CS218 Design and analysis of algorithms **Jan-Apr 2024**

Midsem solutions

Total Marks: 50 *Time: 120 minutes*

Question 1. Consider the following algorithm for multiplying two $n \times n$ matrices. It first splits both matrices into nine smaller matrices, each of size $n/3 \times n/3$. Then it does a constant number of additions and subtractions of these matrices. Then it recursively applies the same algorithm to multiply 23 pairs of $n/3 \times n/3$ matrices (no one knows if smaller than 23 is possible). Finally, there are a few more (constant number) additions and subtractions of $n/3 \times n/3$ matrices. When the recursion reaches to multiplying 1×1 matrices, it multiplies them trivially.

[**4 marks**] Do a runtime analysis and derive an upper bound on the running time of this algorithm. You can write your answer in the form $O(n^{\log_a b})$ or $O(n^{\log b/\log a})$.

[**2 marks**] Compare your answer with $O(n^{\log_2 7})$. Which one is asymptotically smaller?

Here are approximate values of $\log_2 x$ for $x = 2$ to $x = 24$.

1, 1.58496, 2, 2.32193, 2.58496, 2.80735, 3, 3.16993, 3.32193, 3.45943, 3.58496, 3.70044, 3.80735, 3.90689, 4, 4.08746, 4.16993, 4.24793, 4.32193, 4.39232, 4.45943, 4.52356, 4.58496,

Answer 1. Let $T(n)$ be the time taken by this algorithm for multiplying $n \times n$ matrices. Note that an addition/subtraction of $n/3 \times n/3$ matrices will take $O(n^2)$ time, because each entry can be computed in constant time. Then, as per the description, we can write the following recurrence, for some constant c

$$T(n) = cn^2 + 23\,T(n/3).$$

Solving this,

$$\begin{aligned}
T(n) &= cn^2 + 23\,T(n/3) \\
&= cn^2 + 23cn^2/9 + 23 \times 23\,T(n/9) \\
&\vdots \\
&= cn^2 + 23cn^2/9 + (23/9)^2 cn^2 + \cdots + (23/9)^{k-1} cn^2 + (23)^k\,T(n/3^k) \\
&= cn^2 \frac{(23/9)^k - 1}{23/9 - 1} + (23)^k T(n/3^k) \\
&= cn^2 \frac{(23/9)^k - 1}{23/9 - 1} + (23)^k T(1) \qquad (\text{taking } n = 3^k) \\
&\leq 23^k (3c/20 + T(1)) \qquad (\text{since } n^2 = 3^{2k} = 9^k) \\
&= O(23^{\log_3 n}) \\
&= O(n^{\log_3 23}) \\
&= O(n^{\log 23/\log 3})
\end{aligned}$$

Answer in any of the last two forms acceptable. It is completely fine if the steps for solving recurrence are not shown.

Comparing with $O(n^{\log_2 7})$.

$$
\begin{aligned}
n^{\log_2 7} &\stackrel{?}{\geq} n^{\log_3 23} \\
\log_2 7 &\stackrel{?}{\geq} \log_3 23 \\
\log 7/\log 2 &\stackrel{?}{\geq} \log 23/\log 3 \\
\log 7 \times \log 3 &\stackrel{?}{\geq} \log 23 \times \log 2 \\
\log 7 \times \log 3 &\stackrel{?}{\geq} \log 23 \\
2.81 \times 1.59 &\stackrel{?}{\geq} 4.52 \\
4.46 &\stackrel{?}{\geq} 4.52
\end{aligned}
$$

Thus, $O(n^{\log_2 7})$ is asymptotically smaller (faster).

Question 2. In complexity analysis or even in the implementation of algorithms, often we assume that the input size is a power of 2. Suppose the input size is n which is not necessarily a power of 2. Let $n' \geq n$ be the smallest integer that is a power of 2.

[**2+2 marks**] What can we say about n'? Is it $O(n)$, $O(n^2)$, $O(2^n)$, or something in between? Explain in a 1-2 line argument.

Answer 2. Let k be such that $2^{k-1} < n \leq 2^k = n'$. Then, $n' = 2^k = 2 \times 2^{k-1} < 2n$. Hence, it's $O(n)$.

