CS218 Design and analysis of algorithms **Jan-Apr 2025**

Midsem solutions

Total Marks: 50 *Time: 120 minutes*

Instructions.

- Please write your answers clearly and concisely. Please read the questions carefully and answer precisely what is asked.

Question 1 [6 marks]. Just like integer multiplication, we can apply divide and conquer approach for matrix multiplication as well. The idea is as follows. To multiply two $n \times n$ matrices, we will do a certain number of multiplications of smaller size matrices. In addition, we will need to do some matrix additions/subtractions, which will take $O(n^2)$ time. Consider the following variants, where to multiply $n \times n$ matrices we will need

(a) 7 multiplications of $n/2 \times n/2$ matrices and a constant number of $O(n^2)$ time operations.

(b) 25 multiplications of $n/3 \times n/3$ matrices and a constant number of $O(n^2)$ time operations.

(c) 48 multiplications of $n/4 \times n/4$ matrices and a constant number of $O(n^2)$ time operations.

Write the recurrence relations for the time complexity of $n \times n$ matrix multiplication, in each of the above variants. Arrange the solutions in increasing order of time complexity.

Answer 1. The recurrences are as follows.

(a) $T(n) = 7\,T(n/2) + O(n^2)$.

(b) $T(n) = 25\,T(n/3) + O(n^2)$.

(c) $T(n) = 48\,T(n/4) + O(n^2)$.

The solutions for these are

(a) $O(n^{\log_2 7})$,

(b) $O(n^{\log_3 25})$,

(c) $O(n^{\log_4 48})$.

Increasing order of time complexity is (c) < (a) < (b).

4 marks for the recurrences and their solutions. 2 marks for the correct order. Below explanation was not expected.

We can see the order as follows. To compare (c) and (a), observe that

$$\log_2 7 = \log_{2^2} 7^2 = \log_4 49 > \log_4 48.$$

To compare (a) and (b), observe that

$$\log_3 25 = \log_{3^5} 25^5 = \log_{243} 25^5 > \log_{256} 25^5 = \log_{256}(25^4 \times 25) > \log_{256}(25^4 \times 16) = \log_{256}(50^4) > \log_{256}(49^4) = \log_{2^8}(7^8).$$

Question 2 [8 marks]. Two dimensional convolution can also be viewed as multiplying two bivariate polynomials (polynomials in two variables x and y). We are given two $d \times d$ matrices M and N (indexed from 0 to $d-1$), which represent the coefficients of the following two polynomials.

$$\sum_{i=0}^{d-1}\sum_{j=0}^{d-1} M[i,j]\, x^i y^j \quad \text{and} \quad \sum_{i=0}^{d-1}\sum_{j=0}^{d-1} N[i,j]\, x^i y^j.$$

Give an $O(d^2 \log d)$ time algorithm to multiply these two polynomials. You can directly use the fact that multiplication of two degree d polynomials (in one variable) can be done in $O(d \log d)$ time. Equivalently, convolution of two d-length tuples can be done in $O(d \log d)$ time.

Note: $\log(d^2) = 2 \log d$.

4 marks for a correct $O(d^3 \log d)$ time algorithm.

Answer 2. The main idea is to convert the problem into multiplication of polynomials in one variable. We can substitute $x = t$ and $y = t^r$, for some number r, and then do the usual multiplication of two polynomials over variable t. There is a problem with this approach. When we do this substitution two different monomials in x and y may get mapped to the same monomial in t. If that happens, then we will not be able to separately get the two coefficients of the two mnomials. To avoid this, we will choose r to be large enough. Note that the product of two polynomials can have degree at most $2d-2$ in each variable. Let us choose $r = 2d$.

