Universidad de Buenos Aires.
Facultad de Ingeniería.

Teoría de Algoritmos I 75.29
Trabajo Práctico Nro I

Integrantes	Patrón
Blanco, Joaquin	94653
Almirón, Diego	94051
Zanellini, Gaspar Ruben	95768

Índice

Introducción:

El objetivo del siguiente trabajo práctico es la implementación y análisis de algunos algoritmos clásicos. Se especificará cuál algoritmo es el más eficiente dependiendo de la entrada.

El lenguaje de programación utilizado es python.

Primera Parte:

Complejidad de los algoritmos:

Fuerza Bruta:

Es evidente que es de orden $O(n^2)$. Cada iteración está acotada por n (la cantidad de elementos) y el elemento podría ser el último de la lista. $O(n^2)$

Ordenar y Seleccionar:

El algoritmo de ordenamiento es de orden $O(nlog(n))$ y la selección toma tiempo constante, entonces $O(nlog(n))$

k-selecciones:

En cada paso del algoritmo se busca el mínimo en un arreglo que tiene un elemento menos que el del paso anterior. $T(n) <= kn - \sum_{i=1}^{k-1} i <= ckn \Rightarrow O(kn)$

k-heapsort:

El tiempo de pop de un heap está acotado por $log(n).$ Luego

$T(n) <= \sum_{i=0}^{k} log(n-i) <= klog(n) \Rightarrow O(klog(n))$

k-heapselect:

El tiempo de pop está acotado por $log(k)$. Luego $O(nlog(k))$

Quick-select:

En promedio es lineal

Peores y Mejores Casos:

Numeramos a partir de 0, o sea, que k=0 se refiere al elemento más pequeño del array

Fuerza bruta:

Mejor Caso: el k-ésimo elemento está primero en el array.

Con k=0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Con k=8

8	1	2	3	4	5	6	7	9	0	10	11	12	13	14	15

Con k=15

15	1	2	3	4	5	6	7	8	0	10	11	12	13	14	0

Peor Caso: el k-ésimo elementos está en la última posición.

Con k=0

9	1	2	3	4	5	6	7	8	15	10	11	12	13	14	0

Con k=8

15	1	2	3	4	5	6	7	9	0	10	11	12	13	14	8

Con k=15

9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15

Ordenar y Seleccionar:

Mejor Caso: Dado que usamos el algoritmo sort de python para listas el cual es Timsort y este es igual a un insertion sort para listas con menos de 64 elementos. El mejor caso es cuando el arreglo ya está ordenado, independientemente del k.

Con k=0, k=8, k=15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Peor Caso: por las mismas razones dadas a conocer en el mejor caso, el peor caso se da con un arreglo en el orden inverso.

Con k=0, k=8, k=15

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

K-Selecciones:

Dado que en cada paso paso siempre se busca un mínimo en una cadena y que esto siempre lleva n pasos (cantidad de elementos de la cadena actual) podemos decir que el algoritmo es igual de caro para cualquier combinación de números. Por esta razón no daremos ejemplos.

K-Heapsort:

Mejor Caso: El arreglo ya está ordenado

Con k=0, k=8, k=15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Peor Caso: el arreglo está ordenado de cierta forma que al aplicarle heapify todos los nodos “naden” hasta lo más profundo del árbol y el arreglo quede ordenado. Así, cada vez que se hace un pop se reemplaza el mínimo por el máximo elemento y este debe nadar hasta el fondo del árbol nuevamente.

Con k=0, k=8, k=15

15	7	12	3	10	11	14	1	8	9	4	2	5	13	6	0

HeapSelect:

Mejor Caso: El arreglo ya está ordenado

Con k=0, k=8, k=15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Peor Caso: El arreglo está ordenado en sentido inverso

Con k=0, k=8, k=15

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

QuickSelect:

En nuestra implementación de quickselect se toma el último elemento elemento del array como pivote.

Mejor Caso: se da cuando el último elemento es el que ocuparía la posición k-ésima si el arreglo estuviera ordenado.

