CS218 Design and analysis of algorithms **Jan-Apr 2023**

Endsem solutions

Total Marks: 90 *Time: 180 minutes*

Instructions.

- Answers are in blue color.
- Marking scheme is in red color.
- Violet color means additional comments, not expected from students.

Que 1 (Divide and Conquer). You are given a set of 2-dimensional points, say $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$. Two points (x_i, y_i) and (x_j, y_j) are said to be not comparable if

$$(x_i - x_j)(y_i - y_j) < 0.$$

That is, when point i is smaller in one coordinate and larger in the other. Your goal is to find the number of pairs of points which are not comparable. For example, consider the set of points

$$\{p_1 = (1, 4), p_2 = (2, 5), p_3 = (4, 1), p_4 = (5, 4)\}.$$

Here the desired number is 3 (the pairs are (p_1, p_3), (p_2, p_3), and (p_2, p_4)).

We want to design an $O(n \log n)$-time algorithm for finding this number. We will go with the divide and conquer approach. We first sort the given points in **decreasing order of x-coordinates** and then call Count-and-Sort procedure. Count-and-Sort will use MergeY and Count as subroutines. MergeY takes two lists of points sorted in increasing order of y-coordinates and outputs the union of the two lists sorted in increasing order of y-coordinates.

Count-and-Sort:

Input: a list S of 2D points sorted in decreasing order of x-coordinates.

Output: returns the number of pairs of points in S which are not comparable. It also sorts S in increasing order of y-coordinates.

```
1. If $S$ has only one point then return ___
2. Divide $S$ into two halves: let $S_1$ be the first half and $S_2$ be the second half.
3. $c_1 \leftarrow$ Count-and-Sort($S_1$)
4. $c_2 \leftarrow$ Count-and-Sort($S_2$)
5. $c \leftarrow$ Count($S_1, S_2$) + ____
6. $S \leftarrow$ MergeY($S_1, S_2$).
7. return $c$.
```

(a) (2 marks) Fill in the blank in line 1.

(b) (2 marks) Fill in the blank in line 5.

(c) (6 marks) Describe the procedure Count. Assume S_1 and S_2 both have size $n/2$. The running time for this procedure should be $O(n)$.

(a) 0

(b) $c_1 + c_2$

(c) Observe that each point in S_2 has a smaller x coordinate than each point in S_1. To count incomparable pairs across S_1 and S_2, we need to count for any point in S_2, how many points in S_1 have smaller y coordinate. We will traverse over S_1 and S_2, which are sorted in increasing order of y coordinates. We will maintain two pointers: i for S_1 and j for S_2.

Some might have counted the opposite, pairs where point in S_1 has a larger y coordinate than the point in S_2. If the implementation is correct, they can get 3 marks.

```
**Count**($S_1, S_2$):
Initially $i = 0$, $count = 0$.
for $j = 0$ to $n/2 - 1$ {
    Keep increasing $i$ till the first point we get $S_2[j].y < S_1[i].y$
    $count \leftarrow count + i$ (because $S_2[j].y$ is larger than exactly $i$ $y$-coordinates in $S_1$)
}
return count.
```

If someone uses indexing starting from $i = 1$ then the update in the count would be $i - 1$.

Another solution

```
**Count**($S_1, S_2$):
Initially $i = n/2$, $count = 0$.
for $j = n/2 - 1$ to 0 {
    Keep decreasing $i$ till the first point we get $S_2[j].y > S_1[i].y$
    $count \leftarrow count + i + 1$ (because $S_2[j].y$ is larger than exactly $i + 1$ $y$-coordinates in $S_1$)
}
return count.
```

Similar solutions with i in the outer loop.

```
**Count**($S_1, S_2$):
Initially $j = 0$, $count = 0$.
for $i = 0$ to $n/2 - 1$ {
    Keep increasing $j$ till the first point we get $S_2[j].y > S_1[i].y$
    $count \leftarrow count + n/2 - j$ (because $S_1[i].y$ is smaller than exactly $n/2 - j$ $y$-coordinates in $S_2$)
}
return count.
```

Count(S_1, S_2):

```
Initially $j = n/2$, $count = 0$.
for $i = n/2 - 1$ to 0 {
    Keep decreasing $j$ till the first point we get $S_2[j].y < S_1[i].y$
    $count \leftarrow count + n/2 - j - 1$ (because $S_1[i].y$ is smaller than exactly $n/2 - j - 1$ $y$-coordinates in $S_2$)
}
return count.
```

The updates in count could have ± 1 if the array indexing is different.