Claim 1. *Any two different monomials $x^i y^j$ and $x^k y^\ell$ for $0 \le i, j, k, \ell < 2d$, will get mapped to different monomials under the substitution $x \mapsto t$ and $y \mapsto t^{2d}$.*

Proof. Observe that the two monomials get mapped to t^{2dj+i} and $t^{2d\ell+k}$, repsectively. For the sake of contradiction, assume that $2dj + i = 2d\ell + k$. Then we have

$$2d(j - \ell) = k - i.$$

Note that $k - i$ has an absolute value less than $2d$. Hence, the above equation can hold only if both sides are zero. That is, $k - i = j - \ell = 0$. That means the two monomials are same. Contradiction. □

After the mentioned substitution, we will get two polynomials in variable t. Their coeffients will be given by the following tuples.

$$(M[0,0], \ldots, M[d-1,0],\ 0, 0, \ldots, 0,\ M[0,1], \ldots, M[d-1,1],\ \ldots,\ M[0,d-1], \ldots, M[d-1,d-1]).$$

$$(N[0,0], \ldots, N[d-1,0],\ 0, 0, \ldots, 0,\ N[0,1], \ldots, N[d-1,1],\ \ldots,\ N[0,d-1], \ldots, N[d-1,d-1]).$$

We can easily compute these new polynoimals in $O(d^2)$ time, whose degrees are at most $(2d+1)(d-1) = (d^2)$. This is simply a flattening of the two matrices, with zeros inserted in appropriate places.

We can multiply these two polynomials using the FFT method discussed in the class, which will take $O(d^2 \log d)$ time. From the obtained polynomial, we can recover the desired product polynomial in variables x and y, as follows. Any monomial t^q should be replaced by $x^{q \bmod 2d}\, y^{q/(2d)}$.

Question 3 [8 marks]. In bioinformatics, one measure of similarity between two strings is how many times one occur as a subsequence of another. For example, consider two strings $S_1 = a\ b\ a\ c\ b\ a\ c\ b\ a\ c$ and $S_2 = a\ a\ b\ c$. Here, string S_2 appears as a subsequence of S_1 five times. Here are the indices of these 5 subsequences in S_1: (1, 3, 5, 7), (1, 3, 5, 10), (1, 3, 8, 10), (1, 6, 8, 10), (3, 6, 8, 10).

For two given strings, give a polynomial time algorithm to find the number of occurrences of the second string as a subsequence of the first string.

Hint: you can partition the set of occurrences of S_2 in a natural way and then for each part, compute the count recursively and add (keep storing what you already computed).

2 marks for a naive exponential time algorithm. If memoization is not done properly, -2 marks. If Base case is not done then -1.

Answer 3. Let the string S_1 be $s_1 s_2 \cdots s_n$. Let the string S_2 be $t_1 t_2 \cdots t_m$. Let us partition the set of occurrences of S_2 based on the position of the last character t_m. The number of choices of this position is the number of times the last character appears in S_1. For every such choice, we will count the number of occurrences recursively, and then add all the counts. Let the j-th character in S_1, i.e., s_j be same as t_m. Then the number of occurences where t_m appears at the j-th position in S_1 is same as the number of occurences of the remaining string $t_1 t_2 \cdots t_{m-1}$ in the prefix $s_1 s_2 \cdots s_{j-1}$ of string S_1.

Note that all the subproblems in the recursion will be of the following kind. For some $k \leq n$ and some $\ell \leq m$, count the number of occurences of $t_1 t_2 \cdots t_\ell$ in $s_1 s_2 \cdots s_k$. We will maintain a 2D array OCCUR where $[k, \ell]$ entry will denote the above count. Out final answer will be $[n, m]$ entry of this array.

We can compute the entries in this array by the following algorithm.

For any $k \leq n$ and any $\ell \leq m$:

- if $\ell > k$ then OCCUR$[k, \ell]$= 0. This is because a longer string cannot appear as a subsequence of a smaller string.
- if $\ell \leq k$ then

  Initialize OCCUR$[k, \ell]$= 0.

  For $j = 1, 2, \ldots, k$:

  if $s_j = t_\ell$ then OCCUR$[k, \ell]$ = OCCUR$[k, \ell]$ + OCCUR$[j-1, \ell-1]$.

We will run the above procedure for for every $1 \leq k \leq n$ and every $1 \leq \ell \leq m$. Note that to compute any entry of the array, we need the entries with smaller indices. Hence, we should go in increasing order of both k and ℓ. That is, we will have two outer for loops, one for $k = 1, 2, \ldots, n$ and one for $\ell = 1, 2, \ldots, m$.

Total running time of this algorithm is $O(mn^2)$.

