COL351: Analysis and Design of Algorithms **Fall 2024**

Tutorial Sheet 6

Announced on: Aug 30 (Fri)

Problems marked with ($\star$) will not be asked in the tutorial quiz.

1. You are given n jobs, each with *length* ℓ_j and *deadline* d_j. Define the *lateness* λ_j of a job j in a schedule σ as the difference $C_j(\sigma) - d_j$ between the job's completion time and deadline, or as 0 if $C_j(\sigma) \leqslant d_j$.

   Which of the following greedy algorithms produces a schedule that minimizes the *maximum* lateness? Feel free to assume that there are no ties. In each case, either provide a counterexample to show incorrectness or a brief argument (2-3 sentences) to show correctness.

   a) Schedule the jobs in increasing order of length ℓ_j.

   b) Schedule the jobs in increasing order of slack $d_j - \ell_j$.

   c) Schedule the jobs in increasing order of deadline d_j.

   Assuming that only one job is executed at a time uninterruptedly till its completion.
   Note: We use the notation $(x)_+$ to denote $\max(x, 0)$

   a) Consider the following two jobs $\{j_1, j_2\}$ having the specifications: $\ell_1 = 10, d_1 = 10$ and $\ell_2 = 2, d_2 = 12$.

      Thus, the greedy schedule $\{j_2, j_1\}$ has maximum lateness $\max\{(2-12)_+, (12-10)_+\} = 2$, which is strictly worse than that of the optimal schedule $\{j_1, j_2\}$, namely $\max\{(10-10)_+, (12-12)_+\} = 0$.

   b) Consider the following two jobs $\{j_1, j_2\}$ having the specifications: $\ell_1 = 1, d_1 = 2$ and $\ell_2 = 10, d_2 = 10$. So, the slack for each job is: $\{d_1 - \ell_1, d_2 - \ell_2\} = \{1, 0\}$.

      Thus, the greedy schedule $\{j_2, j_1\}$ has maximum lateness $\max\{(10-10)_+, (11-2)_+\} = 9$.

      The optimal schedule $\{j_1, j_2\}$ has maximum lateness $\max\{(1-2)_+, (11-10)_+\} = 1$.

   c) Define an *inversion* in a schedule to be a pair of jobs j_1 and j_2 such that $d_{j_2} < d_{j_1}$ but j_1 is scheduled before j_2. A consecutive *inversion* is an *inversion* where the two jobs have been scheduled consecutively.

      Theorem 1: Swapping a consecutive *inversion* does not increase the maximum lateness.

Proof. Let $\sigma = \{..., p, q, ...\}$ be the schedule in which the jobs p and q represent a consecutive *inversion*. Thus, $d_q < d_p$.

Let C_p and C_q be their respective completion time before swap, and C'_p and C'_q be their respective completion time after swap. Since there is no idle time between any two consecutive jobs, we have $C_q = C'_p$. Also, $C'_q < C_q$ due to swapping.

Let L_p and L_q be their respective lateness before swap, and L'_p and L'_q be their respective lateness after swap. Then,

$$\begin{aligned}
L'_p &= (C'_p - d_p)_+ && \text{(from the definition of lateness)} \\
&= (C_q - d_p)_+ \\
&\leqslant (C_q - d_q)_+ \\
&\leqslant L_q && \text{(from the definition of lateness)} \\
\text{Also, } L'_q &= (C'_q - d_q)_+ && \text{(from the definition of lateness)} \\
&< (C_q - d_q)_+ \\
&< L_q && \text{(from the definition of lateness)}
\end{aligned}$$

□

Theorem 2: The greedy schedule is optimal.

Proof. (By contradiction): Let σ be the greedy schedule having no idle time and no *inversions*. Assume that σ is not optimal. Let σ^* be the optimal schedule that has the fewest number of *inversions* of all the optimal schedules and has no idle time.

Case 1: σ^* has no *inversions*:

Maximum lateness of σ = maximum lateness of σ^*. This contradicts the definition of σ^*.

Case 2: σ^* has an inversion. Let (p, q) be a consecutive *inversion*:

If we swap the jobs p and q in the schedule, the maximum lateness of the schedule does not increase (Theorem 1) and strictly decreases the number of *inversions*. This contradicts the definition of σ^*.

Thus, the greedy schedule σ is optimal. □

- Part (a):
  - Written ”I do not know how to approach this problem” - 0.2 points
  - Correct counterexample - 1 point
- Part (c):
  - Written ”I do not know how to approach this problem” - 0.4 points

- Mentioning it is correct - 0.5 points
  - Correct proof ideas - 1.5 points

2. How will your answer to Problem 1 change if, instead of maximum lateness, the goal is to minimize the *total* lateness $\sum_{j=1}^{n} \lambda_j$.

