, es decir que su complejidad es lineal:

$$\mathcal{O}(n)$$

Búsqueda secuencial

Búsquedas

En la búsqueda secuencial iremos de uno por uno en el contenedor hasta encontrar el elemento que estamos buscando (si existe). Si lo encontramos, informaremos de su posición.

- En términos técnicos, suele hacerse la comparación de buscar *una aguja en un pajar*
- La aguja (*needle*) es el **elemento** a buscar
- El pajar (*haystack*) es el **contenedor** en el que se realiza la búsqueda

No hay manera de acortar la búsqueda ya que no sabemos si el elemento existe en la parte no revisada hasta llegar al final. Por eso en el peor de los casos exploramos todo el contenedor, es decir que su complejidad es lineal:

$$\mathcal{O}(n)$$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

Búsquedas

La búsqueda binaria o *binary search* consiste en buscar más eficientemente si sabemos que el contenedor ya está ordenado:

1) Comparo contra el elemento del centro
2) Si es el que buscaba, dame la posición central
3) Si es más grande que el centro, busca en la mitad de arriba
4) Si es más chico que el centro, busca en la mitad de abajo

De tal manera que iremos cortando en mitades, cuartos, octavos...

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots$$

El peor de los casos es que no lo encuentres. Asumiendo que tienes 8 elementos, habrás eliminado primero 4 elementos, y luego otros 2, y luego 1 más $\therefore \quad \mathcal{O}(\log n)$

Búsqueda Binaria

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

template <typename T>
int binarySearch(T needle, vector<T> haystack){
   int mid, low, hi, n, retval;
   low = 0;
   n = haystack.size();
   hi = n - 1;
   retval = -1;

   while(low <= hi){
       mid = (low + hi)/ 2;
       if (needle == haystack[mid]){
           retval = mid;
           break;
       }
       else if(needle < haystack[mid]){
           hi = mid - 1;
       }
       else{
           low = mid + 1;
       }
   }
   return retval;
}
```