Winter 2023 Analysis and Design of Algorithms: MidSem Solutions *Max: 60 Marks*
Feb 27, 2023 *120 minutes*

Part A

For each of the following, choose the correct answer.

1. **(7 Marks)** Given three algorithms:

   - **Algorithm A:** Partitions a problem of size n into 8 subproblems of size $n/3$ and combines them in time $O(n^2)$.
   - **Algorithm B:** Partitions a problem of size n into 4 subproblems of size $n/2$ and combines them in time $O(n^2 \log n)$.
   - **Algorithm C:** Partitions a problem of size n into 4 problems of size $n/4$ and combines them in time $O(n)$.

   The asymptotic running times of the algorithms in increasing order are:

   (a) A, B, C
   (b) A, C, B
   (c) B, A, C
   (d) B, C, A
   (e) C, A, B (**correct**)
   (f) C, B, A
   (g) All have the same asymptotic running time.

   Solution: For the Algorithm A, the recurrence is $T(n) = 8T(n/2) + O(n^2)$. It means that the algorithm A takes $O(n^3)$-time. For the algorithm B, the recurrence is $T(n) = 4T(n/2) + O(n^2 \log^2 n)$. It means that the algorithm B takes $O(n^2 \log^2 n)$. On the other hand, for algorithm C, the recurrence is $T(n) = 4T(n/4) + O(n)$. It means that the algorithm takes $O(n \log n)$-time. So, the running times at increasing order is C, A, B.

2. **(7 Marks)** Give an asymptotically tightest upper bound in Big-Oh notation for this recurrence relation $T(n) = 4T(\sqrt{n}) + (\log n)^2$. Assume $T(n)$ is constant for $n \leq 2$.

   (a) $O(\log(\log n)^2 \log n)$
   (b) $O((\log n)^2 \log \log n)$ (**Correct**)
   (c) $O((\log n)^2 \log n)$
   (d) $O(\log n \log \log n)$

   Solution: We use variable change method for this. Let $n = 2^k$. Hence, $\log n = k$. Then, this recurrence can be represented as $S(k) = 4S(k/2) + O(k^2)$. Solving this recurrence gives $O(k^2 \log k)$. Hence, $T(n)$ is $O((\log n)^2 \log \log n)$.

3. **(5 Marks)** Suppose that there are two strings $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. Let $L[i, j]$ denotes the length of a longest common subsequence between $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$. If $x_i = y_j$, then recurrence relation for the dynamic programmin to solve $L[i, j]$ is

   (a) $L[i, j] = \max\{L[i, j-1], L[i-1, j]\}$
   (b) $L[i, j] = 1 + \max\{L[i, j-1], L[i-1, j]\}$
   (c) $L[i, j] = 1 + L[i-1, j-1]$. (**Correct**)
   (d) None of the above.

4. **(6 Marks)** Let $LC(n)$ be the sequence of *lucas numbers* defined by $LC(0) = 2, LC(1) = 1$ and for all $n \geq 2$, $LC(n) = LC(n-1) + LC(n-2)$. What is the worst case running time of the fastest algorithm to compute $LC(n)$? Just right down the tightest possible asymptotic running time in Big-Oh notation.

   (a) $O(1.618^n)$.

   (b) $O(n \log n)$.

   (c) $O(n)$.

   (d) $O(n^2)$.

   Solution: Computing this can be built iteratively. Initialize $Lucas[0] = 2$ and $Lucas[1] = 1$. Then, for every $i = 2, \ldots, n$ in this order, $Lucas[i] = Lucas[i-1] + Lucas[i-2]$. Hence, this can be computed in $O(n)$-time.

