COMP 3711 – Design and Analysis of Algorithms
2019 Spring Semester – Written Assignment # 2
Distributed: March 6, 2019 – Due: March 15, 2019
Corrected version of March 8, 2019, 12:00

Your solutions should contain (i) your name, (ii) your student ID #, and (iii) your email address
Notes:

- Please write clearly and briefly.
- Please follow the guidelines on doing your own work and avoiding plagiarism given on the class home page.
  In particular ***don't forget to acknowledge individuals who assisted you, or sources where you found solutions.*** Failure to do so will be considered plagiarism.
- Also follow the submission guidelines on the class web page. Use white, unwatermarked paper, e.g., no student society stationary.
  - If handwritten, solutions should be single sided, start a new page for every problem, be single column and leave space between consecutive lines and more space between paragraphs.
  - If typed, try to use an equation editor, e.g., latex , the equation editor in MS-WORD or whatever the equivalent is in whatever typesetting system you are using
- This assignment is due by 13:00 (1PM) on Friday March 15, 2019 in BOTH hard AND soft copy formats. A hard copy should be deposited in one of the two COMP3711 assignment collection boxes outside of room 4210. A soft copy for our records in PDF format should also be submitted via the online CASS system. See the Assignment 1 page in Canvas for information on how to submit online.
- The default base for logarithms will be 2, i.e., $\log n$ will mean $\log_2 n$. If another base is intended, it will be explicitly stated, e.g., $\log_3 n$.

Update The original released version of this assignment had incorrect formulas in the definitions of the median and quartile problems in problem 1. This version has corrected those errors.

Update The original version asked for a more complicated function (with four input parameters) as a solution to Question 4. This version simplifies the requirements on the problem and only has two parameters. We will also accept a solution to the original version of the problem.

Problem 1 [25 pts] Recall the *Selection* problem. *Given an array A of n unsorted values and an integer $k \leq n$ return the k-smallest item in A.*

Algorithmically, suppose you are given an array$A[1 \ldots n]$, $p < r$ and k. A procedure $Select(A, p, r, k)$ should find and report the k-smallest item in $A[p..r]$

In class you learned a simple $O(n)$ *randomized* algorithm for solving Selection. There also exists a (much more complicated) $O(n)$ worst-case time Selection algorithm.

A special case of the Selection problem is the *Median-Problem* which finds the middle ordered item in the subarray. Formally, $MEDIAN(A, p, r)$ should return $Select\left(A, p, r, \lceil \frac{r-p+1}{2} \rceil\right)$.

A special case of the Selection problem is the *Quartile-Problem* which finds the $\frac{1}{4}$ ordered item in the subarray. Formally, $Quart(A, p, r)$ should return $Select\left(A, p, r, \lceil \frac{r-p+1}{4} \rceil\right)$.

In this problem assume you are given a linear time procedure $Quart(A, p, r)$, i.e., it solves the Quartile problem in $O(r - p + 1)$ time.

Also assume that you are also given the linear time code to implement the Partition procedure taught in class and can call it as a subroutine in your algorithm.

1. **Show how, using $Quart()$ as a subroutine, you can solve the median problem in linear, i.e., $O(r-p+1)$, time.**

   Justify the running time of your algorithm.

   Further restriction: Your solution must be performed in-place (no new array can be created). In particular, one alternative solution would be to extend the array to size m so that all the entries in $A[n+1,\ldots,m] = \infty$ and $m > 2n$. After this, you would be able to solve the median problem in the original array with just one call to the quartile problem in the extended array. With the further restriction, this is not possible.

   Marking Note: The problem asked you to return the VALUE of the quartile or Median element. If your code returned the LOCATION instead, that was fine as well.

   Solution: It is actually easier to solve the full Selection problem than just the Median one. After writing the code for $Select(A,p,r,k)$, we just call it with $Select\left(A,p,r,\lceil\frac{r-p+1}{2}\rceil\right)$.