Con k=0

15	1	2	3	4	5	6	7	8	9	10	11	12	13	14	0

Con k=8

15	1	2	3	4	5	6	7	0	9	10	11	12	13	14	8

Con k=15

8	1	2	3	4	5	6	7	0	9	10	11	12	13	14	15

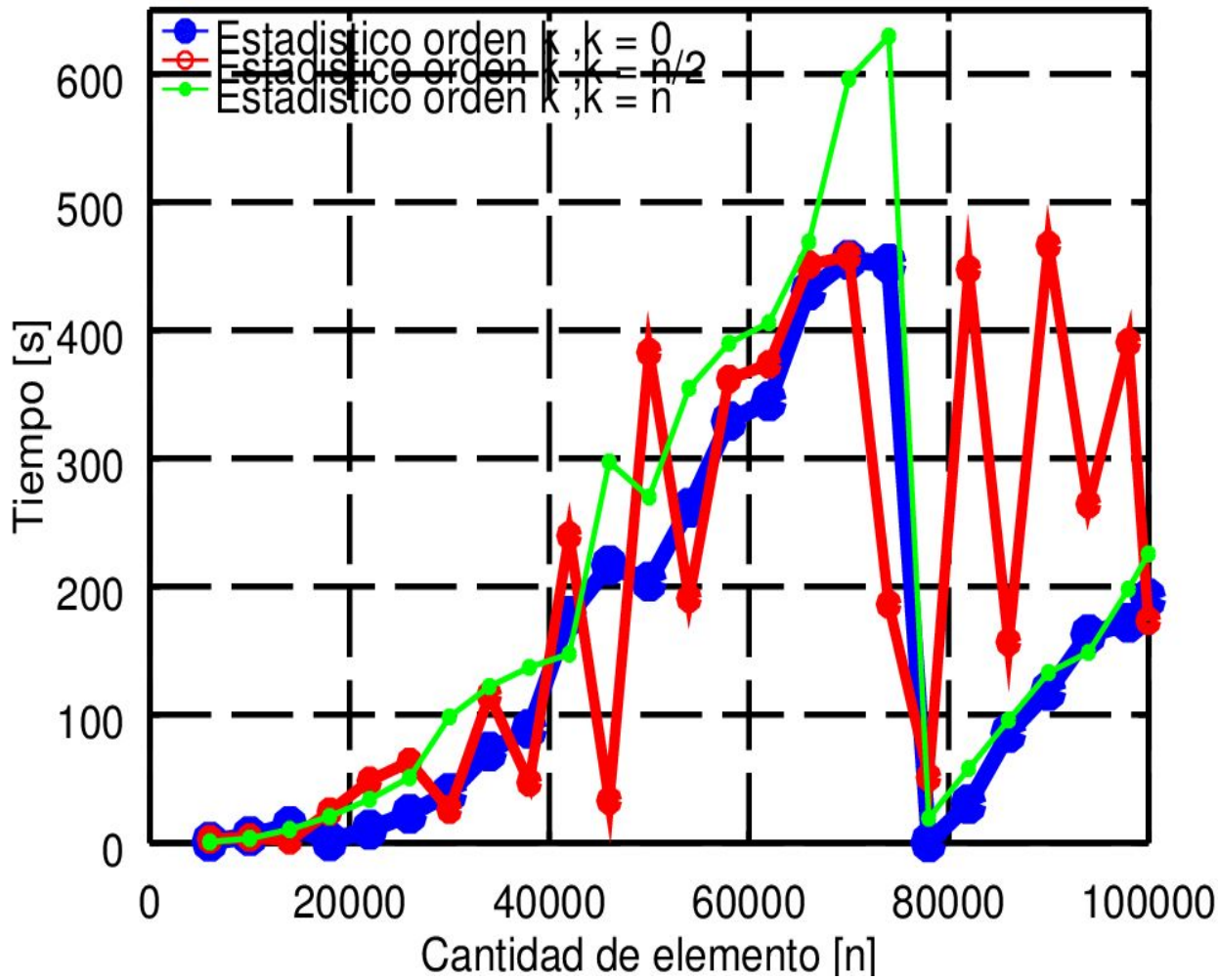
