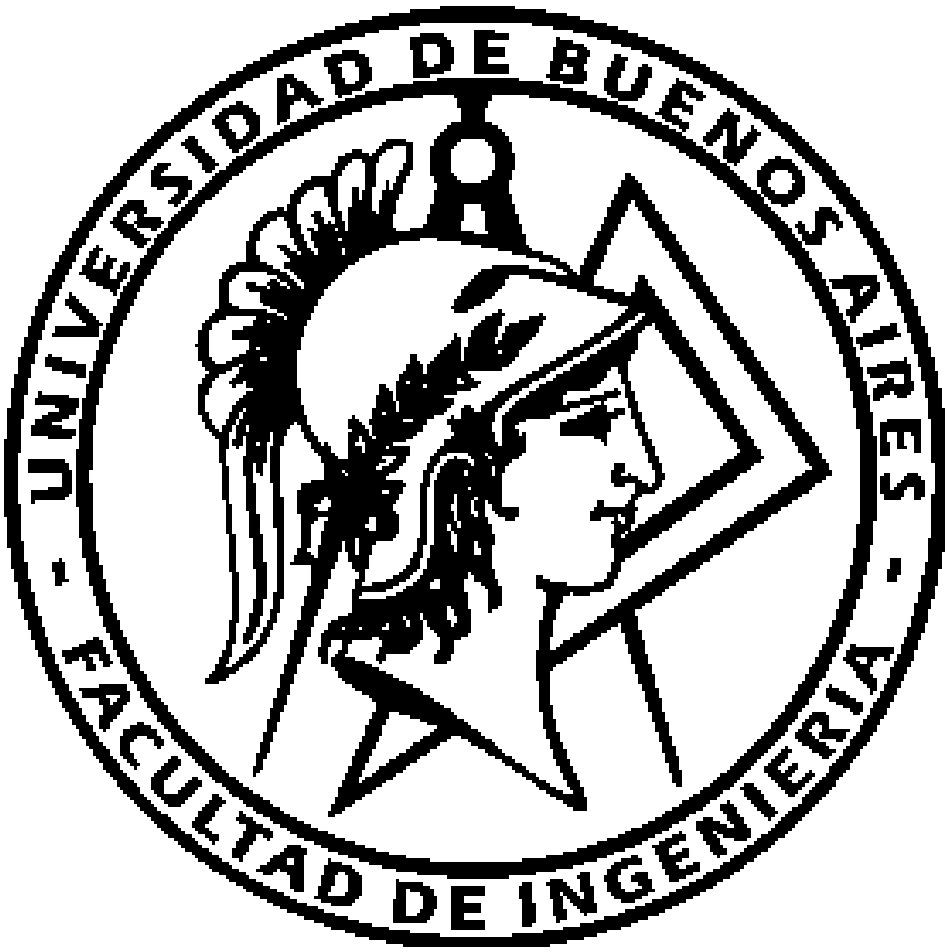
Universidad de Buenos Aires



Teoría de los Algoritmos (75.29)

Reentrega Trabajo Practico Nro 1

2do Cuatrimestre 2016

Padrón	Apellido y Nombre	email
89563	Ferreyra, Oscar	fferreyra38@gmail.com
	Eisner,Ariel	

Fecha de reentrega: 25 de noviembre de 2016

Índice

Modo Ejecución

El programa de inferencias estadísticas recibe 4 parámetros

Ej: datafake.txt 3 8 9

Nombre del Archivo de Datos

Texto plano con los números separados por una “,”

Numero de Método

1.FuerzaBruta

2. Ordenar y Seleccionar

3. K-Selecciones

4.K_HeapSort

5.HeapSelect

6.QuickSelect

La posición a consultar dentro del vector

El valor en si

Calculo de Orden de los Algoritmos Solicitados

Fuerza Bruta

```
public boolean calcularPorFuerzaBruta(int k, int valor) {
      int cantidadDatos = contenedorDatos.size();                          //1

      boolean seEncontroElemento = false;                                  // 1

      int indice = 0;                                                      // 1

      while (!seEncontroElemento) {                                       // K

            int elementoActual = contenedorDatos.elementAt(indice);  // 2

            int menores = 0;                                             // 1

            for (int i = 0; i < cantidadDatos; i++) {                    // N
                  if (i != indice) {//para que no analice el actual      // 1
                  int elementoAComparar = contenedorDatos.elementAt(i); // 2
                  //no se contemplan caso de valores repetidos
                        if (elementoAComparar < elementoActual) {        // 1
                              menores++;                                 // 2
                        }
                  }
            }
            seEncontroElemento = ((menores + 1) == k);// si se encontro// 3

            indice++;          // 2
      }
      return (contenedorDatos.elementAt(indice-1) == valor); // 3
   }
```