   Intuitively, the algorithm presented below is exactly the "randomized" selection algorithm taught in class but with the choice of random pivot replaced by the Quartile item. Since the Quartile item is a "good" pivot this would be equivalent to the randomized algorithm always choosing a good pivot and having each "stage" in its analysis of size 1. The discussion below explicitly fleshes out this idea.

We will assume that $1 \leq k \leq r-p+1$. *This implies that in* $Select(A, p, p, k)$, $k = 1$, *and thus* $Select(A, p, p, k) = A[p]$.

Select$(\mathbf{A}, \mathbf{p}, \mathbf{r}, \mathbf{k})$

1. If $p = r$ *then*
2. *Return*$(A[p])$
3. Else
4. *Set* $q = Quart(A, p, r)$
5. *Partition* $A[p, r]$ *around* q
6. *Let* x *be index of* q *after the partition*
7. *If* $x = k$
8. *then Return*$(A[x])$
9. *Else If* $x < k$
10. *Return* $(Select(A, x + 1, r, k - x))$
11. *Else*
12. *Return* $(Select(A, p, x - 1, k))$

The correctness of this code follows from exactly the same analysis as the correctness of the randomized selection algorithm. The only difference here is that we partitioning around the quartile item and not an arbitrary one.

To analyze the running time notice that if $r - p + 1 = m \geq 4$ *then the algorithm does* $O(m)$ *work (in lines 4-5) and then either stops or calls itself recursively on a problem of size* $\max(r - x, x - 1) \leq \frac{3}{4}m$. *The running time* $T(m)$ *then satisfies*

$$\forall m \geq 4, \quad T(m) \leq T(3m/4) + O(m)$$

which can be solved using the expansion method or a Master Theorem to yield the desired $T(m) = O(m)$.

2. Now suppose that you are given a linear time algorithm for solving the median problem.

   Show how, using this as a subroutine, Quicksort can be modified to run in $O(n \log n)$ *worst-case* time.

   Explain why your modified version runs in $O(n \log n)$ time.

   You MAY assume that all items in the array have distinct values.

   Solution. In every step of quicksort replace
   1. $q = Partition(A, p, r)$

   by

   1'. $s = Select\left(A, p, r, \lfloor \frac{p+r}{2} \rfloor\right)$.
   2'. Swap s *and* $A[r]$ *% this might require linear scan finding its location*
   3'. $q = Partition(A, p, r)$

   This is equivalent to always having the pivot for the partition being the median item. Since lines 1',2',3' take, in total, linear time, the entire running time for the algorithm would be

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

   giving an $O(n \log n)$ *algorithm.*

3. (For this part you may assume that n and k are powers of 2) .
Again assume that you are given a linear time algorithm for solving the median problem.

Given an array of size n, and $k \leq n$ find an $O(n \log k)$ algorithm that reorders the items in A so they are partitioned into k parts with items in each part less than or equal to the items in the next part.

More formally, let $\Delta = n/k$. Then, for $i = 0, 1, \ldots k-2$, all items in

$$A[i\Delta, i\Delta + 1, \ldots, (i+1)\Delta - 1]$$

are less than or equal to all of the items in

$$A[(i+1)\Delta, (i+1)\Delta + 1, \ldots, (i+2)\Delta - 1].$$

As an example if $k = 4$ and the original array was

$$[1, 3, 5, 7, 9, 11, 13, 15, 2, 4, 6, 8, 10, 12, 16, 14]$$

one legal output would be

$$[1, 3, 2, 4, 7, 6, 5, 8, 12, 11, 10, 9, 13, 14, 16, 15]$$

Solution

1. **Reorder**$(\mathbf{A}, \mathbf{p}, \mathbf{r}, \mathbf{k})$
2. *If* $k > 1$ *then*
3. $\quad x = Median(A, p, r, m)$
4. $\quad$ *Swap* x *and* $A[r]$
5. $\quad q = Partition(A, p, r)$
6. $\quad Reorder(A, p, q, k/2)$
7. $\quad Reorder(A, q+1, r, k/2)$

The intuition is to partition $A[p..r]$ around the median and then Reorder each half of the array by splitting them into $k/2$ subarrays properly ordered. Since everything in the first half of the array is $\leq$ everything in the 2nd half, this can be proven inductively to work.

