Selection Problem
Selection, or, finding the kth Largest Element.

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Selection problem: Introduction

- **What is the most efficient algorithm to find the k^{th} smallest element in an array having n unordered elements?**
- Suppose you have a group of N numbers and would like to determine the k^{th} smallest. This is known as the *selection problem*.
- One way to solve this problem would be to read the N numbers into an array, sort the array in decreasing order by some simple algorithm such as bubble sort, and then return the element in position *k*.
- Assuming a simple sorting algorithm, the running time is $O(N^2)$.

Introduction

- Suppose you have a group of N numbers and would like to determine the k^{th} largest. This is known as the *selection problem*.
- One way to solve this problem would be to read the N numbers into an array, sort the array in decreasing order by some simple algorithm such as bubble sort, and then return the element in position *k*.
- Assuming a simple sorting algorithm, the running time is $0(N^2)$.

Introduction

- Given an unsorted array, how quickly can one find the **median element**? Can one do it more quickly than by sorting?
- This was an open question for some time, solved affirmatively in 1972 by (Manuel) Blum, Floyd, Pratt, Rivest, and Tarjan. In this lecture we describe two linear-time algorithms for this problem: one randomized and one deterministic.
- More generally, we solve the problem of finding the **k^{th} smallest** out of an unsorted array of n elements.

Order statistics

- A problem closely related to, but simpler than sorting is that of the selection (also referred to as the *order statistics*) *problem*.
- The **Selection problem**: Given a sequence $A = (a_1 , a_2 , . . . , a_n)$ *of n elements on* which a linear ordering is defined, and an integer *k*, $1 \leq k \leq n$, *find the* k^{th} *smallest* element in the sequence.
- The **selection problem** is the problem of computing, given a set A of n distinct numbers and a number k, $1 \leq k \leq n$, the **kth order statistics** (i.e., the kth smallest number) of A.

Selection

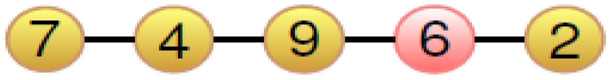
- Selection is a trivial problem if the input numbers are sorted. If we use a sorting algorithm having **O(n log n)** worst-case running time, then the selection problem can be solved in in **O(n log n)** time.
- But using a sorting is more like using a cannon to shoot a fly since only one number needs to computed.

The Selection Problem

The selection problem:

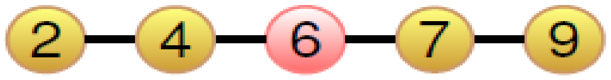
- given an integer k and a list $x_1, \ldots, x_n$ of n elements
- find the k-th smallest element in the list

Example: the 3rd smallest element of the following list is 6



An $O(n \cdot \log n)$ solution:

- sort the list ($O(n \cdot \log n)$)
- pick the k-th element of the sorted list ($O(1)$)



Can we find the k-th smallest element faster?

Solution 1

- Problem: Given a sequence A[1 : : : n] and an integer k, $1 \leq k \leq n$ the kth smallest integer in A.

- Sort the elements in an array and return the element in the kth position

Complexity:

- simple sorting algorithm $\mathbf{O(n^2)}$
- advanced sorting algorithm **O(n log n)**

Solution 2

- Read *k elements in an array and sort them , Let Sk be the smallest element*
- For each next element E do the following:
- If E > Sk remove Sk and insert E in the appropriate position in the array
- Return Sk
- Complexity: O(N*k)
- k2 for the initial sorting of k elements
- (N-k)*k for inserting each next element
- O(k2)+ O((N-k)*k) = O(k2 + N*k - k2) = O(N*k)
- Worst case: k = N/2, complexity: O(N2)

Solution 3

- Assume we change the heap-order property - the highest priority corresponds to the highest key value.
- Read N elements in an array
- Build a heap of these N elements - O(N)
- Perform k *DeleteMax operations - O(klogN)*
- Complexity: O(N) + O(klogN)
- For k = N/2 the complexity is O(NlogN)

Solution 4

- We return back to the usual heap-order property – the smallest element is at the top.
- Build a **Heap** of **k elements. The k-th largest element among k elements is the smallest element in that heap and it will be at the top. Complexity O(k)**
- Compare each next element with the top element O(1)
- If the new element is larger, *DeleteMin the top element and insert the new element in the heap. - O(log(k))*
- At the end of the input the smallest element in the heap is the **k-th largest element in the list of N elements.**
- Complexity: O(k) + O((N-k)*log(k)) = O(Nlog(k))

Solution 5

- **(Use Bubble k times)**
  1) Modify Bubble Sort to run the outer loop at most k times.
  2) Print the last k elements of the array obtained in step 1.
- Time Complexity: O(nk)
- Like Bubble sort, other sorting algorithms like Selection Sort can also be modified to get the k largest elements.

Solution 6

(Use temporary array)

- K largest elements from arr[0..n-1]
- 1) Store the first k elements in a temporary array temp[0..k-1].
  2) Find the smallest element in temp[], let the smallest element be *min*.
  3) For each element *x* in arr[k] to arr[n-1]
  If *x* is greater than the min then remove *min* from temp[] and insert *x*.
  4) Print final k elements of *temp[]*
- Time Complexity: O((n-k)*k). If we want the output sorted then O((n-k)*k + klogk)
- Thanks to nesamani1822 for suggesting this method.

Solution 6

(Use Oder Statistics)

- 1) Use order statistic algorithm to find the kth largest element. [selection in worst-case linear time O(n)]
- 2) Use **Quicksort_Partition** algorithm to partition around the kth largest number O(n).
- 3) Sort the k-1 elements (elements greater than the kth largest element) O(kLogk). This step is needed only if sorted output is required.
- Time complexity: O(n) if we don’t need the sorted output, otherwise O(n + kLogk)

Quick select

Prune-and-Search

- **The idea of this algorithmic paradigm is to perform just enough computation** to detect at least a constant fraction of the data as irrelevant and to remove it.
- After deleting the constant fraction from the data, the algorithm simply recurs.
- The total amount of time spent by the algorithm turns out to be asymptotically the same as the amount of time for one iteration.
- This is because the amount of data involved in the sequence of iterations decreases geometrically.
- It is clear from this description that only problems that produce little output, as opposed to creating an elaborate output structure, qualify as candidates for the application of the prune-and-search paradigm.

Solution of selection problem : Quick select

- How **quick sort** is used to find the solution to selection problem[Find the K^th **Largest**(smallest)]
- If the pivot selected is places at K^th position in the array, then that results the **best case** of the selection problem.
- If not it is possible to discard one of the partition S1 or S2, that does not include the K^th largest element, and apply **QuickSort** to the selected partition

Quick Sort

```
Algorithm QuickSort(p, q)
/ / Sorts the elements a[p], ... ,a[q] which reside in the global
/ / array a[1 : n] into ascending order; a[n + 1] is considered to
/ / be defined and must be ≥ all the elements in a[1 : n].
{
if (p < q) then / / If there are more than one element
{
/ / divide P into two subproblems.
j := Partition(a, p, q + 1);
/ / j is the position of the partitioning element.
/ / Solve the subproblems.
          QuickSort(p, j - 1);
          QuickSort(j + 1, q);
/ /There is no need for combining solutions.
}
}
```

Solution of selection problem : Quick select

Given an array of n elements, determine the kth smallest element. (Clearly k must lie in between 1 and n, inclusive.)

