

MEDIAN AND ORDER STATISTICS

Chapter 9.2 and 9.3 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Selection Order statistics

The selection problem also known as the order statistic problem or more specifically the k -th order statistic problem It is defined as follows.

Definition 1 (*Selection: k -th order statistic*).

Input: A sequence A of n comparable keys and an index k , where $1 \leq k \leq n$.

Output: The k -th order statistic that is defined as the k -th smallest of the n keys of A .

Comparable means that a total order is defined on the n keys of A . The statistic is always a key of A . The definition assumes that the keys are all distinct. If there are duplicates we can label them with an auxiliary value that is the index (address) of storage in the input to make them distinct.

Example 1. Let $A = \langle 20, 30, 10, 40 \rangle$. The $k = 1$ statistic is also known as $\min(A)$ and it is 10. The $k = 4$ statistic is also known as $\max(A)$ and it is 40. The $k = 3$ statistic is 30. It is the result of $\text{Select}(A, 4, 3)$.

Example 2. Let $B = \langle 10, 10, 10, 10 \rangle$. We convert into $B = \langle 10_1, 10_2, 10_3, 10_4 \rangle$. When we compare two keys say $10_1 : 10_2$ we first compare the key values and if they are equal (10) we resolve the comparison by comparing the subscripts. The $k = 1$ statistic is also known as $\min(A)$ and it is 10_1 , i.e. a 10. It is the result of $\text{Select}(A, 4, 1)$. The $k = 4$ statistic is also known as $\max(A)$ and it is 10_4 , i.e. a 10.

Order statistics

Some statistics: $k = 1$ and $k = n$

The following was discussed in the context of SelectionSort.

Lemma 1. *The $k = 1$ order statistic can be found in $n - 1$ comparisons. It is known as Min. The $k = n$ order statistic can be found in $n - 1$ comparisons. It is known as Max.*

Definition 2 (*Median*). The median of n keys is the $\lfloor (n + 1)/2 \rfloor$ -st order statistic. The median is sometimes referred to as lower median.

Definition 3 (*Upper Median*). The upper median of n keys is the $\lceil (n + 1)/2 \rceil$ -st order statistic.

Example 3. Let $A = \langle 20, 30, 10, 40 \rangle$. The median or lower median is 20. The upper median is 30. In a MATH course you might be told that 25 is the median: this is not a MATH section. A statistic, and the median or lower median is a statistic is a key of A and A does not have a 25.

Example 4. Let $A = \langle 50, 20, 30, 10, 40 \rangle$. The upper median is 30. The median is 30. The lower median is 30.

Selection Suboptimal Select

Proposition 2. *SubOptimalSelect(A, n, k) finds the k -th order statistic of n keys in $O(n \lg n)$ time.*

Proof. The following solution finds the k -th order statistic for distinct keys. However it misses the answers to the Example of the previous case that contains duplicates. There is no guarantee that the Min returned will be the first 10; the answer will be correct though.

```
SubOptimalSelect(A[0..n-1], n, k)
1. HeapSort(A, n);
2. return(A[k-1]);
```

An alternative is as follows

```
SubOptimalSelectBetter(A, n, k)
1. MergeSort(A, n);
2. return(A[k-1]);
```

The running time of one or the other approach is $T(n) = O(n \lg n)$. Naturally we can also write $T(n) = \Theta(n \lg n)$. ■

Related problems

MinMax, Min2Min, Max2Max

The $\text{Select}(A, n, 1)$ and $\text{Select}(A, n, n)$ are the Min (A, n) and Max (A, n) problems that compute the $k = 1$ and $k = n$ statistics that is the 1-st order and the n -th order statistics of n keys. Each one of those can be found in $n - 1$ comparisons by the discussion of SelectionSort. Thus $T(n) = \Theta(n)$ is the running time of those instances of problem Select.

Definition 4 (*MinMax*). The MinMax problem finds the Minimum and the Maximum at the same time.

Input: A sequence A of n comparable keys.

Output: The Min and Max of A .

Proof. The MinMax problem is the $k = 1 \wedge k = n$ order statistics problems. We have already described partial solutions explicitly or implicitly.

One solution in $2(n - 1) = 2n - 2$ comparisons is to solve the Min and Max problem independently.

One solution in $2n - 3$ comparisons is to solve the Min problem in $n - 1$ comparisons and then for the remaining $n - 1$ keys solve the Max problem in $n - 2$ further comparisons for a total of $2n - 3$.

If n is a power of two we can solve MinMax in $3n/2 - 2$ comparisons.

If n is even we can pair the keys and with $n/2$ comparisons (one per pair) we find the m_i of each pair, the minimum, and also the M_i of each pair, the maximum. Then we find the Min of the m_i in $n/2 - 1$ comparisons and the Max of the M_i in $n/2 - 1$ comparisons. The total is $3n/2 - 2$. Resolve as an exercise the n 'is odd' case. ■

MinMax Example

Example 5 (MinMax). An example follows. (Based on Problem Set 1, 2, or 3.)

Solution.

50 10 20 30 40 60 70 80	Input	
50:10 20:30 40:60 70:80	Pair up; compare min bottom; max top	$\sim n/2$
50 30 60 80	Find Max of max	$\sim n/2 - 1$
10 20 40 70	Find Min of min	$\sim n/2 - 1$
	Total:	$\sim 3n/2 - 2$
50 10 20 30 40 60 70 80	Divide $n=8$ into 2 x $n=4$	Practice Makes Perfect Fill in details
50 10 20 30 40 60 70 80	Divide $n=4$ into 2 x $n=2$	
50:10 20:30 40:60 70:80	Compare the pairs	$\sim n/2$
50 : 30 60 : 80	Max of Max of $n=2$	$\sim n/4 \times 2$
10 : 20 40 : 70	Min of Min of $n=2$	
50 : 80	Max of Max of $n=4$	$\sim n/8 \times 2$
10 : 40	Min of Min of $n=4$	...
...		$\sim 1 \times 2$
80		Total: $\sim 3n/2 - 2$
10		

■

Definition 5 (*Min2Min*). The Min2Min problem finds the $k = 1$ and $k = 2$ of n keys at the same time.

Input: A sequence A of n comparable keys.

Output: The $k = 1$ and $k = 2$ order statistics of the n keys (smallest and second smallest).

Proof. An equivalent problem is MAX-2MAX for the largest and second largest of n keys i.e. the $k = n - 1$ and $k = n$ order statistics.

Similarly to Min and Max it can be done naively in $2n - 2$ or $2n - 3$ comparisons.

