#

HKUST – Department of Computer Science and Engineering

COMP3711: Design and Analysis of Algorithms – Fall 2022

Midterm

Date: October 20, 2022 Time: 19:30 -21:30 Venue: TBA

Name: ____________________	Student ID: ____________________
Email: ____________________	Lecture L1 / L2

Instructions

- This is a closed book exam. It consists of 5 questions on 26 pages.
- Please write your name, student ID and ITSC email at the top of this page.
- At the top of every subsequent page that you use, please write your student ID in the space provided.
- Please sign the honor code statement on page 2.
- Answer all the questions within the space provided on the examination paper. You may use the back of the pages for your rough work.
- Each question is on a separate page and most have at least one extra page for writing answers. These are provided for clarity and are not meant to imply that each question requires all of the blank pages. Many can be answered using much less space.
- If you write your answers on the back of a page, please indicate conspicuously what we should read. We ignore things written on the back of any page if there is no specific reference to a particular part of the back side that we should read.

Questions	1	2	3	4	5	Total
Points	16	24	15	15	30	100
Score						

Student ID: ____________

As part of HKUST's introduction of an honor code, the HKUST Senate has recommended that all students be asked to sign a brief declaration printed on examination answer books that their answers are their own work, and that they are aware of the regulations relating to academic integrity. Following this, please read and sign the declaration below.

I declare that the answers submitted for this examination are my own work.

I understand that sanctions will be imposed, if I am found to have violated the University regulations governing academic integrity.

Student's Name: ______________________________

Student's Signature: ________________________

1. **Asymptotic Analysis [11 pts]**

   (a) (10 points) We have two algorithms, A and B. Let $T_A(n)$ and $T_B(n)$ denote the time complexities of A and B respectively, with respect to the input size n. In the table shown below, there are four different cases of time complexities for each algorithm. Complete the last column of the following table with "A", "B", or "U", where:

   - "A" means that for large enough n, algorithm A is always faster;
   - "B" means that for large enough n, algorithm B is always faster;
   - "U" means that the information provided is not enough to justify stating that, for large enough n, one algorithm is always faster than the other;
   - We fill in the first line for you as an example.

Case	$T_A(n)$	$T_B(n)$	Faster
0	$\Theta(n)$	$\Theta\left(n^2\right)$	A
1	$\Theta(n^2/\log n)$	$\Theta\left(n^{1.9}\right)$	
2	$\Theta\left(10000^{\sqrt{n}}\right)$	$\Theta(2^n)$	
3	$O\left(\sqrt{\log n}\right)$	$\Omega(\log\log n)$	
4	$\Theta(5^n)$	$\Theta(7^n)$	
5	$\Theta(n^2)$	$T_B(n) = \begin{cases} 2T_B(n/2) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$	

   <u>Answer:</u> 1-B, 2-A, 3-U, 4-A, 5-B.

(b) (6 points) Derive the exact asymptotic solution for the following recurrence:

$$\begin{cases} \forall\, n > 1, \quad T(n) = 27 \cdot T\left(\frac{n}{3}\right) + n^2; \\ \\ T(1) = 1. \end{cases}$$

Assume that n is always an integral power of 3. Show your derivation from first principles. DO NOT use the Master Theorem for this part. Show all your steps. Write your final result in the form $T(n) = \Theta\left(n^\alpha (\log n)^\beta\right)$ for appropriate values of α and β.

Answer:

$$\begin{aligned} T(n) &= 27 \cdot T(n/3) + n^2 \\ &= 3^3 \cdot T(n/3) + n^2 \\ &= 3^6 \cdot T(n/3^2) + 3^3 \cdot n^2/3^2 + n^2 \\ &= 3^9 \cdot T(n/3^3) + 3^6 \cdot n^2/3^4 + 3^3 \cdot n^2/3^2 + n^2 \\ &= 3^{3k} \cdot T(n/3^k) + n^2(1 + 3 + 3^2 + \cdots + 3^{k-1}) \\ &= 3^{3k} \cdot T(n/3^k) + n^2 \cdot (3^k - 1)/2. \end{aligned}$$

We hit the boundary condition of $T(1) = 1$ when $k = \log_3 n$. In this case, $3^{3k} = n^3$ and $3^k - 1 = n - 1$. Therefore, $T(n) = n^3 + n^2 \cdot (n-1)/2 = \Theta(n^3)$.