The termination step is when $k = 1$, i.e., no reordering is needed, and this requires only $O(1)$ time

If $k > 1$ lines $3-5$ use $O(n)$ time where $n = r - p + 1$ is the number of items in the array. This gives a recurrence of

$$T(n, k) = 2T\left(\frac{n}{2}, \frac{k}{2}\right) + n$$

with initial condition $T(n,1) = 1$ for all n. The expansion method yields

$$\begin{aligned}
T(n,k) &= 2T\left(\frac{n}{2},\frac{k}{2}\right)+n \\
&= 2\left(2T\left(\frac{n}{2^2},\frac{k}{2^2}\right)+\frac{n}{2}\right)+n \\
&= 2^2T\left(\frac{n}{2^2},\frac{k}{2^2}\right)+2n \\
&= 2^2\left(2T\left(\frac{n}{2^3},\frac{k}{2^3}\right)+\frac{n}{2^2}\right)+2n \\
&= 2^3T\left(\frac{n}{2^3},\frac{k}{2^3}\right)+3n \\
&\cdots \\
&= 2^iT\left(\frac{n}{2^i},\frac{k}{2^i}\right)+in
\end{aligned}$$

This expansion stops when $\frac{k}{2^i} = 1$, i.e., $i = \log_2 k$. Plugging in this value of i gives

$$T(n) = 2^{\log_2 k}T\left(\frac{n}{2^i},\frac{n}{2^k}\right)+n\log k = 2^{\log_2 k}1+n\log k = k+n\log_2 k = O(n\log k).$$

the same result could also have been derived using the recursion tree method.

Note that, technically, this algorithm only works properly if all items have unique values. If there are repeated values this can be fixed by slightly modifying the code but that was not necessary.

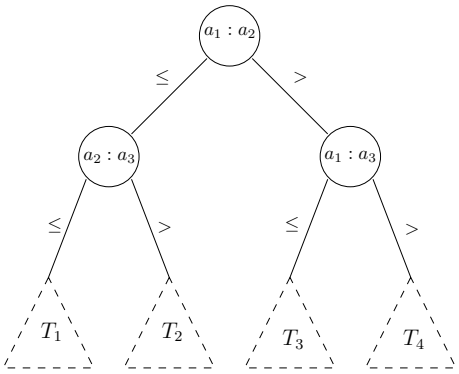
Marking note: A major issue here was ANALYZING the algorithm since the recurrence has two parameters and can not just be plugged into the master theorem.

Problem 2 [25 pts]

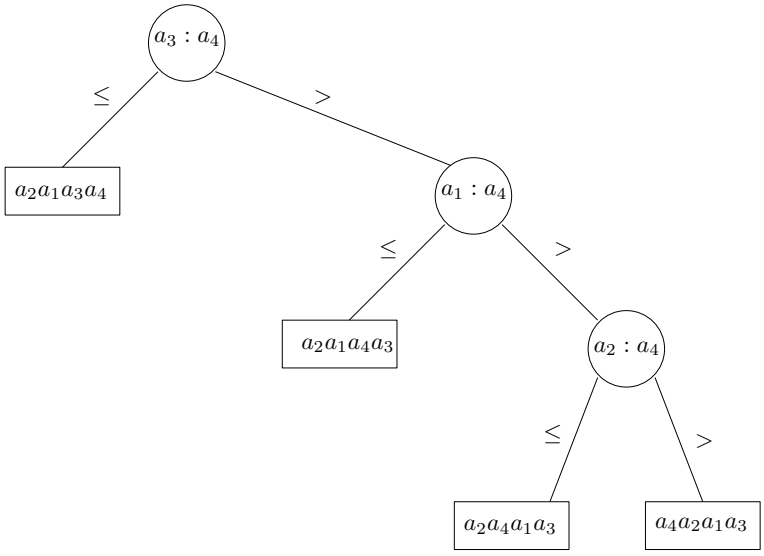
The figure below shows part of the decision tree for Insertion Sort (from page 12 of our introductory set of slides) operating on a list of 4 numbers, a_1, a_2, a_3, a_4.

