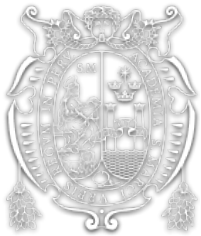
Algoritmos

Búsqueda y ordenación

MSc Edson Ticona Zegarra

Campamento de Programación



Contenido

Contenido

Búsqueda

Definition

Dado un conjunto de números A, y un número x, deseamos saber si x se encuentra en A.

Algoritmo de búsqueda

input : A es un conjunto de n números y x es el número buscado.

output: 0 si $x \in A$, 1 caso contrario

for $a \in A$ **do**
  if $x == a$ **then**
    return 0
  end
end
return 1

Algoritmo de búsqueda

input : A es un conjunto de n números y x es el número buscado.
output: 0 si $x \in A$, 1 caso contrario

for $a \in A$ **do**
  if $x == a$ **then**
    return 0
  end
end
return 1

- Complejidad:

Algoritmo de búsqueda

input : A es un conjunto de n números y x es el número buscado.
output: 0 si $x \in A$, 1 caso contrario

for $a \in A$ **do**
  if $x == a$ **then**
    return 0
  end
end
return 1

- Complejidad: $T(n) = O(n)$

Búsqueda binaria

- Si los elementos se encuentran ordenados se puede obtener un algoritmo más eficiente

Búsqueda binaria

- Si los elementos se encuentran ordenados se puede obtener un algoritmo más eficiente
- En general, **cuando la data tiene cierta estructura, esta puede ser aprovechada para obtener algoritmos más eficientes**

Búsqueda binaria

- Si los elementos se encuentran ordenados se puede obtener un algoritmo más eficiente
- En general, **cuando la data tiene cierta estructura, esta puede ser aprovechada para obtener algoritmos más eficientes**
- Se toma de referencia el elemento central, así se sabe si se debe continuar buscando en la parte superior o en la parte inferior

Búsqueda binaria

input : A es un conjunto de n números y x es el número buscado.

output: 0 si $x \in A$, 1 caso contrario

$m \leftarrow (l + r)/2$

if $x < A[m]$ **then**
| BinarySearch(A, x, l, m-1)
end

if $x > A[m]$ **then**
| BinarySearch(A, x, m+1, r)
end

if $x == A[m]$ **then**
| **return** m
end

return -1

Búsqueda binaria

input : A es un conjunto de n números y x es el número buscado.

output: 0 si $x \in A$, 1 caso contrario

$m \leftarrow (l + r)/2$

if $x < A[m]$ **then**
| BinarySearch(A, x, l, m-1)
end

if $x > A[m]$ **then**
| BinarySearch(A, x, m+1, r)
end

if $x == A[m]$ **then**
| **return** m
end

return -1

- Complejidad:

Búsqueda binaria

input : A es un conjunto de n números y x es el número buscado.

output: 0 si $x \in A$, 1 caso contrario

$m \leftarrow (l + r)/2$

if $x < A[m]$ **then**
| BinarySearch(A, x, l, m-1)
end

if $x > A[m]$ **then**
| BinarySearch(A, x, m+1, r)
end

if $x == A[m]$ **then**
| **return** m
end

return -1

- Complejidad: $T(n) = O(n \log n)$

Contenido

Ordenación

- Definimos el problema de ordenación de la siguiente manera

Definition

Dado un conjunto de números A en un orden arbitrario, deseamos retornar el conjunto en un orden particular, por ejemplo, de menor a mayor.

Definition

Se dice que una ordenación es estable si, para elementos iguales, respeta el orden inicial

Ordenación

- Primer algoritmo: algoritmo de inserción

Ordenación

- Primer algoritmo: algoritmo de inserción
- La idea de este algoritmo es recorrer el arreglo, de izquierda a derecha, colocando cada elemento en su posición adecuada en el lado izquierdo.

Algoritmo de inserción

input : A es un conjunto de n números.
output: A ordenado

for $i = 2, ..., n$ **do**
    $j \leftarrow i$;
    while $A[j] < A[j-1]$ & $j > 1$ **do**
        /* Solo intercambiar $A[j]$ con $A[j-1]$ */
        $tmp \leftarrow A[j-1]$;
        $A[j-1] \leftarrow A[j]$;
        $A[j] \leftarrow tmp$;
        $j--$
    end
end
return 1

Algoritmo de inserción

input : A es un conjunto de n números.
output: A ordenado

for $i = 2, ..., n$ **do**
    $j \leftarrow i$;
    while $A[j] < A[j-1]$ & $j > 1$ **do**
        /* Solo `intercambiar` $A[j]$ `con` $A[j-1]$ */
        $tmp \leftarrow A[j-1]$;
        $A[j-1] \leftarrow A[j]$;
        $A[j] \leftarrow tmp$;
        $j--$
    end
end
return 1