Orden

El algoritmo de fuerza bruta recorre los elementos del arreglo, y a cada uno lo compara contra todos los demás para determinar la cantidad de elementos menores al considerado, en cada. Si encuentra alguno para el cual dicha cantidad sea k, corta el ciclo en ese momento y lo devuelve como resultado.

Complejidad: O(n^2)

Mejores casos: O(n) si el elemento con k menores se encuentra en la primera

Peores casos: O(n^2) si el elemento con k menores se encuentra en la última posición.

$$T(n) = 1 + 1 + 1 + \sum_{i=1}^{K} 2 + 1\ [\sum_{i=1}^{n} 1 + 2 + 1 + 2] + 3 + 2\ = 3 + \sum_{i=1}^{K} 8\ [\sum_{i=1}^{n} 6]$$

$$\mathrm{T(n)} = 3 + 8\ \mathrm{K}\ (\ 6\ \mathrm{N}\ ) = 3 + 48\ \mathrm{K\ N} = \in O(\mathrm{K\ N})\ y\ si\ K\ es\ el\ ultimo\ elemento \in O(n^2)$$

Ordenar y Seleccionar

```
public boolean calcularPorOrdenarSeleccionar(Integer pos, Integer valor) {
      mergeSort(0, contenedorDatos.size()-1);                        //N log(N)

      if (contenedorDatos.elementAt(pos-1) == valor) {   // 3
            return true;                                 // 1
      }
      return false;                                      // 1
}
```

Orden

Este algoritmo tiene dos partes. La primera, que ordena el conjunto por el metodo de mergesort y la segunda, que selecciona un elemento de ese conjunto. El orden del algoritmo dependerá completamente del orden del algoritmo de búsqueda.

Complejidad: O(n log n)

Mejor y peor caso: O(n log n)

$$T(n) = \mathbf{n}.\mathbf{log_2}(\boldsymbol{n}) + 3 \ + 1 = \mathbf{n}.\mathbf{log_2}(\boldsymbol{n}) + 4 \in O(\mathbf{n}.\mathbf{log_2}(\boldsymbol{n}))$$

K-Selecciones

```
public boolean calcularPorKSelecciones(Integer pos, Integer valor){
	Integer minimo = Integer.MAX_VALUE;		//1

	Integer indiceMin = 0;				//1

	for (int i = 0; i < pos; i++) {				//K
		for (int j = 0; j < contenedorDatos.size(); j++) {	//n
			if(contenedorDatos.elementAt(j) < minimo){	//2
				minimo = contenedorDatos.elementAt(j);	//2
				indiceMin = j;				//1
			}
		}
		selecciones.set(i,minimo);			//1
		contenedorDatos.removeElementAt(indiceMin);	//1
		minimo = Integer.MAX_VALUE ; indiceMin = 0;		//2
	}
	return verificarPosPorSeleccion(pos-1,valor);	//1 + T(VerPos)
}

private boolean verificarPosPorSeleccion(Integer pos, Integer valor) {
	if(selecciones.elementAt(pos) == valor)	//1
		return true;			//1

	return false;		//1
}
```

Orden

El ordenamiento por selección es O(n^2) porque recorre todo el conjunto para hallar el menor, lo coloca en el primer lugar y vuelve a repetir el proceso para todos los elementos restantes. Dado que no se necesita ordenar todo el arreglo, sino solo encontrar el k-esimo elemento menor, deberemos recorrer el conjunto de tamaño n, k veces.