   For minimizing *total* lateness, all three proposals turn out to be suboptimal. In each case, there is a simple counterexample with only two jobs.

   a) **Scheduling by length is suboptimal**: Consider two jobs j_1 and j_2 with deadlines $d_1 = 5$ and $d_2 = 4$ and lengths $\ell_1 = 2$ and $\ell_2 = 3$, respectively. Scheduling in increasing order of lengths implies that j_1 precedes j_2, resulting in total lateness of $0 + 1 = 1$. However, scheduling j_2 before j_1 gives an improved total lateness of $0 + 0 = 0$.

   b) **Scheduling by slack is suboptimal**: Consider two jobs j_1 and j_2 with deadlines $d_1 = 3$ and $d_2 = 2$ and lengths $\ell_1 = 3$ and $\ell_2 = 1$, respectively. Scheduling in increasing order of slack implies that j_1 precedes j_2, resulting in total lateness of $0 + 2 = 2$. However, scheduling j_2 before j_1 gives an improved total lateness of $0 + 1 = 1$.

   c) **Scheduling by deadline is suboptimal**: Consider two jobs j_1 and j_2 with deadlines $d_1 = 1$ and $d_2 = 2$ and lengths $\ell_1 = 3$ and $\ell_2 = 1$, respectively. Scheduling in increasing order of deadlines implies that j_1 precedes j_2, resulting in total lateness of $2 + 2 = 4$. However, scheduling j_2 before j_1 gives an improved total lateness of $0 + 3 = 3$.

   Given the failure of the above strategies, one might wonder if there is some other greedy algorithm (or perhaps a non-greedy algorithm) that solves this problem in polynomial time. Turns out that minimizing the total lateness is an NP-hard problem [DL90]. Informally, that means we do not expect to have a polynomial-time algorithm for this problem unless a widely believed conjecture is proven false. We will study NP-hard problems later in the course.

3. You are given as input n jobs, each with a start time s_j and a finish time t_j. Two jobs *conflict* if they overlap in time—if one of them starts between the start and finish times of the other. The goal is to select a maximum-size subset of jobs that have no conflicts. (For example, given three jobs consuming the intervals $[0, 3]$, $[2, 5]$, and $[4, 7]$, the optimal solution consists of the first and third jobs.) The plan is to design an iterative greedy algorithm that, in each iteration, irrevocably adds a new job j to the solution-so-far and removes from future consideration all jobs that conflict with j.

Which of the following greedy algorithms is guaranteed to compute an optimal solution? Feel free to assume that there are no ties. In each case, either provide a counterexample to show incorrectness or a brief argument (2-3 sentences) to show correctness.

a) At each iteration, choose the remaining job with the earliest start time.

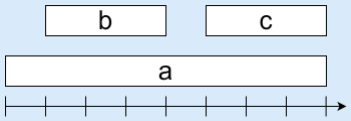
b) At each iteration, choose the remaining job with the earliest finish time.

c) At each iteration, choose the remaining job that requires the least time, i.e., with the smallest value of $t_j - s_j$.

d) At each iteration, choose the remaining job with the fewest number of conflicts with other remaining jobs.

For each counter-example, we have provided a job interval diagram in which the X-axis represents the time and there is a horizontal bar for each job. The left endpoint of this bar represents the start time and the right endpoint of this bar represents the finish time for the respective job.

a) Consider the following job interval diagram:



The optimum schedule {b, c} does not schedule the job 'a' having the earliest starting time.

b) Gives the correct answer. Let two jobs be feasible with each other if they don't *overlap* in time.

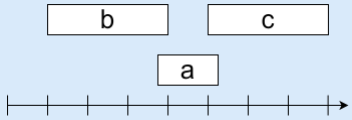
Proof. By contradiction: Assume that the greedy schedule is not optimal. Let the greedy schedule be $\sigma = \{g_1, g_2, ..., g_k\}$. Let $\sigma^* = \{o_1, o_2, ..., o_m\}$ be the optimal schedule, with the longest initial agreement of jobs with σ. Due to the assumed suboptimality of greedy, we have $m > k$, i.e., the optimal schedule includes strictly more jobs compared to the greedy schedule.

Assume that the first $r < k$ jobs scheduled are the same between σ and σ^*, i.e., $g_1 = o_1, g_2 = o_2, ...$ *and* $g_r = o_r$, and the schedules differ at $(r+1)^{th}$ job (**Try it yourself:** Why will such a job always exist?).

