**CS601 Algorithms and Complexity** **2021 Jul-Nov**

# Assignment 2 Solutions

*Total Marks: 50*

**Que 1 [5+5 marks].** We are given a binary encoding scheme for an alphabet of size $n$. We want to design an algorithm to test whether it is uniquely decodable or not. A code is said to be **not** uniquely decodable, if there are two different strings over the alphabet whose binary encodings are same. Consider two examples below.

- $A \to 11$, $B \to 1100$, $C \to 01$, $D \to 1001$, $E \to 101$.

  This code is uniquely decodable. Different strings over $\{A, B, C, D, E\}$ are mapped to distinct binary encodings.

- $A \to 11$, $B \to 110$, $C \to 01$, $D \to 1001$, $E \to 101$.

  This is not uniquely decodable. $ACD$, $BBC$ both have the same binary encoding 11011001.

Someone suggests us to build the following directed graph.

**Vertices.**

$$V = \{w \in \{0,1\}^* : w \text{ is a nonempty prefix or suffix of an alphabet encoding or } w \text{ is an alphabet encoding}\}.$$

To elaborate, the set of vertices will be corresponding to the set of prefixes and suffixes of the alphabet encodings. An alphabet encoding of length $\ell$ can contribute up to $2\ell - 1$ vertices: $\ell - 1$ prefixes, $\ell - 1$ suffixes, and the encoding itself. A binary string can be a prefix/suffix for multiple alphabet encodings, but there will be only one vertex for it. For the second encoding scheme above, the vertex set will have ten vertices labeled $\{1, 11, 10, 0, 110, 01, 100, 001, 1001, 101\}$.

**Edges.**

$$E = \{(w_1, w_2) : w_1 w_2 \text{ is an alphabet encoding}\} \cup \{(w_1, w_2) : w_1 = \alpha w_2 \text{ for some alphabet encoding } \alpha\}.$$

To elaborate, for a vertex labeled $w_1$ and a vertex labeled $w_2$, there is a directed edge from $w_1$ to $w_2$ if and only if any of the following is true

- $w_1 w_2$ is an alphabet encoding.
- $w_1 = \alpha w_2$ for some alphabet encoding $\alpha$.

For the second encoding scheme above, for example, there will be a directed edge from 10 to 01 (because 1001 is $D$). There will also be a directed edge from 110 to 0 (because 11 is $A$). And for example, there will be no edge from 100 to 0 (because neither 1000, nor 10 is an encoding for an alphabet).

- Prove that if there is a path (of nonzero length) from a vertex labeled $w_i$ to a vertex labeled $w_j$ such that both $w_i$ and $w_j$ are alphabet encodings then the encoding scheme is not uniquely decodable.
- Prove that if the encoding scheme is not uniquely decodable then there is a path (of nonzero length) from a vertex $w_i$ to a vertex $w_j$ such that both $w_i$ and $w_j$ are alphabet encodings.

**Ans 1.** In the question, we have confusingly used the word 'alphabet' to mean both, a set of characters and a character. In the following, characters will mean the elements of the alphabet. Word will mean a sequence of characters. String will mean a sequence of bits.

For any character $a$, let $\phi(a)$ be its binary encoding. For any word $S = a_1 a_2 \cdots a_\ell$ over the alphabet, let $\phi(S) = \phi(a_1)\phi(a_2)\cdots\phi(a_\ell)$ be its binary encoding. When a binary string $x$ is an encoding of a character, we denote the corresponding character by $\psi(x)$. That is, $\psi$ is the inverse map of $\phi$ for character encodings.

Suppose there is a path (of nonzero length) from a vertex labeled $w_i$ to a vertex labeled $w_j$ such that both $w_i$ and $w_j$ are character encodings. We will show that there are two different words $S_1$ and $S_2$ over the alphabet such that $\phi(S_1) = \phi(S_2)$. Let the path have vertices $w_i, w_{i+1}, w_{i+2}, \cdots, w_j$. To understand the simple case first, let us assume that all the edges on this path are of the first kind, i.e., $w_p w_{p+1}$ is a character encoding for every $i \le p \le j-1$. In that case, it is straightforward to get two different words: $S_1 = \psi(w_i)\ \psi(w_{i+1}w_{i+2})\ \psi(w_{i+3}w_{i+4})\ \cdots\ \psi(w_{j-1}w_j)$ and $S_2 = \psi(w_i w_{i+1})\ \psi(w_{i+2}w_{i+3})\ \psi(w_{i+4}w_{i+5})\ \cdots\ \psi(w_j)$. We are assuming $j - i$ is even (the case of odd is similar). Clearly $\phi(S_1) = \phi(S_2) = w_i w_{i+1} w_{i+2} \cdots w_j$.

Originally, the above was the simple proof I had in mind. But, at some point Kushagra pointed out that it's not enough to have just the first kind of edges. With the addition of second kind of edges, things become a bit more complicated. The construction of the two words is somewhat similar, but it's not as clean as above. We argue via induction. Without induction, perhaps it would be more intuitive, but requires too much notation.