Que 2 (Binary search). (6 marks) Suppose you have two processes which can possibly be scheduled in parallel. A process will take different amounts of time depending on how much memory you allocate to it. The time taken will be a non-increasing function of the allocated memory. Suppose you have total m bytes of memory, which can be distributed among the two processes. For each process, you are given an array of length m (indexed as $1, 2, \dots, m$), whose ith entry is the time taken by the process when i bytes of memory is allocated. You want to do a memory allocation so as to minimize the time taken for finishing both the processes.

Suppose the two arrays for the processes are T_1 and T_2. If we schedule these process one after the other, then the time taken to finish both will be $T_1[m] + T_2[m]$. We want to check whether scheduling them in parallel can give an advantage. Design an $O(\log m)$ time algorithm to find the minimum time in which both the processes can be finished, when scheduled in parallel.

If we allocate x bytes of memory to the first process and remaining $m - x$ bytes to the second process, then the time to finish both will be $f(x) = \max\{T_1[x], T_2[m-x]\}$. We want to find x such that $f(x)$ is minimized. Observe that $T_1[x]$ is a decreasing function and $T_2[m-x]$ will be a increasing function. When x is small $T_1[x]$ will be larger and when x is large, $T_2[m-x]$ will be larger. We need to find the point where this switch happens. We will do a binary search for this.

Start with $x = m/2$, check whether $T_1[x] > T_2[m-x]$. If yes, go to the right half, otherwise go to the left half. And so on.

```
Set $L = 0$ and $U = m$.
while $U - L > 1$ {
    $x \leftarrow (L + U)/2$.
    If $T_1[x] > T_2[m-x]$ then $L \leftarrow x$.
    otherwise $U \leftarrow x$.
}
Check the values of $f(L)$ and $f(U)$ and output whichever is less.
```

Que 3 (Inverse FFT). Let the dth roots of unity be $\omega^0, \omega^1, \ldots, \omega^{d-1}$.

(a) (2+2 marks) Prove that
$$\sum_{j=0}^{d-1} \omega^{ij} = \begin{cases} 0 & \text{if } 1 \le i \le d-1 \\ d & \text{if } i = 0. \end{cases}$$

(b) (6 marks) You can assume part (a) even if you did not prove it. Let $P(x) = a_0 + a_1 x + \cdots + a_{d-1}x^{d-1}$ be a degree $d-1$ polynomial. Let the evaluations of $P(x)$ on the dth roots of unity be $e_0, e_1, \ldots, e_{d-1}$. That is, for $0 \le i \le d-1$
$$e_i = P(\omega^i).$$
Define a new polynomial $Q(y)$ as $e_0 + e_1 y + \cdots + e_{d-1}y^{d-1}$. Prove that for each $0 \le i \le d-1$
$$Q(\omega^{-i}) = da_i.$$

(a) If $i = 0$ then $\omega^{ij} = 1$ and hence, $\sum_{j=0}^{d-1} \omega^{ij} = d$.

Consider the case when $1 \le i \le d-1$. We can use the formula for the sum of a geometric series.
$$\sum_{j=0}^{d-1} \omega^{ij} = \frac{\omega^{di} - 1}{\omega^i - 1} = 0.$$
The last equality holds because $\omega^{di} = 1^i = 1$.