Question 4 [5 marks]. Suppose you are trading a particular commodity whose price fluctuates every day. For some reason, you cannot wait long to sell it after buying. Once you buy it, you have to sell it whenever the first time price becomes higher than your purchase price. If it never becomes higher, then you cannot sell it. You are given the prices for next n days, say $p_1, p_2, \ldots, p_n$ (assume all are distinct). You want to purchase on the day which will give the maximum profit.

For any j, let $f(j)$ be the first index greater than j such that $p_{f(j)} > p_j$. If no such index exists then take $p_{f(j)} = 0$. You have to find $\max_j(p_{f(j)} - p_j)$.

Design an $O(n)$ algorithm. You should argue the correctness and the runtime bound of your algorithm. No marks for $O(n^2)$ time. Partial marks for something in between.

Example input: Prices: 70, 100, 90, 95, 140, 40, 60, 90, 120, 30, 60.

For example, if you buy at 100, you have to sell at 140, and you will get a profit of 40.

Here, the optimal choice is to buy at 95, sell at 140, giving profit of 45.

3 marks for the correct algorithm, 2 marks for the argument showing correctness. 2 marks for $O(n \log n)$.

Answer 4. There is an $O(n)$ algorithm to compute $f(j)$ for every j. But, we will not need that.

There is one case when all prices are in decreasing order. In this case, no profit can be made. This can be simply checked in $O(n)$ time. In all other cases, a profit can be made.

Claim 2. *To get the maximum profit, there is a choice i, where you purchase on day i and sell on day $i+1$.*

Proof. Suppose the maximum profit is achieved by purchasing on day j and selling on day k with $k > j$. By the given condition in the problem, k must be the first index greater than j such that $p_k > p_j$. That means every price in between $p_{j+1}, \ldots, p_{k-1}$ is at most p_j. Hence, we can buy on day $k-1$ and sell on day k and get profit $p_k - p_{k-1}$ which is at least $p_k - p_j$. □

Knowing the above claim, computing the maximum profit is simple. Compute $\max_k(p_k - p_{k-1})$. This can be done in $O(n)$ time.

Question 5 [5 marks]. You have to meet n people, each of whom has given you an interval of dates in which they are available. It is possible to meet any subset of people on a particular day, if that day is present in each of their intervals. You want to select the minimum number of days, on which you can set up the meetings, so that you cover all n people.

Example input: (1,5), (3, 10), (4, 9), (6, 8). Here is a possible schedule. Meet first three persons on day 4. Meet the last person on day 7.

Here is a proposed algorithm. Find the day which is present in maximum number of intervals (break ties arbitrarily). Schedule a meeting on this day with all the people available on that day. Remove these intervals and repeat this process with the remaining.

Find an example where this algorithm does not give an optimal solution. For your example, clearly show what is the optimal solution and what is the algorithm's solution.

Note: correct algorithm is not required.

Answer 5. We are following the convention that (i, j) includes day i and day j both. It is ok to follow a different convention.

Consider the following example: (1,4), (3,6), (5, 8), (7, 10). Note that there is no day which is present in three of these intervals. Hence, maximum number of intervals which have a common day is 2. For example, day 3 is present in first two intervals. Day 5 is present in second and third intervals. Day 7 is present in second and third intervals. As mentioned in the problem, ties are broken arbitrarily. So, the algorithm could choose Day 5 in the first step, covering second and third interval. After removing them, we are left with first and fourth interval. The algorithm will choose one day from the first interval, then one day from the fourth interval (or vice versa). Here, the algorithm has selected 3 days.

However, there is a better solution. Select days 3 and 7. Day 3 will cover first two intervals. Day 7 will cover last two intervals.

One can construct a more interesting example, where there are no ties. This is not expected in the solution.

Consider (1,4), (3,6), (3,7), (5,9), (6, 9), (8, 10). Here the algorithm will first choose day 5 (or day 6) which covers four middle intervals. Then one day each for first and last intervals. In total, it will select three days.

However, an optimal solution is just two days, say day 3 and day 8. Day 3 will cover first three and day 8 will cover last three intervals.