Since the job g_{r+1} has the earliest finish time feasible with $g_r = o_r$, it must finish before o_{r+1}, i.e., finish time of $g_{r+1} \leqslant$ finish time of o_{r+1}. This means that we can replace the o_{r+1} with g_{r+1} without decreasing the number of jobs scheduled and the newer optimal solution $\{g_1, g_2, ..., g_r, g_{r+1}, o_{r+2}, o_{r+3}, ..., o_m\}$ will remain feasible. Thus, this newer optimal schedule agrees with σ for the first $r+1$ jobs, which contradicts our assumption that σ^* was the optimal schedule with the longest initial agreement of jobs

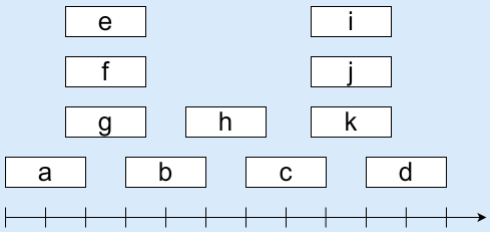
with σ. □

c) Consider the following job interval diagram:



The optimum schedule {b, c} does not schedule the job 'a' having the minimum execution time ($t_a - s_a$)

d) Consider the following job interval diagram:



The optimum schedule {a, b, c, d} does not schedule the job 'h' having the minimum conflicts.

a)
- Written "I do not know how to approach this problem" - 0.2 points
- Correct counterexample - 1 point

b)
- Written "I do not know how to approach this problem" - 0.4 points
- – Mentioning it is correct - 0.5 points
  – Correct proof ideas - 1.5 points

4. Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists a simple undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. That is, if $V = (v_1, v_2, \ldots, v_n)$, then the degree of v_1 should be d_1, the degree of v_2 should be d_2, and so on.

Informal Idea: To check if a list of numbers can represent the degrees of nodes in a simple undirected graph, we repeatedly remove the node with the largest degree d and attempt to connect it to other d nodes with the largest degree. We start by sorting the degree sequence in non-increasing order. Then, we remove the largest degree, say d, and subtract 1 from the d next largest degree values. We then recurse on the remaining values. If, at any stage, this process results in negative degrees or if there are not enough nodes to connect to, the sequence is declared invalid; otherwise, if all degrees eventually become zero, the sequence is declared valid. This algorithm is

known as Havel-Hakimi algorithm.

We call a list of n natural numbers $d_1, d_2, \ldots, d_n$ graphic if there exists a simple undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$.

Claim: Let $A = (s, t_1, \ldots, t_s, d_1, \ldots, d_n)$ be a finite list of nonnegative integers that is nonincreasing and $A' = (t_1 - 1, \ldots, t_s - 1, d_1, \ldots, d_n)$ be another finite list of nonnegative integers that is rearranged to be nonincreasing. Then, list A is graphic if and only if list A' is graphic.

Proof. ($\Leftarrow$) (as given here.)
Assume that $A' = (t_1 - 1, \ldots, t_s - 1, d_1, \ldots, d_n)$ is graphic. Then we add a new vertex v adjacent to the s vertices with degrees $t_1 - 1, \ldots, t_s - 1$ to obtain the degree sequence A.

($\Rightarrow$)
Notation: We will use the upper case of the variable denoting the degree, to denote that vertex, eg: We use S to denote the vertex that has degree s.
Assume that A is graphic. The following two cases are possible on the basis of which vertices are adjacent to the vertex S.

a) In the first case, S is adjacent to the vertices $T_1, \ldots, T_s$ in G. In this case, we remove S with all its incident edges to obtain the degree sequence A'.

b) In the second case, S is not adjacent to some vertex T_i for some $1 \leqslant i \leqslant s$ in G. Then we can change the graph G so that S is adjacent to T_i while maintaining the same degree sequence A. Since S has degree s, the vertex S must be adjacent to some vertex D_j in G for $1 \leqslant j \leqslant n$: Let the degree of D_j be d_j. We know $t_i \geqslant d_j$, as the degree sequence A is in non-increasing order.

   Since $t_i \geqslant d_j$, we have two possibilities: Either $t_i = d_j$, or $t_i > d_j$. If $t_i = d_j$, then by switching the places of the vertices T_i and D_j, we can adjust G so that S is adjacent to T_i instead of D_j. If $t_i > d_j$, then since T_i is adjacent to more vertices than D_j, let another vertex W be adjacent to T_i and not D_j. Then we can adjust G by removing the edges $\{S, D_j\}$ and $\{T_i, W\}$, and adding the edges $\{S, T_i\}$ and $\{W, D_j\}$. This modification preserves the degree sequence of G, but the vertex S is now adjacent to T_i instead of D_j.

   In this way, any vertex not connected to S can be adjusted accordingly so that S is adjacent to T_i while maintaining the original degree sequence A of G. Thus, any vertex not connected to S can be connected to S using the above method, and then we have the first case once more, through which we can obtain the degree sequence A'.