A solution in $n + \lceil \lg n \rceil - 2$ comparisons involves a tournament tree, or a binomial tree or an array of length around $2n$ and clever thinking. We can solve this problem also by building a MinHEAP in $\approx 2n$ comparisons and then perform two ExtractMin operations, or just one ExtractMin and a FindMin operation for a total of $2n + O(\lg n)$ comparisons. ■

Two statistics continued

Continued. 10 Winner is MINIMUM and move to next round
 / \
 10 : 40 Number of rounds $\text{ceil}(\lg n)$
 / \ / \ Number of comparisons : to find MIN $n-1$
 10 : 20 40 : 70 Who can be second smallest: Lost a game to MIN {40,50,20}
 / \ / \ / \ / \ Find the min of those losers : in $\text{ceil}(\lg n)-1$ comparisons
 50:10 20:30 40:60 70:80 Total : $n-1 + \text{ceil}(\lg n)-1$ comparisons

Binomial Tree (not binary tree)

50:10 20:30 40:60 70:80 Pair up

10 : 20 40 : 70 Winner Root; Loser hangs under root (child of)
 | | | |
 50 30 60 80

10 : 40 Winner ...
 | \ | \
 50 20 60 70
 | |
 30 80

10---- MIN : The root
 | \ \ 2MIN : The smallest of the three children
 50 20 40 of the root i.e. $\min(50,20,40)=20$
 | | \ Total : $n-1 + \text{ceil}(\lg n)-1$ comparisons
 30 60 70 (9 comparisons: $8-1$ for MIN, $3-1$ for MIN{50,20,40})
 |
 80



Max2Max Two statistics continued

Input : 70, 20, 40, 80, 60, 10, 50, 30 => 70, 20, 40, 80, 60, 10, 50, 30
 70 20 40 80 60 10 50 30 :n 'trees'

DIFFERENT EXAMPLE

Step 1: Pair keys + compare: max up, min down : n/2 comparisons

```

70      80      60      50
|        |        |        |
20      40      10      30
  
```

: n/2 trees

20, 70, 40, 80, 10, 60, 30, 50

Step 2: Repeat for max found

: n/4 comparisons

```

      80      60
    /  |    / |
  70 40  50 10
    |      |
   20     30
  
```

: n/4 trees

20, 70, 40, 80, 30, 50, 10, 60

Step 3: Repeat for max found

: n/8 comparisons

```

      .... 80
    /  /  |
  60 70 40
 /  |  |
50 10 20
 |
30
  
```

: n/8 trees

Index	0	1	2	3	4	5	6	7
	30,	50,	10,	60	20,	70,	40,	80

Index		7-4,		7-2,7-1
-------	--	------	--	---------

Step 4: Go on until one tree is left.

MAX : root , 2MAX: max of ceil(lgn) children i.e. max(60,70,40)

Let t be index of MAX in the array view i.e. A[t] is MAX

Indexes of its children are at A[t-2**0], A[t-2**1], A[t-2**2], ...

Randomized Selection

Expected linear time but quadratic time

Proposition 3 (*RQuickSelect*). *RQuickSelect*(A, n, k) is a solution to *Select*(A, n, k) by using a randomized algorithm that leads to randomized selection. *RQuickSelect* uses *GTRPartition* and leads to an algorithm with expected running time $\tilde{T}(n) = \Theta(n)$. Its best case running time is $B(n) = \Theta(n)$ and its worst case running time $W(n) = \Theta(n^2)$, the worst case running time of *GTQuickSort* or *GTRQuickSort*. The running time of *RQuickSelect* is $T(n) = O(n^2)$.

A log-linear (i.e. $\Theta(n \lg n)$) algorithm for the selection problem is *SubOptimalSelect*. This suggests there is a better solution and indeed it would be *DetSelect*, i.e. deterministic *Select*. To reach the optimal solution we are going to present *RQuickSelect*, a randomized version of *Select* that uses the *GTRPartition* of randomized quicksort.

To find the k -th order statistic, we use a random splitter to split the n input keys around it. The k -th statistic can be on the left side, on the right side or it is the splitter! In the latter case we are lucky and we are done. In the other two cases we only need to recur on the side that contains the splitter. Whereas *QuickSort* or *GTRQuickSort* must work on BOTH sides, *RQuickSelect* does not need to! Thus a $C(n) = 2C(n/2) + (n - 1)$ recurrence for the comparisons performed in *QuickSort* or *GTRQuickSort* if the split is the best one becomes $C(n) = C(n/2) + (n - 1)$ for *RQuickSelect*. Assuming of course we are not super lucky and the splitter is the statistic! The algorithm uses *GTRPartition* of *GTRQuickSort*. The wrapper function for the recursive pseudocode wraps around function *RQSel* that does all the work.

Randomized Selection

Expected linear time but quadratic time

```
RQuickSelect(A,n,k)    // RandSelect is the generic reference
1. RQSel(A,k,0,n-1);    // A[0..n-1] for k-th order statistic
    | | |
    | | |
    RQSel(A,i,l,r)      // A[l..r] for i-th order statistic (in general)
1. if (l=r) return(A[l]); // Base case
2. m=GTRPARTITION(A,l,r); // A[m] is the splitter at completion of line 2
3. t=m-l+1;             // t : # keys left side + 1 for splitter
4. if (i==t)             // Is statistic i the splitter? A[m]?
5.   return(A[m]);       // YES: return it
6. elseif (i<t)          // NO, is it on left side ?
7.   RQSel(A,i,l,m-1)    // YES, recur on left side
8. else
9.   RQSel(A,i-t,m+1,r)  // NO, recur on right side ; adjust i! to exclude keys
                        that were left out (left side or splitter)
```

Randomized Selection

Example

Example 6.

```
Input :    n=100 keys    A= [ arbitraryorder( 101, ... , 200) ]

                                Solve two problems
                                RQuickSelect(A,n=100,k=64)
                                ||
                                RQSel(A,i=64,0,99)
                                Pick Splitter : 141
                                and split A around splitter 141
                                A[0 .. 39]          A[40]          A[41 ..99]
                                Left: A[1 .. m-1]    splitter=A[m ]    Right: A[m+1 .. r]
                                                141
                                <--- 40keys----->          <----- 59 keys----->
                                <-----41 keys----->
                                    t=41

Step 7:    if  32=i<t=41
            RQSel(A,i=32,0,39)

Step 9:                                else  // (i>=41)
                                           RQSel(A,i-t=64-41,41,99)
```

Randomized Selection

The proof part

Proof. Operations.

The running time and comparison count of RQuickSelect is the running time $T(n)$ and comparisons count $C(n)$ of RQSel for $i = k$ and $l = 0$ and $r = n - 1$ that is being invoked by Line 1 of RQuickSelect and thus RQSel operates on all of the n keys of A . Lines 1-5 of RQSel for $l = 1$, $r = n$ have running time $\Theta(n)$. Line 7 is $T(m)$ and Line 9 is $T(n - m - 1)$ (same rationale as that of GTRQuickSort). Thus the running time of RQuickSelect is either $T(n) = T(m) + \Theta(n)$ if we recur on the left side, or $T(n) = T(n - m - 1) + \Theta(n)$ if we recur on the right side, or $T(n) = \Theta(n)$ if we recur on neither side but exit through Line 5.