Question 6 [10 marks]. In a variant of the above problem, suppose each day has an associated cost. We want to minimize the total cost of selected days, such that for every person, at least one of the selected days lies in their interval. We only care about the cost, and not about the number of days. Give a polynomial time algorithm. Your algorithm should output the optimal cost as well as the optimal set of days.

Answer 6. Note that the input size can be taken as the total number of days in the union of given intervals plus the number of intervals. This is because are given the cost of each day and the list of intervals. Our running time will be polnomial in these two parameters.

Let us sort the intervals in increasing order of finish times. Let the interval in this order be $I_1, I_2, \ldots, I_n$. Without loss of generality, let us assume that the starting day of earliest starting interval is day 1. And the last interval finishes at day t.

We will partition the set of possible solutions, based on whether a particular day is selected or not. We will start with the last day.

- If the last day is selected then, all intervals containing that day are covered. Then the subproblem to solve will be remaining intervals and all days except the last one.
- If the last day is not selected then, the subproblem to solve will be the same set of intervals, and all days excpet the last one.

Observe that our subproblems are parameterized by two parameters, the remaining set of intervals and the remaining set of days. Let us denote by $OPT[i, j]$ the optimal cost of the subproblem where given intervals

are $I_1, I_2, \ldots, I_i$ and we are trying to cover them using days $1, 2, \ldots, j$. Clearly, if any interval among first i intervals starts after day j, then it cannot be covered and hence, we will take $OPT[i, j]$ as ∞.

Otherwise, we can compute $OPT[i, j]$ recursively, as follows. Let $f(j)$ be the last index h such that $I_1, I_2, \ldots, I_h$ have their finish times strictly before j. Let c_j be the cost of day j.

$$OPT[i,j] = \min \begin{cases} OPT[i, j-1] & \text{(that is, we have not selected day } j\text{)} \\ OPT[f(j), j-1] + c_j & \text{(i.e., we have selected day } j \text{ and removed intervals covered by day } j\text{)} \end{cases}$$

Here, $OPT[i, j]$ is computed using OPT values for smaller indices. Hence, we should fill up this 2D array in increasing order of i and j. The final optimal cost is $OPT[n, t]$.

Initialization: For any $i > 0$, put $OPT[i, 0]$ as infinity. Put $OPT[0, j]$ as zero for any $j \geq 0$.

Computing the optimal set: If $OPT[n, t] = \infty$ then say that there is no such set. Otherwise, we will go backwards over the OPT array and compute the optimal set as follows.
Initialize $i = n$, $j = t$. Initialize optimal set as empty.

While($i \geq 1$ or $j \geq 1$):

- if $OPT[i, j-1] > OPT[f(j), j-1] + c_j$ then include j in the optimal set. And set $i \leftarrow f(j)$ and $j \leftarrow j - 1$.
- otherwise do not include j in the optimal set. And set $i \leftarrow i$ and $j \leftarrow j - 1$.

Output the optimal set.

6 marks for the correct recursive equation. 1 mark for base case. 3 marks for the backtracking idea to compute optimal set.

Question 7 [8 marks]. You have a large number of assignments in this semester, each of which takes one full day to finish (if you are not doing any other assignment on that day). The i-th assignment is released on a particular day, say r_i and has a deadline d_i. That means you can schedule the i-th assignment on any day t such that $r_i \leq t \leq d_i$. Given $r_1, r_2, \ldots, r_n, d_1, d_2, \ldots, d_n$, you want to find out whether it is possible make a schedule to finish all the assignments on time. Give a polynomial time algorithm for this. Prove the correctness of your algorithm.

Hint: a greedy approach may work. A typical approach to prove correctness of a greedy algorithm is as follows. You want to prove that if there is a feasible schedule then the greedy algorithm will give a feasible schedule. Consider a feasible schedule that agrees with first $i-1$ choices of your greedy schedule. Modify the feasile schedule to make it agree on first i choices, while maintaining feasibility. Then argue inductively.

Answer 7. There are two approaches. First let us see a greedy algorithm to schedule all the assignments, if it is possible.

Sort the assignments in increasing order of deadlines. Say the assignments in this order are $A_1, A_2, \ldots, A_n$.
For $j = 1, 2, \ldots, n$:

Schedule A_j on the earliest available day between r_j and d_j. If there is no day available in this interval then output that it is not possible to schedule all assignments.

