PROJECT #1: PROJECT REPORT

COMPSCI 260 P FALL 2017

By

APARNA CHINYA RAMACHANDRA 10506242

ISHITA ACHARYA 68923829

PROJECT GUIDE:

PROF. DAN HIRSCHBERG

UNIVERSITY OF CALIFORNIA, IRVINE

1. Analysis

1. Quick Select

Quick select is used to find the kth smallest element in an unsorted array. Given an array 'Arr' of size n and integer $k \leq n$;

1. Pick a median (pivot) element 'm' at random from Arr.
2. Split Arr into subarrays 'Less' and 'Greater' by comparing each element to m as in Quicksort. Count the number of elements in 'Less' as L.
3. (a) If $L = k - 1$, then output 'm'.
(b) If $L > k - 1$, output QuickSelect(Less, k).
(c) If $L < k - 1$, output QuickSelect(Greater, $k - L - 1$)

Let us suppose we have an array of n elements, now if we try breaking the array into 2 pieces randomly, intuitively, the larger piece will have 3/4 elements. Therefore the recurrence can be T(n) <= (n-1) + T(3n/4).
Let T(n,k) be the time taken to find the kth element in an array of n elements and let T(n) = max[T(n,k)].
Firstly, it takes (n-1) comparisons to split the array into 2 pieces. These pieces can have sizes ranging 0 & n-1 or 1 & n-2 or 2 & n-3 so on upto n-1 & 0. The sub-array that we take up for recursion will depend on the value of k. Here let us always take the larger sub-array to recurse upon (since we are finding the upper bound).

T(n) < = (n-1) + Avg [T(n/2),...T(n-1)]
Now, for any i < n, let T(i) <= 4i;
T(n) <= (n-1) + avg [4(n/2), 4(n/2 +1),..,4(n-1)]
T(n) <= (n-1) + 4(3n/4)
T(n) <= 4n => O(n) hence, linear time.

Worst case would give O(n^2).

2. Deterministic Select

D-Select is used to find the kth smallest element in an unsorted array.
The concept behind the DSelect algorithm is that it picks a pivot deterministically in a way that produces a good split. We would want to pick this pivot element to be the median so that the split is unbiased. But, this is the same problem we are trying to solve in the first place! So, instead, we will give ourselves leeway by allowing the pivot to be any element that is "roughly" in

the middle: at least 3/10 of the array below the pivot and at least 3/10 of the array above. The algorithm is as follows:

Given an array 'Arr' of size n and integer $k \leq n$;

1. Group the array into n/5 (or 3 or 7) groups of size 5 (or 3 or 7) and find the median of each group.
2. Recursively, find the true median (m) of the medians.
3. Use m as a pivot to split the array into subarrays 'Less' and 'Greater'.
4. Recurse on the appropriate piece.

Below is the pseudo code for the same:
in our case, k is the median

```
partition Arr into subsets S[i] of five elements each
   (there will be n/5 subsets total).

for (i = 1 to n/5) do
   x[i] = select(S[i],3)

M = select({x[i]}, n/10)

partition Arr into L1<M, L2=M, L3>M
if (k <= length(L1))
   return select(L1,k)
else if (k > length(L1)+length(L2))
   return select(L3,k-length(L1)-length(L2))
else return M
}
```

Analysis for group of 5
n = Total Number of Elements in our array Array[n];
We have groups of elements ranging from i equal to 1 to n/5.
Getting median for EACH group (a[i]) takes 6 comparisons at its worst case. So totally, it is **6n/5** comparisons for our Array[n]. Now we have a set of n/5 elements from which we need to Deterministically Select a median, hence we recursively call the same function with input size n/5;

If T(n) is the time to run the algorithm with n elements, getting median would take **T(n/5).**
Let Median, be this median of medians. We use this Median to partition the elements in the array and recursively call the algorithm.
Let L3 be values greater than M. L1 be values lesser than M.
If we leave out L3, among the n/5 values of a[i] we have n/10 elements larger than M.
This implies there are atleast 3 elements in each of n/10 groups array[n]. So Totally there are atleast **3n/10** elements.
This also shows that, L1 also has atleast 3n/10 elements.
Therefore, finally recursive call will yield : 3n/10 + 3n/10 + n/10 = **7n/10** elements which takes at most **T(7n/10).**

Time complexity analysis:
T(n) <= T(n/5) + T(7n/10) + O(n)
<= cn/5 + 7cn/10 + (6n/5) * 2
= n(12/5 + 9c/10)

If this is at most cn,
n(12/5 + 9c/10) <= cn
=> **c <=24**
So the dselect algorithm with partition with group 5 uses at most 24n comparisons and takes O(n) time.