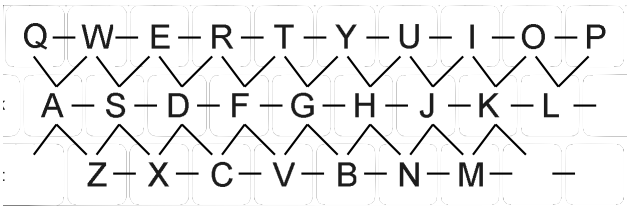

Figure 1: Keyboard with edges showing the adjacent characters.

Question 3. Consider an instance of the edit distance problem where strings are made of english alphabet and the edit costs are as follows.

- inserting or deleting a character – cost 2
- substituting a character with a different character
  - if the two characters are adjacent on the keyboard then – cost 1 (see Figure 1, where edges show the pairs of characters which are adjacent)
  - otherwise – cost 3

[**7 marks**] Find a way to modify string LINUX to DOUBT with cost at most 10. You should clearly show the sequence of edit steps with their costs.

Only 4 marks for cost 11 and only 2 marks for any higher cost.

$$\text{LINUX} \xrightarrow[\text{Insert D}]{\text{cost 2}} \text{DLINUX} \xrightarrow[\text{L} \to \text{O}]{\text{cost 1}} \text{DOINUX} \xrightarrow[\text{I} \to \text{U}]{\text{cost 1}} \text{DOUNUX}$$

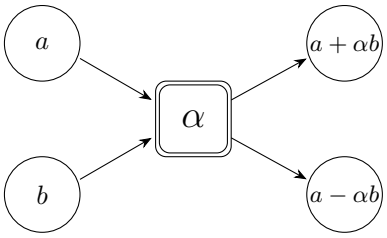

Figure 2: The operation that forms the building block of the FFT algorithm, represented as a gate labeled with a root of unity α

$$\text{DOUNUX} \xrightarrow[\text{N} \to \text{B}]{\text{cost } 1} \text{DOUBUX} \xrightarrow[\text{U} \to \text{T}]{\text{cost } 3} \text{DOUBTX} \xrightarrow[\text{Delete X}]{\text{cost } 2} \text{DOUBT}$$

The edits can be done in any order, not necessarily from left to right. There can be multiple different solutions. There is also a solution with cost 9, which also gets full marks.

Question 4. Suppose we want to evaluate a degree 7 polynomial $P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_7 x^7$ at the eighth roots of unity $1, \omega, \omega^2, \ldots, \omega^7$, where $\omega = e^{2\pi i/8}$. Consider an iterative implementation of the FFT algorithm. We will represent the algorithm as a circuit, with the gate shown in Figure 2. The gate labeled with α takes two numbers a and b (in that order) as inputs and outputs two numbers $a + \alpha b$ and $a - \alpha b$ (in that order).

The FFT circuit shown in Figure 3 shows the computation for evaluations of $P(x)$ at 8th roots of unity. The input to the algorithm will be the values in the leftmost circles, which are the coefficients of $P(x)$, in some order. The output will be the values in the rightmost circles, which will be the evaluations of the polynomial $P(x)$.

[8×1 **marks**] Write what should be the values of $j_0, j_1, j_2, j_3, j_4, j_5, j_6, j_7$ in Figure 3.

$j_0 = 0$,
$j_1 = 4$,
$j_2 = 2$,
$j_3 = 6$,
$j_4 = 1$,
$j_5 = 7$,
$j_6 = 3$,
$j_7 = 5$,

Question 5. Suppose we want to create a visual representation of a certain district-wise statistics, say number of healthcare workers (HW) per 10000 population. We want to color code districts according to this number. Say, for example, all districts with HW number between 25-30 will be colored dark blue and all districts with HW number between 31-33 will be colored light blue, and so on. The choice of these ranges will depend the distribution of the data. Given the number of colors k, we would like to find a good arrangement of data into k color classes. We will do it by ensuring that the variance in each color class is small.