Claim 3. *If it is possible to finish all assignments in time, then the above greedy algorithm will find a day for every assignment.*

Proof. The proof will be based on induction.

Induction hypothesis: the greedy algorithm succeeds in finding a day for first $i-1$ assignments and there is a feasible schedule for all assignments which assigns the same days for the first $i-1$ assignments as the greedy algorithm.

Base case: the above statement is trivially true for $i = 1$.

Induction step: Let the feasible schedule from the induction hypothesis be S. We will show that the greedy algorithm will find a day for the ith assignment and moreover, S can be modified so that it assigns the same days for the first i assignments as the greedy algorithm.

- Note that S has assigned the same days for first $i-1$ assignments as the greedy algorithm, and it also has scheduled a day for the i-th assignment. That means, after scheduling first $i-1$ assignments, the greedy algorithm will surely find an available day for the ith assignment.
- Let the first available day for the ith assignment be t. Greedy will schedule ith assignment on day t. If S also assigns the same day t for assignment i, then we are done. Otherwise, suppose S assigns another day t' for assignment i, which can only be later than t. Now, we would like to modify S as follows: move assignment i to day t. If S had scheduled another assignment A_j on day t, then move that to day t'. Note that A_j must be a later assignment (i.e., $j > i$), because the earlier assignments are scheduled as in the greedy algorithm. In particular, A_j has a deadline greater than (or equal to) d_i. Hence, there is no problem in moving A_j to day t', because we know that $t' \leq d_i$.

The induction hypothesis for first n assignments follows by induction. □

Second approach is based on the Hall's marriage theorem (on bipartite matching). Consider a bipartite graph, where assignments form one side of vertices and days form another side of vertices. There is an edge between an assignment and a day if and only if the day is between the assignment's release time and deadline. Clearly, it is possible to schedule all assignments, if and only if there is a matching in the above graph which matches all assignment vertices. Hall's theorem states that in a bipartite graph all the vertices on the left side can be matched if and only if for every subset T of left side vertices, we have

$$|N(T)| \geq |T|,$$

where $N(T)$ is the set of vertices neighboring to vertices in T. Applying Hall's theorem in our setting, we get that all the assignments can be matched if and only if for any subset T of assignments, the total number of days in the union of their intervals ($N(T) := \cup_{A_i \in T}(r_i, d_i)$) is at least $|T|$.

Now, there are exponentially many possible subsets of assignments, so we cannot check the above condition for all the subsets. The following observation implies that we only need to check polynomially many subsets.

Claim 4. *Suppose there is a subset T of assignments, such that $|N(T)| < |T|$. Then there are two indices j, k such that the following subset $T_{j,k}$ of assignments has $|N(T_{j,k})| < |T_{j,k}|$*

$$T_{j,k} = \{A_i : r_i \geq r_j \text{ and } d_i \leq d_k\}.$$

That is, the set of assignments released on r_j or later and having deadline d_k or earlier.

Proof. Any union of intervals can be viewed as a disjoint union of intervals. For example, union of $(1,3),(2,6),(7,9),(8,12)$ is simply $(1,6) \cup (7,12)$. Hence, the subset T can be naturally partitioned into subsets $T_1, T_2, \ldots, T_h$, such that for each $N(T_\ell)$ is a single interval and $N(T_1), N(T_2), \ldots, N(T_h)$ are disjoint. Clearly, $|T| = |T_1| + |T_2| + \cdots + |T_h|$ and also$|N(T)| = |N(T_1)| + |N(T_2)| + \cdots + |N(T_h)|$.

If $|N(T)| < |T|$, then there must be one subset T_ℓ such that $|N(T_\ell)| < |T_\ell|$. Since $N(T_\ell)$ is a single interval which is a union of given intervals, its starting point must be r_j for some j and ending point must be d_k for some k. Hence, we get that $|N(T_{j,k})| < |T_{j,k}|$. □

The above claim means we only need to check the following: for any j, k, whether $|N(T_{j,k})| < |T_{j,k}|$? This can be easily checked in polynomial time.

In either approach, 4 marks for the algorithm, 4 marks for the correctness.