Due: Thursday, February 6, 202 4:59PM on MarkUs

You will receive 20% of the points for any (sub)problem for which you write "I do not know how to answer this question." You will receive 10% if you leave a question blank. If instead you submit irrelevant or erroneous answers you will receive 0 points. You may receive partial credit for the work that is clearly "on the right track."
You mnay choose to spend your time looking for solutions on the internet and may likely succeed in doing so but you probably won't understand the concepts that way and will then not do well on the quizzes, midterm and final. So at the very least try to do the assignment iinitially without searching the internet. If you obtain a solution directly from the internet, you must cite the link to avoid plagarizing.

1. (20 pts) You are writing a blog about restaurants in various cities. You have a list of the 16 top restaurants in Moscow but you don't trust the reviewer. You have a friend in Moscow is willing to help and all you want to do is provide the best of these 16 restaurants and the worst of these 16 restaurants. So you ask your friend to compare restaurants and give you his opinion on the best and the worst. How many retaurants does he/she have to compare?

   (a) (5 points)It is reasonably obvious how your friend can provide his/her opinion doing 30 comparisons. Say why?

   Solution sketch: We can maintain a candidate for the maximum element (i.e., best restaurant) and keep updating it. This takes $n-1$ comparisons when there are n elements in an array. Once we eliminate the maximum elemenet we have n-1 elements and finding the minium element will then take $n-2$ more comparisons. So we have a total of $2n-3$ and thus 30 comparisons when $n=16$.

   (b) (10 points) It is possible to provide the desired opinion in just 22 comparisons. Provide a divide and conquer algorithm that will achieve this bound. State your algorithm (for any arbitrary number n of restaurants) in pseudo code.

   Solution sketch: The idea is to partition the array in two subarrays of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ and apply recursion.

   (c) (5 points) State the recurrence (when there are $n = 2^k$ restaurants) that describes the number comparisons. (Do not forget the base case.)
   When $n = 2^k$ we don't have to worry about floors and ceilings and the resursion is then simply $T(n) = 2T(n/2) + 2$ with the base case $T(2) - 1$. Then by induction, we can then prove that $T(n) = \frac{3}{2}n - 2$. When we don't assume $n = 2^k$, we can prove $T(n) = \lceil 3n/2 \rceil - 2$. Whe $n = 16$, we get $T(16) = 22$.

2. (20 points) Let $a(x)$ be a polynomial of degree $n-1$ (say over the reals). We can evaluate $a(x)$ at a given point y in $O(n)$ $+, -, \times$ arithmetic operations. In fact, the precise number is $2(n-1)$ arithmetic operations using Horner's rule. Your goal is to provide an algorithm that will evaluate $a(x)$ at n distinct points $y_1, \ldots, y_n$. Evaluating at each y_i separately will result in a totasl of $O(n^2)$. operations. We are aiming for an algorithm that uses asymptoticallly much fewer arithmetic operations.

   Assume we can multiply two degree n polynomials in $O(n \log n)$ arithmetic operations. (Note: this can be done as we will indicate in class.) We also know that we can also do polynomial division in $O(n \log n)$ arithmetic operations. By polynomial division, we mean that for polynomials $a(x)$ and $b(x)$ we can compute polynomials $q(x)$ and $r(x)$ such that $a(x) = q(x)b(x) + r(x)$ with degree $r <$ degree b.
   You may assume $n = 2^k$ for some $k \geq 1$. For simplicity (but without loss of generality), let $n = 2^k$.

   (a) (5 points)
   With and without recursion, show how to efficiently compute the polynomial $\prod_{i=1}^{n}(x - y_i)$ where $y_1, \ldots, y_n$ are n real numbers. Using the above assumption on the complexity of polynomial multiplication, what is the time complexity for computing this polynomial?

   (b) (10 points) Provide a divide a conquer algorithm for evaluating a degree $n-1$ polynomial $a(x)$ at n distinct points $y_1, \ldots, y_n$ using only $T(n) = O(n \log^2 n)$ airthmetic operations. Hint: Divide $a(x)$ by an appropriate $b(x)$ of degree $n/2$ that can be efficiently computed.

