

Clase de Ordenamiento, Búsqueda y otros Algoritmos sobre Secuencias

Algoritmos y Estructuras de Datos I

Gonzalo Guillamon

Departamento de Computación
Universidad de Buenos Aires

Primer Cuatrimestre 2018

Contenido

Ordenamiento

Insertion Sort

Selection Sort

Bubble Sort

Counting Sort

Algoritmos de Búsqueda

Búsqueda Lineal

Búsqueda Binaria

Ejercicios de implementación con secuencias

Ejercicio 1

Ejercicio 2

Ejercicio 3

Ordenamiento

¿Para qué ordenar?

Ordenamiento

¿Para qué ordenar?

- ▶ Para buscar elementos en una lista de forma fácil y rápida

Ordenamiento

¿Para qué ordenar?

- ▶ Para buscar elementos en una lista de forma fácil y rápida
- ▶ Para verificar unicidad de elementos

Ordenamiento

¿Para qué ordenar?

- ▶ Para buscar elementos en una lista de forma fácil y rápida
- ▶ Para verificar unicidad de elementos
- ▶ Para acceder al máximo, al mínimo, al k -ésimo

Y mucho más...

Algoritmos y Estructuras de Datos II

Ordenamiento

¿Para qué ordenar?

- ▶ Para buscar elementos en una lista de forma fácil y rápida
- ▶ Para verificar unicidad de elementos
- ▶ Para acceder al máximo, al mínimo, al k -ésimo

Y mucho más...

Algoritmos y Estructuras de Datos II

- ▶ ¿Cómo ordenamos?

Insertion Sort

Idea

Consiste en recorrer la lista e ir agregando cada elemento en la posición que le corresponde de la parte ya recorrida.

Invariante

- ▶ Los elementos de la lista entre $0..i$ son los mismos que los de la original pero se encuentran ordenados.
- ▶ La lista es una permutación de la lista original.

Insertion Sort

```
vector<int> insertionSort(vector<int> lista){
    for(int i=0; i < lista.size() ; i++){
        insertar(lista,i);
    }
    return lista;
}

void insertar(vector<int> &lista, int i){
    while(i > 0 && lista[i] < lista[i-1]){
        swap(lista,i,i-1);
    }
}

void swap(vector<int> &lista, int i, int j){
    int k=lista[i];
    lista[i]=lista[j];
    lista[j]=k;
}
```

Insertion Sort

¿Complejidad?

Insertion Sort

¿Complejidad? $\mathcal{O}(n^2)$

(intuición: insertar se hace hasta n veces y hace $1, \dots, n$ operaciones cada vez)

Selection Sort

Idea

Consiste en ir seleccionando los elementos de la lista de a uno en orden creciente, ubicando el mínimo en la primer posición, y así sucesivamente.

invariante

- ▶ En la k -ésima iteración, los primeros k elementos ya están ordenados en su posición final.
- ▶ La lista es una permutación de la lista original.

Selection Sort

```
vector<int> selectionSort(vector<int> lista){
    for(int i=0; i<lista.size(); i++){
        seleccionarMinimo(lista,i);
    }
    return lista;
}

void seleccionarMinimo(vector<int> &lista, int i){
    int posMinimo= i;
    for(int j=i; j<lista.size(); j++){
        if(lista[posMinimo] > lista[j]){
            posMinimo = j;
        }
    }
    swap(lista,i,posMinimo);
}
```

Selection Sort

¿Complejidad?

Selection Sort

¿Complejidad? $\mathcal{O}(n^2)$

(intuición: seleccionarMinimo se hace n veces y hace $n, \dots, 1$ operaciones cada vez)

Bubble Sort

Idea

Consiste en comparar cada elemento de la lista con el siguiente, intercambiándolos si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios.

Invariante

- ▶ En la k -ésima iteración, el elemento $k-1$ es menor al elemento k .
- ▶ El arreglo es una permutación del arreglo original.

Bubble Sort

```
void burbujeo(vector<int> &lista, int i){
    for(int j=lista.size()-1; j>i; j--){
        if(lista[j] < lista[j-1]){
            swap(lista, j, j-1);
        }
    }
}

vector<int> bubbleSort(vector<int> lista){
    for(int i=0; i<lista.size(); i++){
        burbujeo(lista,i);
    }
    return lista;
}
```

Bubble Sort

¿Complejidad?

Bubble Sort

¿Complejidad? $\mathcal{O}(n^2)$

(intuición: burbujeo se hace n veces y hace $n, \dots, 1$ operaciones cada vez)

Counting Sort

Idea

Asumimos que los elementos de la lista están entre 0 y k (con k arbitrario). El algoritmo calcula la cantidad de apariciones de cada elemento, que almacena en una secuencia nueva. Dicha información es utilizada para determinar el valor en cada posición en el arreglo de salida.

Invariante

- ▶ En la i -ésima iteración, la cantidad de apariciones de cada número de la lista entre las posiciones $0..i$ se encuentra en su posición correspondiente de la secuencia de conteo.
- ▶ La sumatoria del arreglo de conteo es i
- ▶ El arreglo original se mantiene.