Quickselect :

- Partition the array. Let's say the partition splits the array into two sub-arrays, one of size m with the m smallest elements and the other of size **n - m - 1**.
- If $k \leq m$, then we know that the kth smallest element of the original array was in the first partition.
- If $k=m+1$, then we know our first partition element is the correct value,
- otherwise, we know to search for the kth smallest value in the second partition of the original array.

Quick select

QuickSelect: Given array A of size n and integer $k \leq n$,

1. Pick a pivot element p at random from A.
2. Split A into subarrays LESS and GREATER by comparing each element to p as in Quicksort. While we are at it, count the number L of elements going in to LESS.
3. (a) If $L = k - 1$, then output p.

   (b) If $L > k - 1$, output QuickSelect(LESS, k).

   (c) If $L < k - 1$, output QuickSelect(GREATER, $k - L - 1$)

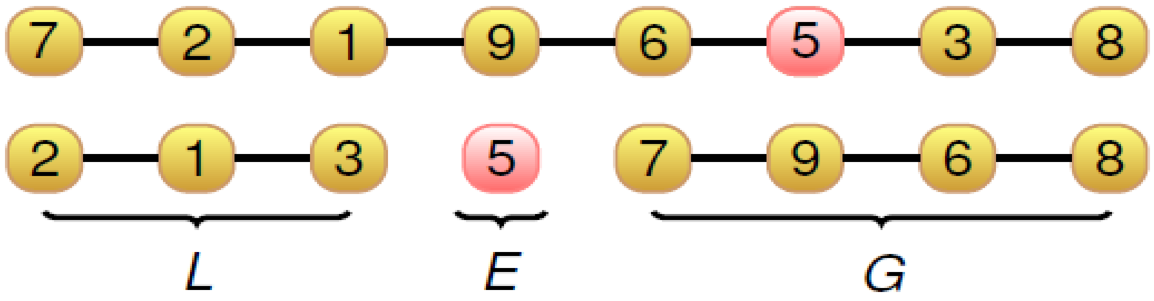
Randomized Quick select

```
function $QuickSelect(S, k)$;   (*finds the $k$-th smallest element of $S$*)
begin
    $m \leftarrow$ a random element of $S$;        (*the pivot element*)
    $S_1 \leftarrow \{ a \in S \mid a < m \}$;
    $S_2 \leftarrow \{ a \in S \mid a > m \}$;
    if $|S_1| \geq k$ then return $QuickSelect(S_1, k)$
      else if $|S| - |S_2| \geq k$ then return $m$
        else return $QuickSelect(S_2, k - |S| + |S_2|)$
end
```

Quick-Select

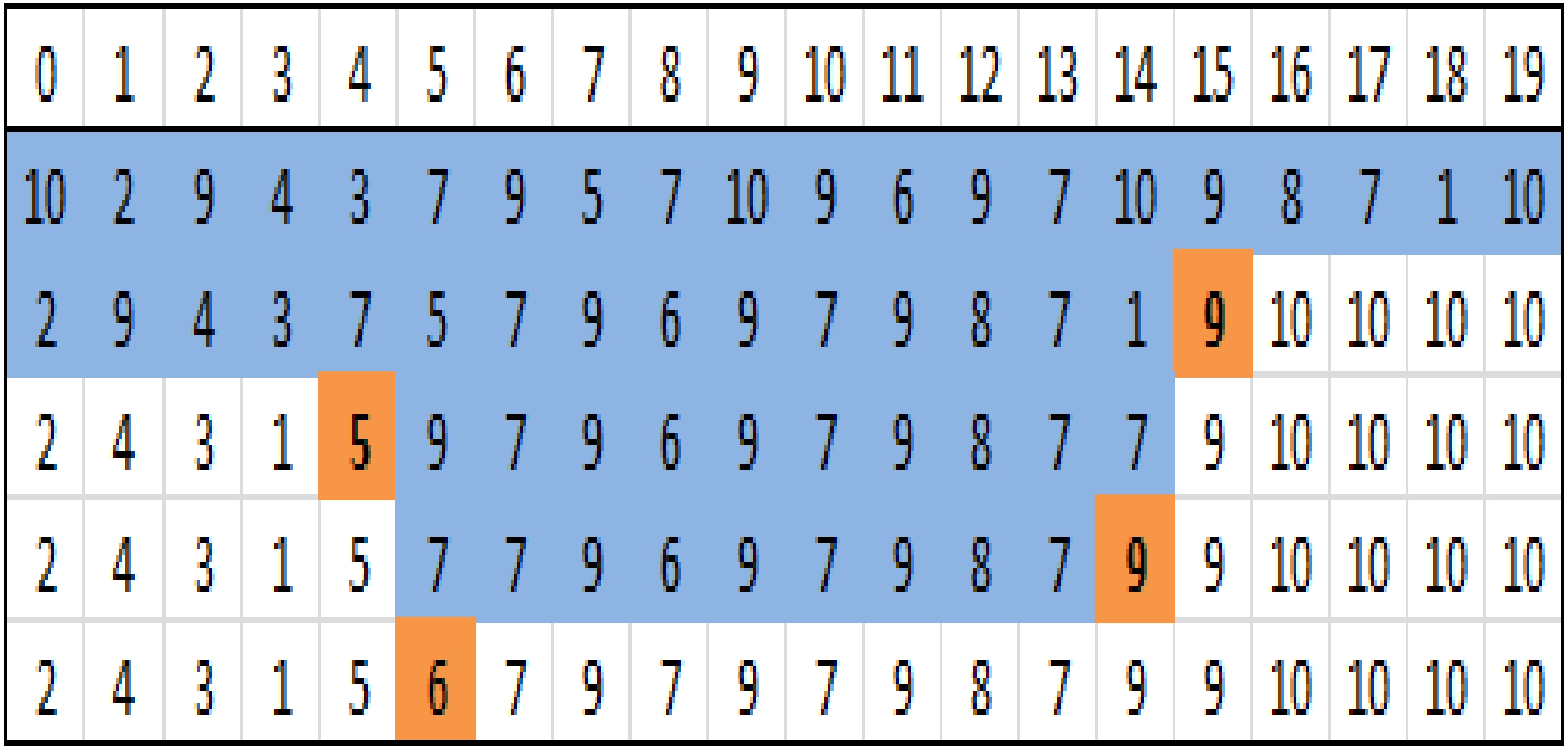
Quick-select of the k-th smallest element in the list *S*:

- based on **prune-and-search** paradigm
- Prune: pick random element x (pivot) from *S* and split *S* in:
  - *L* elements $< x$, *E* elements $== x$, *G* elements $> x$



  - partitioning into *L*, *E* and *G* works precisely as for quick-sort
- Search:
  - if $k \leq |L|$ then return quickSelect(k, L)
  - if $|L| < k \leq |L| + |E|$ then return x
  - if $k > |L| + |E|$ then return quickSelect$(k - |L| - |E|, G)$

Finding 6th largest element

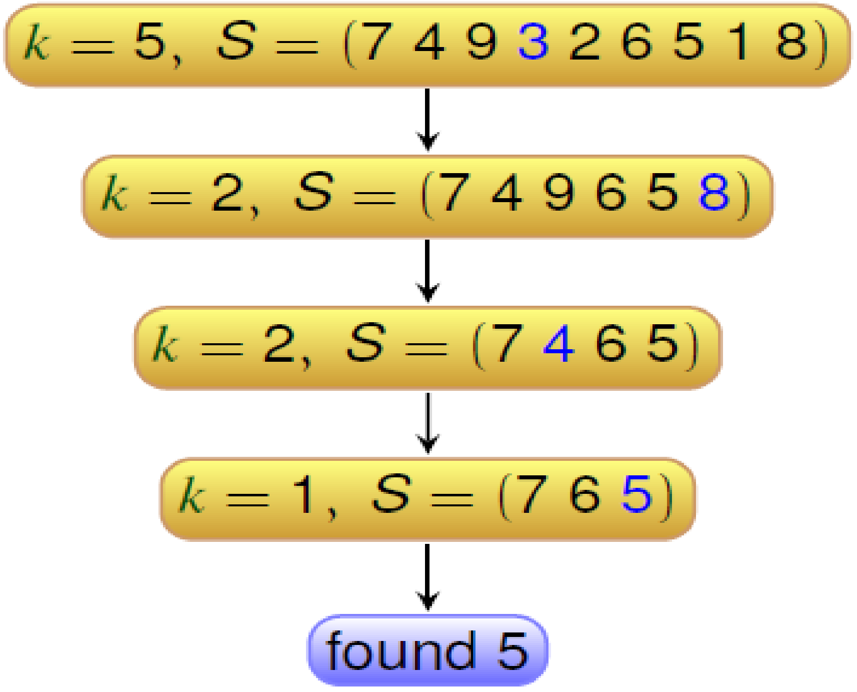


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
10	2	9	4	3	7	9	5	7	10	9	6	9	7	10	9	8	7	1	10
2	9	4	3	7	5	7	9	6	9	7	9	8	7	1	9	10	10	10	10
2	4	3	1	5	9	7	9	6	9	7	9	8	7	7	9	10	10	10	10
2	4	3	1	5	7	7	9	6	9	7	9	8	7	9	9	10	10	10	10
2	4	3	1	5	6	7	9	7	9	7	9	8	7	9	9	10	10	10	10

Quick-Select Visualization

Quick-select can be displayed by a sequence of nodes:

- each node represents recursive call and stores: k, the sequence, and the pivot element


$k = 5,\ S = (7\ 4\ 9\ 3\ 2\ 6\ 5\ 1\ 8)$
$k = 2,\ S = (7\ 4\ 9\ 6\ 5\ 8)$
$k = 2,\ S = (7\ 4\ 6\ 5)$
$k = 1,\ S = (7\ 6\ 5)$
found 5

Average case Analysis Quick select :O(N)

$T(1) = 1$ for $N = 1$

$T(N) = T(i)$ **OR** $T(N-i-1) + cN$, for $N > 1$ (15)

- The average value of $T(i) = {}^{1}/_{N} \sum_{j=0}^{N-1} T(j)$

T(0)		T(N-1)
T(1)	OR	T(N-2)
T(2)		T(N-3)
.		.
.		.
.		.
T(N-1)		T(0)

Total sub problems $= \sum_{j=0}^{N-1} T(j)$

Average case Analysis Quick select :O(N)

Using **Total sub problems** = $\sum_{j=0}^{N-1} T(j)$ in equation 15 $\Rightarrow$

$T(1) = 1$ **for N =1**

$T(N) = \frac{1}{N}\sum_{j=0}^{N-1} T(j) + cN,$ **for** $N > 1$ **(16)**

Multiplying the equation (16) with N

$\Rightarrow NT(N) = \sum_{j=0}^{N-1} T(j) + cN^2$ **(17)**

Inorder to remove the summation sign from equation (17), we can telescope with one more equation:

$\Rightarrow (N\text{-}1)T(N\text{-}1) = \sum_{j=0}^{N-2} T(j) + c(N\text{-}1)^2$ **(18)**

- Subtracting (18) from (17)

$\Rightarrow NT(N) - [(N\text{-}1)T(N\text{-}1)] = [\sum_{j=0}^{N-1} T(j) + cN^2] - [\sum_{j=0}^{N-2} T(j) + c(N\text{-}1)^2]$

$\Rightarrow NT(N) - (N\text{-}1)T(N\text{-}1) = T(N-1) + cN^2 - cN^2 - c + 2cN$

Average case Analysis Quick select :O(N)

- $NT(N) - (N\text{-}1)T(N\text{-}1) = T(N-1) + cN^2 - cN^2 - c + 2cN$

$\Rightarrow NT(N) - (N\text{-}1)T(N\text{-}1) = T(N-1) + 2cN - c$

By dropping the insignificant term –c

$\Rightarrow NT(N) - (N\text{-}1)T(N\text{-}1) = T(N-1) + 2cN$

$\Rightarrow NT(N) = T(N-1) + (N-1)T(N-1) + 2cN$

$\Rightarrow NT(N) = T(N-1)[1 + N - 1] + 2cN$

$\Rightarrow NT(N) = N\,T(N-1) + 2cN$

$\Rightarrow T(N) = T(N-1) + 2c$ (19)

$\Rightarrow T(N\text{-}1) = T(N-2) + 2c$

$\Rightarrow T(N\text{-}2) = T(N-3) + 2c$

.

$\Rightarrow$.

.