Let $a_1, a_2, \ldots, a_n$ be the HW numbers for the n districts. Without loss of generality, let us assume $a_1 \leq a_2 \leq \cdots \leq a_n$. Then naturally, we want each color class to be of the form $\{a_i, a_{i+1}, \ldots, a_j\}$ for some i, j. Let $S_1 \cup S_2 \cup \cdots \cup S_k$ be such a partitioning of $\{a_1, a_2, \ldots, a_n\}$ into k color classes. Then, we define the following objective function:

$$\sum_{h=1}^{k} \left(\sum_{a \in S_h} (a - \mu_h)^2 \right),$$

where μ_h is the average of the numbers in S_h. We want to find the partition that minimizes the above objective function. Design an efficient algorithm that for given $a_1 \leq a_2 \leq \cdots \leq a_n$ and k, outputs the optimal partition, i.e., the ranges for color classes $S_1, S_2, \ldots, S_k$. What is the running time of your algorithm?

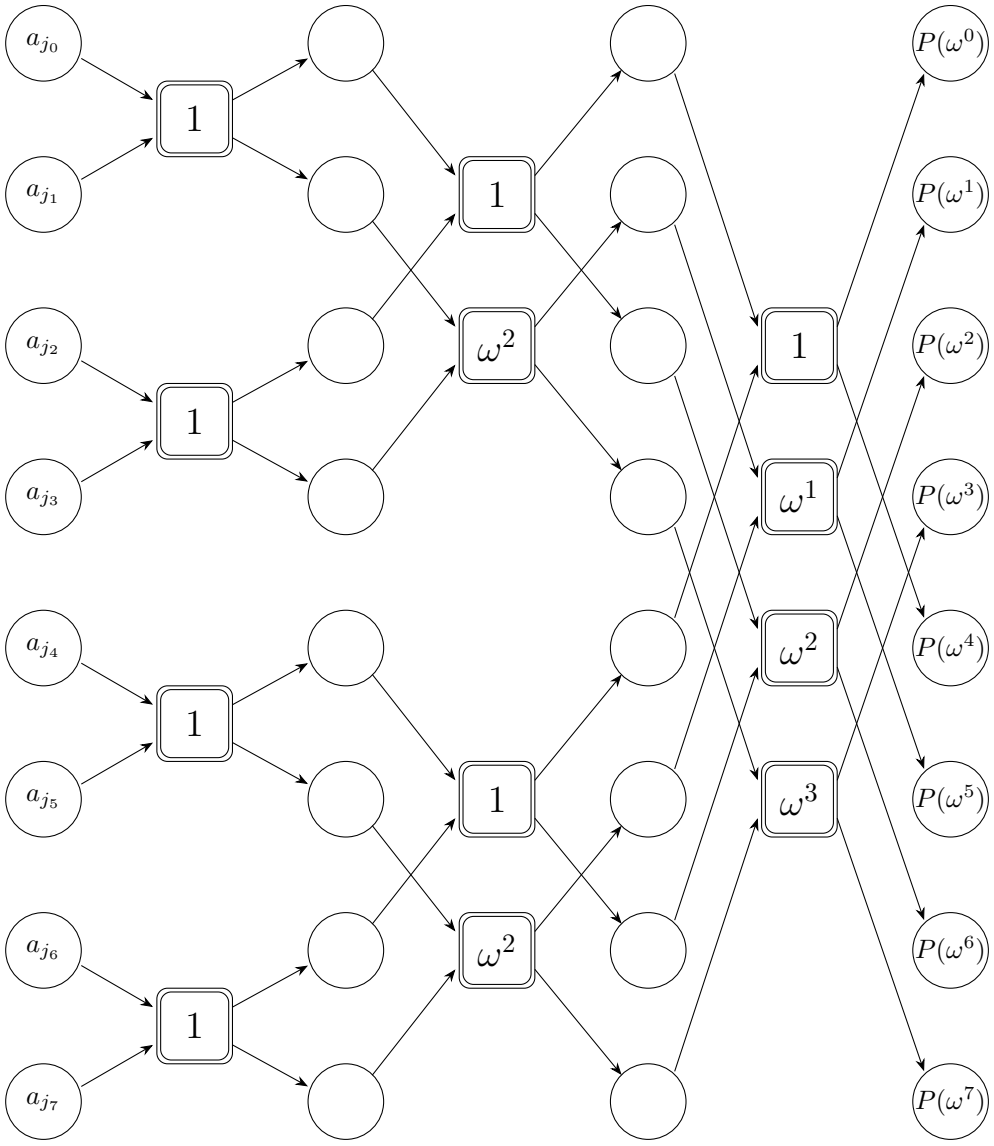

Figure 3: The FFT algorithm for a degree 7 polynomial $P(x)$ evaluated at 8th roots of unity. Here ω is the primitive 8th root of unity.