(b)
$$\begin{aligned}
Q(\omega^{-i}) &= e_0 + e_1\omega^{-i} + \cdots + e_{d-1}\omega^{-i(d-1)} \\
&= P(\omega^0) + P(\omega^1)\omega^{-i} + \cdots + P(\omega^{d-1})\omega^{-i(d-1)} \\
&= \left(\sum_{j=0}^{d-1} a_k\right) + \omega^{-i}\left(\sum_{k=0}^{d-1} a_k\omega^k\right) + \cdots + \omega^{-i(d-1)}\left(\sum_{k=0}^{d-1} a_k\omega^{(d-1)k}\right) \\
&= \sum_{k=0}^{d-1} a_k \left(1 + \omega^{-i}\omega^k + \cdots + \omega^{-i(d-1)}\omega^{(d-1)k}\right) \\
&= \sum_{k=0}^{d-1} a_k \left(\sum_{j=0}^{d-1} \omega^{(k-i)j}\right) \\
&= da_i
\end{aligned}$$

The last equality holds because from part (a), the inner summation is zero whenever $k - i \ne 0$ and it is d when $k - i = 0$.

Que 4 (Greedy). Consider a set of m jobs which needs to be scheduled. Each job takes exactly one day to complete and on a given day at most one job can be scheduled. Each job j, has a release time r_j and a deadline d_j ($r_j \leq d_j$), which is known in advance. The job j can be scheduled on a day k only if $r_j \leq k \leq d_j$. We want to maximize the number of jobs that can be scheduled.

We use the following greedy strategy: go over the jobs in increasing order of deadlines. Schedule a job j on the earliest available day k such that $r_j \leq k \leq d_j$. If no day is available in this interval then the job is not scheduled.

Example: suppose there are 7 jobs with their release times and deadlines as

Job	Release time	Deadline
1	2	5
2	2	3
3	3	4
4	4	5
5	2	4
6	3	5
7	1	7

(a) (2 marks) What is the maximum number of schedulable jobs in the above example?

We want to prove that the above greedy algorithm will always give an optimal solution. We can prove this inductively. Let S_{greedy} be the schedule obtained by the greedy algorithm and S^* be an optimal schedule. Suppose S^* agrees with S_{greedy} on the schedule of first $i-1$ jobs (in increasing order of deadlines). Give short arguments for the following.

(b) (3 marks) If i-th job is not scheduled in S_{greedy}, then argue that it is not scheduled in S^* either.

(c) (4 marks) Suppose i-th job is scheduled on a day t in S_{greedy} and on a different day t' in S^*. Modify S^* to construct another optimal schedule where i-th job is scheduled on day t.

(d) (3 marks) Suppose i-th job is scheduled on a day t in S_{greedy} and not scheduled in S^*. Modify S^* to construct another optimal schedule where i-th job is scheduled on day t.

(a) 5

(b) If i-th job is not scheduled in S_{greedy}, that is because after scheduling first $i-1$ jobs, there was no available day left for the i-th job. Since S^* agrees with S_{greedy} on the schedule of first $i-1$ jobs, there will be no available day for the i-th job in S^*. Hence, i-th job is not scheduled in S^*.

(c) Since S^* agrees with S_{greedy} on the schedule of first $i-1$ jobs and S_{greedy} schedules the i-th job on the earliest available day, we can say that $t' > t$. In the schedule S^*, move i-th job to day t. If there was any job j scheduled on day t, move it to t'. (2 marks)

First we need to argue that it remains a valid schedule. Note that the job j must have its deadline after job i, because S^* agrees with S_{greedy} on all jobs before i. The job i was on day t', so its deadline and also job j's deadline is at least t'. The release time of job j was clearly t or before. Hence, it is valid to schedule job j on day t'. (1 marks)

The number of jobs scheduled in S^* remains unchanged. Hence, the schedule is still optimal. (1 marks)

(d) If job i is not scheduled in S^*, then we schedule it on day t. If there was any job j scheduled on day t, then we just remove it from the schedule. Clearly, the number of jobs remains same and hence the schedule is still optimal.

Que 5 (Dynamic Programming). We have n jobs and two processors to complete them. The processing times of jobs are dependent on the processors. Some job may be faster on processor A, while some other job may be faster on processor B. Let the processing times of the jobs be $a_1, a_2, \ldots, a_n$ on processor A and $b_1, b_2, \ldots, b_n$ on processor B.