$\Rightarrow T(2) = T(1) + 2c$ (20)

Adding all N equations starting from equation () to ()

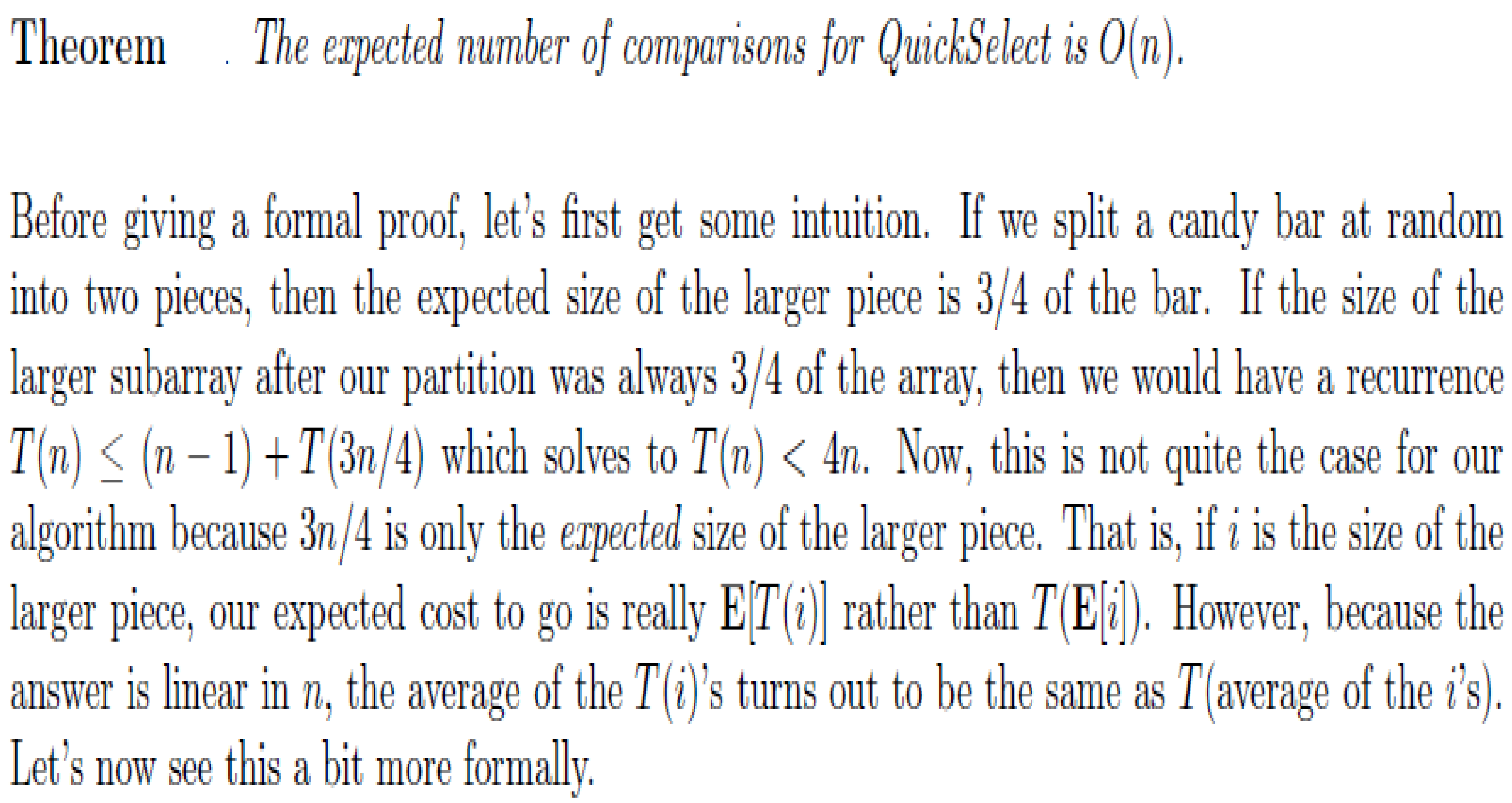
$\Rightarrow T(N) = T(1) + 2cN \Rightarrow T(N) = O(N)$ □

Quick-Select

- **Theorem 1:** The expected number of comparisons for QuickSelect is O(n).

$$\bar{T}(n) = cn + \frac{1}{n}\sum_{i=0}^{n-1} \bar{T}(i)\,.$$

Theorem: The expected number of comparisons for QuickSelect is O(n).



Theorem . *The expected number of comparisons for QuickSelect is $O(n)$.*

Before giving a formal proof, let's first get some intuition. If we split a candy bar at random into two pieces, then the expected size of the larger piece is 3/4 of the bar. If the size of the larger subarray after our partition was always 3/4 of the array, then we would have a recurrence $T(n) \leq (n-1) + T(3n/4)$ which solves to $T(n) < 4n$. Now, this is not quite the case for our algorithm because $3n/4$ is only the *expected* size of the larger piece. That is, if i is the size of the larger piece, our expected cost to go is really $\mathrm{E}[T(i)]$ rather than $T(\mathrm{E}[i])$. However, because the answer is linear in n, the average of the $T(i)$'s turns out to be the same as T(average of the i's). Let's now see this a bit more formally.

Proof (Theorem): Let $T(n,k)$ denote the expected time to find the kth smallest in an array of size n, and let $T(n) = \max_k T(n,k)$. We will show that $T(n) < 4n$.

First of all, it takes $n-1$ comparisons to split into the array into two pieces in Step 2. These pieces are equally likely to have size 0 and $n-1$, or 1 and $n-2$, or 2 and $n-3$, and so on up to $n-1$ and 0. The piece we recurse on will depend on k, but since we are only giving an upper bound, we can imagine that we always recurse on the larger piece. Therefore we have:

$$\begin{aligned} T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=n/2}^{n-1} T(i) \\ &= (n-1) + \operatorname{avg}\left[T(n/2), \ldots, T(n-1)\right]. \end{aligned}$$

We can solve this using the "guess and check" method based on our intuition above. Assume inductively that $T(i) \leq 4i$ for $i < n$. Then,

$$\begin{aligned} T(n) &\leq (n-1) + \operatorname{avg}\left[4(n/2), 4(n/2+1), \ldots, 4(n-1)\right] \\ &\leq (n-1) + 4(3n/4) \\ &< 4n, \end{aligned}$$

and we have verified our guess. ∎

Theorem

The *k*th smallest element in a set of n elements drawn from a linearly ordered set can be found in $\Theta(n)$ time. In particular, the median of n elements can be found in $\Theta(n)$ time.

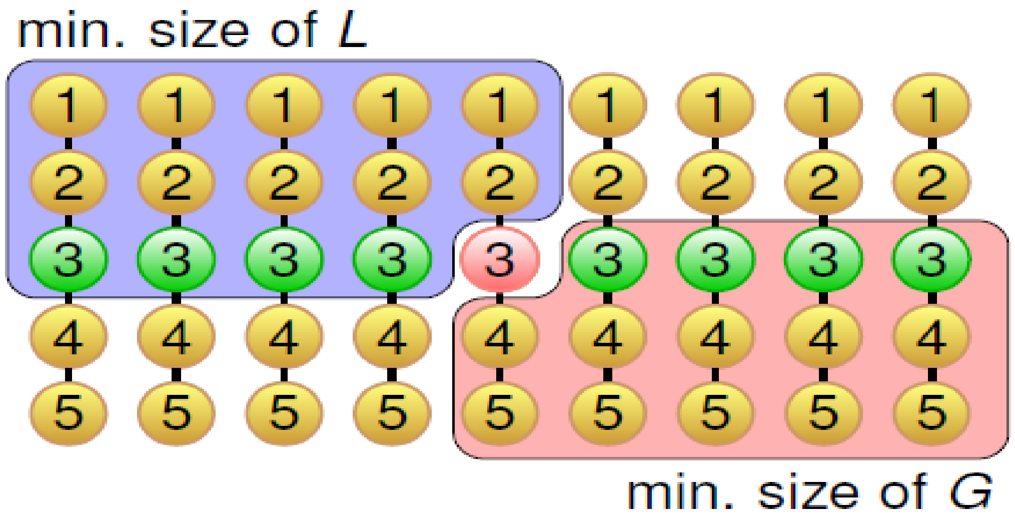
Quick-Select: Median of Medians

Quick-Select: Median of Medians

We can do selection in $O(n)$ worst-case time.