Please expand subtree T_3, i.e., show all the internal (comparison) nodes and leaves in subtree T_3.

Note that the left branch of a comparison $(a : b)$ is the case $a \leq b$ and the right branch is $a > b$.



Solution



Problem 3: [25 pts]

Consider a long river, along which n houses are located. You can think of this river as an x-axis; the houses locations are given by their coordinates on this axis in a sorted order.

Your company wants to place cell phone base stations along the river, so that every house is within 8 kilometers of one of the base stations. Give an $O(n)$-time algorithm that minimizes the number of base stations used.

As well as giving the algorithm and showing that it runs in $O(n)$ time, you must prove that it yields an optimal solution.

Solution Let the house locations $x_1 < x_2 < \cdots < x_n$.

Let S *be the set of base stations chosen so far. Start with* $S = \emptyset$.

A house will be covered by S *if it is within 8km of some item in* S.

The algorithm will start with x_1 *and process the houses in order. If the current house* x_i *is already covered it will just go on to the next house. Otherwise it will add a base station at* $x_i + 8$ *and then go on to the next house.*

It will keep repeating this until all houses are covered.

The pseudocode is.

1. $i = 1$; $current = x_1 + 8$; $S = \{current\}$;
2. *For* $i = 2$ *to* n *Do*
3. *If* $x_i - current > 8$ *then*
4. $current = x_i + 8$; $S = S \cup \{current\}$;

Note that since this algorithm only does $O(1)$ *work for every* i, *it runs in* $O(n)$ *time.*

Marking Note:

Another possible solution would be to notice that the problem is equivalent to trying to cover the x_i with as few as possible intervals of length 16 (with the base stations at the centers of the intervals). This could then be solved using the interal covering algorithm from the tutorial.

First proof of correctness: *Let $G = g_1, g_2, \ldots, g_k$ be the solution returned by this greedy algorithm. Our goal is to show that G is optimal.*

Let $Y = y_1, y_2, \ldots, y_m$ be an optimal solution. (We assume the stations in each solution are sorted from smallest to largest).

If $G = Y$ then G is optimal. So, suppose that they are not the same. Consider the first base station where Y is different from X, i.e., the first r such that $g_r \neq y_r$. Note the following facts

- *The only way that r could NOT be well defined is if*

  $$g_1, g_2, \ldots, g_k = y_1, y_2, \ldots, y_k$$

  and either $k = m$ or $k < m$. The first is impossible because this would mean $G = Y$ which we know is not the case. The second is impossible because, by the definition of optimality, $m \leq k$.
- *By the definition of the greedy algorithm $g_r = x_t + 8$ where x_t is the leftmost point not covered by $g_1, \ldots, g_{r-1}$, Since Y must cover x_t as well we must have $y_r \leq x_t + 8 = g_r$. Since $y_r \neq g_r$ we must have $y_r < g_r$.*
- *From the above, $g_1, g_2 \ldots, g_r$ must cover all houses covered by $y_1, y_2, \ldots, y_{r-1}, y_r = g_1, g_2, \ldots, g_{r-1}, y_r$.*
- *Thus, if we replace y_r in Y with g_r to get*

  $$Y' = g_1, g_2, \ldots, g_{r-1}, g_r, y_{r+1}, \ldots, y_m$$

  we find that Y' also covers all of the houses. Since Y' also has length m, Y' is also optimal.

Repeatedly applying this transformation will convert Y into X. Thus X is also an optimal solution.

Second proof of correctness: *It is also possible to prove correctness of the greedy algorithm by induction as follows.*

Let S be the set of points, $G(S)$ the solution given by greedy and $OPT(S)$ some optimal solution.