Student ID: ________

2. **Sorting [24 pts]**

   (a) (12 points) An array $A[1..n]$ stores n **possibly non-distinct numbers**. A **range query** specifies two integers x and y that form an interval $(x, y]$. The answer to a range query is the number of elements in A that are greater than x and less than or equal to y. There is NO requirement that x and y are elements of A.

   As an example if $A = [4, 5, 2, 6, 8, 3, 4, 4]$, the range query for the interval $(3, 6]$ would return 5 because there are five elements in A that lie in $(3, 6]$, namely $A[1]$, $A[2]$, $A[4]$, $A[7]$, and $A[8]$. Similarly, the range query for the interval $(7, 10]$ would return 1.

   Describe an algorithm that generates a data structure C such that C uses $O(n)$ space and you can use C to answer any range query in $O(\log n)$ time. Explain the running time of your algorithm for constructing C. Describe **in detail** how you use C to answer a range query and explain why it takes $O(\log n)$ time.

Answer: We need to count the number of elements in A that are greater than x but at most y. We sort the elements of A in non-decreasing order in C. The sorting can be done by an optimal sorting algorithm in $O(n \log n)$ time. Clearly, the array C uses $\Theta(n)$ space as there are n elements in the array. We answer a range query $(x, y]$ as follows.

We use a variant of binary search Findnext$(C, 1, n, x)$ that returns the position of the smallest integer y in the subarray $C[1..n]$ that is larger than x. If there are multiple occurrences of y in $C[1..n]$, the position of the leftmost occurrence of y is reported. If x is the largest in $C[1..n]$, then $n + 1$ is reported.

```
Findnext($C, p, r, x$)
  **if** $p \geq r$ **then**
    **if** $p > r$ or $x < C[p]$ **then**
      **return** $p$
    **else**
      **return** $p + 1$
    **end if**
  **end if**
  $q \leftarrow \lfloor (p + r)/2 \rfloor$
  **if** $x \geq C[q]$ **then**
    Findnext($C, q + 1, r, x$)
  **else**
    Findnext($C, p, q - 1, x$)
  **end if**
```

A range query for $(x, y]$ can be answered by calling the following procedure:

```
Range($C, x, y$)
  $X \leftarrow$ Findnext($C, 1, n, x$)
  $Y \leftarrow$ Findnext($C, 1, n, y$)
  **return** $Y - X$
```

The running time is $O(\log n)$ because we are basically calling binary search twice on the array $C[1..n]$.

(b) (12 points) Consider an array A of n **possibly non-distinct numbers**. Suppose that the numbers in A come from the range $[1..k]$ for some positive integer k given to you. A **range-sum query** specifies two integers x and y in the range $[1, k]$ that form an interval $(x, y]$. The answer to a range-sum query is the sum of elements in A that are greater than x and less than or equal to y.

As an example if $k = 9$ and $A = [4, 5, 2, 6, 8, 3, 4, 4]$, the range-sum query for the interval $(3, 6]$ would return 23. The interval $(7, 10]$ does not define a range-sum query because 10 is outside the range $[1, 9]$.

Describe **in detail** how you would organize the data structure C and construct C so that each range-sum query can be answered in $O(1)$ time. Explain the running time of your algorithm for constructing C. Explain why a range-sum query can be answered in $O(1)$ time.

Answer: We need to sum the number of elements in A that are greater than x but at most y. We introduce an array $D[1..k]$. For every $i \in [1, k]$, $D[i]$ should store the number of occurrences of i in $A[1..n]$. This can be done as follows in $O(n + k)$ time. First, initialize $D[i] = 0$ for every $i \in [1, k]$. Then, we scan the elements $A[i]$ for $i = 1, 2, \cdots, n$, and for every $A[i]$ encountered, we increment the element $D[A[i]]$.

Next, we compute an array $C[1..k]$ such that for every $i \in [1, k]$, $C[i]$ stores the sum of elements in A that are at less than or equal to i. This can be done in $O(k)$ time inductively as follows. We first compute $C[1] := D[1]$. Then, for $i = 2, 3, \cdots, k$, we compute $C[i] := i \cdot D[i] + C[i - 1]$. Due to the inductive computation, we could have replaced the array $D[1..k]$ by $C[1..k]$ without affecting correctness.

Finally, given a range-sum query $(x, y]$, we return $C[y] - C[x]$ in $O(1)$ time.