Best and Worst case running time.

Note 1. There are two best cases:

- (a) Splitter $A[m]$ is the statistic (Line 4 $i == t$)
- (b) Splitter $A[m]$ is NOT the statistic but split is even.

If we exit through Line 5 (i.e. Line 4 $i == t$ evaluates true) and we are in best case (a), the running time is obviously $T(n) = \Theta(n)$. Another best case is for $m = n - m - 1 \approx n/2$ i.e. case (b) describing the second best case. Then $T(n) = T(m) + \Theta(n) = T(n/2) + \Theta(n)$ or $T(n) = T(n - m - 1) + \Theta(n) = T(n/2) + \Theta(n)$. The master method settles $T(n) = \Theta(n)$. The worst case scenario derives from the splitter being MAXIMUM or the MINIMUM of the input and neither is the statistic that needs to be determined. Then $T(n) = T(n - 1) + \Theta(n)$, and it yields a $T(n) = \Theta(n^2)$ worst case running time, just like the worst-case of QuickSort.

Expected Running time of RandSelect (aka quickSelect).

We use the argument utilized in GTR-QuickSort with the GoodSplitters and BadSplitters. We have on the average a BadSplitter-GoodSplitter sequence. Thus in two consecutive iterations (Bad follow by Good on the average) an original RS problem of size n becomes an RS problem of size $n/4$ or $3n/4$. Thus the expected running time $\tilde{T}(n) = \tilde{T}(n/4) + \Theta(n)$ or $\tilde{T}(n) = \tilde{T}(3n/4) + \Theta(n)$. The solution using the master method is asymptotically the same (for either recurrence) with that of $\tilde{T}(n) = \tilde{T}(n/2) + \Theta(n)$ i.e. $\tilde{T}(n) = \Theta(n)$. ■

Randomized Selection

The (pictorial) proof part

Example 7. Input is $A[1..n]$. If $B = \text{Sort}(A)$ then $B[1], \dots, B[n]$ are the sorted keys of A .

'Worst' Case Split	0 - n-1	'Best' Case Split	n/2 - n/2
but also others :	$< n/4 - > 3n/4$	but also others :	$\geq n/4 - \leq 3n/4$
'Bad' :	$B[1], \dots, B[n/4],$	'Better' :	$B[n/4+1], \dots, B[n]$
splitters	$B[3n/4+1] \dots, B[n]$	splitters	
Total :	n/2	Total :	n/2

Assume all 'Bad' splitter splits
are WORST i.e. 0 - (n-1)

Assume all 'Better' splitter splits
are least better i.e. $n/4 - 3n/4$

	n		
	/ \		
Bad	0	n-1 ~ n ~~~~~~	$n \cdot (3/4)^{**0}$ at Iteration 1
	/ \		
Better	n/4	~3n/4-1	$n \cdot (3/4)^{**1}$ at Iteration 2
	/ \		
Bad	0	~3n/4 -1 ~~~~~~	$n \cdot (3/4)^{**1}$ at Iteration 3
	/ \		
Better	.25*3n/4	~0.75*3n/4~	$n \cdot (3/4)^{**2}$ at Iteration 4
	\		
	...		
	\		
~		~ $n \cdot (3/4)^{**i}$	at Iteration $2i+1$ or $2i+2$

$T(n)$ = expected running time ~ expected number of comparisons

$\leq n + n + (3n/4) + (3n/4) + \dots + (3/4)^{**i} + (3/4)^{**i}$

$\leq 2*n + 2*n \sum_i (3/4)^{**i}$, i is bounded

$\leq 2*n + 2*n \sum_i (3/4)^{**i}$, i is unbounded to infinity

$\leq 2*n + 2*n * 1/(1- 3/4) \leq 10*n$ i.e. $\Theta(n)$

Deterministic selection

Linear time algorithm

Proposition 4 (*DetSelect*). *DetSelect*(A, n, k) solves *Select*(A, n, k) in worst-case linear time. The running time of *DetSelect* is thus $T(n) = \Theta(n)$.

The function *DetSelect* is a wrapper function to *DSelect*. The latter resembles *RQSel* with one exception: lines 1-3 or *RQSel* are replaced with lines that determine a splitter deterministically. The selection of a splitter is more complicated, involves a recursive separate call to *DSelect* itself but guarantees that the recursive calls to *DSelect* similar to the ones of line 7 and line 9 of *RQSel* will NEVER involve a subarray that will have either few keys or all of the keys; a recursive call will involve at least $3n/10$ keys or about so and at most $7n/10$ keys or about so (about so has to do with whether n is a multiple of 4 or 5). Avoiding the extremes cases (good or bad) is all it then takes to get a linear time algorithm.

DetSelect's splitters. *DetSelect* through *DSelect* picks a splitter as follows. The keys are split into groups of 5 keys (why 5 and not 3,7,9, 11 leads to an interesting discussion later; why not 4,6,8,10 will become clear soon). Each group of five keys is sorted with InsertionSort with the intent of finding the group's median, which is the third smallest of those 5 keys. We do so for every one of the $n/5$ groups or so and we collect those medians. The median of those medians will become the splitter! Thus we must solve a selection problem involving $q = \lceil (n/5) \rceil$ keys, all the medians of all groups of 5 keys, and the median of those medians is a statistic whose order will be $\lfloor (q+1)/2 \rfloor$.

DetSelect and DSelect. The pseudo-code of *DSelect* maintains as much as possible in lines 4-9 the structure of *RQSel*. Lines 0-3 are different to reflect the new splitter selection method.

Deterministic selection DetSelect aka DSelect

	DetSelect(A,n,k) // A[0..n-1]	TIME ANALYSIS
	1. DSelect(A,k,0,n-1) // A[0..n-1]	T(n) =
	k-th order statistic	T(n) =
	DSelect(A,i,l,r) // A[l..r]	
	i-th order statistic	
0.	Group keys into groups of 5;	Theta(n)
	Sort each group with InsertionSort;	Theta(n)=n/5*(5(5-1)/2)
	Identify median of each group	Theta(n)
	(third smallest of five sorted keys)	
1.	Collect q medians into set M;	Theta(n)
	// q = ceil((r-l+1)/5)	
2.	x = DSelect (M, floor((q+1)/2),0,q-1);	T(n/5)
	// swap x with M[q-1] implicitly	
3.	m= GTP(A,l,r) around x;	Theta(n)
	A[l..m-1]U A[m] = { <= x}, . =t;	
	A[m+1..r] = { > x}, . =(r-l+1)-t;	
4.	if (i==t)	
5.	return(A[m]);	Theta(n)
6.	elseif (i<t)	or
7.	DSelect(A,i,l,m-1);	T(Left)=T(7n/10)
8.	else	or
9.	DSelect(A,i-t,m+1,r);	T(Right)=T(7n/10)
		=====
		T(n) = T(n/5)+T(7n/10)+Theta(n)

Deterministic selection

Performance

Proof. Performance. The running time of DetSelect for n keys is the running time DSelect with $i = k$ and $l = 0$ and $r = n - 1$ involving those n keys. Let the latter be $T(n)$. We can isolate a comparison count by using $C(n)$.

