COMP 3711 – Honors Design and Analysis of Algorithms
2021 Spring Semester – Written Assignment # 2
Distributed March 4, 2021 – Due March 14, 2021

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

Some Notes:

- Please write clearly and briefly. Your solutions should follow the guidelines given at
  https://canvas.ust.hk/courses/36180/pages/assignment-submission-guidelines

  In particular, your solutions should be written or printed on *clean* white paper with no watermarks, i.e., student society paper is not allowed.

- Please also follow the guidelines on doing your own work and avoiding plagiarism as described on the class home page.
  You must acknowledge individuals who assisted you, or sources where you found solutions. Failure to do so will be considered plagiarism.

- The term *Documented Pseudocode* means that you must include clear comments *inside* your pseudocode.

- Many questions ask you to explain things, e.g., what an algorithm is doing, why it is correct, etc. To receive full points, the explanation must also be *understandable* as well as correct.

- Please make a *copy* of your assignment before submitting it. If we can't find your submission, we will ask you to resubmit the copy.

- Submit a SOFTCOPY of your assignment to CASS by the deadline. The softcopy should be one PDF file (no word or jpegs permitted, nor multiple files).

  If your submission is a scan of a handwritten solution, make sure that it is of high enough resolution to be easily read. At least 300dpi and possibly denser.

P1: [18 pts] **Recurrence Relations**

Give asymptotic upper bounds for $T(n)$ satisfying the following recurrences. Make your bounds as tight as possible. For example, if $T(n) = \Theta(n^2)$ then $T(n) = O(n^2)$ is a tight upper bound but $T(n) = O(n^2 \log n)$ is not. Your upper bound should be written in the form $T(n) = O(n^\alpha (\log n)^\beta)$ where α, β are appropriate constants.

A correct answer will gain full credits. It is not necessary to show your work BUT, if your answer was wrong, showing your work steps may gain you partial credits.

If showing your work, you may quote theorems shown in class.
For (a) and (b) you may assume that n is a power of 2;
for (c) and (d) you may assume that n is a power of 4;
for (e) and (f) you may assume that n is a power of 7.

(a) $T(1) = 1; T(n) = 8T(n/2) + n^2\sqrt{n}$ for $n > 1$.

(b) $T(1) = 1; T(n) = 5T(n/2) + n^3 \log_2 n$ for $n > 1$.

(c) $T(1) = 1; T(n) = 3T(n/4) + \sum_{i=1}^{n} i$ for $n > 1$.

(d) $T(1) = 1; T(n) = 5T(n/4) + 1$ for $n > 1$.

(e) $T(1) = 1; T(n) = 2T(n/7) + 2^{\log_7 n}$ for $n > 1$.

(f) $T(1) = 1; T(n) = 49T(n/7) + \log_3 (n!)$ for $n > 1$.

P2: [20 pts] **Monotone Matrices**

An $N \times N$ matrix is *monotone* if each of its rows and columns are sorted in nondecreasing order. For example,

$$A = \begin{pmatrix} 1 & 12 & 17 & 19 \\ 15 & 18 & 20 & 25 \\ 16 & 23 & 26 & 30 \\ 24 & 27 & 31 & 35 \end{pmatrix}$$

is monotone.

Given monotone matrix A and value a, the *membership problem* is to test whether $a \in A$ or $a \notin A$.

More specifically $MEM(a, m_1, n_1, m_2, n_2)$ will return `TRUE` if there exist i, j such that $a = A[i, j]$ where $m_1 \leq i \leq m_2$ and $n_1 \leq j \leq n_2$ and `FALSE` otherwise.

(Note that we are using standard matrix notation here. The m's are the rows and the n's are the columns. The top row is 1 and the bottom one N, while the left column is 1 and the right column is N. Thus $A[3, 2] = 23$.)

For example, in the given matrix, $MEM(31, 1, 1, 4, 4)$ == `TRUE` while $MEM(31, 1, 1, 3, 3)$ == `FALSE`.

The next page gives pseudocode for $MEM(a, m_1, n_1, m_2, n_2)$

(a) Prove that this algorithm (i) terminates and (ii) correctly solves the membership problem for monotone matrices.