The induction hypothesis is that the greedy algorithm will be optimal for all inputs of size n, i.e., $|G(S)| = |OPT(S)|$.

This is obviously true for $|S| = n = 1$ since then $|G(S)| = |OPT(S)| = 1$.

Now assume that the induction hypothesis is true for all values $< n$.

Let $|S| = n$ and consider the set of houses H covered by the first base station in the Greedy algorithm. Then

$$G(S) = 1 + G(S \setminus H).$$

Now let H' be the set of houses covered by the leftmost base station. Since this base station MUST cover x_1, it can not be to the right of the first greedy base station. Thus $H' \subseteq H$. Furthermore, the optimal solution must cover the remaining houses with an optimal number of base stations so

$$|OPT(S)| = 1 + |OPT(S \setminus H')|.$$

Thus

$$\begin{aligned} |G(S)| &= 1 + |G(S \setminus H)| \\ &= 1 + |OPT(S \setminus H)| \qquad \textit{(from induction hypothesis)} \\ &\leq 1 + |OPT(S \setminus H')| \\ &= |OPT(S)|. \end{aligned}$$

where the inequality comes from the fact that

$$(S \setminus H) \subseteq S \setminus H')$$

so any set of base stations that covers $S \setminus H'$ must cover $S \setminus H$ and, in particular,

$$|OPT(S \setminus H)| \leq |OPT(S \setminus H')|.$$

We have seen that $|G(S)| \leq |OPT(S)|$ and, by the definition of optimal, $|G(S)| \geq |OPT(S)|$ and thus $|G(S)| = |OPT(S)|$, i.e., the induction hypothesis is also correct for n.'

Note: This proof is similar to the proof of correctness of the interval covering problem from the tutorial.

Problem 4 [25 pts] **Updated March 8, 2019, 12:00.**

Suppose that X and Y are two sequences of equal size, sorted in increasing order. The first sequence is stored in sorted order in the array $X[1..n]$; the second sequence is stored in sorted order in the array $Y[1..n]$;

The goal of this problem is to find an efficient algorithm to find the median element in the combined set of $2n$ elements.

As an example, if

$$X = [1, 3, 7, 8, 10, 11],\ Y = [2, 4, 9, 13, 17, 19],$$

then the median item is $X[4] = 8$.

Note that this could easily be "solved" in $O(n)$ time if you merge the two sorted arrays into one sorted array and return its n'th element. You should find something that runs in time $O(\log n)$. (Hint: Use a variation of binary search).

More specifically, you should design a procedure that is called as $MED(X, Y)$ that returns the correct answer.

In writing your solution you may assume that n is known as a global constant (so it does not need to be passed to the function).

Your solution to this problem should be separated into three parts.

(a) First provide clearly documented pseudocode for the procedure.

(b) Next, explain what your algorithm does and why it is correct. Be explicit in proving the correctness for all cases of the algorithm.

(c) Finally, derive and state a tight upper bound on the running time of your algorithm.

Marking Note:

There are many algorithms available on the internet for finding the "median of two sorted arrays". Most of those assume that the two arrays are of *different* sizes. Much of the complexity of those algorithms derived from dealing with issues arising from the sizes being different. In this problem, the arrays are of the same size so the algorithm is much easier to describe.

Solution: We answer (b) before (a) so that the code is more understandable.

(b) Let M denote the median value being searched for and consider the structure of a solution. For simplicity, we assume that all items have distinct values (the analysis works even without this assumption).

First note that if $X[n] \leq Y[1]$ then $M = X[n]$.
If $Y[n] \leq X[1]$ then $M = Y[n]$
If $X[n] > Y[1]$ and $Y[n] > X[1]$, then $M \neq X[n]$ and $M \neq Y[n]$.

Our algorithm will check for $X[n] \leq Y[1]$ and $Y[n] \leq X[1]$ at the start so, in what follows, we will assume that they do not occur.

In what follows m', n' are integers.