Idea: recursively use select itself to find a good pivot:

- divide S into $n/5$ sets of 5 elements
- find a median in each set (baby median)
- recursively use select to find the median of the medians



The minimal size of L and G is $0.3 \cdot n$.

Quick-Select: Median of Medians

We know:

- The minimal size of L and G is $0.3 \cdot n$.
- Thus the maximal size of L and G is $0.7 \cdot n$.

Let $b \in \mathbb{N}$ such that:

- partitioning of a list of size n takes at most $b \cdot n$ time,
- finding the baby medians takes at most $b \cdot n$ time,
- the base case $n \leq 1$ takes at most b time.

We derive a recurrence equation for the time complexity:

$$T(n) = \begin{cases} b & \text{if } n \leq 1 \\ T(0.7 \cdot n) + T(0.2 \cdot n) + 2 \cdot b \cdot n & \text{if } n > 1 \end{cases}$$

Three linear-time algorithms for the selection problem

Quickselect

First selection algorithm is a simple adaptation of Quicksort, which we'll call **Quickselect**.

Given the n elements, we choose a pivot uniformly at random, split the elements into those greater than and those less than the pivot, and recurse on *one of* the two sets. Which set of course depends on k and on the size of the sets – if

there are t elements less than the pivot, we recurse on

the smaller-than set if k t, simply return the pivot if

k = t+1, and recurse on the greater-than set if k > t+1.

```
Function FIND-KLARGEST(A, $k$, $n$)
 1: We assume that the array elements are stored in A[1] through A[$n$] and that $k$ is an integer $\in [1, n]$. We also assume
    without loss of generality, we assume that the numbers are distinct.
 2: if ($n = 1$) then
 3:    {$k$ has to be 1 as well}
 4:    return(A[$n$])
 5: end if
 6: We consider a variation of the PARTITION() procedure in which elements *larger* than the pivot are thrown in the left
    subarray and elements *smaller* than the pivot are thrown in the right subarray.
 7: Partition A using A[1] as the pivot, using the above PARTITION() procedure. Let $j$ denote the index returned by
    PARTITION(). {As per the mechanics of PARTITION(), elements A[1] through A[$j - 1$] are greater than A[$j$] and
    elements A[$j + 1$] through A[$n$] are smaller than A[$j$].}
 8: Copy the elements larger than A[$j$] into a new array C and the elements smaller than A[$j$] into a new array D.
 9: if (($k = j$) then
10:    return(A[$j$])
11: else
12:    if ($k < j$) then
13:       return( FIND-KLARGEST(C, $k$, ($j - 1$)))
14:    else
15:       return( FIND-KLARGEST(D, $k - j$, ($n - j$)))
16:    end if
17: end if
```

LazySelect by Motwani and Raghavan

- **LazySelect** finds the k'th smallest element from a totally ordered set S, of size n, as follows:
- Choose $\mathbf{n}^{3/4}$ elements uniformly and independently, with replacement, from S.
- Sort this multiset R completely.
- Let $\mathbf{t = kn^{-1/4}}$. Let a be the **t - √n'th** smallest element of R, and let b be the **t + √ n'th** smallest.
- Compare each element against a and/or b to find the elements that fall between them, and how many fall before a and after b,
- If the k'th smallest element is guaranteed to fall between a and b, and there are no more than $4n^{3/4}$ elements between them, sort those elements and find the target.
- If the conditions above are not true, repeat the entire algorithm with independent choices.

Algorithm Select1

```
Algorithm Select1($a, n, k$)
// Selects the $k$th-smallest element in $a[1:n]$ and places it
// in the $k$th position of $a[\,]$. The remaining elements are
// rearranged such that $a[m] \leq a[k]$ for $1 \leq m < k$, and
// $a[m] \geq a[k]$ for $k < m \leq n$.
{
    $low := 1$; $up := n + 1$;
    $a[n+1] := \infty$; // $a[n+1]$ is set to infinity.
    repeat
    {
        // Each time the loop is entered,
        // $1 \leq low \leq k \leq up \leq n + 1$.
        $j :=$ Partition($a, low, up$);
            // $j$ is such that $a[j]$ is the $j$th-smallest value in $a[\,]$.
        if ($k = j$) then return;
        else if ($k < j$) then $up := j$; // $j$ is the new upper limit.
            else $low := j + 1$; // $j + 1$ is the new lower limit.
    } until (false);
}
```

Algorithm Select2

```
Algorithm Select2($a, k, low, up$)
// Find the $k$-th smallest in $a[low : up]$.
{
        $n := up - low + 1$;
        if ($n \leq r$) then sort $a[low : up]$ and return the $k$-th element;
        Divide $a[low : up]$ into $n/r$ subsets of size $r$ each;
        Ignore excess elements;
        Let $m[i]$, $1 \leq i \leq (n/r)$ be the set of medians of
        the above $n/r$ subsets.
        $v :=$ Select2($m, \lceil (n/r)/2 \rceil, 1, n/r$);
        Partition $a[low : up]$ using $v$ as the partition element;
        Assume that $v$ is at position $j$;
        if ($k = (j - low + 1)$) then return $v$;
        elseif ($k < (j - low + 1)$) then
                        return Select2($a, k, low, j - 1$);
                else return Select2($a, k - (j - low + 1), j + 1, up$);
}
```

Algorithm Partition

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p - 1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

        if ($i < j$) then Interchange($a, i, j$);

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}
```

Algorithm Interchange