Useful to note: $\sum_{a \in S_h}(a - \mu_h)^2 = (\sum_{a \in S_h} a^2) - (\sum_{a \in S_h} a)^2/|S_h|$.

[**10 Marks**] for $O(n^2k)$ algorithm, only 8 marks for $O(n^3k)$, 6 marks for any larger polynomial time. If complexity analysis is not done for some part of the algorithm, we will assume whatever is the trivial upper bound.

Since it was not clarified in the question, there are two ways to interpret it: (1) the set S_js are all non-empty, i.e., we want exactly k colors and (2) the set S_js can be empty, which means we want at most k colors. Both interpretations are fine.

Let us first consider S_js as all non-empty. We can categorize the possible solutions $S_1 \cup S_2 \cup \cdots \cup S_k$ based on the number of elements in the last color, i.e., S_k. Let $C(n, k)$ be the optimal objective value for n data points and k color classes. Let $D(i, j)$ be the sum of squared deviations for the color class that contains

$a_i, a_{i+1}, \ldots, a_j$, that is,

$$D(i,j) = (\sum_{h=i}^{j} a_h^2) - (\sum_{h=i}^{j} a_h)^2/(j-i+1)$$

Then we can write

$$C(n,k) = \min_{\ell \geq 1}\{C(n-\ell, k-1) + D(n-\ell+1, n)\}.$$

In the bottom up implementation, we can compute $C(\cdot,\cdot)$ values in increasing order of n and k.

Let us first pre-compute the $D(i,j)$ values for all $i \leq j$. To do this, we just need to compute two terms $\sum_{h=i}^{j} a_h^2$ and $(\sum_{h=i}^{j} a_h)$ for each $i \leq j$. These can be computed in $O(n^2)$.

```
for i ← 1 to n {
    sum ← 0;
    sum_sq ← 0;
    for j ← i to n {
      sum ← sum +a_j;
      sum_sq ← sum_sq +a_j^2;
      D[i,j] ← sum_sq − sum^2/(j − i + 1);
      }
    }
```

Pseudocode is not expected here, but it should be clearly mentioned how all the sums can be computed in $O(n^2)$ time. If not explained, $O(n^3)$ will be assumed.

Now, we describe how to compute the $C(\cdot,\cdot)$ values. $C[i,j]$ will store the optimal objective value dividing first $i$points into j color classes. We will also store $L[i,j]$ for each i, j, the optimal choice of the number of words in the last cluster.

Initialization: $C[0,0] \leftarrow 0$; $C[0,j] \leftarrow \infty$ for $j > 0$;
for $i \leftarrow 1$ to n {
  for $j \leftarrow 1$ to k {
    $C[i,j] \leftarrow \min_{\ell \geq 1}\{C[n-\ell, k-1] + D[n-\ell+1, n]\}$.
    $L[i,j] \leftarrow$ the optimal choice of ℓ in the above line.
    }
  }

Once we have compute the $L[i,j]$ values, we can compute the optimal partition as follows.

$i \leftarrow n$;
for $j \leftarrow k$ to 1 {
  The jth color class S_j has $L[i,j]$ data points, that is, the points from $i - L[i,j] + 1$ to i.
  $i \leftarrow i - L[i,j]$
  }

---

2 marks for $O(n^2)$ time computation of $D[i,j]$. 4 marks for writing the correct recurrence. 2 marks for correct implementation for computing $C[i,j]$, including initialization. 2 marks for reconstructing the partition, using $L[i,j]$ values.