(b) Show that without monotonicity, the algorithm might not be correct. To do this, give an example of a 5×5 non-monotone matrix A and value a such that $a \in A$ but $MEM(a, 1, 1, 5, 5)$ would return false.

(c) Analyze, from scratch (no use of other theorems allowed) the running time of the algorithm when run on an $N \times N$ monotone matrix, i.e., when $MEM(a, 1, 1, N, N)$ is called.

You may assume that N is always of the form $N = 2^k + 1$, k a non-negative integer, if that makes the analysis easier. The running time should measure the worst case number of comparisons made and be given using $O()$ notation.

Your running time should be as precise as possible, e.g, if the running time is actually $\Theta(N^2)$ then $O(N^2 \log N)$ would not be tight enough but $O(N^2)$ would be.

MEM$(\mathbf{a}, \mathbf{m_1}, \mathbf{n_1}, \mathbf{m_2}, \mathbf{n_2})$:

If $((m_1 > m_2)$ OR $(n_1 > n_2))$ % Check Validity of indices
    Return(`FALSE`)

Else if $((m_1 = m_2)$ AND $(n_1 = n_2))$ % Check if only one element
    { If $(a = A[m_1, n_1])$ Return(`TRUE`)
      Else Return(`FALSE`)
    }

Else if $((m_1 + 1 = m_2)$ AND $(n_1 = n_2))$ % Check if 2×1 submatrix
    { If $((a = A[m_1, n_1])$ OR $(a = A[m_2, n_1]))$
        Return(`TRUE`)
      Else Return(`FALSE`)
    }

Else if $((m_1 = m_2)$ AND $(n_1 + 1 = n_2))$ % Check if 1×2 submatrix
    { If $((a = A[m_1, n_1])$ OR $(a = A[m_1, n_2]))$
        Return(`TRUE`)
      Else Return(`FALSE`)
    }

Else if $((m_1 + 1 = m_2)$ AND $(n_1 + 1 = n_2))$ % Check if 2×2 submatrix
    { If $((a = A[m_1, n_1])$ OR $(a = A[m_1, n_2])$ OR $(a = A[m_2, n_1])$ OR $(a = A[m_2, n_2]))$
        Return(`TRUE`)
      Else
        Return(`FALSE`)
    }

Else { $u := \lfloor \frac{m_1+m_2}{2} \rfloor$; $v := \lfloor \frac{n_1+n_2}{2} \rfloor$; % Recurse on smaller matrices
    If $(Mem(a, u, n_1, m_2, v) =$ `TRUE`$)$ % lower-left submatrix
      Return(`TRUE`)
    If $(Mem(a, m_1, v, u, n_2) =$ `TRUE`$)\}$ % upper-right submatrix
      Return(`TRUE`)
    If $(a \leq A[u, v])$ % upper-left submatrix
      If $(Mem(a, m_1, n_1, u, v) =$ `TRUE`$)$
        Return(`TRUE`)
    If $(a > A[u, v])$ % lower-right submatrix
      If $(Mem(a, u, v, m_2, n_2) =$ `TRUE`$)$
        Return(`TRUE`)
    Return(`FALSE`)
    }

P3: [22 pts] **Selection**

Recall the *Selection* problem. *Given an array $A[1 \ldots n]$ of n unsorted values and an integer $k \leq n$, return the (index of the) k^{th} smallest item in A.*

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

For this problem, assume that you are given a black-box procedure for implementing this worst-case $O(n)$ selection algorithm. That is, given an array$A[\,]$, $p < r$ and k, $BB(A, p, r, k)$ finds and reports the index of the k^{th} smallest item in $A[p \ldots r]$ in $O(r - p + 1)$ time.

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

(i) Provide documented pseudocode for your $O(n \log n)$ *worst-case* time Quicksort.

(ii) Explain why the psuedocode runs in $O(n \log n)$ *worst-case* time.

Your new Quicksort pseudocode can call the black-box $BB(A, p, r, k)$ procedure as often as it wants.

(b) **Find a good algorithm for reporting the k middle items in the array in sorted order.**

The middle k items are item $\lfloor \frac{n-k}{2} \rfloor + 1$ to item $\lfloor \frac{n+k}{2} \rfloor$.