Analysis for group of 7
n = Total Number of Elements in our array Array[n];
We have groups of elements ranging from i equal to 1 to n/7.
Getting median for EACH group (a[i]) takes **10** comparisons at its worst case. So totally, it is **10n/7** comparisons for our Array[n].
(Ref: "Minimum-Comparison Selection" in Volume III of The Art of Computer Programming)
Now we have a set of n/7 elements from which we need to Deterministically Select a median, hence we recursively call the same function with input size n/7; If T(n) is the time to run the algorithm with n elements, getting median of each group would take T(n/7).
Let Median, be this median of median.
Let Median, be this median of medians. We use this Median to partition the elements in the array and recursively call the algorithm.
Let L3 be values greater than M. L1 be values lesser than M.
If we leave out L3, among the n/7 values of a[i] we have **2n/7** elements larger than M.

This implies there are atleast 4 elements in each of n/7 groups array[n]. So Totally there are atleast 2n/7 elements.
This also shows that, L1 also has atleast 2n/7 elements.
Therefore, finally recursive call will yield : 2n/7 + 2n/7 + n/7 = **7n/10** elements which takes at most T(5n/7).

T(n) <= T(n/7) + T(5n/7) + O(n)
<= cn/7 + 5cn/7 + (10n/7) * 2
= n(6c/7 + 20/7)

If this is at most cn,
n(6c/7 + 20/7) <= cn
=> **c <=20**
So the dselect algorithm with partition with group 7 uses at most 20n comparisons and takes O(n) time.

Analysis for group of 3
n = Total Number of Elements in our array Array[n];
We have groups of elements ranging from i equal to 1 to **n/3**.
Getting median for EACH group (a[i]) takes 2 comparisons at its worst case. So totally, it is **2n/3** comparisons for our Array[n].
Now we have a set of n/3 elements from which we need to Deterministically Select a median, hence we recursively call the same function with input size n/3; If T(n) is the time to run the algorithm with n elements, getting median of each group would take **T(n/3).**
Let Median, be this median of median.
We use this Median to partition the elements in the array and recursively call the algorithm.
Let L3 be values greater than M. L1 be values lesser than M.
If we leave out L3, among the n/3 values of a[i] we have n/10 elements larger than M.
This implies there is atleast 1 element in each of n/3 groups array[n]. So Totally there are atleast n/3 elements.
This also shows that, L1 also has atleast n/3 elements. So, if we recurse, we get a sub array with n-2n/3 elements. So this is **T(2n/3)**.

T(n) <= T(n/3) + T(2n/3) + O(n)
<= cn/3 + 2cn/3 + (2n/3) * 2
= n(c + 4/3)

Solving this, for group of 3 we know that it is **not in linear time.**

2. Implementation

1. **Quick Select**

The quick select function takes an array and k (k is the kth smallest element that is to be returned, in our case it is size/2). The first step is to check the size of the array; if the size is 1, return the element itself; otherwise we select a random element. In this case, we are taking the last element to be our pivot element (random median). We then divide the array into 3 groups – L, E, G such that E is our pivot element, all the elements on the left are less than and all the elements on the right are greater than the pivot element. We now check for the "if K is less than index of pivot element, then we consider only L and recursively call the quick select function with array being "L" and k. Similarly, we will follow the same check for sets L+E and G.
Finally, the kth smallest element is returned.

Timer result:
Input set: 49
Time: 0.000007 s

Input set: 1001
Time: 0.002191 s

2. D Select

The D Select function takes an array and k (k is the kth smallest element that is to be returned, in our case it is size/2). The first step is to check the size of the array; If the size of the array is less than or equal to 50, we simply sort and return the median, otherwise we carefully select a median by dividing the array into groups of 5 and selecting medians of each of the groups. This goes on until we get a single median (pivot element). We then divide the array into 3 groups – L, E, G such that E is our pivot element, all the elements on the left are less than and all the elements on the right are greater than the pivot element. We now check for the "if K is less than index of pivot element, then we consider only L and recursively call the quick select function with array being "L" and k. Similarly, we will follow the same check for sets L+E and G.
Finally, the kth smallest element is returned. The same function is to be used for partition size as 3 and 7.