   (c) (5 points) Indicate the recursion and justfiy the bound on the number of arithmetic operations using the assumptions about the complexity of polynomial multiplication and division. What would the bound be if we only assume that polynomial multiplication and division can be done in $O(n^{\log_2 3})$.

   Solution sketch:

   In the division theorem, if we let $b_1(x) = \prod_{i=1}^{\frac{n}{2}}(x - y_i)$, then $a(y_i) = r(y_i)$ for $1 \leq i \leq \frac{n}{2}$. Similarly, we can let $b_2(x) = \prod_{i=\frac{n}{2}+1}^{n}(x - y_i)$

   Using the first part of the question, we can assume that the products $b_1(x)$ and $b_2(x)$ are all available (as well as the product of polynomials needed in the recursive calls) then the recursion is $T(n) = 2T(n/2) + O(n)$. The master theorem gives the desired asymptotic bound.

3. (20 points) Maximum subarray sum

   (a) Given an array A of n signed integers, design a divide and conquer algorithm in time $O(nlog(n))$ to find a subarray $A[i, ..., j]$ such that $A[i] + A[i + 1] + ... + A[j]$ is maximized ($1 \leq i < j \leq n$). In other words, find a pair (i, j) such that $\forall(x, y)\Sigma_{k=i}^{j}A[k] \geq \Sigma_{k'=x}^{y}A[k']$. Note: empty set is not a valid input.

- In English, describe your algorithm.
  Solution Partition array A into two almost equally size subarrays, L (for left) and R (for right). The solution is maximum the following cases:
  - Solution of the L
  - Solution of the R
  - Solution will cross the middle of A.

  The first two cases can be solved recursively. If the third case is the solution, then the left element of R subarray, R_l, and the right element of L subarray, L_r, must be in the solution. Focusing on the R we start from R_l and scan R to the left to find the MSS that included R_l. We denote it as T_r. For L, we start from L_r and move to the right to find the MSS that included L_r. We denote it as T_l. Finally we return the maximum of these three cases.
- Describe your algorithm in pseudocode.

```
FIND-MSS(A, p, r)
 1  if p = r
 2     then return {p, p, A[p]}
 3  q ← ⌊(p + r)/2⌋
 4  {i_l, j_l, T_l} ← FIND-MSS(A, p, q)
 5  {i_r, j_r, T_r} ← FIND-MSS(A, q + 1, r)
 6  {i_s, j_s, T_s} ← MAX-SPAN(A, p, q, r)
 7  T ← max(T_l, T_s, T_r)
 8  switch
 9     case T = T_l :
10           return {i_l, j_l, T}
11     case T = T_s :
12           return {i_s, j_s, T}
13     case T = T_r :
14           return {i_r, j_r, T}
```

```
MAX-SPAN(A, p, q, r)
 1  ▷ Find maximum sum to the left, beginning at q
 2  i ← q, T_l ← −∞, S ← 0
 3  for k ← q downto p
 4     do S ← S + A[k]
 5        if S ≥ T_l
 6           then i ← k, T_l ← S
 7  ▷ Find maximum sum to the right, beginning at q + 1
 8  j ← q + 1, T_r ← −∞, S ← 0
 9  for k ← (q + 1) to r
10     do S ← S + A[k]
11        if S > T_r
12           then j ← k, T_r ← S
13  T ← T_l + T_r
14  return {i, j, T}
```

- Using recurrence analyze the running time of your algorithm (Do not forget the base case).
  Solution In *MAX-SPAN* in lines 3 to 9 we have only one loop over the full array, so its complexity is $\theta(n)$. In *MAX_MSS* we have two recursive calls. So time complexity is:

$$T(n) = \begin{cases} \theta(1) & \text{for } n = 1 \\ 2T(n/2) + \theta(n) & \text{for } n > 1 \end{cases}$$

  Based on Master Theorem, a = 2, b = 2, and d = 1, so $T(n) = O(nlogn)$

4. (15 points)

Consider the following greedy algorithm for graph colouring. Without loss of generality we will let the input $G = (V, E)$ be a connected graph with $V = \{v_1, v_2, \ldots, v_n\}$ $(n \geq 1)$ and let the colours be $C = \{1, 2, \ldots\}$. We also let $Nbhd(v)$ denote the neighbourhood (i.e. adjacent vertices) of node v.