```
**Algorithm** Interchange$(a, i, j)$
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i];$
    $a[i] := a[j]; a[j] := p;$
}
```

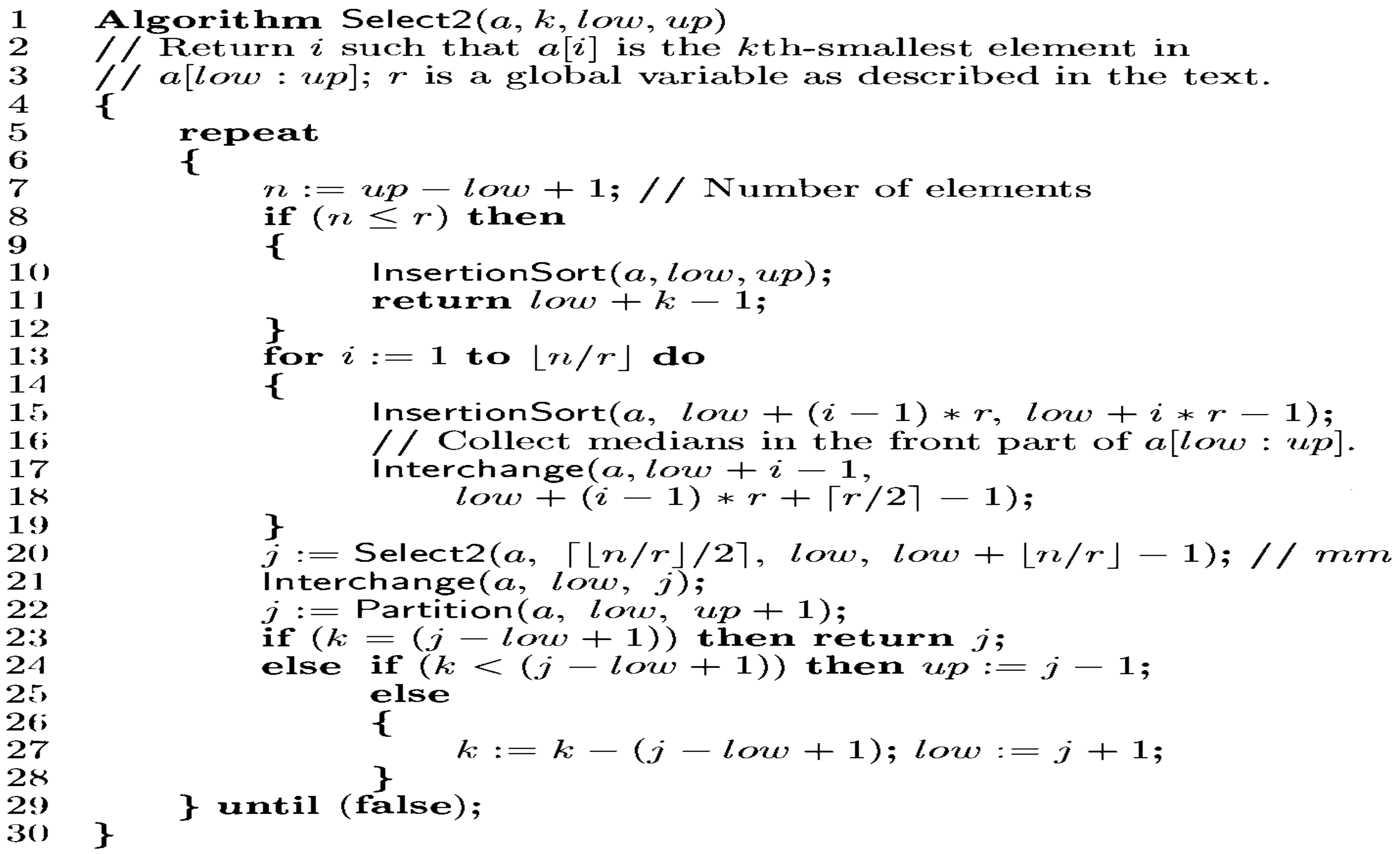
Algorithm InsertionSort

```
Algorithm InsertionSort($a, n$)
// Sort the array $a[1 : n]$ into nondecreasing order, $n \geq 1$.
{
    for $j := 2$ to $n$ do
    {
        // $a[1 : j - 1]$ is already sorted.
        $item := a[j]$; $i := j - 1$;
        while (($i \geq 1$) and ($item < a[i]$)) do
        {
            $a[i + 1] := a[i]$; $i := i - 1$;
        }
        $a[i + 1] := item$;
    }
}
```

Pseudocode Selection

```
Algorithm Select2($a, k, low, up$)
// Return $i$ such that $a[i]$ is the $k$th-smallest element in
// $a[low : up]$; $r$ is a global variable as described in the text.
{
    repeat
    {
        $n := up - low + 1$; // Number of elements
        if ($n \leq r$) then
        {
            InsertionSort($a, low, up$);
            return $low + k - 1$;
        }
        for $i := 1$ to $\lfloor n/r \rfloor$ do
        {
            InsertionSort($a$, $low + (i-1) * r$, $low + i * r - 1$);
            // Collect medians in the front part of $a[low : up]$.
            Interchange($a, low + i - 1$,
                $low + (i-1) * r + \lceil r/2 \rceil - 1$);
        }
        $j :=$ Select2($a$, $\lceil \lfloor n/r \rfloor /2 \rceil$, $low$, $low + \lfloor n/r \rfloor - 1$); // $mm$
        Interchange($a$, $low$, $j$);
        $j :=$ Partition($a$, $low$, $up + 1$);
        if ($k = (j - low + 1)$) then return $j$;
        else if ($k < (j - low + 1)$) then $up := j - 1$;
            else
            {
                $k := k - (j - low + 1)$; $low := j + 1$;
            }
    } until (false);
}
```

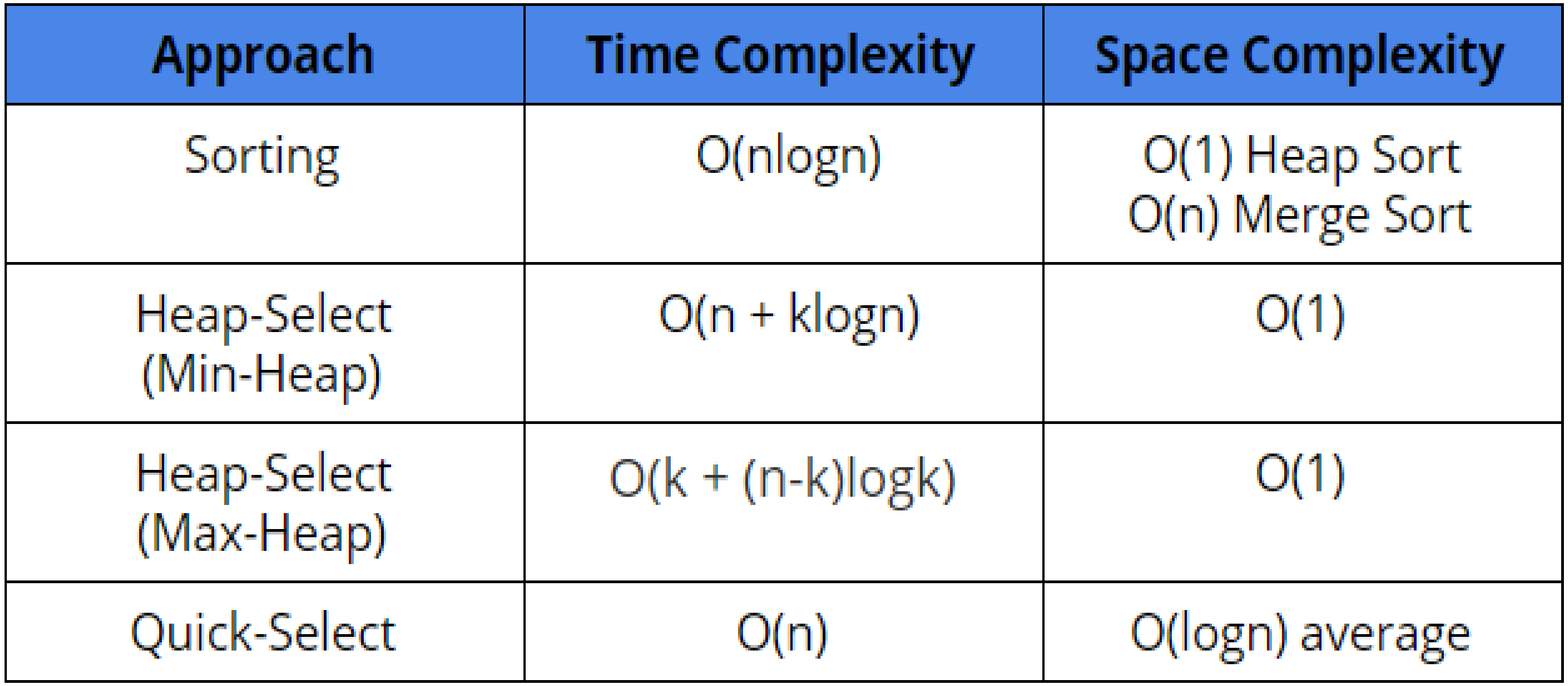
Algorithm Select2