Peor Caso: El arreglo está ordenado

Con k=0, k=8, k=15

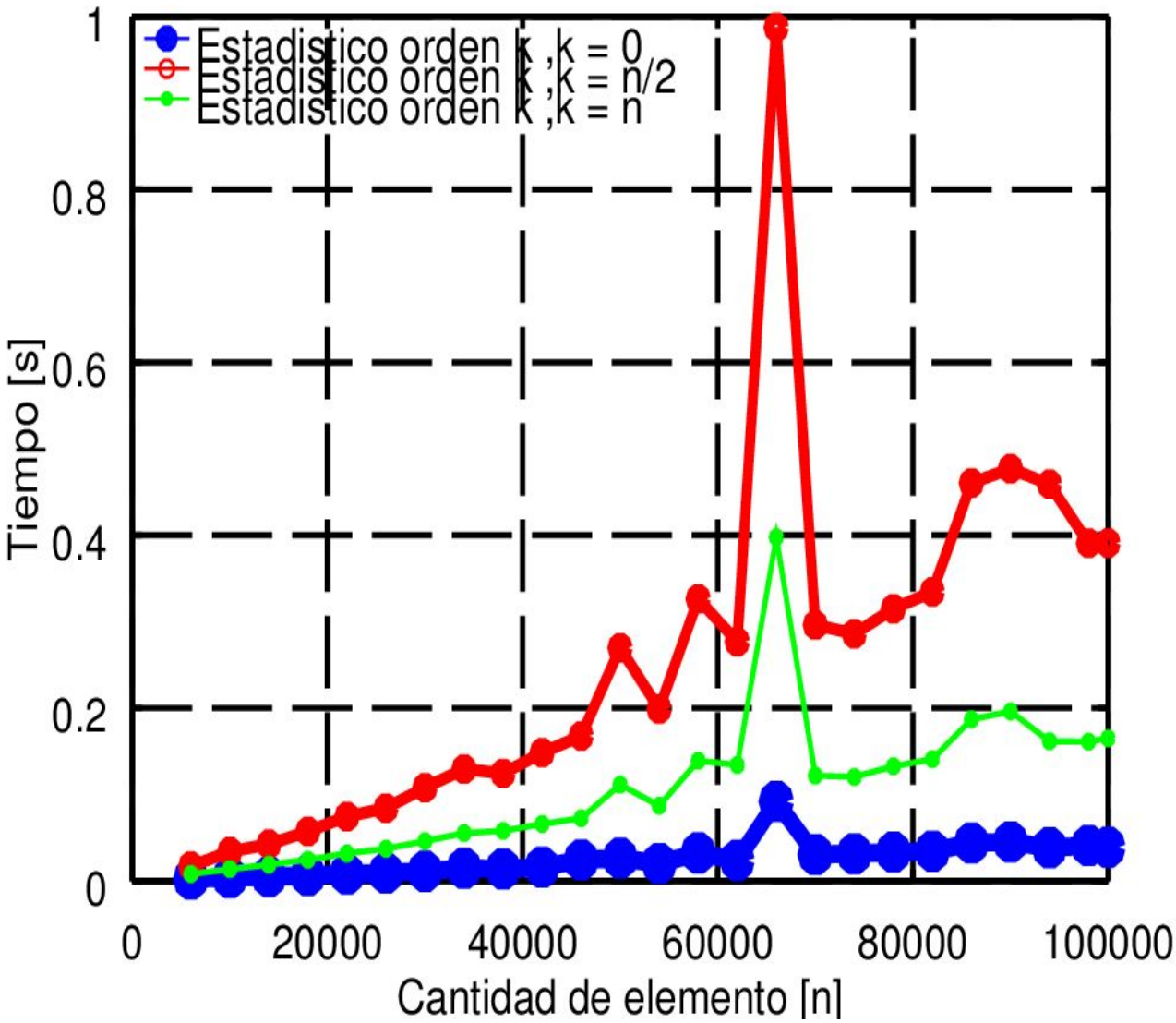
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Gráficos de tiempos de ejecución