Lines 0-1. The running time of line 0 and line 1 of DSelect is obviously $\Theta(n)$. We identify the group of say the 12th element by a simple $12/5 = 2$ and thus the 12-th element belongs to group 2 which is the third group after 0 and 1. Sorting with InsertionSort 5 keys, per Subject 1, requires $5(5 - 1)/2 = 10$ comparisons. Repeating so for $q \approx n/5$ groups Line 0 uses up $2n$ comparisons. Collectively the running time of line 0 and line 1 is $\Theta(n)$ and the comparison count approximately $2n$.

Line 2. The running time of line 3 involves a recursive call to DSelect. Its running time is $T(q)$, where q is the number of groups of 5 i.e. the number of medians collected, which is $q = \lceil n/5 \rceil$. This is asymptotically $T(n/5)$ and the comparison count $C(n/5)$.

Line 3. In $n - 1$ comparisons we split the keys around the chosen splitter x . Running time is $\Theta(n)$.

Lines 4-9. Beyond this the analysis follows the analysis of RS. We either recur on the left side for $T(7n/10 + 2)$ or on the right side for $T(7n/10 + 2)$ in the worst case. Either one give $T(7n/10 + 2)$ and comparison-wise $C(7n/10 + 2)$. (The $7n/10 + 2$ bound will be proven separately after the end of this proof.)

Therefore the running time $T(n)$ is given by the following recurrence if we add up all contribution.

$$T(n) = T(n/5) + T(7n/10 + 2) + \Theta(n).$$

Comparison wise line 0 uses $2n$ and line 3 uses $n - 1$ comparisons; the rest are either the recursive call of line 2 or one of line 7 and line 9.

$$C(n) = C(n/5) + C(7n/10 + 2) + (3n - 1).$$

The recurrence was solved earlier using the substitution method. The running time of DSel and also DetSelect is thus $T(n) = \Theta(n)$. ■

Lemma 5. *In line 7 or line 9 of DSelect the recurrence involves a subarray of A with no more than $7n/10 + 2$ keys.*

Proof. If the median of the medians is x , consider in the figure below the bottom left side and top right side. The former contains the keys in each group that are guaranteed to be smaller than the corresponding median of that group. The left side contains groups with median less than or equal to x . Likewise for the top right side that contains keys greater than x . The other two areas (bottom right or top left) can contain keys less or greater than x or equal to x . Thus the bottom side contains roughly 3 keys per group for half of the groups ($\approx n/10$) involved and so does the top right side. If in DSelect the two sides of recursion in lines 7 and 9 each contains at least $\approx 3 \cdot n/10 \approx 3n/10$ of the n keys, the other side must contain no more than $7n/10$ and this proves the claim of the previous slide. For more accurate counting, the number of groups is q with $q = \lceil (n/5) \rceil \geq n/5$.

Case 1. q is even i.e. $q/2$ is integer. Then half of the groups have medians $\leq x$ and half $> x$ and each group including the median of the group has a total of 3 keys $\leq x$ or $> x$. Thus left side has at least $3q/2 - 2 \geq 3n/10 - 2$ keys, likewise for the right side. If n is not a multiple of 5, a group can have between 1 and 4 keys, thus number of keys less than $\leq x$ or not is 1 or 2 then and not 3. This also implies that the maximum number of keys involved in the recursive call of Line 7 or Line 9 is $7n/10 + 2$.

Case 2. q is odd i.e. $(q+1)/2$ is integer. There are $(q+1)/2$ groups with 3 keys $\leq x$ contributing a total of at least $3n/10$ keys. Moreover there are $(q-1)/2$ groups with 3 keys $> x$ plus the two keys of the group of x which are $> x$. The total is $3(q-1)/2 + 2 - 2 \geq 3n/10 - 1.5$. This also implies that the maximum number of keys involved in the recursive call of Line 7 or Line 9 is $7n/10 + 1.5 \leq 7n/10 + 2$.

Median x of the medians of 5-element groups.

x is the median of medians (half are larger than or equal to x)								
	Grp 1	Grp 2	Grp 3	...				
largest within group	10000	709	20000		1020	5000	6000	1750
	1300	301	800		1010	4000	3000	1550
medians-of-5	500	300	700		x=1000	2000	2500	1500
	100	5	600		8	11	2200	1450
smallest within group	10	1	2		7	3	1600	1400



Moreover a comparison count can be estimated for DetSelect (or DSelect).

Example 8.

x is the median of medians (half are larger than or equal to x)

	Grp 1	Grp 2	Grp 3	...				
Group max	10000	709	20000		1020	5000	6000	1750
	1300	301	800		1010	4000	3000	1550
medians-of-5	500	300	700		$x=1000$	2000	2500	1500
	100	5	600		8	11	2200	1450
Group min	10	1	2		7	3	1600	1400

The bottom **Left** side contains the keys guaranteed to be $\leq x$.

Thus $|\text{Left}| \geq 1/2 * n/5 * 3 = 3n/10$.

The top **Right** side contains the keys guaranteed to be $> x$.

Thus $|\text{Right}| \geq 1/2 * n/5 * 3 = 3n/10$.

Since $|\text{Right}| + |\text{Left}| = n$, we have that $|\text{Left}| \leq 7n/10$ and $|\text{Right}| \leq 7n/10$.

Lemma 6. *The comparison count $C(n)$ of DSelect is given by the following expression.*

$$C(n) \leq 33n \quad n \geq 350.$$

Proof. If $n > 100$, and $q = \lceil n/5 \rceil \leq n/5 + 1$, we have a refinements of the recurrence for $C(n)$ as follows.

$$C(n) = C(n/5 + 1) + C(7n/10 + 2) + 3n - 1 \quad n \geq 100$$

else if $n < 100$, $C(n) = \Theta(1)$ and it takes a constant number of comparisons to solve such problems. For the upper bound of $C(n)$ we have,

$$\begin{aligned} C(n) &= C(n/5 + 1) + C(7n/10 + 2) + 3n - 1 \\ &\leq c(n/5 + 1) + c(7n/10 + 2) + 3n - 1 \\ &= 9cn/10 + 3c + 3n - 1 \end{aligned}$$

We observe that $9cn/10 + 3c + 3n - 1 \leq cn$ for say $c = 33$ as long as $n \geq 350$. Larger values of c are OK as well. Therefore, for $n \geq 350$

$$C(n) \leq 33n$$

which also implies that $T(n) = \Theta(n)$, as needed. In other words this method is competitive to sorting for $n > 2^{33}$ or so. ■

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 1.