```
Algorithm Select2($a, k, low, up$)
// Return $i$ such that $a[i]$ is the $k$th-smallest element in
// $a[low : up]$; $r$ is a global variable as described in the text.
{
    repeat
    {
        $n := up - low + 1$; // Number of elements
        if ($n \leq r$) then
        {
            InsertionSort($a, low, up$);
            return $low + k - 1$;
        }
        for $i := 1$ to $\lfloor n/r \rfloor$ do
        {
            InsertionSort($a$, $low + (i - 1) * r$, $low + i * r - 1$);
            // Collect medians in the front part of $a[low : up]$.
            Interchange($a, low + i - 1$,
                $low + (i - 1) * r + \lceil r/2 \rceil - 1$);
        }
        $j :=$ Select2($a$, $\lceil \lfloor n/r \rfloor /2 \rceil$, $low$, $low + \lfloor n/r \rfloor - 1$); // $mm$
        Interchange($a$, $low$, $j$);
        $j :=$ Partition($a$, $low$, $up + 1$);
        if ($k = (j - low + 1)$) then return $j$;
        else if ($k < (j - low + 1)$) then $up := j - 1$;
            else
            {
                $k := k - (j - low + 1)$; $low := j + 1$;
            }
    } until (false);
}
```

Straightforward selection algorithm