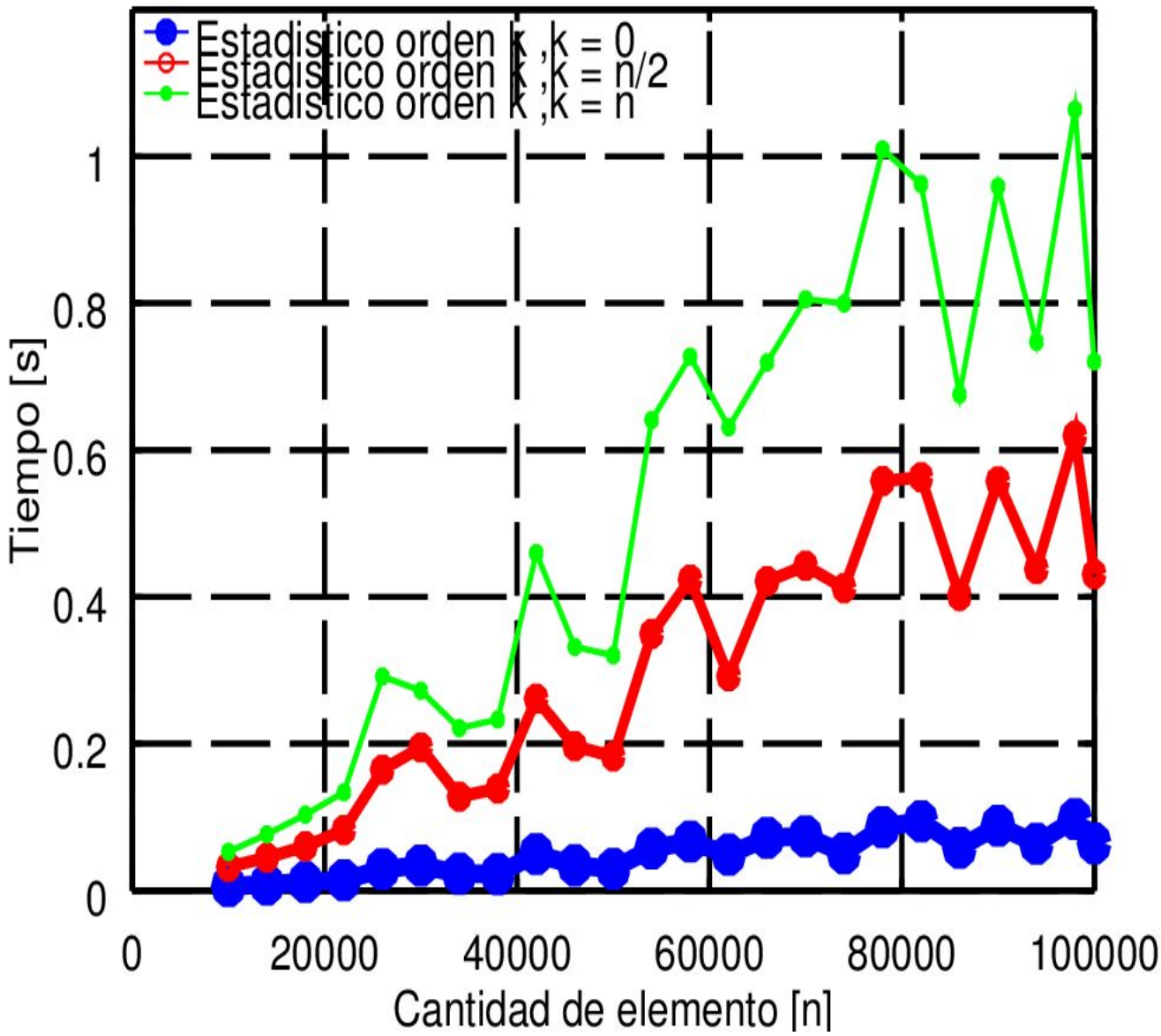
Fuerza Bruta



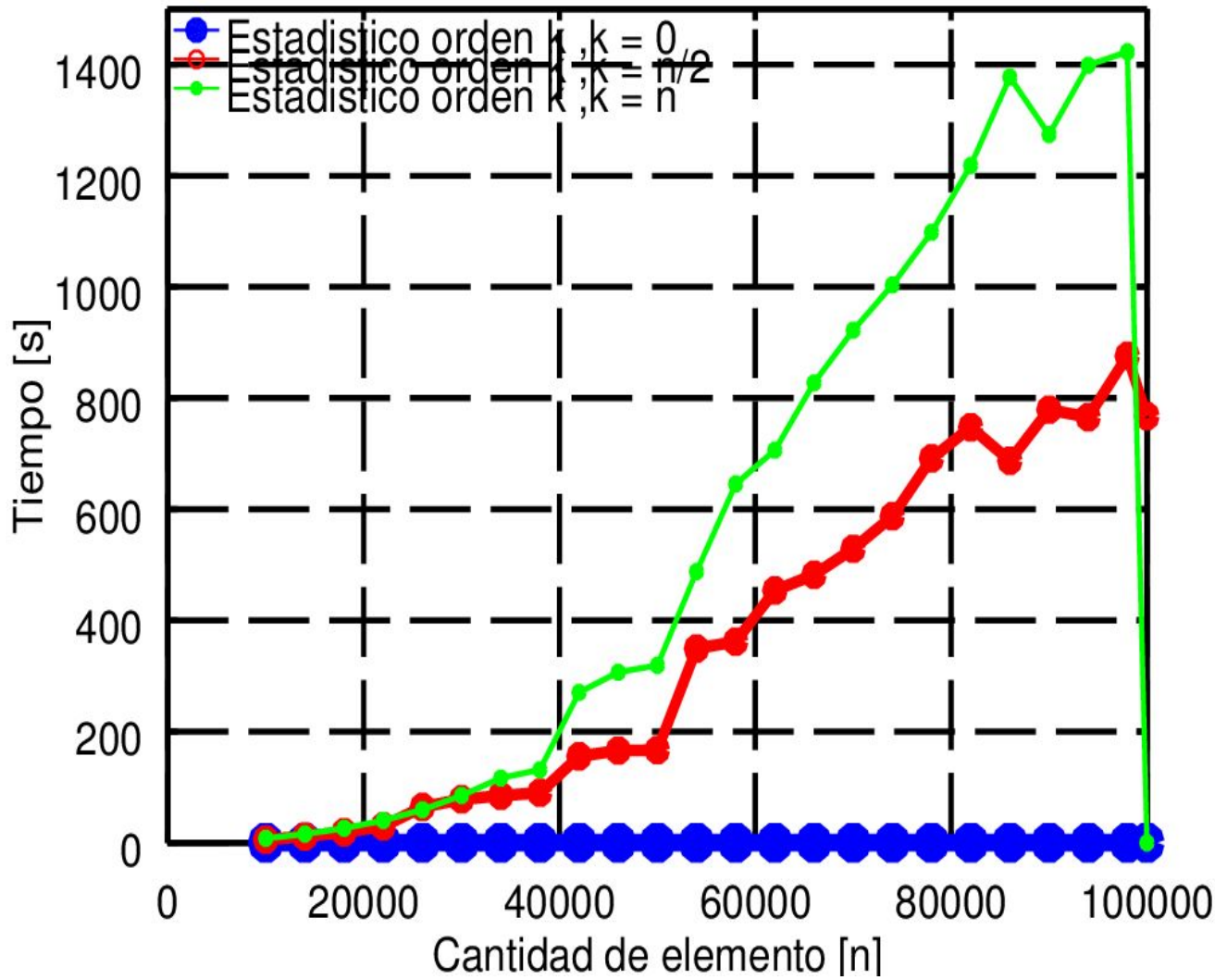
Heapselect


Estadistico orden k ,k = 0
Estadistico orden k ,k = n/2
Estadistico orden k ,k = n
Tiempo [s]
0
0.2
0.4
0.6
0.8
1
0
20000
40000
60000
80000
100000
Cantidad de elemento [n]