Here the implementation was from right to left. Alternatively, an implementation can be done from left to right. That is, categorization based on the number of points in the first color class.

Now, let's see an implementation for the second interpretation: the number of colors should be at most k.

Initialization: $C[0,0] \leftarrow 0$; $C[0,j] \leftarrow 0$ for $j > 0$;
for $i \leftarrow 1$ to n {
  for $j \leftarrow 1$ to k {

$C[i,j] \leftarrow \min_{\ell \geq 0}\{C[n-\ell, k-1] + D[n-\ell+1, n]\}$.
// minimization over $\ell \geq 1$ is also correct here.
$L[i,j] \leftarrow$ the optimal choice of ℓ in the above line.
}
}

Once we have compute the $L[i,j]$ values, we can compute the optimal partition as follows.

$i \leftarrow n$;
for $j \leftarrow k$ to 1 {
The jth color class S_j has $L[i,j]$ data points, that is, the points from $i - L[i,j] + 1$ to i.
$i \leftarrow i - L[i,j]$
}

Wrong algorithm: Many students have used the following idea, which is wrong. Below we show an example, where it doesn't give the optimal coloring.

For putting i data points in j color classes, we can consider two cases: (i) when the jth data points forms its own color class and (ii) when jth data point falls into the color class of the $j-1$th data point. We take the minimum of the two. Let $L[i,j]$ be the number of points in the last color, when compute the optimal coloring for first i points with j color classes.

$$C(i,j) = \min\{C(i-1,j-1), C(i-1,j) + D(i - L(i-1,j) + 1, i) - D(i - L(i-1,j) + 1, i-1)\}.$$

The problem with this update rule is that when we extend the optimal coloring for $(i-1,j)$ by just inserting the ith data point in the last color, it need not be optimal for first i points.

Example: 0, 2, 4, 6, 7, 10.

Observe that the optimal coloring for the six points is $(0,2,4),(6,7,10)$. But, the optimal coloring for the first 5 points is $(0,2),(4,6,7)$. Hence, the optimal coloring for the six points is not covered in either of the two cases.

no marks for this solution.

Question 6. Suppose you own a function hall for which you have received several booking requests in advance. Each request is asking for a one day booking and comes with an interval (i,j), which means, they need the hall for one of the days among day i, day $i+1$, ..., day j. Naturally, the hall can cater to only one of the requests on any particular day. You want to accept the maximum possible number of booking requests.

You want to design an $O(n \log n)$ time algorithm for this, where n is the number of requests. The input is $(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)$. The output should say which requests are accepted and which day is assigned to an accepted request.

Example 1. (1, 4), (1, 2), (1, 2). Here all requests can be accepted and days assigned can be $3, 1, 2$.

Example 2. (1, 3), (2, 3), (1, 2), (2, 2). Here, for example, first three requests can be accepted and days assigned can be $1, 3, 2$.

Example 3. (1, 4), (2, 3), (1, 3), (2, 2). Here, all requests can be accepted and days assigned can be $4, 3, 1, 2$.

Consider the following greedy ideas:

(i) Process the requests in increasing order of their starting days s_i (for tie breaks, use increasing order of interval lengths). If any day in the interval is available then assign the first available day for the request, otherwise reject this request.

(ii) Process the requests in increasing order of their ending days f_i (for tie breaks, use increasing order of interval lengths). If any day in the interval is available then assign the first available day for the request, otherwise reject this request.

(iii) Process the requests in increasing order of their interval lengths ($f_i - s_i + 1$) (for tie breaks, use increasing order of s_i). If any day in the interval is available then assign the first available day for the request, otherwise reject this request.

[2+2 marks] Two of these ideas don't work. For each of the two, give one example and show that the optimal solution is larger than what the greedy gives.

[2+4 marks] Which one is the correct idea? Prove its correctness.

You can prove correctness as follows: consider an optimal assignment of days to requests, say O^*. Modify the optimal solution so that it agrees with the first greedy step and remains optimal. You will need to consider two cases, the case when the first request is not assigned, and the case when it is assigned to a different day.

Rest of the proof is by induction, which is not expected here.