Hence, A is graphic if and only if A' is graphic. □

5. Consider the following change-making problem: The input to this problem is an integer L. The output should be the minimum cardinality collection of coins required to make L shillings of change, that is, you want to use as few coins as possible. The coins are worth $1, 5, 10, 20, 25$, and 50 shillings. Assume that you have an unlimited number of coins of each type. Formally prove or disprove that the greedy algorithm that takes as many coins as possible from the highest denominations correctly solves the problem. So, for example, to make a change for 234 Shillings, the greedy algorithm would require ten coins: four 50 shilling coins, one 25 shilling coin, one 5 shilling coin, and four 1 shilling coins.

   The greedy approach will not work here.

   Proof. By counterexample: Let $L = 40$, then the optimal solution is $\{20, 20\}$ i.e. 2 coins. But the greedy approach takes $\{25, 10, 5\}$. Thus, greedy approach takes a total of 3 different shilling coins making it not optimal for the given denomination of coins. □

6. Consider another change-making problem: The input to this problem is again an integer L, and the output should again be the minimum cardinality collection of coins required to make L nibbles of change (that is, you want to use as few coins as possible). Now the coins are worth $1, 2, 2^2, 2^3, \ldots, 2^{1000}$ nibbles. Assume that you have an unlimited number of coins of each type. Prove or disprove that the greedy algorithm that takes as many coins of the highest value as possible solves the change-making problem.

   Hint: The greedy algorithm is correct for one of the above two subproblems and is incorrect for the other. For the problem where greedy is correct, use the following proof strategy: Assume, to reach a contradiction, that there is an input I on which greedy is not correct. Let $\texttt{OPT}(I)$ be a solution for input I that is better than the greedy output $G(I)$. Show that the existence of such an optimal solution $\texttt{OPT}(I)$ that is different than greedy is a contradiction. So what you can conclude from this is that for every input, the output of the greedy algorithm is the unique optimal/correct solution.

   The greedy algorithm is correct, which we'll show from the following claims.

   Claim 1. *Given input L, the algorithm `OPT` can take any number of coins of denomination 2^{1000}, but it can take at most one coin of each of the denominations $1, 2, 2^2, 2^3, \ldots, 2^{999}$ to make L.*

   Proof. Assume, for the sake of contradiction, that there exists an input L where `OPT` selects two or more coins of some denomination 2^a for $a < 1000$.
   Now, consider replacing these two coins of denomination 2^a with a single coin of denomination 2^{a+1}. This results in a less number of coins, which contradicts the assumption that `OPT` is the optimal algorithm, as an optimal algorithm should minimize the number of coins. □

Claim 2. *The greedy algorithm stated in the problem is optimal.*

Proof. Assume our algorithm is not optimal. Let $\texttt{OPT}(L)$ be an optimal solution for input I and hence, it uses lesser coins than our greedy output $\texttt{G}(L)$. Let $\texttt{OPT}(L)$ and $\texttt{G}(L)$ use the coins $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_m)$ respectively, sorted in non-increasing order of their values. We know that $n < m$ and $\sum_{j=1}^{n} x_j = L = \sum_{k=1}^{m} y_k$. Consider the smallest index $i < n$ such that $x_i \neq y_i$ (**Try yourself**: if such an i doesn't exist then $G(L)$ is optimal). Then, it can be seen that $\sum_{j=i}^{n} x_j = \sum_{k=i}^{m} y_k$.
Following two cases are possible on index i:

a) $x_i < y_i$: As given, the coins are of the form 2^j where $0 \leqslant j \leqslant 1000$. This means that $\exists\ a$ and b such that $x_i = 2^a$ and $y_i = 2^b$ and $a < b$. We define *rem* as follows:

$$rem = L - (y_1 + y_2 + \cdots + y_{i-1}) = \sum_{k=i}^{m} y_k = \sum_{j=i}^{n} x_j \tag{1}$$

Since $G(L)$ chooses 2^b, we know that $rem \geqslant 2^b$. Note that $x_i = 2^a < 2^{1000}$ and hence, by Claim 1, $\texttt{OPT}(L)$ can use at most 1 coin each of denominations $\leqslant 2^a$. Thus, $\sum_{j=i}^{n} x_j \leqslant 2^0 + 2^1 + \cdots + 2^a = 2^{a+1} - 1 < 2^b \leqslant rem$. So, $\sum_{j=i}^{n} x_j < rem$, which is a contradiction to the Equation 1. Hence, our supposition was wrong.

b) $x_i > y_i$: This case is not possible because our greedy algorithm always chooses maximum coin which is upper bounded by the remaining value.