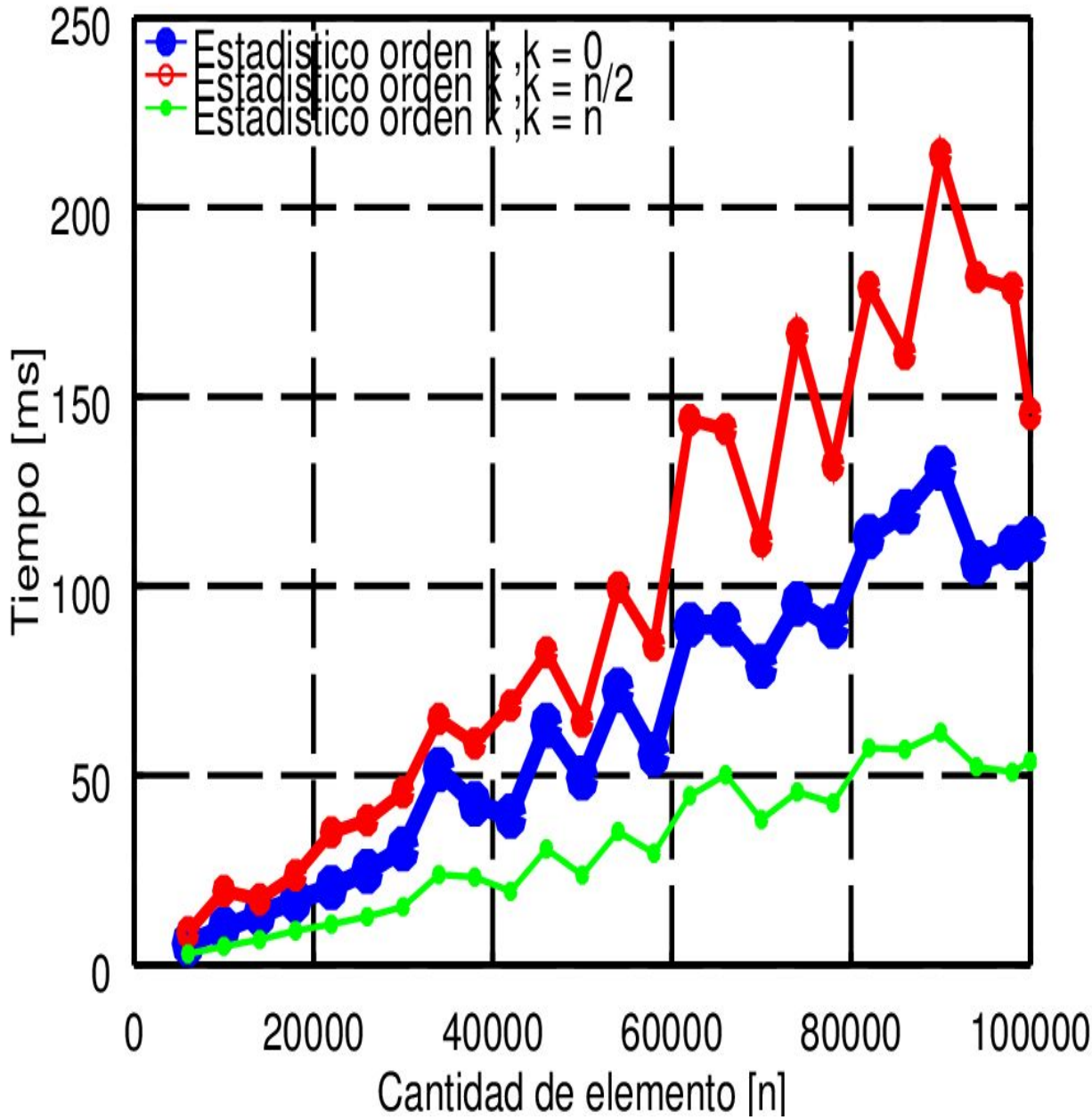
K-Heapsort


Estadistico orden k ,k = 0
Estadistico orden k ,k = n/2
Estadistico orden k ,k = n
Tiempo [s]
0
0.2
0.4
0.6
0.8
1
0
20000
40000
60000
80000
100000
Cantidad de elemento [n]