- Complejidad:

Algoritmo de inserción

input : A es un conjunto de n números.
output: A ordenado

for $i = 2, ..., n$ **do**
    $j \leftarrow i$;
    while $A[j] < A[j-1]$ & $j > 1$ **do**
        /* Solo intercambiar $A[j]$ con $A[j-1]$ */
        $tmp \leftarrow A[j-1]$;
        $A[j-1] \leftarrow A[j]$;
        $A[j] \leftarrow tmp$;
        $j--$
    end
end
return 1

- Complejidad: $T(n) = O(n^2)$

Algoritmo de inserción

- En el mejor de los casos el algoritmo es lineal
- Es decir $T(n) = \Omega(n)$
- En el peor de los casos el algoritmo es cuadrático
- Es decir $T(n) = O(n^2)$

Ordenación

- Segundo algoritmo: *mergesort* (algoritmo de mezcla)

Ordenación

- Segundo algoritmo: *mergesort* (algoritmo de mezcla)
- La idea de este algoritmo es dividir el arreglo en dos partes iguales, ordenar los dos sub-arreglos recursivamente y luego mezclarlos de tal manera que que los elementos queden ordenados.

Mergesort

input : A es un conjunto de n números.
output: A ordenado

```
/* Toda recursion tiene un caso base                    */
if i == j then
    return
end
mitad ← (i − j)/2 ;
B ←= MergeSort(A, i, mitad) ;
C ←= MergeSort(A, mitad + 1, j) ;
/* Cada subarray esta ordenado, ahora solo unirlos
   ordenadamente                                        */
return Merge(B, C)
```

Mergesort

input : Arreglos B y C
output: $A = B \cup C$ ordenado

```
it, i, j ← 1 ;                                  /* Iteradores */
while it < size(B) + size(C) do
    if B[i] < C[j] then
        A[it] ← B[i] ;
        i++
    end
    if C[j] <= B[i] then
        A[it] ← C[j] ;
        j++
    end
    it++
end
return A
```

Mergesort

- Complejidad:

Mergesort

- Complejidad: $T(n) = 2T(n/2) + O(n)$

Ordenación

- Segundo algoritmo: *quicksort* (Algoritmo rápido)

Ordenación

- Segundo algoritmo: *quicksort* (Algoritmo rápido)
- En este algoritmo tomamos un elemento, al que se denomina *pivot*, por conveniencia puede ser el primero, y lo colocamos en su lugar adecuado respecto al resto. Esta operación va a crear dos sub-arreglos, uno a la izquierda y otro a la derecha de este elemento. Repetimos el mismo procedimiento con ambos arreglos.

Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow A[0]$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow A[0]$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

- Complejidad:

Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow A[0]$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

- Complejidad: $T(n) = T(n/a) + T(n/b) + O(n)$

Quicksort

input : A un conjunto y p el pivot
output: L, R elementos menores y mayores que p

```
L, R ← [] ;                                  /* Inicializacion */
for a ∈ A do
    if a < p then
        L.push(a)
    end
    if a >= p then
        R.push(a)
    end
end
return L, R
```

Randomized Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow random(A)$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

Randomized Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow random(A)$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

- Complejidad:

Randomized Quicksort

input : A es un conjunto de n números.
output: A ordenado

if $size(A) == 1$ **then**
  return A
end
$p \leftarrow random(A)$;
$left, right \leftarrow Pivot(A \setminus \{p\}, p)$;
$S \leftarrow QuickSort(left) \cup \{p\} \cup QuickSort(right)$;
return S

- Complejidad: $T(n) = 2T(n/2) + O(n)$

Implementación

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main(){
  vector<int> V;
  V.push_back(5);
  V.push_back(2);
  V.push_back(3);
  sort(V.begin(), V.end());
  for(auto it : V)
    cout << it << "-";
  cout << endl;
}
```

Implementación estable

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool compare(pair<int,int> a, pair<int,int> b) {
  return a.first < b.first;
}

int main(){
  vector<pair<int,int>> V;
  V.push_back(make_pair(2,5));
  V.push_back(make_pair(7,2));
  V.push_back(make_pair(7,3));
  V.push_back(make_pair(1,3));
  stable_sort(V.begin(), V.end(), compare);
  for(auto it : V)
    cout << "(" << it.first << "-" << it.second << ")-";
  cout << endl;
}
```