Part B

1. **(10 Marks)** Given n objects, how many different sets of size k can be chosen? Write down the pseudocode of an algorithm. Give an explanation of the running time of the algorithm. ANY ALGO AND COMPATIBLE COMPLEXITY IS FINE. DP IS DESIRABLE; PSEUDOCODE/STEPS

   - **Number of different sets of size k:** (2 Marks) $\binom{n}{k}$.
   - **Pseudocode of the algorithm:** Here is a basic result from discrete mathematics (not for grading).

$$\text{For all } n > k \text{ and } k \geq 1, \text{ it holds that } \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

If $n = k$, then $\binom{n}{k} = 1$ and if $k = 0$, then $\binom{n}{k} = 1$.
The pseudocde works as follows. We describe both the recursive algorithm and iterative algorithm. Both carry equal marks. One student would write one of them. Then, give marks accordingly.

Recursive algorithm: (5 Marks)

Algorithm 1: Computing the number of subsets of size k

Data: Function $Count(n, k)$
if $n < k$ **then**
  return 'undefined';
if $k = n$ *or* $k = 0$ **then**
  return 1;
Return $Count(n-1, k-1) + Count(n-1, k)$;

Iterative Algorithm: (5 Marks)

Algorithm 2: Computing the number of subsets of size k

Data: Function $Count(n, k)$
if $n < k$ **then**
  return 'undefined';
if $n = k$ *or* $k = 0$ **then**
  Return 1;
for $i = 1, \ldots, n$ **do**
  $Count[i, i] = 1$; and $Count[i, 0] = 1$;
for $i = 2, \ldots, n$ **do**
  for $j = 1, \ldots, \min(i-1, k)$ **do**
    $Count[i, j] = Count[i-1, j-1] + Count[i-1, j]$;
Return $Count[n, k]$;

- Running time of the algorithm: (3 Marks)
  Running time of the recursive algorithm has the recurrence

  $$T(n,k) = T(n-1,k-1) + T(n-1,k)$$

  The iterative algorithm builds up table in the bottom-up fashion. Hence, the running time of the algorithm is $O(nk)$.
  (1 Mark for mentioning the correct running time and 2 Marks for justification)

2. (10 **Marks**) In this problem we consider a monotonously decreasing function $f : N \to Z$ (that is, a function defined on the natural numbers taking integer values, such that $f(i) > f(i+1)$). Assuming we can evaluate f at any i in constant time, we want to find $n = \min\{i \in N | f(i) \leq 0\}$ (that is, we want to find the value where f becomes negative). Write the pseudo-code of an $O(\log n)$-time algorithm and give a justification why your algorithm runs in $O(\log n)$-time. **The solution lies in [a,b] s.t. a,b \in N; there is a guarantee that the sol will be converged at**

   - **Pseudocode of the Algorithm:** (7 Marks).

Algorithm 3: Finding out n

Data: Input is $f : N \to Z$

Initialize $k = 0$ and $r_s = 0$;
while do
    $r_s \leftarrow r_s + 2^k$ and $r_m \leftarrow r_s + 2^{k+1}$;
    if $r_m = r_s + 1$ *and* $f(r_s) > 0$ *but* $f(r_m) \leq 0$ **then**
        return r_m;
    if $f(r_s), f(r_m) > 0$ **then**
        $k \leftarrow k + 1$;
    else
        if $f(r_m) = 0$ **then**
            Return r_m;
        else
            (it must be that $f(r_s) > 0$ but $f(r_m) < 0$)
            $j \leftarrow$ Binary-Search($upper = r_s, lower = r_m, f$);
            (this above function returns an index i that is at least r_s and at most r_m such that $f(i) = 0$ if exists. It returns i with $f(i) < 0$ if $f(i-1) > 0$)
            Return j;

The binary search subroutine described above does a binary search with lower index r_s and upper index r_m and at every stage, checks if $f(i) \leq 0$ but $f(i) > 0$.

- **Explanation of the running time of your algorithm:** (3 Marks) Observe that the binary search subroutine is used with lower index r_s and upper index r_m. Also, $r_m - r_s = 2^k$ and $r_m = 2r_s$. As $2^k = r_m - r_s$, it only requires to invoke $f(i)$ for $k+1$ distinct values. Then, the binary search runs in $O(k)$-time in for this range of values. Also, $n \leq 2^{k+1}$.
  Hence, the running time of the algorithm is $O(k)$, i.e. $O(\log n)$.