Answer the following questions briefly by justifying your answers. If needed provide asymptotically tight bounds as answers.

- (a) What's the running time of $\text{RandSelect}(A, n, i)$?
- (b) What's the worst-case running time of $\text{RandSelect}(A, n, i)$?
- (c) What's the expected running time of $\text{RandSelect}(A, n, i)$?
- (d) What's the worst-case running time of (deterministic) $\text{Select}(A, n, i)$?
- (e) What's the running time of (deterministic) $\text{Select}(A, n, i)$?

Proof. (a) $T(n) = O(n^2)$

(b) $T(n) = \Theta(n^2)$

(c) $T(n) = \Theta(n)$

(d) $T(n) = \Theta(n)$

(e) $T(n) = \Theta(n)$

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

■

Problem Set
Problem 2.

Google ranks web-pages with an arbitrary numerical value. One cannot make any assumptions about the range of the rank. Suppose that a query returns n pages (n is part of the input) and information about rank is stored for the i -th page in $A[i]$. We are interested in determining the top 20%, the bottom 40% and the middle 40% of all pages. For that we need to determine two values m and M , the values for which 40% of the pages have ranks less than m , 20% more than M and those whose rank is between m and M . Give an efficient algorithm (whose performance cannot be improved upon) that solves this problem, i.e. it finds m , and M given n and A .

Proof. Selection. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem 3.

Close $2k + 1$ elements. Let A be an array of n arbitrarily ordered keys that are comparable. Let $k \leq n/2$ be an input parameter. We would like to find $2k + 1$ keys of A whose median is the same as the median of A itself. Give an algorithm for $\text{Find}(A, n, k)$ whose worst-case running time is $o(n)$ (little-oh). (You just need to find some set of $2k + 1$ keys; if there are many candidates for such sets, return just one of them.)

Proof. Time efficient means skip sorting. Skip sorting mean linear-time. Linear-time means selection. Proof by example. Say the keys are the integer from 1 to 20. Median is 10. For a set of $2k + 1 = 9$ keys that has median also 10. I'll pick 4 smaller than 10, 4 larger than 10, and of course 10!. I can pick the 4 smaller to be 1, 2, 3, 4 or 1, 7, 8, 6, or you guessed it, didn't you? ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 4.

There are two types of NJIT professors: **nice** that give 100% of their students an **A** and **mean** that gives 80% of their students an **A** and 20% a **B+**. You need to determine whether professor **I.M.Cute** is **nice** or **mean**. Towards this you have access to his grading this past semester in the form of array $A[1..n]$, $n > 25$. An entry $A[i]$, $1 \leq i \leq n$, can be an **A** or a **B+**. However reading entry $A[i]$ for some arbitrary i , is going to cost you \$1. If your budget is only \$20, can you determine whether **I.M.Cute** is **nice** or **mean** with 90% or more confidence and at a cost of no more than \$20? Explain.

Disclaimer. Do not take seriously the definition of a **mean** NJIT Professor :-) I am neither **nice** nor **mean**!

Proof. That's the Google problem! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 5.

How could you modify quicksort (give pseudocode or describe convincingly the changes) so that its worst-case performance is $O(n \lg n)$? Describe your changes, analyze the worst-case running time of the modified quicksort and show that it is indeed $O(n \lg n)$ in the worst-case.

Proof. Pick the splitter (pivot) key to be the median of the keys of the input participating in the recursive call. Finding the median requires linear time (though with a very large constant). However it guarantees a 50-50 split (plus or minus one key). Thus the recurrence for the running time of this modified quick-sort will be (plus or minus one key): $T(n) = 2T(n/2) + Cn$, where C is a big constant. The asymptotic solution is $\Theta(n \lg n)$, which is acceptable and asymptotically faster than $\Theta(n^2)$ in the worst-case. Pseudocode-wise we can just change line 1 of Partition slightly.

OLD CODE

```
1. splitter= A[r];
```

NEW CODE

```
// DSelect 'returns' the index of the statistic not the statistic itself
1a. splitter-index=DSelect(A,floor((r-l+1)/2),l,r);
1b. swap(A[r],A[splitter-index]);
1c. splitter=A[r];
```

■

Problem Set Problem 6.

For a set of $n = 2^m$ unordered keys we are interested in finding the 1-st, 2-nd, 4-th, $\dots$, 2^{m-2} -th, 2^{m-1} -th order statistic. Give an algorithm whose running time is $O(n)$ in the worst case that finds all m statistics. **Hint: Be careful. If you use Select m times on the set of n keys, it takes $O(n)$ per statistic for a total of $O(nm) = O(n \lg n)$, which is too much.**

Proof.

```
FindAllStatistics(A[1..n],n,m)
1. S= A[1..n];
2. for (i=m-1 ; i >=0 ; i--) {
3.     m = Median(S);
4.     Sleft= { all x of S such that x <=m };
5.     Output('2*i-th statistic is m');
6.     S=Sleft;
7. }
```

Start from the right statistic (the rightmost one). Solve the first selection problem. Filter out the unneeded keys and continue with the next problem and so on. In the beginning of the for loop, $S = A$.

An inductive argument proves that in the i -th iteration S contains the $2 \cdot 2^i$ smallest keys of A , and therefore median m of S is the 2^i -th order statistic of A . Lines 4-6 prepare S for the following iteration by maintaining in S the 2^i smallest keys of A . In the i -th iteration, S contains the $2 \cdot 2^i$ smallest keys of set A . We then find the median of S which is m and identify the 2^i smallest keys of S , i.e. the 2^i smallest keys of A itself. This also means that m is the 2^i -th order statistic of A . These keys are temporarily stored in $Sleft$ and then transferred (copied) into S before the next iteration.

Analysis of the Algorithm. Step 1 takes $O(n)$ steps for copying n keys. The body of the loop is executed m times, but the running time each time is different. In the i -th execution, S has size $2 \cdot 2^i$, and therefore median finding takes worst case $O(2^i)$ time using Select, Step 4 takes the same amount of time as well, and so is Step 6; Step 5 takes $O(1)$ time. Therefore the i -th execution of the loops takes time $O(2^i)$. Therefore as $n = 2^m$, we have that

$$\sum_{i=0}^{m-1} O(2^i) = O\left(\sum_{i=0}^{m-1} 2^i\right) = O(2^m - 1/(2 - 1)) = O(2^m) = O(n).$$

■

Problem Set

Problem 7.

For n distinct numbers $x_1, x_2, \dots, x_n$ with positive weights $w_1, w_2, \dots, w_n$ such that $\sum_{i=1}^n w_i = 1$, the **weighted median** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \leq 1/2, \quad \sum_{x_i > x_k} w_i \leq 1/2$$

i. Argue that the median of $x_1, x_2, \dots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, \dots, n$.

ii. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time.