The input is a non-sorted array. As an example, let the input be the array

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

which is a permutation of the first 15 integers. If $k = 7$, the output should be the items $5, 6, 7, 8, 9, 10, 11$ in sorted order. If $k = 8$, the output should be the items $4, 5, 6, 7, 8, 9, 10, 11$ in sorted order.

The "obvious" way to solve this problem is to first sort the items and then report the values of the k middle ones. This requires $\Theta(n \log n + k) = \Theta(n \log n)$ time.

Use the black-box selection procedure to design an algorithm that is never worse (and sometimes better) than $\Theta(n \log n)$. In particular, for all $k \leq \frac{n}{\log_2 n}$, your algorithm should run in $\Theta(n)$ time. **Algorithms that do not satisfy this last condition will be considered wrong.**

(i) Provide documented pseudocode for your algorithm.

(ii) State the running time of your new procedure as tightly as possible in terms of *both* n and k. Explain why the pseudocode runs in that time.

(c) **Show how to use a solution to the *Tierce problem* to solve the Selection problem.**

The *Tierce problem* is to find the index of the $\frac{1}{3}$'rd item in the array, i.e., the index that would be returned by $BB\left(A, p, r, \left\lfloor \frac{r-p+1}{3} \right\rfloor\right)$.

Suppose that you lost the code for $BB(A, p, r, k)$ but were given a new black box for solving $Tierce(A, p, r)$ in $O(r - p + 1)$ worst case time. Show how, using $Tierce(A, p, r)$ as a subroutine, you can write a new linear time Selection algorithm.

(i) Provide documented pseudocode for your algorithm.

(ii) Explain why your algorithm is correct

(iii) Prove that it runs in $O(r - p + 1)$ worst case time.

Rules, Recommendations and Hints:

- Similar to the randomized selection algorithm taught in class, $BB(A, p, r, k)$ may move items in the array. More specifically, it may permute the items in $A[p \ldots r]$ (but does not move any of the other items). It returns the index of the k^{th} item in $A[p \ldots r]$ **at the time of return**.
- You may assume that you are given the $O(n)$ time $Partition(A, p, r)$ procedure and the $O(n \log n)$ time $Mergesort(A, p, r)$ (and associated $Merge(A, p, q, r)$) procedures taught in class and that your various routines can call them. You may not use any other procedures from class or anywhere else without writing the full code.
- Similar to our Quicksort implementation, your new Quicksort algorithm may move items around in A but is not allowed to add or delete items in the array.
- If you prove running times by first deriving recurrences, you may then use the Master theorem to derive the running time implied by that recurrence.
- For all problems, you may assume that all values in A are distinct.
- For (b). Note that because of the design of the problem, $k \leq n$. Thus, as an example, a running time of $\Theta(n \log n + k)$ could be written as $\Theta(n \log n)$. On the other hand, a running time of $\Theta(n + k^2)$ could not be simplified any further.
- For (c). Similar to the Quicksort and Selection algorithms taught in class, your procedure for *Tierce* can only use $O(1)$ extra space (per call). In particular, it may not create a new array or extend the size of A.

P4: [20 pts] **Sorting**

(a) (Radix Sort)
You are given a set of 10 decimal integers in the range of 1 to 65535:

$A = [23825, 29914, 14359, 39515, 62095, 49896, 63678, 37642, 40263, 52107]$.

(i) Please conduct Radix Sort on A using Base 10. Illustrate your result after each step following the worked example on Page 35 in the 08_Linearsort lecture slides.

(ii) Now convert these decimal integers to hexadecimal and conduct Radix sort again, this time using Base 16. Illustrate your result after each step following the worked example on Page 35 in the 08_Linearsort lecture slides.
Note: a digit in hexadecimal is in the range of
$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]$.
To start you off, note that 23825 *is* $5D11$ *in hexadecimal.*
There are many Decimal to Hex converters available online that you can use to translate A into Hex.

(b) (Heapsort)
Apply Heapsort on an initial input array $A = [84, 22, 19, 57]$. Illustrate the heap after each step following the worked example in 07a_Example_Heapsort slides.