3. Test Cases

Case 1: (Base case)
When the input length is 1
Return the element as median

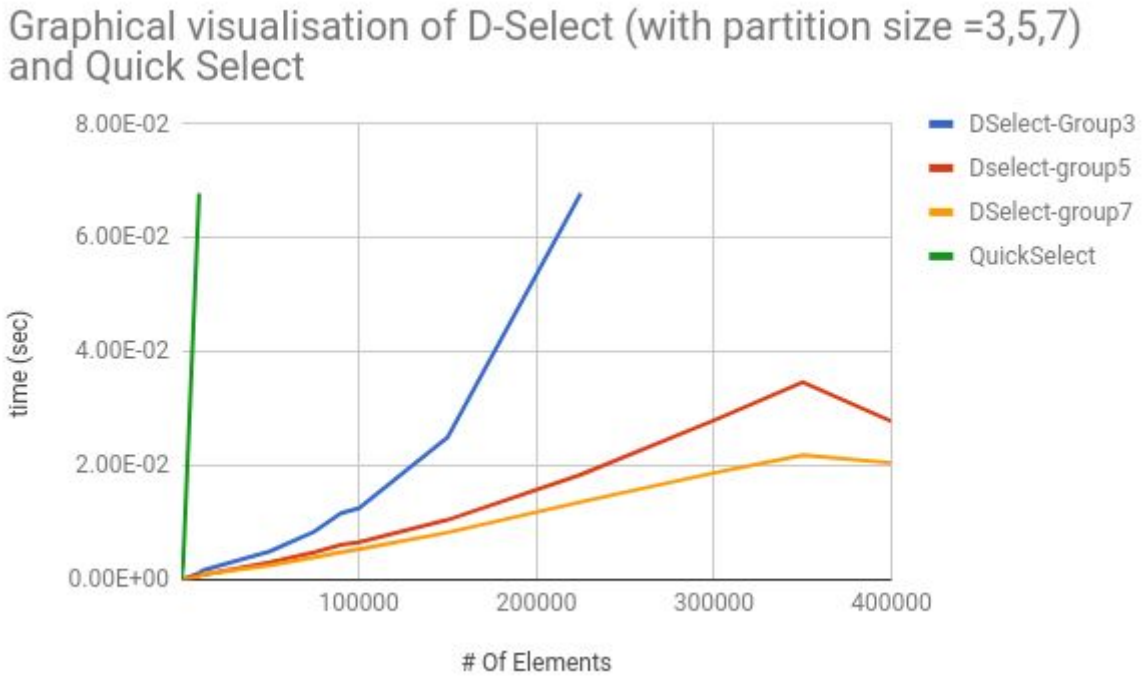
Case2: (Base case)
When the input length is 2
Return the average of 2 elements as median

Case3:
When the input length is 10001, 12345, 50001, 75001, 90001, 100001, 150001, 225001, 300001, 350001, 400001
Return the median using Quick select, D select (partition size =3,5,7)

4. Comparison Analysis

# Of Elements	DSelect-Group3	Dselect-group 5	DSelect-group 7	QuickSelect
13	3.00E-06	6.00E-06	2.00E-06	4.00E-06
25	5.00E-06	5.00E-06	2.00E-06	4.00E-06
49	7.00E-06	6.00E-06	5.00E-06	9.00E-06
101	1.40E-05	1.10E-05	1.00E-05	2.80E-05
1001	1.09E-04	9.60E-05	7.50E-05	0.002946
10001	0.001163	0.000803	0.000538	0.22508
12345	0.001645	0.000836	0.000982	0.321777
50001	0.005	0.003064	0.002508	5.23196
75001	0.008487	0.004833	0.00396	11.9222
90001	0.011673	0.006137	0.004786	17.1491
100001	0.012532	0.006567	0.005319	Large value
150001	0.025028	0.0105	0.008295	Large value
225001	0.067813	0.018459	0.013654	Large value
300001	0.151717	0.028003	0.018744	Large value
350001	0.236691	0.034677	0.021872	Large value
400001	0.246691	0.027805	0.020507	Large value

On comparing the 4 methods namely: Quick Select, D Select for partition size = 3,5,7 on different input sizes; we are getting the graph as below:



As it is evident from the graph, D-Select takes linear time and Quick Select takes n^2 (worst case) time to find the median.

Conclusion:
D-Select with partition size as 7 is the most optimum of the four approaches with a slightly better performance than that of partition size = 5.
D-Select with partition size as 3 takes non-linear time.