Student ID: ________

3. **Heap [15 points]**

   Let Q be a first-in-first-out queue that stores w numbers initially. We define an operation update(Q, x) which removes the number at the beginning of Q, inserts x at the end of Q, and outputs the minimum number in the modified Q afterwards. The operation update(Q, x) is useful when the data arrives one by one in an online fashion, and we are only interested in the most recent w data items. **We assume that the numbers in Q are distinct at all times.**

   As an example, let $w = 3$ and let the numbers in Q be $(4, 3, 8)$ initially in this order. Suppose that new numbers 9, 2, 4 arrive in this order. Then, update($Q, 9$) outputs 3; update($Q, 2$) outputs 2; update($Q, 4$) outputs 2.

   A brute-force implementation of update(Q, x) finds the minimum number in the modified Q by scanning the entire Q once. This takes $\Theta(w)$ time. We are interested in a more efficient implementation using a min-heap H, organized as an array as taught in class.

   Describe **in detail** an algorithm that performs update(Q, x) in $O(\log w)$ time using a min-heap H. You can assume that a library function position(x) is given to you such that if x is a number in Q, then position(x) returns in $O(1)$ time the index i such that x is stored in $H[i]$ (the top of the heap is at $H[1]$). You can also use as black boxes the following operations:

   - The function dequeue(Q) deletes the first number in Q and returns it. It runs in $O(1)$ time.
   - The procedure enqueue(Q, x) adds x at the end of Q. It runs in $O(1)$ time.
   - The procedure extract-min(H) removes the minimum number in H and make the necessary modifications so that the resulting H remains a min-heap. This procedure does not output the number at the root of the heap. It runs in $O(\log m)$ time, where m denotes the size of H.
   - The procedure insert(H, x) inserts x into H and make the necessary modifications so that the resulting H remains a min-heap. It runs in $O(\log m)$ time, where m denotes the size of H.

   Explain why your algorithm runs in $O(\log w)$ time.

<u>Answer:</u> The operation update(Q, x) can be implemented as follows.

```
(1) set $y$ to be the first element of $Q$;
(2) $i :=$ position($y$);
(3) dequeue($Q$);
(4) $H[i] := -\infty$;
(5) While ($i > 1$) do { swap $A[i]$ and $A[\lfloor i/2 \rfloor]$; $i := \lfloor i/2 \rfloor$ };
(6) extract-min($H$);
(7) insert($H, x$);
(8) output $H[1]$.
```

Steps 1, 2, 3, and 7 take $O(1)$ time. Step 4 repeatedly swaps $-\infty$ up the tree until it reaches the root. Therefore, the running time of this step is bounded by the tree height which is $O(\log w)$. After step 4, H is a min-heap because $-\infty$ is smaller than all other numbers in H. Step 5 runs in $O(\log w)$. By the end of step 5, y has effectively been removed from H. Step 6 runs in $O(\log w)$. Since the minimum is stored at the root, it is stored in $H[1]$, so outputting $H[1]$ is correct.

Student ID: ________

4. **Divide and Conquer [15 pts]**

   Let $A[1..n]$ be an array of n distinct integers. We assume that n is an integral power of 2. You are given the following two procedures:

   - Select(A, p, r, k): It returns the kth smallest number in the subarray $A[p..r]$, provided that $k \in [1, r-p+1]$. The worst-case running time of a call Select(A, p, r, k) is $\Theta(r-p+1)$.
   - Split(A, p, r, x): It uses the number $x \in A[p..r]$ as the pivot to rearrange the numbers in A. Afterwards, there is an index $q \in [p, r]$ such that all numbers in $A[p..q]$ are less than or equal to x, and all numbers in $A[q+1..r]$ are bigger than x. The worst-case running time of a call Split(A, p, r, x) is $\Theta(r-p+1)$. Note that Split does not return anything; it only rearranges the elements in $A[p..r]$.

(a) (5 points) Explain how you can use the procedures Select and Split to rearrange the elements in $A[p..r]$ in $\Theta(r-p+1)$ time so that every number in the left half is smaller than every number in the right half. Assume that $r-p+1$ is even. The left half means the first $(r-p+1)/2$ numbers in the rearranged $A[p..r]$; the right half means the last $(r-p+1)/2$ numbers in the rearranged $A[p..r]$.