We want to distribute all the jobs to the two processors so as to minimize the time to finish all the jobs (makespan, i.e., the maximum total time among the two processors). Design an algorithm to find the minimum makespan.

Example input: Processing times: 7, 9, 11 on processor A. And 8, 6, 10 on processor B.

Output: 14

(8 marks) if the algorithm takes $O(nT^2)$ time and (12 marks) if the algorithm takes $O(nT)$ time, where $T = \max\{\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} b_i\}$. Assume addition, comparisons etc between processing times are unit time. Please follow the below outline to describe your algorithm.

(i) Running time of your algorithm

(ii) Size and dimension of the array, which you will use for maintaining solutions for the subproblems.

(iii) What does an entry of this array supposed to compute?

(iv) Base case: which entries of the array you will initialize and how?

(v) How will you compute an entry of the array and in which order?

(vi) How will you find the optimal makespan from this array?

Solution for 8 marks

(i) $O(nT^2)$

(ii) 3-dimensional array M of size $n \times (T+1) \times (T+1)$.

1 marks

(iii) The entry $M(j, s, t)$ will be TRUE if it is possible to distribute first j jobs to the processors such that s is the total time for processor A and t is the total time for processor B. Otherwise the entry will be FALSE.

2 marks

(iv) Base case: $M(1, a_1, 0)$ = TRUE and $M(1, 0, b_1)$ = TRUE. For all other entries $M(1, \cdot, \cdot)$=FALSE. Alternatively, if our array starts from $j = 0$ then $M(0, 0, 0)$ =TRUE and FALSE for every other $M(0, \cdot, \cdot)$

1 marks

(v)
```
for $j = 2$ to $n$
    for all $s, t$ from 0 to $T$
        $M(j, s, t) \leftarrow M(j-1, s-a_j, t)$ OR $M(j-1, s, t-b_j)$
```

3 marks

(vi) Go over all s, t such that $M(n, s, t)$ is TRUE and among them find the minimum value of $\max\{s, t\}$.

1 marks

Solution for 12 marks

(i) $O(nT)$

(ii) 2-dimensional array of size $n \times (T+1)$.

1 marks

(iii) If it is possible to distribute first j jobs such that s is the total time for processor A, then $M(j,s)$ will be the minimum possible time on processor B among all distributions where total time for A is s. Otherwise $M(j,s)$ will be infinity.

4 marks

(iv) Base case: $M(1,a_1) = 0$ and $M(1,0) = b_1$. For other entries $M(1,\cdot) = \infty$. Alternatively, if our array starts from $j = 0$ then $M(0,0) = 0$.

1 marks

(v)
```
for $j = 2$ to $n$
    for all $s$ from 0 to $T$
        $M(j,s) \leftarrow \min\{M(j-1, s-a_j), M(j-1,s) + b_j\}$.
```

3 marks

(vi) Go over all s and find the minimum value of $\max\{s, A(n,s)\}$.

3 marks

Que 6 (Linear Programming). Consider the minimum makespan problem from the previous question, with n jobs and two processors. Let the processing times of the jobs be $a_1, a_2, \ldots, a_n$ on processor A and $b_1, b_2, \ldots, b_n$ on processor B.

(a) (6 marks) Write an integer linear program for the minimum makespan problem. You should have $O(n)$ variables and some linear constraints among them and a linear optimizing function. The optimal value for an integer solution should be equal to the minimum makespan.

Hint: you can take a variable z (say) for the makespan, minimize z in the optimizing function, and constraints can ensure that the makespan is at most z.

(b) (4 marks) In the above integer linear program, let us allow variables to take real values and thus, get a linear program. Show an example where the LP optimal value is 1/2 times the minimum makespan. In the example, you should mention the processing times, minimum makespan and an optimal LP solution with optimal value.

(a)

$$\begin{gathered}\min z \text{ subject to}\\ x_i, y_i \geq 0 \text{ for } 1 \leq i \leq n\\ x_i + y_i = 1 \text{ for } 1 \leq i \leq n\\ \sum_{i=1}^{n} x_i a_i \leq z\\ \sum_{i=1}^{n} y_i b_i \leq z\end{gathered}$$

2 marks for first two constraints and 4 marks for the last two.