Therefore, greedy algorithm gives optimal solution. □

- Written "I do not know how to approach this problem" - 0.6 points
-
  - Mentioning Claim 1 - 1 point
  - Proof idea for Claim 1 - 0.5 points
  - High-level proof for optimality of greedy - 1.5 points

7. Suppose there are n agents and m items. The value of agent i for item j is given by a nonnegative integer $v_{i,j}$. An agent's value for a set of items is the sum of its values for individual items in that set. The goal is to partition the m items among the n agents in a *fair* manner.

   Denote an allocation by $A := (A_1, A_2, \ldots, A_n)$, where A_i is the subset of items assigned to agent i. We require that for any $i \neq k$, $A_i \cap A_k = \emptyset$ (i.e., items are not shared between bundles) and $\cup_i A_i$ is the entire set of items (i.e., no item is left unallocated). An allocation is deemed fair if, for any pair of agents i and k, the value derived by agent i from its bundle A_i is "within an item" of the value it derives from agent k's bundle A_k; specifically, for every pair of agents i and k and for every item $j \in A_k$, we have that $v_i(A_i) \geqslant v_i(A_k \setminus \{j\})$, where $v_i(S)$ denotes

the value of agent i for a subset S of items.

Design a polynomial-time algorithm for computing a fair allocation when the agents have *identical* valuations, i.e., item j is valued at $v_j \geqslant 0$ by every agent (though, for distinct items j and j', the values v_j and $v_{j'}$ may differ).[1]

We give an equivalent definition of a fair allocation, the proof of which is trivial (**Try it yourself**).

Claim 3. *An allocation A is fair if and only if for all $i, k \in [n]$:*

$$v(A_i) \geqslant v(A_k) - v_g, \textit{where } g = \operatorname*{argmin}_{j \in A_k} v_j$$

Note: Items are some times referred to as goods also.

Algorithm Description: We greedily allocate unassigned item with largest value to the agent with the lowest value of bundle assigned till now. Formally the greedy algorithm to achieve fair allocation is as follows:

ALGORITHM 1: Fair Allocation

Input: Agents $A = \{a_1, \ldots, a_n\}$, goods $G = \{g_1, \ldots, g_m\}$, value of goods $V = \{v_1, \ldots, v_m\}$
Output: Allocation $\mathcal{X} = \{X_i : i \in [n]\}$ which is fair

```
G' ← goods sorted in order of non-increasing value
𝒳 ← Initialize with empty sets for each agent
currValues ← min-heap with n zeros pushed along with the agent identity.
for g'_i ∈ G' do
    (val, agent) ← currValue.pop()
    X_agent ← X_agent ∪ {g'_i}
    currValue.push((val + v_{g'_i}, agent))
end
return 𝒳
```

It can be seen that the algorithm terminates. Now, we prove that the allocation given by the algorithm is indeed fair.

Loop Invariant: At the end of each iteration, the partial allocation (i.e., the allocation of items assigned till now) is fair.

Proof. Before the first iteration, the invariant holds as no agent has been assigned any items and hence, everyone has equal valued bundles (of value 0). To complete the proof, we now show that if the invariant holds till first x $(x \geqslant 0)$ iterations, then it holds after first $x + 1$ iterations too.

Terminology: We denote by g'_{x+1} the good assigned during the $(x+1)^{th}$ iteration. Also, we denote by $A^x = (A_1^x, A_2^x, \ldots, A_n^x)$ the partial allocation at end of x^{th} iteration. Then, by the algorithm description, g'_{x+1} is assigned to the agent having the least

[1] If you can show the existence of a fair allocation without the identical valuations assumption, please meet Rohit.

valued bundle in A^x (we refer to this agent has α_x). This gives us A^{x+1}. By our assumption, A^x is fair and now we need to show that A^{x+1} is fair.

As A^x is fair, we have for all $i, k \in [n]$:

$$v(A_i^x) \geqslant v(A_k^x) - v_g, \text{where } g = \underset{j \in A_k^x}{\operatorname{argmin}} v_j$$

Since $A_i^{x+1} = A_i^x \ \forall i \neq \alpha_x$, and $A_{\alpha_x}^{x+1} = A_{\alpha_x}^x \cup \{g'_{x+1}\}$, we get that for all $i \in [n], k \in [n] \setminus \{\alpha_x\}$:

$$v(A_i^{x+1}) \geqslant v(A_k^{x+1}) - v_g, \text{where } g = \arg\min_{j \in A_k^{x+1}} v_j \tag{2}$$

Now we show the following inequality for all $i \in [n]$:

$$v(A_i^{x+1}) \geqslant v(A_{\alpha_x}^{x+1}) - v_g, \text{where } g = \arg\min_{j \in A_{\alpha_x}^{x+1}} v_j \tag{3}$$