  (deduct 1 Mark if the explanation is partially correct. If the justification is absent, then deduct 2 Marks)

3. **(20 Marks)** Suppose that you are running a large computing job in which you need to simulate a physical system for as many discrete steps as you can. The lab you are working as two large supercomputers A and B. But your job can only run one of the machines at any given minute. Over each of the next n minutes, you have a 'profile' of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps if your job uses machine A and $b_i > 0$ steps of simulation if your job uses machine B. ou can move your job from one machine to the other. But moving job from one machine from other costs you a minute and no processing can be done in that minute. So, given a sequence of n minutes, a *plan* is specified by a choice A, B or *move* for each minute with the property that two distinct machines cannot appear in two consecutive minutes. It means that if you use machine A in minute i and you want to switch to machine B, then choice for minute $i+1$ must be *move*. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so, it's the sum of a_i over all the minutes in which the job is on A, plus the sum of b_i in which your job is in B. Design a dynamic programming based algorithm to compute the the value of an *optimal plan* (i.e. a plan with maximum value). **OVER n MINUTES**

- **Define the sub-problems and state the number of sub-problems** (4 Marks)
  For minutes $\{1, \ldots, k\}$, we define two subproblems.
  $StepA(k)$ is the maximum value of a plan if machine A is used at the k-th minute.
  $StepsB(k)$ is the maximum value of a plan if machine B is used at the k-th minute.
  Hence, there are 2 subproblems. If somebody also adds $ST(k) = \max\{StepsA(k), StepsB(k)\}$ then also no deduction of marks.
  Rubric: If somebody writes the subproblem definitions correctly, but his/her number of subproblems mentioned is inconsistent, then deduct one mark.

- **The recurrence relation including base case(s):** (6 Marks)
  Base cases are: $StepsA(1) = a_1$ and $StepsA(2) = a_1 + a_2$; $StepsB(1) = b_1$ and $StepsB(2) = b_1 + b_2$. For $k \geq 3$, the recurrence is

$$StepsA(k) = a_k + \max\left\{StepsA(k-1), StepsB(k-2)\right\}$$

$$StepsB(k) = b_k + \max\left\{StepsB(k-1) + StepsA(k-2)\right\}$$

  (**Rubric:** 2 Marks for base cases and 4 Marks for correct recurrences. Deduct 1 mark if partially incorrect base case and deduct 2 additional marks if partially correct recurrence).

- **The subproblem that solves the actual problem:** (2 Marks)

$$\max\{StepsA(n), StepsB(n)\}$$

  .

- **A brief description dynamic programming algorithm**: (5 Marks) We use two different arrays $A[1, \ldots, n]$ and $B[1, \ldots, n]$ as follows.
  - Initialize $A[1] = a_1$, $A[2] = a_1 + a_2$, $B[1] = b_1$ and $B[2] = b_1 + b_2$.
  - For $k = 3, \ldots, n$; assign $A[k] = a_k + \max\{B[k-2], A[k-1]\}$ and assign $B[k] = b_k + \max\{B[k-1] + A[k-2]\}$.
  - Finally, output $\max\{A[n], B[n]\}$.

  (**Rubric:** deduct 2 marks for partially correct algorithm description or partially correct pseudocode)

- **Explanation of the running time**: (3 Marks) Initialization of $A[1], A[2], B[1], B[2]$ values require $O(1)$-time. Computing $A[k]$ requires $O(1)$ many comparison operations and other arithmetic operations. Hence, the second step requires $O(n)$-time. Finally, the output requires $O(1)$-time. Hence, the running time of the algorithm is $O(n)$.
  (**Rubric:** deduct 1 mark for incorrect justification while the running time is correct and consistent with the algorithm)