One may also add $x_i, y_i \leq 1$ in the above program. Alternatively, we can just replace y_i with $1 - x_i$ and write the following program.

$$\begin{gathered}\min z \text{ subject to}\\ 1 \geq x_i \geq 0 \text{ for } 1 \leq i \leq n\\ \sum_{i=1}^{n} x_i a_i \leq z\\ \sum_{i=1}^{n} (1 - x_i) b_i \leq z\end{gathered}$$

2 marks for the first constraint and 4 marks for the last two.

(b) The simplest example is we have one job, with $a_1 = 10$ and $b_1 = 10$. Clearly minimum makespan is 10. However, the first program has a real solution $x_i = 1/2$, $y_i = 1/2$, $z = 5$ which satisfies the constraints. The optimal LP value is 5 which is half of the makespan 10.

Que 7 (Network flow). A vertex cover in a graph $G(V, E)$ is a set of vertices $S \subseteq V$ such that every edge has at least one end point in S. We are given a bipartite graph G with weights on vertices. We want to compute a minimum weight vertex cover in the bipartite graph G. We want to reduce this problem to minimum s-t cut problem in a network.

Let L, R be the sets of left side and right side vertices in G. Let w_v denote the weight of a vertex v. We design the following network:

- direct all edges in G from left to right
- add a source vertex s and a sink vertex t
- add edges from s to all left side vertices
- add edges from all right side vertices to t.

(a) (4 marks) Assign capacities on the edges of this network such that outgoing capacity of the minimum s-t cut is equal to the weight of the minimum weight vertex cover in G. You might need to use infinite capacities for some edges (in practice, a large enough number can be used).

(b) (3+3 marks) Suppose the set $\{s\} \cup L' \cup R'$ is a minimum s-t cut, where $L' \subseteq L$ and $R' \subseteq R$. Describe how a vertex cover can be obtained from this and explain why it's a vertex cover.

(c) (4 marks) To show optimality, argue that for any vertex cover in G, we have an s-t cut whose outgoing capacity is equal the weight of the vertex cover.

(a) For any left side vertex v, the edge (s, v) will have capacity w_v. For any right side vertex u, the edge (u, t) will have capacity w_u. All other edges will have infinite capacity. 2 marks for infinite capacity and 2 marks for the other capacitites

(b) The vertex cover will be $(L - L') \cup R'$.

(3 marks)

We want to argue that it is indeed a vertex cover. That is, there is no edge in G which is not covered by this vertex set. That is, there is no edge between L' and $R - R'$. Why is this true? If there was such an edge, then it would be an outgoing edge from the given cut $\{s\} \cup L' \cup R'$. But, such an edge has infinite capacity, and hence, the given cut will have infinite outgoing capacity and cannot be minimum cut.

(3 marks)

(c) Consider any vertex cover $P \cup Q$ in G, where P comes from the left side vertices and Q comes from the right side vertices. The s-t cut we consider will be $\{s\} \cup (L - P) \cup Q$. What are the outgoing edges from this cut? Edges from s to P and the edges from Q to t. There are no more outgoing edges, because by the definition of a vertex cover, there are no edges from $L - P$ to $R - Q$. The outgoing capacity will be precisely $\sum_{v \in P} w_v + \sum_{u \in Q} w_u$, which is equal to the weight of the vertex cover.

4 marks

Que 8 (Approximation algorithms). Consider the problem of assigning roommates. We have n students and a set of rooms where 3 students can stay. For every pair of students we are given the information whether they are happy to be in one room or not. You can think of this as a graph on n vertices. Three students can be assigned a common room only if all three are happy with each other. We want to maximize the number of students who can be put into the triple occupancy rooms. Equivalently, we want to find the maximum number of vertex-disjoint triangles (size 3 clique) in the graph. We go with the following greedy algorithm.

```
$S$ is initially an empty set.
While the graph has triangles{
    choose an arbitrary triangle $(u, v, w)$ in the graph and put it in $S$.
    delete $u$, $v$ and $w$ and all their incident edges from the graph.
}
```

(a) (2 marks) Give an example where this algorithm does not give an optimal solution.

