1. **Quick Select:**

-Pseudocode

```
quickSelect(S,k):
Input: Sequence S of n comparable elements, and an integer k ∈ [1, n]
Output: The k^th smallest element of S
if n =1
    then return the (first) element of S
pick a random element x of S remove all the elements from S and put them into three sequences:
• L, storing the elements S less than x in a list
• G, storing the elements in S greater than or equal to x in a list
if k <|L|
    then quickSelect (L, k)
else if k =|L|
    then return x // each element in E is equal to x
else
    quickSelect (G, k−|L|)
```

-Theoretical Analysis

Since we select pivot randomly the number of recursions is dependent solely on a good pivot, the probability of selecting a good pivot is half. So, it's equally likely to have a bad pivot in the recursions too. Let's assume we are lucky to get a good pivot in each of the recursion calls, then in every call we try to cut down the size of the list by half.

If we have a good pivot then we have ¾ elements left with us. By master theorem we have, $T(n) <= T(3n/4) + c.n$, where n is the size in S and f(n) is the comparison function we try to do to separate/create the left and right sub lists. And the complexity of $f(n) = O(n)$ for all $n>=n_0$ and a constant $c > 0$. Clearly, it's a case 3 of master's theorem. Hence, the overall complexity is O(f(n)) which is O(n).

If we choose a bad pivot each time, then we have n-1 elements that gets recused every time. This will be sum of n-1 + n-2 + n-3 +…+ 1, whose summation is n^2. Thus, worst-case complexity will be $O(n^2)$.

-Design Choice

Have used an integer type array list as an input to represent set of elements(numbers). K represents the k^{th} element to be found.

Firstly, we take a random pivot in the list and try to bifurcate the list into two smaller lists one for elements smaller than pivot and other list has numbers equal to or greater than pivot, for this reason I have used a dynamic array list (an array would have been inefficient in this type of implementation as we cannot estimate the size of the smaller and greater than arrays to be used as array needs pre initiation of size). I iterate through the list ones completely to do this separation. Then based on the length of the list the recursive action of quick select with the new (less than or greater than list) is called and when the kth element is found I return its value.

2. **DSelect:**

-Pseudocode (explained for groupings in 5)

```
DeterministicSelect(S, 0, S.length(), k):
Input: Sequence S of n comparable elements, and an integer k ∈ [1, n]
Output: The kth smallest element of S
if n =1
    then return the (first) element of S
Divide S into g = (n/5) groups, S1, …, Sg, such that
```

each of groups S1, ..., Sg−1 has 5 elements and group Sg has at most 5 elements.

for i ←1 to g
//to Find the baby median, x_i, in S_i
Use sorting on 5 elements group to find the median xi
and maintain all xi's in the front of the list.

x ←DeterministicSelect ({x1, ..., xg}, 0, g, (g/2)).
//considering only first g elements in the list, trifurcate the elements based on less than, equals to and greater that the median of medians x (given by index m) in the same list by doing it in place thereby reducing extra space complexity overhead. We store the index of the median of median as m.

m ← Trifurcate (list, x); //index of x
if k < m
then DeterministicSelect(S, 0, m, k)
else if k == m
then return x // each element in E is equal to x
else
DeterministicSelect(S, m+1, S.length(), K);

- Theoretical analysis of asymptotic complexity of the variants of DSelect

Generally, the *geometric series* sum(x^i) (for i from 0 to n-1) solves to (1-x^n)/(1-x), and whenever x is less than 1 the limit of the sum as n goes to infinity becomes 1/(1-x). The sum above is just a case of this formula in which x=9/10. The same tree-expansion method then shows that, more generally, if T(n) <= cn + T(an) + T(bn), where a+b<1, the total time is c(1/(1-a-b)) n.

-Groups of three

With groups of only three elements, the resulting list of medians to search in is length *n*/3, and reduces the list of recursions into length, since its greater than ½ * 2/3 = 1/3 of the elements and the less than ½ * 2/3 = 1/3 of the elements. Thus, this still leaves *n* elements to search in, not reducing the problem sufficiently. The individual lists are shorter, however, and one can bound the result of the complexity to O (*n log n).*

i.e $T(n) \leq T(n.\ 1/3) + T(n.\ 2/3) + b.n$, where $b>0$ and $T(n) \leq c.n$ such that $c>0$ is a constant. Sum of 1/3 + 2/3 = 1 which implies the recursion term is of the order O(n) likewise to f(n) = O(n) which suffices case 2 of master theorem leading to an overall complexity of O (*n log n*).

-Groups of five

The median-calculating recursive call does not exceed worst-case linear behavior because the list of medians is 20% of the size of the list, while the other recursive call recurses on at most 70% of the list. Let *T(n)* be the time it takes to run a median-of-medians Quickselect algorithm on an array of size *n*. Then we know this time is: where $T(n) \leq T(n.\ 1/5) + T(n.\ 7/10) + b.n$,

- The *T(n·2/10)* part is for finding the *true* median of the *n/5* medians, by running an (independent) Quickselect on them (since finding the median is just a special case of selecting a *k*-largest element)
- The O(*n*) term *c·n* is for the partitioning work to create the two sides, one of which our Quickselect will recurse (we visited each element a constant number of times, in order to form them into n/5 groups and take each median in O (1) time).
- The *T(n·7/10)* part is for the actual Quickselect recursion (for the worst case, in which the *k*-th element is in the bigger partition that can be of size n·7/10 maximally)

From this, using induction one can easily show that $T(n) \leq 10.\ c.\ n \in O(n)$.

-Groups of seven

With the groups of seven result in search of medians in the length of 7 and reduces the list into recursions into length n*5/7. Therefore, the time complexity is: $T(n) \leq T(n.\ 1/7) + T(n.\ 5/7) + b.n$, the summation of the elements in the recursion is a converging sequence that is less than n .Using general rule of geometric series ($1/7 + 5/7 < 1$)we can show the complexity to be $\theta(n)$.