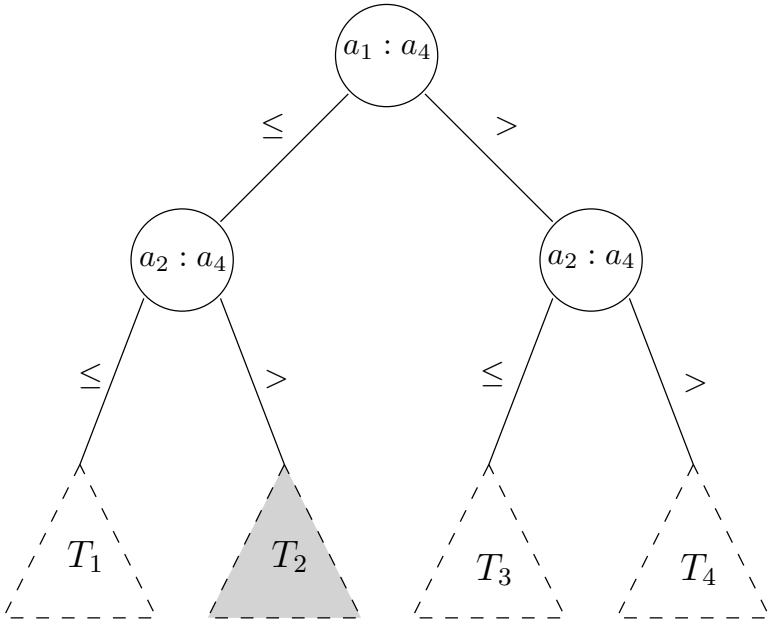
P5: [10 pts] **Decision Trees**

Recall that a comparison-based sorting algorithm can be represented in the (binary) decision tree model.

Following the worked example of Tutorial SS13, expand the subtree T_2 of the decision tree for sorting a list of four items a_1, a_2, a_3, a_4 using Quicksort with the last item as pivot.

Note: You should use the "Quicksort" and "Partition" code on pages 5-6 of 06_Quicksort as your definition of Quicksort. Edges in the tree must be labelled with $\leq$ or $>$ and leaves must show the final sorted order.

Be aware that the subtree might be large and/or deep. Make sure that you leave enough space so that all the internal and leaf nodes are clearly readable.



P6: [10 pts] **Randomization**

Let $A[1..n]$ be an array of n distinct numbers.
In class we said that if $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A. In Tutorial R5 we used indicator random variables to show that, if the elements of A form a uniform random permutation of $\langle 1, 2, ..., n\rangle$, then the expected number of inversions in A is $\frac{n(n-1)}{4}$.

For this problem let $n \geq 2$ be an even number and assume that $A[1 \ldots n]$ is initialized with $A[i] = i$.

- (a) Now run the following random procedure on A:
  For every i, $1 \leq i \leq n/2$, swap the values in $A[i]$ and $A[\frac{n}{2} + i]$ with probability $\frac{1}{2}$ (and leave them unswapped with probability $\frac{1}{2}$.)

  Answer the following question:
  what is the expected number of inversions in A after running this swapping procedure?

  Show how you mathematically derived this answer.

- (b) Now assume again that $A[1 \ldots n]$ is initialized with $A[i] = i$ and run the following different random procedure on A:
  For every i, $1 \leq i \leq n/2$, swap the values in $A[i]$ and $A[n - i + 1]$ with probability $\frac{1}{2}$ (and leave them unswapped with probability $\frac{1}{2}$.)

  Answer the following question:
  what is the expected number of inversions in A after running this different swapping procedure?

  Show how you mathematically derived this answer.

Rules, Recommendations and Hints:

- You are expected to use Indicator Random Variables and Linearity of Expectation to solve this problem.
- *Hint: Consider all pairs (i, j) with $i < j$ and calculate the probability that $A[i]$ and $A[j]$ form an inversion after the process is performed. Note that this probability will somewhat depend upon the locations of i and j.*

  Partition the set of pairs $I = \{(i, j) : 1 \leq i < j \leq n\}$ into subsets such that, for (i, j) in each subset, it is easy to calculate $\Pr(A[i] > A[j])$ after the swap.
- *To ensure that you understand the processes, consider $n = 8$. Then $[1, 6, 7, 4, 5, 2, 3, 8]$ is a valid swap resulting from the process in (a) but not (b). Similarly, $[8, 2, 3, 5, 4, 6, 7, 1]$ is a valid swap resulting from the process in (b) but not (a).*