Counting Sort

```
vector<int> conteo(vector<int> &lista){  
    //creo un vector inicializado en 0  
    //cuya longitud sea igual a una cota máxima  
    vector<int> conteo(COTA,0);  
    for(int i=0;i<lista.size();i++){  
        conteo[lista[i]]++;  
    }  
}  
  
vector<int> countingSort(vector<int> &lista){  
    vector<int> conteo = contar(lista);  
    return reconstruir(lista,conteo);  
}
```

Counting Sort

```
vector<int> reconstruir(vector<int> &lista, vector<int>
    > conteo){
    vector<int> resultado(lista.size());
    int indice_conteo = 0;
    for(int i = 0; i<lista.size();i++){
        while(conteo[indice_conteo]==0){
            indice_conteo++;
        }
        lista[i] = indice_conteo;
        conteo[indice_conteo]--;
    }
}
```

¿Complejidad?

¿Complejidad? $\mathcal{O}(n)$

IMPORTANTE: Para utilizar este algoritmo necesitamos una cota del elemento mas grande que puede tener el arreglo

Algoritmos de búsqueda

Un algoritmo de búsqueda, es un algoritmo que sirve para saber si un elemento pertenece (o no) a una estructura de datos. En general devuelve verdadero o falso, en algunas implementaciones (la función *find* de c++ por ejemplo).

Hay distintos algoritmos de búsqueda dependiendo de la estructura, nosotros vamos a ver algoritmos de búsqueda para secuencias (válidos también para arreglos, vectores o cualquier otra estructura con **acceso aleatorio**).

Búsqueda Lineal

La búsqueda lineal consiste en recorrer la secuencia un elemento a la vez, viendo si el elemento existe o no. En peor caso, vamos a ver una vez cada elemento de la secuencia (Nota: Esto es lo mejor que podemos lograr **si no tenemos ninguna precondition** acerca de la secuencia).

Búsqueda Lineal

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

Búsqueda Binaria

La búsqueda binaria es un algoritmo que podemos utilizar si la secuencia está **ordenada**. El mismo consiste en ver que mitad del arreglo se debería encontrar al elemento y repetir este procedimiento recursivamente hasta tener un elemento (o encontrar el elemento que buscamos).

Búsqueda Binaria (versión recursiva)

```
bool binarySearch(vector<int> lista, int desde, int
    hasta, int k){
    if (hasta >= desde){
        int medio = desde + (hasta - desde)/2;

        if (lista[medio] == k){ //encontre el
            elemento
            return true;
        }
        if (lista[medio] > k){ //esta en la mitad
            izquierda
            return binarySearch(lista, desde, medio-1,
                k);
        }
        //esta en la mitad derecha
        return binarySearch(lista, medio+1, hasta, k);
    }
    //no esta
    return false;
}
```

Búsqueda Binaria (versión iterativa)

```
bool binarySearch(int lista[], int desde, int hasta,
    int k)
{
    while (desde <= hasta){
        int m = desde + (hasta-desde)/2;
        if (lista[m] == k){ //encontre el elemento
            return true;
        }

        if (lista[m] < k){ //esta en la mitad derecha
            desde = m + 1;
        }else{ //esta en la mitad izquierda
            hasta = m - 1;
        }
    }
    // el elemento no esta
    return false;
}
```

Ejercicio 1: Pico mas alto

Pico más Alto

Implementar un algoritmo que dado una secuencia de numeros devuelva el pico mas alto. Definimos como pico a un elemento que es el mayor de una subsecuencia creciente (de al menos dos elementos) de la secuencia y de una subsecuencia decreciente (de al menos dos elementos) de la secuencia. El mas alto es el que mayor diferencia tenga con el menor elemento de ambas subsecuencias.

Ejercicio 2: Pico mas largo

Pico más Largo

Implementar un algoritmo que dado una secuencia de numeros devuelva el pico mas largo. Definimos como pico a un elemento que es el mayor de una subsecuencia creciente (de al menos dos elementos) de la secuencia y de una subsecuencia decreciente (de al menos dos elementos) de la secuencia. El mas largo es el que sume mas elementos en la concatenación de ambas subsecuencias.

Ejercicio 3: k-ésimo restringido

k-ésimo restringido

Implementar un algoritmo que dado una secuencia de numeros devuelva el k-ésimo natural que NO pertenezca a dicha secuencia. Asumir que la secuencia no posee valores repetidos. Ayuda: Considere ordenar la secuencia.

Ejercicio 3: k-ésimo restringido

k-ésimo restringido

Implementar un algoritmo que dado una secuencia de numeros devuelva el k-ésimo natural que NO pertenezca a dicha secuencia.

Asumir que la secuencia no posee valores repetidos. Ayuda:

Considere ordenar la secuencia.

¿Tendrían que modificar el algoritmo si la secuencia tuviera valores repetidos?