GREEDY COLOURING ALGORITHM (using breadth first search)

Let $\chi(v_1) = 1; L(0) := \{v_1\}; i := 0$ % $\chi()$ is the colouring function
    % $L(i)$ will denote the nodes in the current level of the breadth first search
Let $A := \{v_1\}$ % A will be the nodes already coloured
Let $U := V - \{v_1\}$ % U will be the nodes not yet coloured
While $U \neq \emptyset$
  $L(i+1) := \emptyset$
  For $j := 1..n : v_j \in L(i)$
    For $k = 1..n : v_k \in Nbhd(v_j) \cap U$
      % colour the node being added to the next level
      $\chi(v_k) := min_{c \in C} : c \notin \cup_{v_h \in Nbhd(v_k) \cap A} \chi(v_h)$
      $U := U - \{v_k\}; A := A \cup \{v_k\}$
      $L(i+1) := L(i+1) \cup \{v_k\}$
    END For
  END For
  $i := i + 1$
END While

(a) (10 points) Give a short but convincing argument showing that the above greedy algorithm will colour every 2-colourable graph using 2 colours; that is, the greedy algorithm is optimal for 2-colourable graphs.

Solution sketch: Without loss of generality we can consider the graph to be connetced. Consider the breadth first search tree that is being constructed by the colouring algorithm. If the graph is 2 colourable, each level of the search tree will correspond to one side of a bipartite graph. Each node at level $2i - 1$ is colored with colour 1 and each node at level $2i$ will be coloured with clolour 2. If there was a conflit (i.e, twlo adjacent nodes getting the same col,our, then there wlould have to be an lodd lenght cycle and any odd length cycle requires 3 colours.

(b) (5 points) Give an example of a 3-colourable graph for which the above greedy algorithm will use more than 3 colours. You may label the verticess in any way (which then fixes the order in which the algorithm assigns colours,)

Solution: Let $\{v_1, v_2, v_6\}$ be $\{v_3, v_4, v_5\}$ be triangles and let (v_3, v_6) be an edge. Then the greedy colouring algorithm will require a 4th colour.

5. (20 points) Recall the EFT algorithm for interval scheduling on one machine. We wish to extend this algorithm to m machines. That is, when considering the i^{th} interval I_j we will schedule it if there is an available machine. But on which machine? That is, what is the tie breaking rule?

(a) Consider the First-Fit EFT greedy algorithm:

- Sort the intervals $\{I_j\}$ so that $f_1 \leq f_2 \ldots \leq f_n$
- For $j = 1 \ldots n$
  For $\ell = 1 \ldots m$
  If I_j fits on M_ℓ then schedule I_j on M_ℓ
  EndFor
  EndFor

Is First-Fit an optimal algorithm? If yes, provide a proof; otherwise provide a counter-example.

Solution: First-Fit EFT is not an optimal algorithm even for $m = 2$ machines. Consider the following 4 intervals (sorted by finishing times): $[1, 3), [2, 5), [8, 10), [4, 11)$. First-Fit EFT would schedule 3 intervalks while the optimum solution can schedule all 4 intervals.

(b) Consider Best-Fit EFT greedy algorithm:

- Sort the intervals $\{I_j\}$ so that $f_1 \leq f_2 \ldots \leq f_n$
- For $j = 1 \ldots n$
  If there is a machine M_ℓ on which I_j can be scheduled, then
  schedule I_j on M_ℓ where $\ell = argmax_k\{f_k \leq s_j$ and I_k scheduled on $M_k\}$
  EndFor

Is Best-Fit an optimal algorithm? If yes, provide a proof; otherwise provide a counter-example.

Solution: Yes Best-Fit is an optimal algorithm. The proof is a "promising argument" just like the proof for one machine but now there is one extra case to consider. That is, for every i we need to show that there is an optimum solution OPT_i that extends the partial solution S_i being created by Best-Fit. The extra case is that Best-Fit schedules interval i on some machine m and OPT_i schedules this interval on some macine $m' \neq m$. This extra case is handled by swapping all the intervals $j \geq i$ on these two machines.

