Clase de Ordenamiento, Búsqueda y otros Algoritmos sobre Secuencias

Algoritmos y Estructuras de Datos I

Gonzalo Guillamon → Lucas Somacal → Matias G. Marset

Departamento de Computación
Universidad de Buenos Aires

Primer Cuatrimestre 2020

Contenido

Algoritmos sobre secuencias:

- Ordenamiento
- Búsqueda

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);
    i--;
  }
}

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 $1, ..., n$ 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 $n, ..., 1$ 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.
- La lista es una permutación de la lista original.

Bubble Sort

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

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);
    }
  }
}
```

Bubble Sort

¿Complejidad?

Bubble Sort

¿Complejidad? $\mathcal{O}(n^2)$
(intuición: burbujeo se hace n veces y hace $n, ..., 1$ 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. Luego se rearma la lista original poniendo la cantidad correspondiente de i-ésimos elementos de la segunda lista.

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 de la lista de conteo es i
- La lista original se mantiene.

Counting Sort

```
vector<int> countingSort(vector<int> &lista){
  vector<int> conteo = contar(lista);
  return reconstruir(lista,conteo);
}

vector<int> contar(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]]++;
  }
```

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++){
    // Ignoro valores nulos
    while(conteo[indice_conteo]==0){
      indice_conteo++;
    }
    lista[i] = indice_conteo;
    conteo[indice_conteo]--;
  }
```

¿Complejidad?

¿Complejidad? $\mathcal{O}(n + COTA)$ (en gral $\mathcal{O}(n)$ si $COTA$ es similar o más chico que n)

IMPORTANTE: Para utilizar este algoritmo necesitamos una cota ($COTA$) del elemento más grande que puede tener la lista

Algunos videos demostrativos:

- Insertion Sort:
  `https://www.youtube.com/watch?v=ROalU37913U`
- Selection Sort:
  `https://www.youtube.com/watch?v=Ns4TPTC8whw`
- Bubble Sort:
  `https://www.youtube.com/watch?v=lyZQPjUT5B4`

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 (como en 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 arbitrario**).

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 precondición** acerca de la secuencia).

Búsqueda Lineal

```
int 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 de la lista 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, r, k);
   }
 //no esta
   return false;
}
```

Búsqueda Binaria (versión iterativa)

```
int 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;
}
```