[5 marks] Now, consider another version of this problem. Each request also comes with an offer price p_i, that they are willing to pay for the booking. You want to maximize the total price you can get. Design a polynomial time algorithm (proof of correctness not required, but marks will be given only if the algorithm is correct).

Ideas (i) and (iii) don't work. Here are the respective counter examples.

- Example 3 from the question is a counter example for strategy (i).
- For strategy (iii), the counter example is $(3, 4), (1, 3), (1, 3), (1, 3)$.

The correct strategy is (ii). Correctness: let the optimal assignment of days to requests be O^*. The greedy algorithm first takes earliest ending request I_1 and assigns it the first available day, say d_1. Case one, when I_1 is not assigned any day in O^*. Then in O^*, the day d_1 must have been assigned to some other request ending after I_1, say I_2. We can remove d_1 from I_2 and assign it for I_1. Total number of assigned requests remain same.

Case two, when the request I_1 is assigned to a day d_2, which comes after d_1. Suppose there is another request I_2 assigned to day d_1. We know that ending day of I_2 is after I_1, and hence after d_2. Hence, it is possible to move request I_2 to day d_2 and move request I_1 to day d_1.

In both cases, the number of assigned requests remain same. And the modified solution agrees with the greedy first step. As mentioned in the question, the rest of the proof works by induction.

Part (b).

Algorithm 1: Greedy with greedy. Process the requests in decreasing order of prices. Let S be set of requests selected so far, which is initially empty. When we are at ith request, test whether it is possible to assign a day to each of the requests in $S \cup \{i\}$. If possible, the put request i in S, otherwise discard it.

How do we test if it is possible to assign a day to each of the requests in a given set. This can be done with the algorithm described in part (a), i.e., by sorting with respect to ending day.

Note that in this algorithm, we are not maintaining a list of assigned days to requests. We are just maintaining a list of requests, such that it is possible to assign days to all of them.
Pseudocode with two functions isSchedulable() and MaxPrice().

---

isSchedulable (list of requests S)

// takes a list of requests with intervals as input. It outputs yes if and only if it is possible to assign a distinct day to each request in S

- Sort the requests in S in increasing order of ending days (break ties arbitrarily).
- Process the requests in this order. For each request, assign it the first available day in its interval. If no day is available for a request, then return False;
- return True.

MaxPrice (list of requests L with prices)

// takes a list of requests with intervals and prices as input. It outputs the maximum possible total price that can be obtained by assigning distinct days to the requests.

- Sort the requests in L in decreasing order of prices (break ties arbitrarily).
- Initialize S as an empty set.
- for r in L

  if isSchedulable($\{r\} \cup S$) then $S \leftarrow S \cup \{r\}$
- return the total price of S.

Proof of correctness is left as an exercise.

Algorithm 2: augmentation

This algorithm is similar to the max flow algorithm (or the bipartite matching algorithm). This solution was not really expected.

Go over the requests in decreasing order of prices and maintain an assignment of days to requests. For a request r, assign an arbitrary day available in its interval. If no day is available then we can look for another request r' such that r' is currently assigned to a day d from the interval of r and moreover, r' has an unassigned day d' in its interval. If we are able to find such r' then we can re-assign day d to request r and assign day d' to request r'.

We need to generalize this idea and look for a possibly longer sequence of swaps to reach to an available day. This is called augmentation, which is formally defined as follows. We want to find a sequence of requests $(r = r_0, r_1, r_2, \ldots, r_k)$ and a sequence of days $(d_1, d_2, \ldots, d_{k+1})$ such that

- Day d_i is currently assigned to request r_i for $1 \leq i \leq k$.
- Day d_{k+1} is currently unassigned.
- Day d_i lies in the interval of request r_{i-1} for $1 \leq i \leq k+1$.

If we find this sequence then we can do a reassignment of days as follows: assign d_i to r_{i-1} for $1 \leq i \leq k+1$.

How do we find such an augmenting sequence? You can model this problem as finding a path (from r to an unassigned day) in a directed graph, and then, you can use any algorithm like DFS. If there is no such augmenting sequence, then we reject the current request.

The proof of correctness of this algorithm is similar (actually simpler) to the correctness of max flow algorithm.