6. (20 pts) You are given two arrays D (of positive integers) and P (of positive reals) of size n each. They describe n jobs. Job i is described by a deadline $D[i]$ and profit $P[i]$. Each job takes one unit of time to complete. Job i can be scheduled during any time interval $[t, t+1)$ with t being a positive integer as long as $t+1 \leq D[i]$ and no other job is scheduled during the same time interval. Your goal is to schedule a subset of jobs on a single machine to maximize the total profit - the sum of profits of all scheduled jobs. Design an efficient greedy algorithm for this problem.

   (a) Describe your algorithm in plain English (maximum 5 short sentences).

   Solution: We can assume that $D[i] \leq n+1$ for all i since all jobs only take one unit of time. This is not needed for correctness but does allow a time complexity just in terms of n. We can also also argue that in any valid schedule the jobs will be scheduled in increasing order of deadlines. This is not needed for the proof but might help in viksualizing things. Mainly, we need to sort the jobs so that $P[1] \leq P[2] \leq \cdot \leq P[n]$. We can then take jobs greedily, shceduling each job at its deadline.

   (b) Describe your algorithm in pseudocode.

   (c) Prove correctness of your greedy procedure. One possibile proof is to argue by induction that the partial solution constructed by the algorithm can be extended to an optimal solution. But you can use any type of valid proof argument.

   Solution: Once again, you can use a promising argument.

   (d) Analyze the running time of your algorithm (in terms of the total number of operations).

   Solution: The sorting will take time $O(n \log n)$ and then after that it is time $O(n)$.

7. (30 points) A is an array of n integers. Design a dynamic programming algorithm to find a subset $S \subseteq \{1, 2, 3, ..., n\}$ that maximizes $\sum_{i \in S} A[i]$, subject to the constraint that no two members of S can point to two consecutive elements of A, i.e., if $i \in S$ then $(i + 1) \notin S$.

   - (15 points) Describe your algorithm by giving a semantic array M (i.e. what is in each entry of the array) and a recursively defined array M' that will be equal to M in each entry. Don't forget any base case(s) and explain how the desired set S is obtained from M. Provide a brief explanation that $M = M'$.

     Solution: To solve the problem using DP we need to define: (1) the optimal substructure solution, and (2) the recursive solution. Let $M[i]$ the value of the optimal solution for the subarray A[1, ..., i]. Now the array M can be extended to add a solution corresponding to the optimum value in $M[i]$.

     Here quickly is the recursive algorithm. The base case can be $i = 0$ and $M[n] = 0$. For the recursion, $M[i] = \max\{S_1, S_2\}$ where $S_1 = M[i - 1]$ and $S_2 = A[i] + M[i - 2]$.

     To find the members of the optimal solution, here is what we do: If A[i] is part of the optimal solution, then A[i + 1] cannot be in the optimal solution and thus, the optimal solution will be the union of A[i] with the optimal solution for A[i + 2, ..., n]. If A[i] is not part of the optimal solution then the optimal solution will be the best solution for A[i + 1, ..., n].
   - (10 points) Implement your algorithm as an iterative (non recursive) algorithm.

     Solution: Note that since we want to know elements of S in addition to the optimal value, we need to traverse M and construct the optimal solution for S.

```
Find-MNCS(A)
  % Compute the lookup table M
    n ← length(A)
    M[n + 1] ← 0
    if A[1] > 0
        then M[1] ← A[1]
        then M[1] ← 0
        for i ← 2 to n
            do if A[i] > 0
                then S1 ← A[i] + M[i − 2]
                else S1 ← 0
            S2 ← M[i − 1]
            M[i] ← max(S1, S2)

     T ← M[n]
  % Determine members of S
    S ← ∅
    i ← 1
    while i ≤ n
        do if M[i] > M[i + 1]
              then S ← S ∪ i
                   i ← i + 2
              else  i ← i + 1
    return S and T
```

   - (5 points) What is the aysmptotic time complexity of your algorithm whether executed

iteratively or recursively (with memoization).

Solution: The complexity is $O(n)$ as no sorting is needed.