(b) (6 marks) Argue that this is a 1/3-approximation algorithm. That is, argue that the maximum number of vertex-disjoint triangles is at most $3T$, where T is the number of triangles given by the algorithm.

Hint: consider the vertices left out by the greedy algorithm. Is there any triangle consisting of only those vertices?

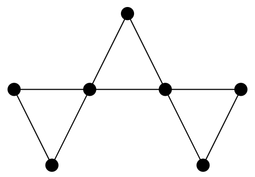

Figure 1: (a) Example where the algorithm is not optimal. If the algorithm picks the upper triangle first, there won't be anymore triangles left. However, the optimal solution is to pick two lower triangles.

(b) If the algorithm outputs T triangles, the number of vertices involved in those triangles will be $3T$. The algorithm stops only when there are no more triangles left. That means any triangle in the graph must have at least one vertex from the chosen $3T$ vertices. Hence, the number of disjoint triangles can be at most $3T$.

Que 9 (Randomized algorithms). Suppose you have a website and you want to find out the median time duration a visitor spends on your website during a visit, on a particular day (we don't care whether they are repeat visitors or new). For each visit, you can measure the time duration. However, your memory size is too small to store all the time durations. So, you go with the sampling idea. Suppose you already know the number of visitors you expect during the day, say n. Choose indices $i_1, i_2, \dots, i_k$, each uniformly randomly and independently from $\{1, 2, \dots, n\}$. During the day, store time durations of the i_1th, i_2th, $\dots$, i_kth visits. Finally output the median of the stored durations. We would like this to be an *approximate median* for the n time durations. The output is called an approximate median, if among the n durations, its rank is between $n/4$ and $3n/4$.

(8 marks) We choose $k = 6 \log n$. Prove that the output is an approximate median with probability at least $1 - 2/n$.

You may use the following fact: suppose a coin gives heads with probability $1/4$ and tails with probability $3/4$. If we toss the coin k times, the probability of getting at least $k/2$ heads is at most $(3/4)^{k/2}$.

Consider the event when the output has rank more than $3n/4$. Since the output was the median of the k stored durations, we can say that at least $k/2$ of them are larger than the output, and hence, have rank more than $3n/4$. What is the probability that at least $k/2$ of k stored durations have rank more than $3n/4$? Recall that each index i_j was chosen uniformly randomly from $\{1, 2, \dots, n\}$. Hence, the probability that the duration of the visit i_j has rank more than $3n/4$ is at most $1/4$.

Thus, our question can be put as follows. We toss k coins. Each coin gets heads with probability at most $1/4$. What is the probability of getting at least $k/2$ heads. By the given fact, it is at most $(3/4)^{k/2}$. Hence,

$$\begin{aligned}&\Pr(\text{output has rank more than } 3n/4) = \\ &\Pr(\text{at least } k/2 \text{ of the stored durations have rank more than } 3n/4) \le (3/4)^{k/2}.\end{aligned}$$

By symmetry, the same bound can be given for the event that the output has rank less than $n/4$. Taking union bound,

$$\begin{aligned}\Pr(\text{output has does not have rank between } n/4 \text{ and } 3n/4) &\le 2(3/4)^{k/2} \\ &= 2(3/4)^{3\log n} \\ &= 2(27/64)^{\log n} \\ &\le 2(1/2)^{\log n} \\ &\le 2/n.\end{aligned}$$

Hence, the output is an approximate median with probability at least $1 - 2/n$.

Que 10 (Red Blue path). (0 marks) This question is challenging. Consider a colored undirected graph where each edge is either red or blue. We need to find a red-blue s-t path if it exists, that is a path that starts at vertex s, alternates between red and blue edges, and ends at vertex t. By definition a path cannot have repeated vertices. Design a polynomial time algorithm for it. Alternatively, reduce it to the matching problem in (not necessarily bipartite) graphs. Given a graph, the matching problem asks for a largest size set of disjoint edges.

If you want to know the answer, you can discuss with the instructor.