We claim that, if $1 \leq m', n' \leq n$ then

$$(A) \quad m' + n' = n \quad AND \quad \max(X[m'], Y[n']) \leq \min(X[m'+1], Y[n'+1]).$$

if and only if

$$(B) \quad m' + n' = n \quad AND \quad M = \max(X[m'], Y[n']).$$

If (A), then, because both X and Y are sorted, (B) is true by definition.

If (B), then, since M is the median and the n items in $X[1 \ldots m'] \cup Y[1 \ldots n']$ are all $\leq M$, the remaining n items in $X[m'+1 \ldots n] \cup Y[n'+1 \ldots n]$ are all $\geq M$.

Next suppose that $\max(X[\bar{m}], Y[\bar{n}]) > \min(X[\bar{m}+1], Y[\bar{n}+1])$.

- *This can only occur if $X[\bar{m}] > Y[\bar{n}+1]$ or $Y[\bar{n}] > X[\bar{m}+1]$.*
- *if $X[\bar{m}] > Y[\bar{n}+1]$ then $X[\bar{m}] > Y[\bar{n}+1] \geq Y[\bar{n}]$*
- *if $Y[\bar{n}] > X[\bar{m}+1]$ then $Y[\bar{n}] > X[\bar{m}+1] \geq X[\bar{m}]$.*
- *Thus it is impossible for both $X[\bar{m}] > Y[\bar{n}+1]$ AND $Y[\bar{n}] > X[\bar{m}+1]$ to simultaneously occur.*

This immediately provides an algorithm. We binary search on $\bar{m}$ to find a pair $\bar{m} = m'$, $\bar{n} = n - \bar{m}$ that identifies the median.

1. *If $\max(X[\bar{m}], Y[\bar{n}]) \leq \min(X[\bar{m}+1], Y[\bar{n}+1])$ then $m' = \bar{m}$ so the median is $M = \max(X[\bar{m}], Y[\bar{n}])$*
2. *If $X[\bar{m}] > Y[\bar{n}+1]$ then all the $n+1$ items in $X[1 \ldots \bar{m}] \cup Y[1 \ldots \bar{n}+1]$ are $\leq X[\bar{m}]$ so $m' < \bar{m}$.*
3. *If $Y[\bar{n}] > X[\bar{m}+1]$ then all the $n+1$ items in $X[1 \ldots \bar{m}+1] \cup Y[1 \ldots \bar{n}]$ are $\leq Y[\bar{n}]$ so $n' < \bar{n}$ which implies $m' > \bar{m}$.*

(a) This implements the above

% First test for boundary case
1. If $X[n] \leq Y[1]$
2. *Return*$(X[n])$
3. Else If $Y[n] \leq X[1]$
4. *Return*$(Y[n])$

% Binary search for m'
5. Else
6. $m_1 = 1$; $m_2 = n - 1$;
7. *While* $m_1 < m_2$ *do*
8. *Set* $\bar{m} = \left\lfloor \frac{m_1+m_2}{2} \right\rfloor$; $\bar{n} = n - \bar{m}$.
9. *If* $\max(X[\bar{m}], Y[\bar{n}]) \leq \min(X[\bar{m}+1], Y[\bar{n}+1])$
10. *Return (*$\max(X[\bar{m}], Y[\bar{n}])$*)*
11. *Else if* $X[\bar{m}] > Y[\bar{n}+1]$
12. $m_2 = \bar{m} - 1$
13. *Else*
14. $m_1 = \bar{m} + 1$

% If solution has not been found yet then $\bar{m}, \bar{n}$ *is a good solution pair*
15. Return ($\max(X[\bar{m}], Y[\bar{n}])$*)*

Note that, from the discussion above, in lines 2, 4 and 10, the value the algorithm returns is the median. The algorithm always correctly maintains $m_1 \leq m' \leq m_2$*. So, if* $m_1 = m_2$ *(line 15) the algorithm correctly returns* $m' = m_1 = m_2$*.*

Finally, the algorithm is essentially binary searching on $\bar{m}$*, so it is* $O(\log n)$*.*