<u>Answer:</u> Compute $k = (r-p+1)/2$ and $x :=$ Select(A, p, r, k). By the setting of k, x is the $\left(\frac{r-p+1}{2}\right)$-th smallest number in $A[p..r]$. Therefore, there are exactly $(r-p+1)/2$ numbers in A that are smaller than or equal to x, and there are exactly $(r-p+1)/2$ numbers in A that are greater than x. Next, we call Split(A, p, r, x).

Student ID: ______

(b) (10 points) Let m be a power of 2 that is less than n. Consider a division of A into m parts $S_1, S_2, \ldots, S_m$ such that for $i \in [1, m]$, S_i contains n/m consecutive elements in A. A division of A into m parts is *legal* if and only if for all $i < j$, every number in S_i is less than every number in S_j.

Describe an algorithm that uses the procedures Select and Split to rearrange the elements in A and construct a legal division of the rearranged A. Your algorithm should output the elements in each part. Your algorithm should run in $O(n \log m)$ time. Explain the correctness of your algorithm and why it runs in $O(n \log m)$ time.

Answer: We use the algorithm in (a) to divide A into a legal division of two parts. This takes $\Theta(n)$ time by (a). By the legality of the division, every number in the left part is less than every number in the right part. Therefore, if we further divide the left and right parts into legal divisions of $m/2$ parts each, then we are done. This motivates the following recursive procedure Divide(A, p, r, s) which outputs a legal division of A into s parts. The recursion bottoms out when the parameter s is equal to 1. In this case, the call just output all elements in the input array and then return. The top-level call is Divide$(A, 1, n, m)$.

```
Divide($A, p, r, s$)
  **if** $s = 1$ **then**
    Output every number in $A[p..r]$
  **else**
    $k \leftarrow (r - p + 1)/2$
    $x \leftarrow$ Select$(A, p, r, k)$
    Split$(A, p, r, x)$
    Divide$(A, p, p + k - 1, s/2)$
    Divide$(A, p + k, r, s/2)$
  **end if**
```

Let $T(n,m)$ denote the worst-case running time of Divide computing a legal partition of a subarray of size n into m parts. We have the following recurrence:

$$T(n,m) \leq 2 \cdot T(n/2, m/2) + O(n).$$

The boundary condition is $T(n,1) = O(n)$. By repeated expansion, we get:

$$\begin{aligned}
T(n,m) &\leq 2 \cdot T(n/2, m/2) + O(n) \\
&\leq 4 \cdot T(n/4, m/4) + 2 \cdot O(n/2) + O(n) \\
&\leq 2^k \cdot T(n/2^k, m/2^k) + \\
&\quad\; 2^{k-1} \cdot O(n/2^{k-1}) + 2^{k-2} \cdot O(n/2^{k-2}) + \cdots + O(n) \\
&\leq 2^k \cdot T(n/2^k, m/2^k) + k \cdot O(n).
\end{aligned}$$

We hit the boundary condition when $k = \log_2 m$. In this case, we get $T(n,m) \leq m \cdot T(n/m, 1) + O(n \log m) = m \cdot O(n/m) + O(n \log m) = O(n \log m)$.

Student ID: ________

5. **Greedy Algorithm [30 pts]**

   You are given n tasks. Each task has a unique integer ID i from the range $[1, n]$. Task i has a profit $p[i]$, which is a positive real number, and a deadline $d[i]$, which is a positive integer from the range $[1, n]$. Note that two different tasks may have the same deadline. Each task can be completed within a day, and you choose a task to perform each day. If you can complete a task i on or before day $d[i]$, you collect profit $p[i]$.

   For example, suppose that you are given the following five tasks:

ID	profit	deadline
1	100	2
2	20	1
3	40	2
4	25	1
5	15	5

   An optimal schedule S with a total profit 155 can be obtained by scheduling task 3 on day 1, task 1 on day 2, and task 5 on day 5. Another optimal solution S', with the same profit, could assign task 1 on day 1, task 3 on day 2, and task 5 on day 3. Observe that tasks 2 and 4 are not performed in any optimal solution because they lead to a smaller profit.

   Describe a greedy algorithm that computes a schedule of tasks that maximizes the total profit.

   The input to your algorithm consists of the number n of tasks, an array $p[1..n]$ of profits, and an array $d[1..n]$ of deadlines for the n tasks.