```
Algorithm Select3($a, n, k$)
// Rearrange $a[\,]$ such that $a[k]$ is the $k$-th smallest.
{
    if ($k \leq \lfloor n/2 \rfloor$) then
        for $i := 1$ to $k$ do
        {
            $q := i$; $min := a[i]$;
            for $j := i + 1$ to $n$ do
                if ($a[j] < min$) then
                {
                    $q := j$; $min := a[j]$;
                }
            Interchange($a$, $q$, $i$);
        }
    else
        for $i := n$ to $k$ step $-1$ do
        {
            $q := i$; $max := a[i]$;
            for $j := (i - 1)$ to $1$ step $-1$ do
                if ($a[j] > max$) then
                {
                    $q := j$; $max := a[j]$;
                }
            Interchange($a$, $q$, $i$);
        }
}
```



Approach	Time Complexity	Space Complexity
Sorting	O(nlogn)	O(1) Heap Sort O(n) Merge Sort
Heap-Select (Min-Heap)	O(n + klogn)	O(1)
Heap-Select (Max-Heap)	O(k + (n-k)logk)	O(1)
Quick-Select	O(n)	O(logn) average

Thanks for Your Attention!

Related Questions

1. Given an array, what is the most efficient way to find whether a given sum can be formed by the elements of the array?
2. How do I find the smallest and second smallest element in an array?
3. What is the way to find a missing element in an array in the most optimized way?
4. How do I get recurrence equation about finding the k smallest element in a array in best way with O(n) time complexity?
5. What's an algorithm for finding the median element in an unsorted array in linear time, constant space?
6. How do I find the top n elements in a list efficiently?

Related Questions

7. What is the lower bound for finding the k smallest elements in a set of n numbers?
8. Given a min-heap of N elements, is it possible to retrieve the K smallest elements of the heap in O(K) time?
9. What is an efficient algorithm provided an n-ary tree, convert the elements in such a manner that every parent is median of its children?
10. I have a sorted array of N integers. The most frequent element of the array appears greater than or equal to N/3 times? What is the fastest way to find this integer?
11. How do I implement the problem of “finding the k^{th} smallest element in the union of two sorted arrays” in Java? What is the O(log k) solution?

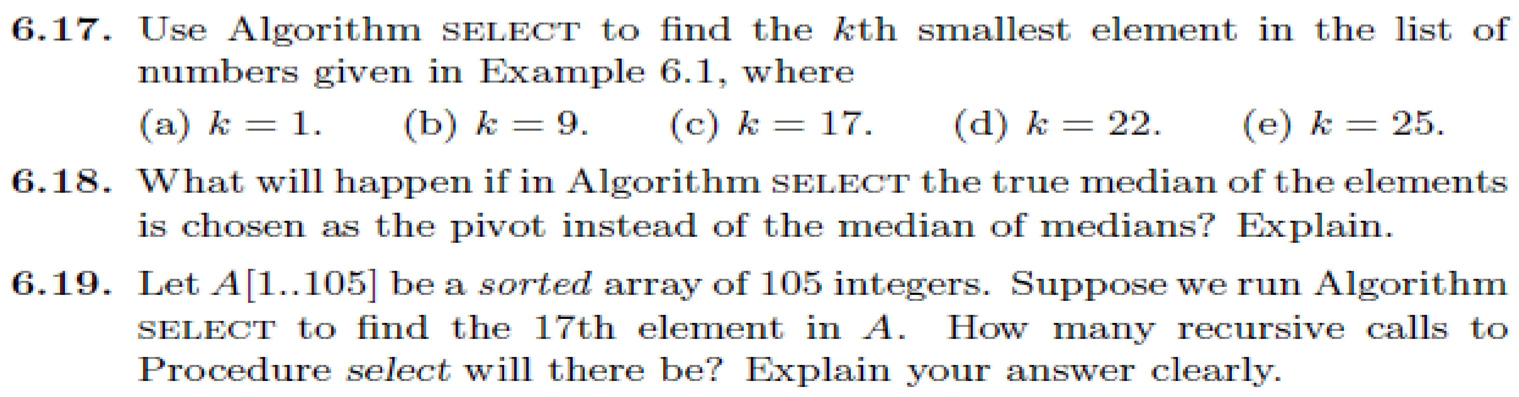
Related Questions

12. Given an array of N integers, how can you find M elements to remove so that the array will end up in sorted order?
13. What is the most efficient algorithm for calculating the mode of an array of integers?
14. What is the most efficient algorithm to find the kth smallest element in an array having n unordered elements?
15. What's the best way to delete duplicates from a sorted array in O(log N) time and O(1) space?
16. What is the most space-efficient algorithm for finding the N/2th element of a stream at the end if the number of elements in the stream is less than M?

Related Questions

17. If I have an array of n elements, how many elements will be in the segment tree to apply minimum range query operation?
18. Write an efficient algorithm for printing k largest elements in an array. Elements in array can be in any order. For example, if given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30 and 23.

Related Questions

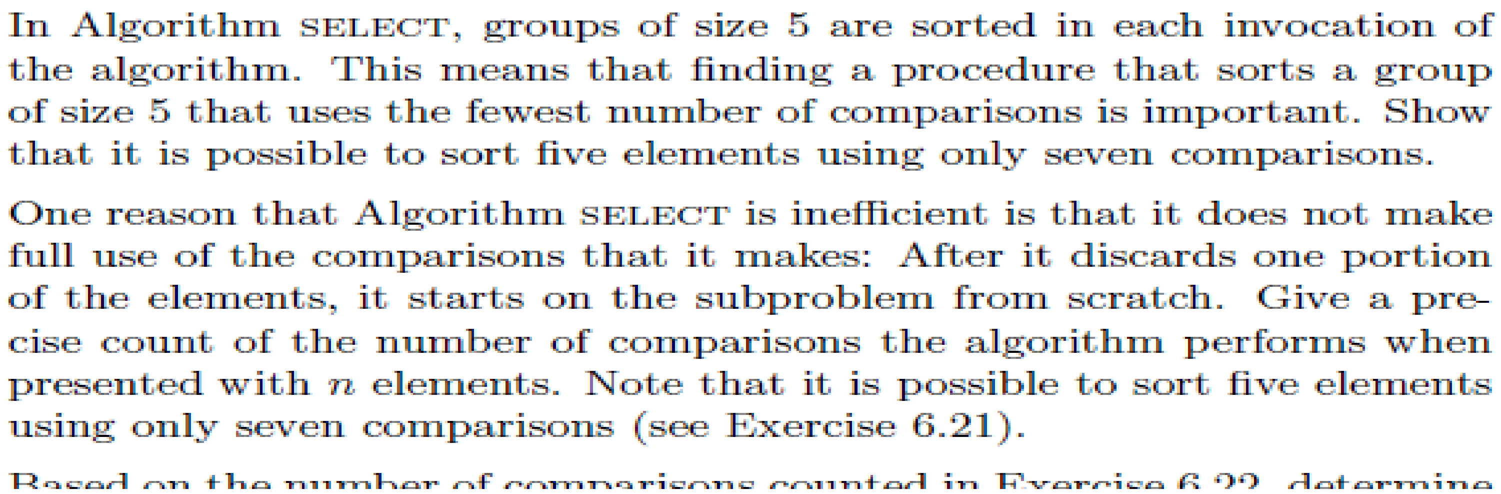


6.17. Use Algorithm SELECT to find the kth smallest element in the list of numbers given in Example 6.1, where

(a) $k = 1$. (b) $k = 9$. (c) $k = 17$. (d) $k = 22$. (e) $k = 25$.

6.18. What will happen if in Algorithm SELECT the true median of the elements is chosen as the pivot instead of the median of medians? Explain.

6.19. Let $A[1..105]$ be a *sorted* array of 105 integers. Suppose we run Algorithm SELECT to find the 17th element in A. How many recursive calls to Procedure *select* will there be? Explain your answer clearly.



In Algorithm SELECT, groups of size 5 are sorted in each invocation of the algorithm. This means that finding a procedure that sorts a group of size 5 that uses the fewest number of comparisons is important. Show that it is possible to sort five elements using only seven comparisons.

One reason that Algorithm SELECT is inefficient is that it does not make full use of the comparisons that it makes: After it discards one portion of the elements, it starts on the subproblem from scratch. Give a precise count of the number of comparisons the algorithm performs when presented with n elements. Note that it is possible to sort five elements using only seven comparisons (see Exercise 6.21).

Based on the number of comparisons counted in Exercise 6.22, determine

Exercises

(1) **The Fasility Location Problem (FLP):** We are given n points in the d dimensional space. We want to compute the location of the optimum facility point whose sum of distances to the given data points is minimized. Imagine we want to establish a communication center (the facility location) and run communication lines from that center to the users (the given points). The objective is to minimize the total length of the communication lines between the center and the users.

(a) Show the one dimentional version of FLP ($d = 1$) can be solved in $O(n)$ time.

(b) How would you solve the problem for $d = 2$ (the planar case) if the communication lines are only allowed to go along horizontal and vertical segments (suppose the communication lines are supposed to follow along the streets which are all North-South or East-West).

Exercises

(2) **The Majority Problem (MP):** We are given a set S of n elements. A majority value, if it exists, is one that appears more than $n/2$ times in the input set. The problem is to determine if S has a majority element, and if yes, find it.

(a) Suppose we are allowed to compare pairs of elements of S using the comparisons from the set $\{=, \neq, <, \leq, >, \geq\}$. Within this model we can solve MP in $O(n \lg n)$ worst-case time by first sorting S. Describe the rest of the process.

(b) Within the same comparison based model as in part (a), show the problem can be solved in $O(n)$ worst-case time using selection.

(c) Now suppose the only comparisons we are allowed to make are from the set $\{=, \neq\}$. So, we cannot sort. Show how to solve MP in worst-case $O(n \lg n)$ time in this model using divide-and-conquer.

(d) Within the same comparison based model as in part (c), show the problem can be solved in $O(n)$ worst-case time.

Thank You

Class Test: Duration 30minutes

Write a recursive search algorithm BINARYSEARCHREC , that facilitate the searching of an element that present is a sorted linear array.

Design a search algorithm that divides a sorted array into one third and two thirds instead of two halves as in Algorithm BINARYSEARCHREC. Analyze the time complexity of the algorithm.

Modify Algorithm BINARYSEARCHREC so that it divides the sorted array into three equal parts instead of two as in Algorithm BINARYSEARCHREC. In each iteration, the algorithm should test the element x to be searched for against two entries in the array. Analyze the time complexity of the algorithm.