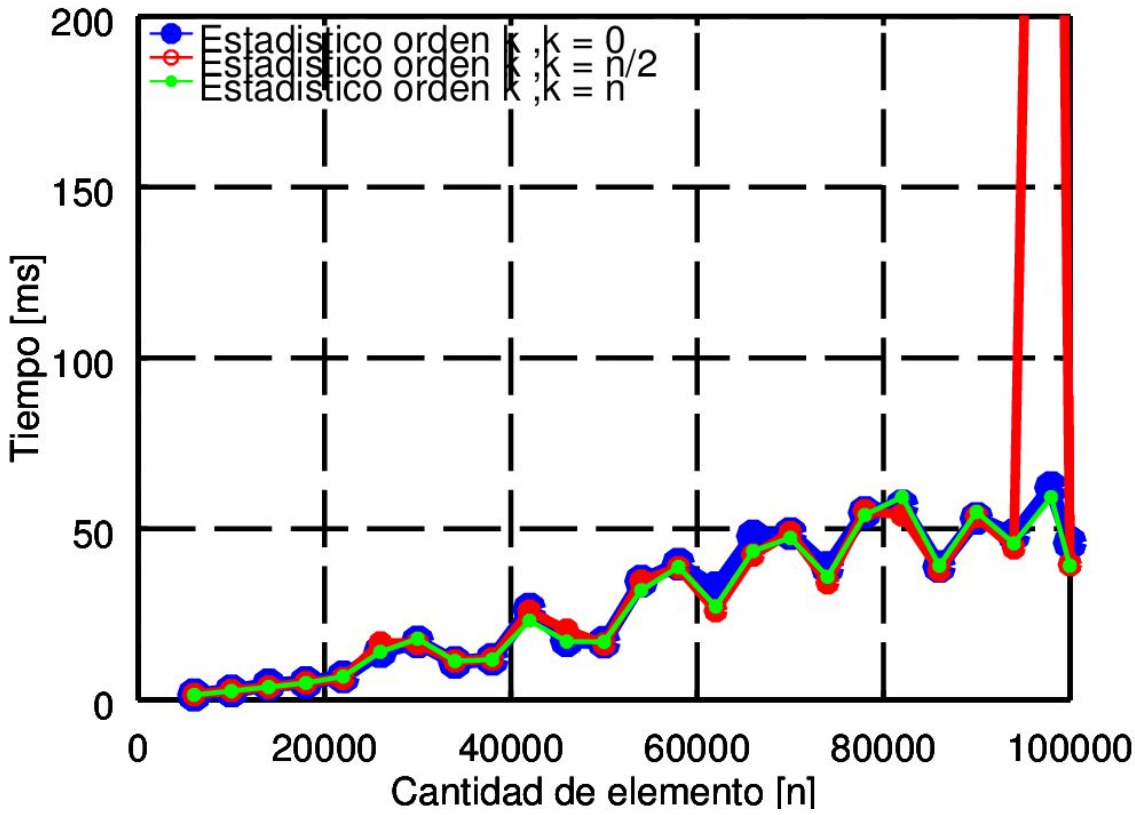
KSelecciones



QuickSelect



Ordenar y Seleccionar



Algoritmo Óptimo para cada entrada:

En este análisis no se contempla nuestras implementaciones particulares de los algoritmos para no hacerlo dependiente del lenguaje elegido. Un ejemplo de estas dependencias es que no existe un maxheap en python y tuvimos que pushear el inverso aditivo de cada elemento en vez de llamar a heapify de una vez.

Los algoritmos se listan en orden decreciente respecto de su optimalidad para el tamaño de cada entrada.

Se supone que el quick-select es mejor que el heap en promedio debido a la forma que tienen las entradas en la vida real.

K=0

- **k-selecciones:** solamente hay que buscar el mínimo.
- **quick-select:** debido a la distribución de las entradas en la vida real
- **k-heapsort:** al armar un heap con todos los elementos se ahorran muchas comparaciones y swaps comparado con las utilizadas en el **heap-select**.
- **heap-select:** su tiempo está acotado linealmente y el del siguiente no.
- **ordenar y seleccionar:** esta acotado por n.log(n).

- **fuerza bruta:** orden cuadrático.

K=n/2

- **quick-select:** debido a la distribución de las entradas en la vida real.
- **ordenar y seleccionar:** se supone que el algoritmo que usa python es más eficiente que **heapsort** (a menos es promedio). El orden de usar **k-heapsort** es igual al de **ordenar y seleccionar** para esta entrada específica e intuitivamente si llamamos al pop del heap n/2 veces hemos hecho la mitad (o un poco más) del trabajo necesario para ordenar el array.
- **k-heapsort:** para escoger este algoritmo en vez de **heapselect** voy a basarme en el peor caso. Notemos que una vez que hemos armado el array de k elementos en el **k-heapsort**, para cada uno de los restantes elementos en el array (la mitad) será necesario un pop y push (en el peor caso) en un árbol de altura $log(n/2) = log(n) - 1$ (aproximadamente) mientras que en el **k-heapsort** solo se necesita un pop en un árbol de altura $log(n)$. Construir el heap del **heapselect** en primer lugar está acotado por cn/2 (c una constante). Como podemos ver, el trabajo de construir el heap para la mitad de los elementos es similar al del construir el heap para todos los elementos.
- **heapselect:** Su tiempo está acotado por $O(nlog(k))$
- **k-selecciones:** Su tiempo es cuadrático al igual que **fuerza bruta** pero se detiene después de n/2 ejecuciones. Mientras que **fuerza bruta** necesita n^2 pasos (en el peor de los casos).
- **fuerza bruta:** si bien fuerza bruta puede terminar antes que **k-selecciones**, vemos que hay 50% de chances de que el elementos esté en la mitad de la derecha, y eso significa que al menos la mitad de la veces **fuerza bruta** es más lento. Además si consideramos que k-selecciones se invoca en arreglos cada vez más pequeños y que **fuerza bruta** siempre en el mismo, vemos que las chances de que **fuerza bruta** sea peor que **k-selecciones** aumentan.

K=n-1

- **quick-select:** debido a la distribución de las entradas en la vida real.
- **heapselect:** se arma un max heap con todos los elementos y se invoca a pop una única vez.
- **ordenar y seleccionar:** se supone que el algoritmo sort en python se comporta mejor que heapsort en promedio.
- **k-heapsort:** es equivalente a ordenar el arreglo con un heapsort.
- **fuerza bruta:** aunque en el peor de los casos, puede tardar un poco más que **k-selecciones**, intuitivamente vemos que hay muchas más chances de que termine antes. Por ende, en promedio se comporta mejor que **k-selecciones.**
- **k-selecciones:** por las mismas razones dadas en **fuerza bruta**.