   The output of your algorithm should be an array $S[1..n]$ that stores the schedule of tasks. Specifically, for every $j \in [1, n]$, if $S[j]$ belongs to the range $[1, n]$, it means that task $S[j]$ is performed on day j; if $S[j] = 0$, it means that no task is scheduled on day j. For example, the output array for the first optimal solution S in the previous example is $S = [3, 1, 0, 0, 5]$. You should also return the maximum total profit obtained by your schedule.

   Derive the running time of your algorithm. Prove the correctness of your algorithm.

Student ID: ________

Answer:

```
Greedy($n, p, d$)
  sort the task IDs in decreasing order of task profits $t_1, t_2, \ldots, t_n$
  profit $\leftarrow 0$
  **for** $i = 1$ to $n$ **do**
    $S[i] \leftarrow 0$
  **end for**
  **for** $i = 1$ to $n$ **do**
    $j \leftarrow d[t_i]$
    **while** $j > 0$ **do**
      **if** $S[j] = 0$ **then**
        $S[j] \leftarrow t_i$
        profit $\leftarrow$ profit $+ p[t_i]$
        break
      **else**
        $j \leftarrow j - 1$
      **end if**
    **end while**
  **end for**
  **return** $S$, profit
```

The running time is dominated by the running time of the nested loops. The while-loop iterates $O(n)$ times for each task. The total running time is thus $O(n^2)$. A worst-case example is that every task has the same deadline n. Then, the most profitable task will be scheduled on day n, the next on day $n-1$, and so on.

Let $S[1..n]$ be the output schedule by Greedy. Let $S'[1..n]$ be an optimal schedule. Assume that $S[k] = S'[k]$ for $k \in [j+1, n]$ and that $S[j] \neq S'[j]$. We make two observations.

First, $S[j] \neq 0$, i.e., Greedy must schedule a task on day j. Assume to the contrary that $S[j] = 0$. As $S'[j] \neq S[j]$, the optimal schedule puts a task on day j, say t_k. Therefore, $d[t_k] \geq j$, and t_k is not an element of $S[j+1..n]$ or $S'[j+1..n]$. When Greedy examines task t_k, it tries to schedule it on a day as close to its deadline $d[t_k]$ as possible; moreover, day j must be an available option because $S[j] = 0$ at the end. But then t_k must appear in $S[j..n]$, a contradiction.

Second, either $S'[j] = 0$ or $p[S'[j]] \leq p[S[j]]$. Assume to the contrary that $S'[j] \neq 0$ and $p[S'[j]] > p[S[j]]$. So Greedy examines task $S'[j]$ before task $S[j]$. When Greedy examines task $S'[j]$, the slot j is available because Greedy puts the later task $S[j]$ on day j. Since task $S'[j]$ can be scheduled on any day between day j and day $d[S'[j]]$, Greedy must have scheduled task $S'[j]$ on such a day in S. Task $S'[j]$ cannot belong to $S[j+1..n]$ which is equal to $S'[j+1..n]$ by assumption. But then Greedy must have scheduled task $S'[j]$ on day j, a contradiction.

We transform S' so that $S'[j..n] = S[j..n]$ afterwards. Then, repeating this transformation shows that the schedule produced by Greedy is optimal. By the first observation above, we have $S[j] = t_i$ for some i. There are two cases to consider in the transformation:

- Case 1: Task t_i is not scheduled in S'. We change $S'[j]$ to t_i. By the second observation above, changing $S'[j]$ to t_i cannot decrease the total profit of S'.
- Case 2: Task t_i is scheduled in S' on a day earlier than j. That is, $S'[a] = t_i$ for some $a < j$. As $S[j] = t_i$, we know that $d[t_i] \geq j$. Therefore, we can swap the contents of $S'[a]$ and $S'[j]$ without violating the deadline of any scheduled task in S'. (It is possible that $S'[j] = 0$, but the swapping can also proceed in this case.) The total profit resulting from S' is preserved. The swap ensures that $S[j..n] = S'[j..n]$.

In the previous example, the greedy solution is $S = [3, 1, 0, 0, 5]$ and $S' = [1, 3, 5, 0, 0]$ is an optimal solution. By swapping $S'[3]$ and $S'[5]$, we obtain a new schedule $S' = [1, 3, 0, 0, 5]$. Later, by swapping $S'[1]$ and $S'[2]$, we obtain $S' = [3, 1, 0, 0, 5]$.

Student ID: ________

Student ID: ________