Complejidad: O(n k)

Mejor casos: O(n) si k es uno

Peores casos: O(n^2) si k es igual a n

$$T(n) = 2 + \sum_{i=1}^{K}\sum_{j=1}^{n}(2+2+1) \;+\; 1+1+2] + 1 + 2 = 5 + \; k(n+5)$$

$$\mathrm{kn} + 5k \;\in O(\mathrm{k\,n})\; y\; si\; K\; es\; el\; ultimo\; elemento \in O(n^2)$$

K-HeapSort

```
public boolean calcularPorKSeleccionesEnHeap(Integer k, Integer valor){
        cargarDatosAlheapMinimo();            // nlog2(n)

        return buscarEnHeapMinimo(k,valor);  // klog2(n)
 }

private void cargarDatosAlheapMinimo() {
   for (int i = 0; i < contenedorDatos.size(); i++) {   //n
      heapMinimo.agregar(contenedorDatos.elementAt(i)); // log2(n)
   }
}


private boolean buscarRefEnHeap(Integer k, Integer valor) {
      --k;                                  //1
      while(k > 0){                         //K
            heap.eliminarMin();             //log2(n)
            --pos;                          //1
      }
      if (heap.obtenerMin() == valor)       //2
            return true;                    //1

      return false;                         //1

}
```

Orden

Cargar los datos en el heap tiene una complejidad de O(n log n). El extraer un elemento de este obliga a realizar un downheap, el cual se realiza en un tiempo de O(log n). Si se realizan k extracciones, se deberá realizar un downheap k veces. Acceder al elemento en la raiz se realiza en orden constante. Ergo la complejidad del K-Heapsort será O(n log n+k log n), el tiempo de la carga del heap mas el de la búsqueda en él mismo. Como k puede ser a lo sumo igual a n, el orden del algoritmo será como máximo

O(n log n + n log n) → O(2n log n), es decir O(n log n).

Complejidad: O(n log n)

Mejores casos: O(n log n) si k es igual a 1. Porque solo debo realizar el ordenamiento de heapsort y evito los downheap de cada extracción.

Peores casos: O(2n log n) si k es igual a n.

$$T(n) = \sum_{i=1}^{n} \mathbf{log_2}(\mathrm{n}) + \ 1 + \sum_{i=1}^{k} [\mathbf{log_2}(\mathrm{n}) + 1] + 2 + 1 =$$

$$T(n) = \sum_{i=1}^{n} \mathbf{log_2}(\mathrm{n}) + \sum_{i=1}^{k} [\mathbf{log_2}(\mathrm{n}) + 1] + 4$$

$$\mathbf{nlog_2}(\mathrm{n}) + k\mathbf{log_2}(\mathrm{n}) \ \in O((\mathrm{k} + \mathrm{n})\mathbf{log_2}(\mathrm{n}))$$

K-HeapSelect

```
public boolean calcularPorHeapSelect(int k, int valor) {
      for (int i = 0; i < k; i++){                          // K
            heapMaximo.agregar(contenedorDatos.elementAt(i)); //log (K)
      }

      for (int i = k; i < contenedorDatos.size(); i++){ //N - K
            if(heapMaximo.obtenerMax() > contenedorDatos.elementAt(i)){
                  heapMaximo.eliminarMax();//log (K)

                  heapMaximo.agregar(contenedorDatos.elementAt(i)); //log (K)
            }
        }
       return heapMaximo.obtenerMax() == valor; // 2
}
```

HeapSelect usa un heap de maximo, pero este heap tiene un total de k elementos. Esto implica que las operaciones, como la de remover la raiz, seran O(log k). Para determinar los menores elementos se comparan todos contra el maximo del heap, el cual será el k-iesimo valor buscado. En caso de que alguno sea menor que el máximo, se lo agrega al heap, removiendo el máximo anterior. Esto implicar dos operaciones O(log k).

Complejidad: O(n log k)

Mejores casos: O(k log k) si los k elementos menores están en las primeras posiciones del conjunto, lo que evita modificar y cargar el heap luego de su creación. En este caso, nunca se cumpliría el if del segundo bucle for.

Peores casos: O(n log k) si los k elementos menores están en las últimas posiciones del conjunto, lo que evita que implica que todos los elementos del conjunto serán agregados al heap y la n k serán removidos.