Segunda Parte

Complejidad de los algoritmos:

Aclaraciones: el número de vértices se representa por la letra n y el de aristas por m.

BFS

Dado a que se recorren todas la aristas de los vértices agregados a la cola y que cada vértice se agrega una única vez, resulta que $O(n + m)$.

Dijkstra

Utilizando una cola de prioridad con una operación change key el orden del algoritmo es $O(m.log(n))$ ya que en la cola hay a lo sumo n elementos y se hacen a lo sumo m operaciones de change key. Pero, la cola de nuestra implementación no tiene esta operación. Lo que agregamos a la cola son las aristas en vez de los nodos. La condición de corte es que la cola esté vacía o encontrar el nodo destino. Cómo a lo sumo podemos tener m aristas en la cola y podemos hacer a lo sumo m pops, se concluye que $O(m.log(n))$ es el orden de nuestra implementación.

Búsqueda con heurística

En el peor de los casos se hace un pop de una cola de prioridad hasta que quede vacía. Ahora, en nuestra implementación, un nodo puede ser agregado varias veces a la cola ya que puede ser adyacente a varios nodos (consultar código fuente para mayor entendimiento). A lo sumo puede haber m elementos elementos en la cola (uno por cada arista), así que el orden es $O(m.log(m))$

A*

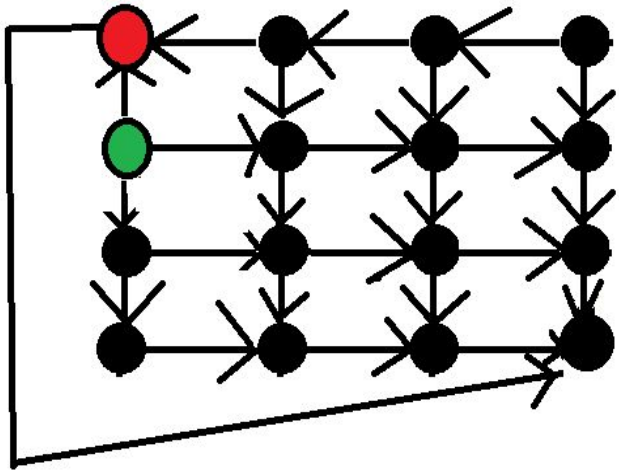
El algoritmo es exactamente igual al de **Dijkstra** (al menos en nuestra implementación) excepto que invoca dos veces a la heurística (para destino y fuente) cada vez que se agrega una arista a la cola de prioridad. Cómo se supone que la función heurística toma tiempo constante podemos decir que el orden es el mismo que el de **Dijkstra**, o sea, orden $O(m.log(m))$

Peores y Mejores Casos:

A menos que se indique lo contrario el peso de una arista es 1 por defecto. El nodo origen está pintado de verde y el destino de rojo. Todos los grafos en los ejemplos son grillas. Se usa la distancia Manhattan como heurística en todos los casos correspondientes.

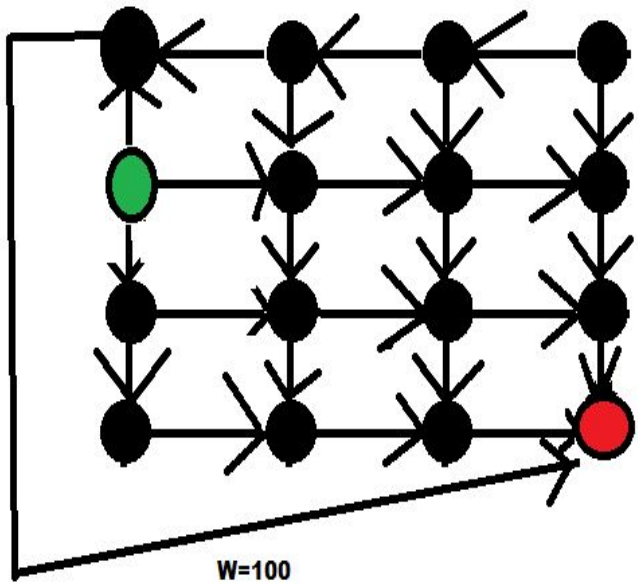
BFS, DIJKSTRA, Búsqueda con heurística, Mejor Caso:

El destino es adyacente al origen y todas las aristas tienen el mismo peso



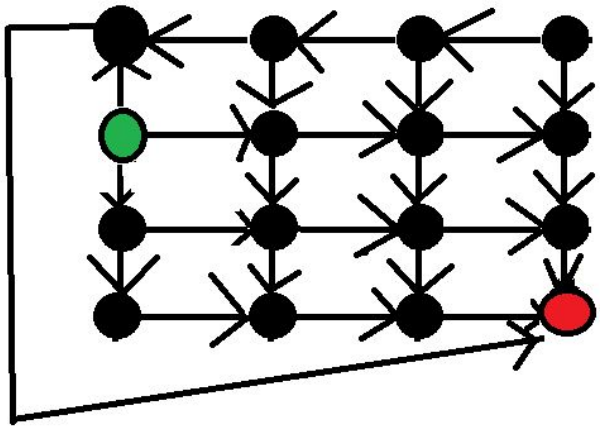
BFS DIJKSTRA Peor Caso:

Todas las aristas tiene el mismo peso, exceptuando una, y se toma el nodo más lejano como destino. Notar que la arista de peso diferente no afecta en nada al algoritmo ya que su peso es muy grande.