Proof. (i) Let the median of the n keys be M . Then M is the $\lfloor (n+1)/2 \rfloor$ smallest of the n keys. If all keys have the same weight the $\lfloor (n+1)/2 \rfloor - 1$ keys smaller than the median have weight less than $1/2$, as $\lfloor (n+1)/2 \rfloor - 1 \leq (n+1)/2 - 1 = (n-1)/2$ and $(n-1)/2 \cdot 1/n < 1/2$. Similarly the keys greater than the median have total weight less than or equal to $1/2$ as well. (One easy way to prove it is to consider two cases.)

Case 1. $n = 2k + 1$ i.e. n is odd. Then the number of keys greater than M is k for a total weight of $k/(2k+1) < 1/2$.

Case 2. $n = 2k$ i.e. n is even. Then the number of keys greater than M is also k for a total weight of $k/(2k) = 1/2$.)

(ii) The algorithm for finding M is given below.

```
WM( A[1..n] ) //A[i].key is the key A[i].weight is the weight of key A[i].key
1.  Y = MergeSort(A,n);
2.  Z = empty-set; // Z will contain keys less than weighted median M
3.  sum = 0;       //stores weight of Z
4.  for (i=1;i<=n;i++){
5.      Z = Z U { Y[i] }
6.      sum = sum + Y[i].weight;
7.      if sum < 1/2 continue;
8.      else {
9.          Z = Z - { Y[i] };
10.         report Y[i] as weighted median
11.         exit;
12.     }
13. }
```

The time for step 1 is $O(n \lg n)$, for 2-3 is $O(1)$, and for the loop of lines 5-13 is $O(n)$. Briefly, we sort the keys in increasing order and if for the first $i-1$ keys the total weight is $< 1/2$ but the weight of the first i keys is greater than or equal to $1/2$, this means that the i -th sorted key is M and it is reported in line 10. ■

Problem Set Problem 8.

Find the median of the $2n$ keys stored in $X[1..n]$ and $Y[1..n]$. Both X and Y are pre-sorted. Given a time efficient algorithm. (Merging and HeapSort does not count as a time efficient algorithm!!!)

Proof.

```
Median(X,i,j) //Returns the median of the sorted subsequence X[i..j]
1. return(X[(i+j)/2]);

WhichMedian(X,Y,n) // We implement WhichMedian
1. return(FindMedian(X,1,n,Y,1,n)); // by writing code for FindMedian

FindMedian(X,i,j,Y,k,l);
1. key Xmedian, Ymedian;
2. Xmedian = Median(X,i,j);
3. Ymedian = Median(Y,k,l);
4. if (Xmedian == Ymedian) return(Xmedian);
5. else
6.     if (Xmedian < Ymedian)
7.         return(FindMedian(X,(i+j)/2+1,j,Y,k,(k+1)/2));
8.     else
9.         return(FindMedian(X,i,(i+j)/2,Y,(k+1)/2+1,l));
```

Note that the two sequences X and Y of size n each are already sorted. We first collect the medians X_{median} and Y_{median} of X and Y respectively. This can be done in $O(1)$ time as the two sets are sorted. X_{median} splits X into two sets, the "left" set of keys with value less than or equal to X_{median} and the "right" set of keys with values greater than X_{median} .

If the two medians X_{median} and Y_{median} are equal, $n/2$ keys of X (left set) are $\leq X_{\text{median}}$ and $n/2$ keys of Y (right set) are $\leq Y_{\text{median}} = X_{\text{median}}$, i.e. n out of $2n$ keys are $\leq$ the common median X_{median} . The remaining n keys of both sequences belong to the right sets i.e. they are greater than $X_{\text{median}} = Y_{\text{median}}$. Therefore $X_{\text{median}} = Y_{\text{median}}$ is the median of $X \cup Y$. Suppose that $X_{\text{median}} < Y_{\text{median}}$ (the other case is treated symmetrically). Then the left set of X and the right set of Y can not contain the median of $X \cup Y$ as they include the $n/2 - 1$ smallest keys of $X \cup Y$ and the $n/2$ keys of $X \cup Y$. Therefore the median of $X \cup Y$ must be in the right set of X or the left set of Y .

Problem Set Continued

This way the original problem of finding the median of two sorted sequences each of size n has been reduced to finding the median of two sorted sequences each of size $n/2$. an $n/2$ plus $n/2$ problem. Let $T(n)$ be the running time of FindMedian on two sets each one of size n . We have shown that $T(n) = T(n/2) + 1$, where the 1 is the comparison that determines the relative order of Xmedian and Ymedian. The solution of this recurrence (cf master method) is $T(n) = O(\lg n)$. The algorithm is a "binary-search"-like method in obtaining the median of two sorted sequences. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 9.

Given a black box worst-case linear time median function, give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Proof. Let's call the black box function $B(X, n)$ that if given a set X of size n returns the median of X . Let the arbitrary statistic we are interested in finding be k . The idea behind the solution is simple. We are going to use a binary search motivated procedure to find the k statistic. Binary search searches the middle element of a sorted set (which happens to be the median). As X in our case is not sorted we call $B(X, n)$ to find the median. If $k = m$, where $m = \lfloor (n + 1)/2 \rfloor$ then we are done as the k statistic is the median which is returned by $B(X, n)$. Otherwise we split X into two sets X_1 and X_2 around the median of X and decide which one contains the k statistic (depending on whether k is less than or greater than m). We repeat recursively the same method on either X_1 or X_2 . In case X_2 is searched, we are then interested in finding not the k statistic of X_2 but the $k - m$ statistic which is the k statistic of X .

```
FindStatWithBinSearch(X,k,n)
// find k statistic in X of size n
1. begin
2.   if k = floor((n+1)/2) then return B(X,n)
3.   else
4.     if k < floor((n+1)/2) then {
5.       temp=B(X,n)
6.       collect in X1 all keys of X less than temp
7.       let m=floor((n+1)/2)
8.       FindStatWithBinSearch(X1,k,m-1)
9.     }else{
10.      temp=B(X,n)
11.      collect in X2 all keys of X greater than temp
12.      let m=floor((n+1)/2)
13.      FindStatWithBinSearch(X2,k-m,n-m)
14.    }
15. end
```

Let $T(n)$ be the time for finding the k statistic of n keys using the previously defined procedure. Sets X_1 and X_2 are of about the same size and each one is of size at most $n/2$.

Problem Set Continued

Step 2 takes $O(n)$ worst case time (median finding, by assumption on the properties of the black box) and so do Steps 5 and 11. Step 6 or 11 takes time $O(n)$ and 7 or 12 $O(1)$. Step 8 or 13 takes time $T(n/2)$.

Note that either step 1 is executed, or steps 5-8, or steps 10-13. Therefore the worst case running time $T(n)$ is given by the following recurrence.

$$T(n) = T(n/2) + O(n)$$

The solution to this recurrence is $T(n) = O(n)$ as required (solution by any of the three known methods). ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 10.