Proof. Since α_x is the agent with least valued bundle in A_x, we can say that $v(A_i^x) \geqslant v(A_{\alpha_x}^x), \ \forall i \in [n]$.
Therefore, for any agent $i \in [n]$ the following holds:

$$v(A_i^x) \geqslant v(A_{\alpha_x}^x) = v\left(\left(A_{\alpha_x}^x \cup \{g'_{x+1}\}\right) \setminus \{g'_{x+1}\}\right) = v(A_{\alpha_x}^{x+1} \setminus \{g'_{x+1}\}) \tag{4}$$

As the order of goods was in non-increasing order of their values in our algorithm, g'_{x+1} is the least valued good in $A_{\alpha_x}^{x+1}$. So, from Inequality 4, we get that for all $i \in [n]$, the following holds:

$$v(A_i^{x+1}) \geqslant v(A_{\alpha_x}^{x+1}) - v_{g'_{x+1}}, \text{where } g'_{x+1} = \arg\min_{j \in A_{\alpha_x}^{x+1}} v_j$$

This proves the Inequality 3 □

From Inequalities 2 and 3, we get that A^{x+1} is fair. This proves our loop invariant across the iterations.

□

As the loop invariant holds, we get that the allocation that we get after the iterations end is indeed fair.

Time Complexity: Sorting goods take $O(m \log m)$ time and for each iteration over the goods, $O(\log n)$ computation is made. Hence the total run time for the algorithm is $O(m(\log m + \log n))$ which is polynomial in input size. If the original ordering of goods is already sorted in decreasing order of their value then the time taken is $O(m \log n)$.

8. ($\star$) Imagine you have a set of n course assignments given to you today. For each assignment i, you know its deadline d_i and the time ℓ_i it takes to finish it. With so many assignments, it may not be possible to finish all of them on time. If you finish an assignment after its deadline, you get zero marks. Therefore, you must either complete the assignment by the deadline or not at all. How can you determine the maximum number of assignments you can complete within their deadlines?

High-level idea: We would try to accommodate as many assignment as possible within each deadline. To achieve this, we sort the assignments in increasing order of their deadlines and then one by one check if we can complete the assignment with current set of assignments. The algorithm was proposed by Pritesh Mehta (2022CS11916). Iterating over the assignments, we will claim that so far our set of assignments is optimal till now. This claim eventually will lead to the final set being optimal. Formally the algorithm is described below:

ALGORITHM 2: MaxScoringAssignments

Input: Set of assignments $A = \{a_i = (l_i, d_i) : i \in [n]\}$ sorted by deadlines $(d_i \leqslant d_j \forall i < j)$
Output: Maximum number of assignments that can be completed within deadline

```
currOptimal ← Max heap initially empty
sumHeap ← Sum of lengths in currOptimal, initially 0
for a_i = (l_i, d_i) in A do
    push l_i in currOptimal
    sumHeap ← sumHeap + l_i
    while sumHeap > d_i do
        maxLength ← currOptimal.pop()
        sumHeap ← sumHeap − maxLength
return size(currOptimal)
```

Proof: We have the following inductive claim after i iterations of the algorithm.

a) Size of the heap is the max number of assignments that could be completed within deadlines till d_i.

b) The total length of assignments in the heap is minimum among the other possible optimal answers so far.

Now now make the following observations:

a) At any step, the size of optimal solution and greedy solution weakly increases by at most 1.

b) At any step, if the size of optimal solution (or greedy solution) does not increase, then the total length of shortest length optimal solution (or greedy solution) weakly decreases.

c) **Lemma 1:** For any two schedule G and H such that $|G| = |H| - 1$ and $\mathsf{len}(H) - \mathsf{len}(G) \geqslant \mathsf{len}(\max(H))$, then $\exists h \in H$ such that $G \cup \{h\}$ is also a valid schedule, where $\max(H)$ represents the assignment with maximum length in H.

d) **Lemma 2:** If H is a shortest length valid schedule for size $|H|$ then $H \setminus \{\max(H)\}$ is a shortest length valid schedule for size $|H| - 1$.