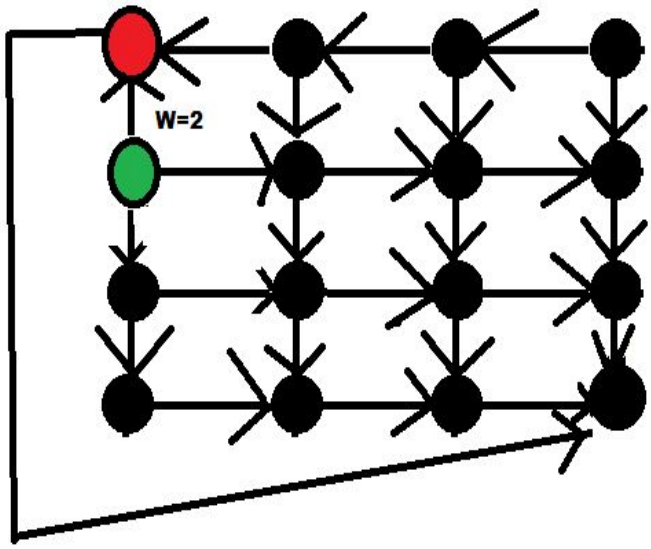
Búsqueda con Heurística Peor Caso:

Todas las aristas tiene el mismo peso. No solamente tarda más que **BFS**, además da una respuesta errónea.



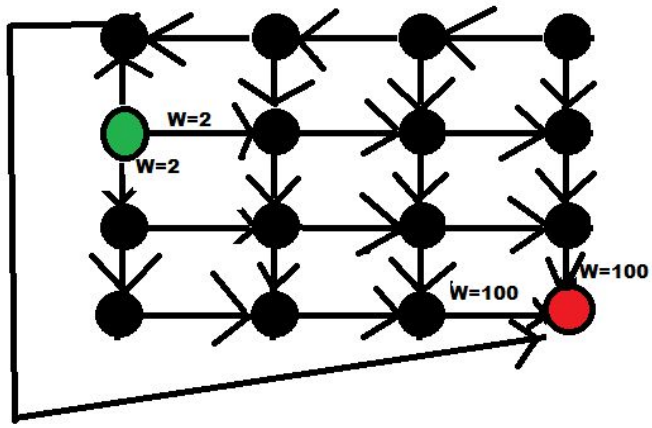
A* Mejor Caso:

El nodo destino es adyacente al origen. Notar que tarda menos que **Dijkstra**.



A* Peor Caso:

Notar que el nodo que está arriba del origen, es el más cercano al destino. Pero con los peso modificados debido a la heurística, este dista 2 del origen, mientras que los demás nodos distan 1. Este nodo será descubierto después de haber recorrido todos los nodos que distan 1 (con los nuevos pesos) que son la mayoría. Dijkstra sería más eficiente en este caso particular.



Algoritmo Óptimo para cada entrada:

Los algoritmos se listan en orden decreciente respecto de su optimalidad y eficiencia para cada entrada.

BFS:

Es el algoritmo más eficiente en los grafos sin peso, los demás algoritmos usan una cola de prioridad la cual es innecesaria cuando todos los pesos de las aristas son iguales. Este es el único algoritmo con orden lineal. Sin embargo, si la heurística funciona sin error, entonces el mejor algoritmo sería **SBH** ya que en cada paso visitará a un nodo perteneciente al camino entre el origen y el destino suponiendo que este path exista, sino, su orden sería peor que **BFS**.

Por último, este algoritmo no es óptimo si el grafo tiene peso.

DIJKSTRA:

Cuando la heurística no es muy buena, **Dijkstra** sería la mejor solución para un grafo con peso. En mi opinión personal, usaría este algoritmo siempre que el espacio de búsqueda sea pequeño para ahorrar el trabajo de buscar una función heurística y lo reemplazaría por **A*** cuando el espacio haya crecido tanto que **Dijkstra** ya sea lento.

Este algoritmo siempre es óptimo (a menos que los pesos sean negativos).

A*:

Cuando la heurística es buena es la mejor opción para un grafo con peso. Ya que priorizamos los nodos que están más cerca del destino que los más alejados.

En los casos de heurística consistentemente optimista o pesimista el algoritmo debería ser igual o peor al del **Dijkstra** ya que las aproximaciones de distancia están muy alejadas de las reales.

Si se puede asegurar $w + h(v) - h(u) > 0$ donde **w** es peso de una arista, **v** su cabez , **u** su cola y **h** la función heurística, entonces **A*** es óptimo.

Búsqueda con Heurística:

intuitivamente es más rápido que **Dijkstra** y que **A*** ya que no se preocupa por el peso de los nodos. Sin embargo solo se puede asegurar que sea óptimo cuando la heurística es perfecta.