The input is an array A of n keys in no particular order. We are interested in determining whether there exists a key x in A that appears more than $n/3$ times in A . Give a worst-case linear-time algorithm that determines whether such a key exists or not. If such a key exists, you return the key otherwise you return "NO KEY". Give the algorithm in pseudocode, explain why it works, and analyze (correctly) its worst-case running time.

```

FindKey(A,n,n/3)
1. a2= Select(A,n,ceiling(n/3));
2. a3= Select(A,n,2ceiling(n/3));
3. f2=Find-Frequency (a2,A,n);
4. f3=Find-Frequency (a3,A,n);
5. if (f2 > n/3) return(a2);
6. if (f3 > n/3) return(a3);
7. return(NO-KEY);

```

```

Find-Frequency(key,A,n)
1.int count=0;
2.for(i=1;i<=n;i++)
3. if (A[i]== key) count++;
4 return(count);

```

Example.

n=8	Indices :	1	2	3	4	5	6	7	8	n/3 = 2
3keys	:	x	x	x	x					Case 1
			x	x	x					Case 2
				x	x	x				Case 3
					x	x	x			Case 4
						x	x	x		Case 5
							x	x	x	Case 6

ceil(n/3) times replicated

Problem Set Continued

Proof. We provide some intuition about the solution. If you have difficult understanding the fifth smallest of $\langle 1, 1, 1, 1, 1, 1 \rangle$, read the Subject. Consider the keys in A in sorted order; we are not going to sort but say someone did it for us for free! If a key in A appears more than $n/3$ times, then in the sorted sequence of these keys a block of $n/3$ such consecutive values or more will span the $\lceil n/3 \rceil$ and $2\lceil n/3 \rceil$ positions (we assume a Pascal indexing). Call the keys in those index positions a_2 and a_3 respectively. Thus if a key appears more than $n/3$ times it will be a_2 or a_3 or possibly both. Our first action is thus to determine the $\lceil n/3 \rceil$ -th and $2\lceil n/3 \rceil$ -th order statistic of the (unordered) A i.e. determine a_2, a_3 . These two keys that are found in lines 1 and 2 of FindKey. It suffices to find these two keys, there is not need to sort all of A to find those keys! So after computing these two keys, in lines 3 and 4 we determine how often these two keys appear in the unsorted A by scanning the input sequence left-to-right and determining the frequency of a_2 and a_3 respectively. If one of them appears more than $n/3$ times such a value of a_2 or a_3 is returned through lines 5 and 6. If neither a_2 nor a_3 appears more than $n/3$ times, this means there is no key that appears that often. Line 7 returns NO-KEY then. The running time of lines 1 and 2 is $O(n)$ if we ran the worst-case-linear-time selection algorithm, the running time of lines 3 and 4 is obviously $O(n)$ (observe the code of Find-Frequency) and lines 5,6,7 take $O(1)$ time. Overall $O(n)$ is the asymptotic worst-case running time of FindKey. We give an example with $n = 8$ and thus $n/3 \approx 2.6$ (i.e. determining whether a key exists more than twice, i.e. 3 times or more). If such a key exists, it is either the $n/3$ -rd order statistic (i.e. a_2) or the $2n/3$ -rd order statistic (i.e. a_3). ■

Problem Set

Problem 11.

Give an $O(n)$ worst case running time algorithm that, given a set S of n distinct numbers and a positive integer k , $k \leq n$, determines the k numbers in S that are closest to the median of S .

Proof. We first find the median of the n number stored originally in A . This takes $O(n)$ worst-case running time and is performed in line 1. So far we have found the median m . Next we create a new array that stores the n numbers in its value field. Attached to each such value is a key field. The key field indicates the distance between $A[i]$ and the median. It does not matter whether $A[i]$ is less or greater than m , and thus we take absolute values to determine the distance. This creation of a new array B takes place in lines 2-4. In line 3, the $A[i]$ values are copied and the distance to the median m is computed in line 4. Total running time for lines 2-5 is also $O(n)$.

In line 6, we select the k -th smallest element of B , i.e. find the k -th statistic of B by using as keys the distance/key values previously created NOT the original numbers. This has the effect of finding the k -th number closest to m . We record its distance to m in s . This takes $O(n)$ time.

In lines 7-9 we use s to find the k numbers that are close to the median m by checking in turn which of the n numbers (value fields) are at distance s or closer to m . This also takes $O(n)$ time. The overall time complexity is also $O(n)$.

Note. The algorithm above assumes that all keys and also distances are distinct. If they are not then we can modify line 8 to output only the first k matches. We just need to add a counter in line 8. As soon as k results are output the loop is terminated.

Problem Set Continued

```
Closest(A[1..n],k)
1. m= Select(A[1..n],n,floor((n+1)/2))    //Find the median m of A[1..n]
2. for(i=1;i<=n;i++) {                  //Find the distance of A[i] from m
3.   B[i].value = A[i];                  //(distance is absolute value of
difference
)
4.   B[i].key   = |A[i]-m|; // |x| is the absolute value of x.
5. }
6. s= Select(B[1..n].key,n,k,);
   // select the k-th statistic use .key values for the selection

7. for(i=1;i<=n;i++) {                  // Find the k keys with smallest distance from median m
8.   if B[i].key <= s                    // i.e. the keys with distance at most s.
       print(B[i].value);              // The values of these k keys is the ANSWER.
9 }


```

■

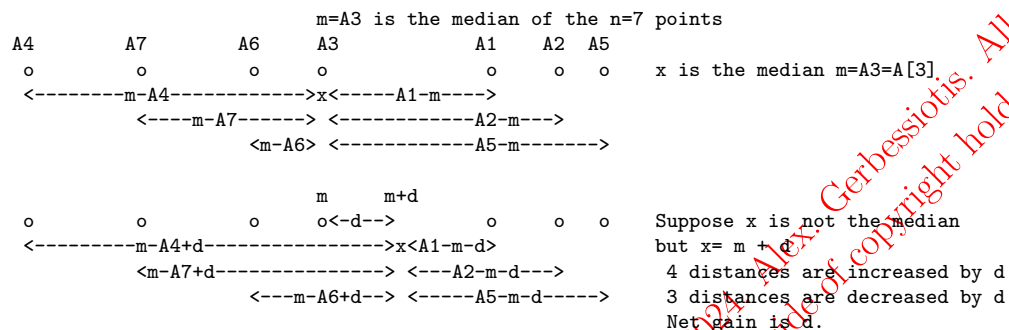
Problem Set Problem 12.

You are given n numbers in the form of array $A[0..n-1]$. Each number can be an integer in an arbitrary range or similarly, a real; no generic assumptions thus can be made about the range and type of these numbers. You are asked to give a linear-time algorithm that finds the number x that minimizes $\sum_{i=0}^{n-1} |A[i] - x|$ where $|y|$ is the absolute value of y . The output x can be one of the input values, or can be computed from a subset of the input values. Justify the correctness of your algorithm and analyze its worst-case running time to verify the linear-time claim.