Proof. **Lemma 1** can be shown by induction on size of H.
Base case is when $|H| = 1$ and only G that satisfies the conditions is $G = \phi$. Therefore we can add the singleton assignment present in H to G to make it a valid schedule.
Inductive step: Say the claim was true for any H, $|H| \leqslant k - 1$. Now for any H such that $|H| = k$ and for any G satisfying the conditions in the claim we show that the claim holds. We can order the assignments in increasing order of their deadlines. This would not change the feasibility (correctness) of the optimal schedule. Now, the assignment with farthest deadline among H, say a_j, is either in G or not. If it is in G then by induction hypothesis we can say that there exists $h \in H \setminus \{a_j\}$ such that $(G\{a_j\}) \cup \{h\}$ is a valid schedule. Otherwise we see that $G \cup \{a_j\}$ is a valid schedule.
As H is a valid schedule, we have

$$\mathsf{len}(H) \leqslant d_j \Rightarrow \mathsf{len}(H \setminus \{a_j\}) \leqslant d_j - \mathsf{len}(a_j)$$

Also, $\mathsf{len}(H) - \mathsf{len}(G) \geqslant \mathsf{len}(\max(H)) \geqslant \mathsf{len}(a_j) \Rightarrow \mathsf{len}(H \setminus \{a_j\}) \geqslant \mathsf{len}(G)$ and hence all the assignments in G have been completed at least within $\mathsf{len}(H \setminus \{a_j\})$ which shows that if a_j is valid in H then it would also be valid in $G \cup \{a_j\}$. Hence the claim is true. □

Proof. (**Lemma 2**) The proof for Lemma 2 follows from Lemma 1. Say for contradiction there exists a schedule G with size $|H| - 1$ and length less than that of $H \setminus \{\max(H)\}$. By Lemma 1, there exists $h \in H$ such that $G \cup \{h\}$ is a valid schedule with size $|H|$.

$$\mathsf{len}(G \cup \{h\}) = \mathsf{len}(G) + \mathsf{len}(h) < \mathsf{len}(H \setminus \{\max(H)\}) + \mathsf{len}(\max(H)) = \mathsf{len}(H)$$

This shows that H was not of shortest length and therefore contradicts our assumption. □

Coming back to our original claim, base case $i = 1$ is true as the only job that can potentially be successfully completed by the first deadline is the first job, and our algorithm explicitly checks for this condition in the while-loop. Observe that there always exists an optimal scheduling of assignments where assignments are done in increasing order of their deadline. (Why? Hint: Use exchange argument.)

Denote the solution maintained by our algorithm after i iterations by G_i, and let $\mathsf{len}(G_i)$ and $\mathsf{count}(G_i)$ respectively denote the total length and the number of assignments in G_i.

Let O_i be a schedule with the shortest total length among the set of schedules with the maximum number of assignments that could be completed before their respective deadline for the first i assignments.

Proof. Our claim is that $\mathsf{len}(G_i) = \mathsf{len}(O_i)$ and $\mathsf{count}(G_i) = \mathsf{count}(O_i)$. Say the claim is true after $i - 1$ iterations. For the i^{th} iteration, we have the following:

- Suppose no optimal scheduling with the shortest length contains the assignment a_i. Then $\mathsf{count}(O_i) = \mathsf{count}(O_{i-1})$ as all the assignments that are being completed within d_i are to be completed at least by d_{i-1} to get marks. According to our algorithm, $\mathsf{len}(G_{i-1}) \leqslant d_{i-1}$. Therefore $\mathsf{len}(G_{i-1}) \leqslant d_i$ and $\mathsf{count}(G_i) \geqslant \mathsf{count}(G_{i-1}) = \mathsf{count}(O_{i-1}) = \mathsf{count}(O_i)$. Also by definition of optimal answer we have $\mathsf{count}(G_i) \leqslant \mathsf{count}(OPT_i)$. This shows that $\mathsf{count}(G_i) = \mathsf{count}(O_i)$. Also, as a_i is not in the optimal answer then $\mathsf{len}(O_i) = \mathsf{len}(O_{i-1})$. One can show with similar reasoning that $\mathsf{len}(G_i) = \mathsf{len}(O_i)$. This show that the claim holds true in this case even for i.

- Now, say there exists a scheduling with shortest length containing a_i. There are again two cases:

  a) If $\mathsf{count}(O_i) = \mathsf{count}(O_{i-1}) + 1$. Then, $d_i \geqslant \mathsf{len}(O_i) = \mathsf{len}(O_{i-1}) + l_i = \mathsf{len}(G_{i-1}) + l_i$. This shows that adding the a_i to partial solution for $i-1$ does not exceeds the deadline and hence we get $G_i = G_{i-1} \cup \{a_i\}$ and $\mathsf{len}(O_i) = \mathsf{len}(G_{i-1}) + l_i = \mathsf{len}(G_i)$ and $\mathsf{count}(O_i) = \mathsf{count}(O_{i-1}) + 1 = \mathsf{count}(G_{i-1}) + 1 = \mathsf{count}(G_i)$.