-Groups of nine

With the groups of seven result in search of medians in the length of 9 and reduces the list into recursions into length n*13/18. Therefore, the time complexity is: $T(n) \leq T(n.\ 1/9) + T(n.\ 13/18) + b.n$, the summation of the elements in the recursion is a converging sequence that is less than n. Using general rule of geometric series ($1/9 + 13/18 < 1$) we can show the complexity to be $\theta(n)$.

-Best asymptotic characterization: (Perform a theoretical analysis to determine the best asymptotic characterization of each of these algorithms as a function of the input size, n.

-Design Choice

I have designed the in-place algorithm for Dselect wherein I try to move the medians from all the groups in front of the list that I have used for input. Next recursively select median of medians from the first g medians represented by g groups. The input structure used is a Array list since java has inbuilt functions for swap , sort and shuffle in Collections library that applies on list.

- Experimental results

- Measuring elapsed time:

To calculate the time elapsed I have used the timer library and functions given by timer.java as a support. I have captured the user start time just before calling the select function that returns the median (n/2 th largest element) and have captured the user end time as soon as we hit a return from this select function. i.e. double startUserTimeNano = getUserTime( ); and double taskUserTimeNano = getUserTime( ) – startUserTimeNano; and (taskUserTimeNano/1000000000) gives us the number of seconds.

- Comparative results of 5 algorithms (your visual plots. Also analyze the difference in their performances and see if it matches your theoretical analysis)

Table 1. Shows the average values observed experimentally for each type of input run 20 times.

I/P size	DS 9 (μs)	DS 7 (μs)	DS 5 (μs)	DS 3 (μs)	Quick (μs)
100	0	0	0	0	0
1000	0.1764705882	0.1176470588	0.5	0.5	0.01
10000	2.941176471	2.941176471	3.5	5	10
100000	14.11764706	15.29411765	15.5	26	40
1000000	147.6470588	168.2352941	204.5	347.5	300

The corresponding graph plot is as shown in the fig1.

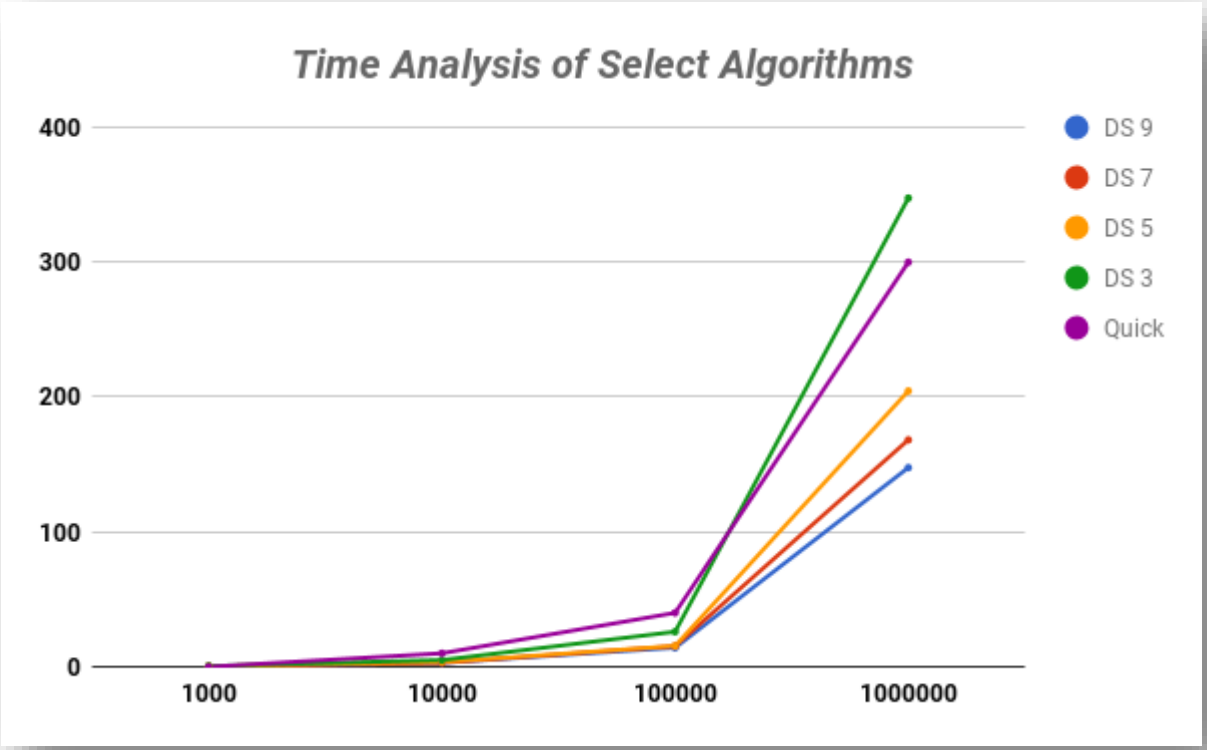

Fig1. Plot for all various Select Algorithms for different input size.

It was observer while testing for average time for Quickselect that the variance in the time the code took was a lot implying that the pivots choosen were good in some cases and were worst in others while others had a mix. The one when good pivots were selected gave faster time than the ones with the worst times

We see from analysis that group by 3 in Dselect and QuickSelect that uses random pivot selection works worse that other Dselect algorithms. We see theoretically that time complexity for group by 9 and 7 of Dselect has $\theta(n)$, while Dselect group by 5 has O(n) complexity, which works well when compared to Dselect group by 3 which has O(n log n) and quick select has $O(n^2)$ as the worst case time.