Proof. Case 1. $n = 2k + 1$.

“Order” the numbers. Let x be the median m of the n numbers. We claim that $D(x) = \sum_{i=0}^{n-1} |A[i] - x|$ is then minimized for $x = m$, i.e. $D(m)$ is minimal. Why?

$n=2k+1$



Suppose that x is not the median but a value $x = m + d$ to the right of the median by a displacement d . Then the distance $|A[i] - x|$ of (at least) $k + 1$ points (including the median) has been increased by d and the distance of (at most) k points has been decreased by d . The end result is that the sum $\sum_{i=0}^{n-1} |A[i] - x|$ has been increased by at least d , i.e. $D(m + d) \geq D(m) + d$. If x is on the left of the median m we have a similar situation and an increase by at least d of the sum (i.e. $D(m - d) \geq D(m) + d$). Therefore the median m is the optimal x because it gives the lowest value for the sum.

We do not need to sort the numbers to find the median, i.e. x . We just use the linear-time Select algorithm. This takes $\Theta(n)$.

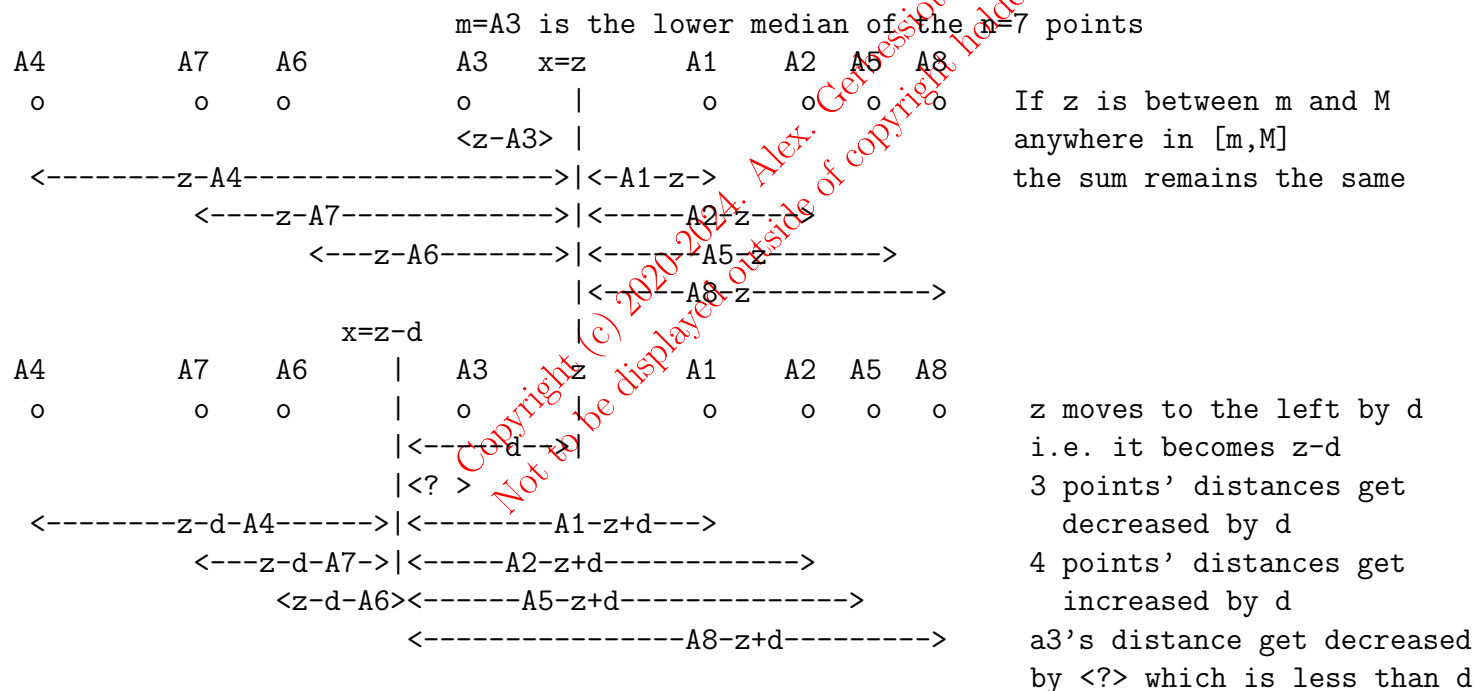
Problem Set Continued

Case 2. $n = 2k$. In this case we have two medians, the lower m and the upper M median. Let us choose $x = z$ to be between m and M i.e. $m \leq z \leq M$. Let the sum for this $x = z$ be $D = D(z)$. If we move x by d to the left (i.e. $x = z - d$) but still in that interval or by d to the right (i.e. $x = z + d$) but still in that interval k points' displacement is increased by d and k points' displacement is reduced by d . So the net-gain/loss is 0 and the sum remains equal to D , i.e. $D(z + d) = D(z - d) = D = D(z)$ as long as $m \leq z, z + d, z - d \leq M$. The d is an arbitrarily chosen value.

However if x is moved to the left of m or to the right of M then the sum gets increased. For example if x moves to the left by d and falls on the left of m , for $k - 1$ points or less the displacement gets smaller by d but for k points or more the displacement gets bigger by d and for m the displacement gets decreased by less than d because x moves from the right to the left of x . This is shown in the figure.

Therefore for Case 2, the optimal x is any value z such that $m \leq z \leq M$. We could report $x = m$ just as we did in Case 1, or use the upper median or any other value in-between.

$n=2k$ median $m=a3$



Problem 13.

Select with groups of 13 or may be 3. In worst-case linear time selection algorithm $\text{Select}(A, n, i)$ as described in Subject 7 and also a problem in the textbook, we used in class groups of 5 keys. If instead we used groups of 13 what changes in the derivation for the running time of the algorithm? Explain and justify your answer.

Oops typo.. What if we picked group of 3 instead?

And by now you know 2, 4, 6, 8, 10, etc are not suitable numbers. They are ... 'odd' numbers, even if they are even, i.e. not odd!

Proof. Won't be able to prove a suitable upper bound for size of L or R . ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 14.

TRUE or FALSE?

1. By reducing the problem of sorting n keys with comparisons and exchanges to the problem of building a heap on n elements, one can prove that the time it takes to build a heap on n elements is $\Omega(n \lg n)$ in the worst case.
2. The $\lg^2 n$ -th smallest key of an unsorted sequence of n keys can be determined in $o(n \lg n)$ worst case time.
3. For any $k \leq n$, we can find the k smallest keys and have them output in sorted order in time $\Theta(n + k \lg k)$.

Proof. 1. FALSE! What the heck is this?

2. TRUE.

3. TRUE.



Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page