  b) If $\mathsf{count}(O_i) = \mathsf{count}(O_{i-1})$. It is easy to show that our greedy approach will give a solution that has equal size as that of the optimal. Therefore, $\mathsf{count}(G_i) = \mathsf{count}(O_i)$ holds.
  Now, if the total length of optimal solution does not changes, then our algorithm is correct and $\mathsf{len}(G_i) = \mathsf{len}(O_i)$ is also satisfied. Say $\mathsf{len}(O_i) < \mathsf{len}(O_{i-1})$. By the optimality of O_i, we can say that $O_i \setminus \{a_i\}$ is a shortest length schedule of size $|O_i| - 1$ within assignments $A[1 : i-1]$. Also by lemma 2, we have that $G_{i-1} \setminus \{\max(G_{i-1})\}$ is a shortest length schedule of size $|G_{i-1}| - 1$ within assignments $A[1 : i-1]$. This implies that

  $$\mathsf{len}(G_{i-1} \setminus \{\max(G_{i-1})\}) = \mathsf{len}(O_i \setminus \{a_i\})$$

  $$\Rightarrow \mathsf{len}(a_i) - \mathsf{len}(\max(G_{i-1})) = \mathsf{len}(O_i) - \mathsf{len}(G_{i-1}) = \mathsf{len}(O_i) - \mathsf{len}(O_{i-1})$$

  As $\mathsf{len}(O_i) < \mathsf{len}(O_{i-1})$, we get $\mathsf{len}(a_i) < \mathsf{len}(\max(G_{i-1}))$. This shows that our greedy algorithm will not pop the i^{th} assignment from the heap in this case and therefore $a_i \in G_i$. This shows that

  $$\mathsf{len}(G_i) = \mathsf{len}(G_{i-1} \setminus \{\max(G_{i-1})\}) + \mathsf{len}(a_j) = \mathsf{len}(O_i \setminus \{a_i\}) + \mathsf{len}(a_i)$$

  $\Rightarrow \mathsf{len}(G_i) = \mathsf{len}(O_i)$. Hence proved.

□

9. ($\star$) Construct a five-symbol alphabet and an associated frequency distribution where the "top-down" Shannon-Fano encoding is suboptimal. You may construct the frequencies such that the encoding always finds an exact split.

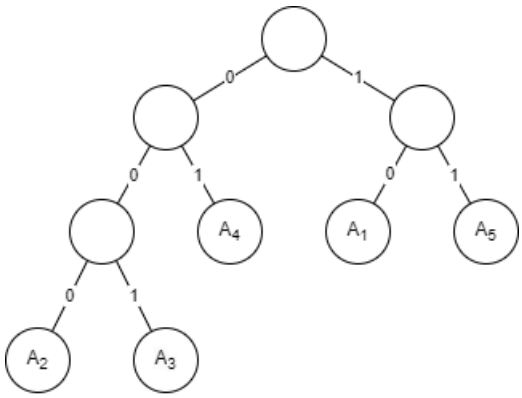


Figure 1: Σ-tree for Shanon-Fano

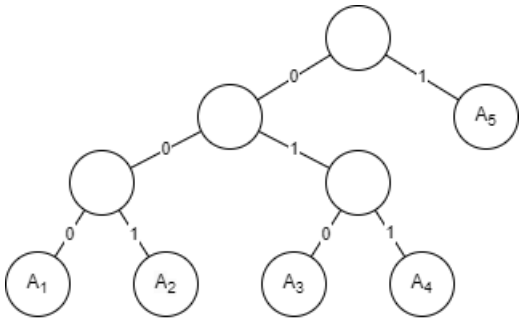


Figure 2: Σ-tree for Huffman

Consider the alphabet $\{A_1, A_2, A_3, A_4, A_5\}$ such that $f_1 = \frac{1}{18}, f_2 = \frac{3}{18}, f_3 = \frac{3}{18}, f_4 = \frac{3}{18}, f_5 = \frac{8}{18}$.
Then, It can be seen that Shannon-Fano encoding constructs the Σ-Tree as given in Figure 1, whereas an optimal Σ-Tree is given in Figure 1.
The average leaf depth of Shannon-Fano tree is:

$$\frac{1}{18} \times 2 + \frac{3}{18} \times 3 + \frac{3}{18} \times 3 + \frac{3}{18} \times 2 + \frac{8}{18} \times 2 = \frac{42}{18}$$

Whereas the average leaf depth of optimal Σ-Tree is:

$$\frac{1}{18} \times 3 + \frac{3}{18} \times 3 + \frac{3}{18} \times 3 + \frac{3}{18} \times 3 + \frac{8}{18} \times 1 = \frac{38}{18}$$

Hence, the Shanon-Fano encoding is suboptimal for the given instance.

References

[DL90] Jianzhong Du and Joseph Y-T Leung. Minimizing Total Tardiness on One Machine is NP-Hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.