Before proceeding to the general case, let's see an example. Let $A \to 11$, $B \to 11010$, $C \to 0$, $D \to 1011$. The corresponding graph will have a path $11 \to 010 \to 10 \to 11$. Here the first and the third edge are of the first kind, while the middle edge is of the second kind. To construct a binary string with two decodings, we should just put together all the vertices on the path, except the heads of second kind of edges. In the above example, this would be 11 010 11. The two decodings will be $ACD$ and $BA$.

Now let us describe the construction in general. We start with a slightly more general claim.

**Claim 1.1.** *If there is a path (of nonzero length) from a vertex labeled $w_i$ to a vertex labeled $w_j$ such that $w_i$ is a character encoding, then there exist two words $S_1$ and $S_2$ such that*

- $\phi(S_1) = \phi(S_2) w_j$
- *and the first characters in $S_1$ and $S_2$ are different.*

*Proof.* The proof will be based on a induction on the path length.

*Base case:* The path length is 1. That is, there is an edge from $w_i$ to $w_j$. The edge can mean one of two things: (i) $w_i w_j$ is an encoding of a character. In this case, define $S_1 = \psi(w_i w_j)$ and $S_2 = \psi(w_i)$. We get the claim. (ii) $w_i = w w_j$ where $w$ is a character encoding. Define $S_1 = \psi(w_i)$ and $S_2 = \psi(w)$. We get the claim.

*Induction hypothesis:* The claim is true for any path up to length $k-1$.

*Induction step:* Consider a path of length $k$, with vertices $w_i, w_{i+1}, \ldots, w_{i+k}$. By applying induction hypothesis on the path $(w_i, w_{i+1}, \ldots, w_{i+k-1})$ we get that there exist two words $S_1$ and $S_2$ such that

- $\phi(S_1) = \phi(S_2) w_{i+k-1}$
- and the first characters in $S_1$ and $S_2$ are different.

Now, the edge $(w_{i+k-1}, w_{i+k})$ can mean one of two things: (i) $w_{i+k-1}w_{i+k}$ is an encoding of a character. In this case, define $S_2' = S_1$ and $S_1' = S_2 \psi(w_{i+k-1}w_{i+k})$. Observe that

$$\phi(S_1') = \phi(S_2) w_{i+k-1} w_{i+k} = \phi(S_1) w_{i+k} = \phi(S_2') w_{i+k}.$$

Hence, the two words $S_1'$ and $S_2'$ satisfy the claim.

(ii) $w_{i+k-1} = w w_{i+k}$ where $w$ is a character encoding. Define $S_1' = S_1$ and $S_2' = S_2\ \psi(w)$. Observe that

$$\phi(S_1') = \phi(S_1) = \phi(S_2) w_{i+k-1} = \phi(S_2) w w_{i+k} = \phi(S_2') w_{i+k}.$$

Hence, the two words $S_1'$ and $S_2'$ satisfy the claim. $\square$

Now, from Claim 1.1, it is straightforward to prove our main statement. Take $S_1$ and $S_2$ as guaranteed by the claim. If $w_j$ is also a character encoding, then $S_1$ and $S_2$ $\psi(w_j)$ are two words which are different but have the same binary encoding $\phi(S_1) = \phi(S_2)w_j$.

Now, we prove the other direction. Suppose there are two different words with same encoding. Then we will show the desired path in the graph. Again it will be convenient to argue via induction. We make the following stronger claim, which will immediately imply what we want.

**Claim 1.2.** *Suppose there are two words $S_1$ and $S_2$ (with at least two characters in total) such that*

- *$\phi(S_1) = w_i\phi(S_2)$, where $w_i$ is the label of some vertex, and $w_i$ is not equal to the encoding of the first character of $S_1$.*

*Then there exists a path in the graph that starts from the vertex $w_i$ to a vertex $w_j$, where $w_j$ is a character encoding.*

*Proof.* The proof will be via an induction based on the total number of characters in $S_1$ and $S_2$.

*Base case:* When total number of characters in $S_1$ and $S_2$ is 2. This is possible in two ways (i) $S_1 = a$ and $S_2 = b$ and (ii) $S_1 = ab$ and $S_2 = \varepsilon$ (empty string). In case (i), we have $\phi(a) = w_i\phi(b)$. This means there will be an edge from $w_i$ to $\phi(b)$. That edge is the desired path. In case (ii) we have $\phi(a)\phi(b) = w_i$. This means there will be an edge from $w_i$ to $\phi(b)$. That edges is the desired path.

*Induction hypothesis:* Assume that the claim is true when the total number of characters in $S_1$ and $S_2$ is at most $k + \ell - 1$.

*Induction step:* We prove the claim for two words $S_1 = a_1a_2\cdots a_\ell$ and $S_2 = b_1b_2\cdots b_k$. By the assumption in the claim, $\phi(S_1) = w_i\ \phi(S_2)$. That means either $w_i$ is a prefix of $\phi(a_1)$ or $\phi(a_1)$ is a prefix of $w_i$ We consider both the cases one by one.

(i) $w_i$ is a prefix of $\phi(a_1)$. Let $\phi(a_1) = w_iw_{i+1}$. Clearly there is an edge from $w_i$ to $w_{i+1}$. If $w_{i+1}$ is a character encoding, then we already have our path. Otherwise define $S'_1 = a_2a_3\cdots a_\ell$. Observe that

$$w_i\ \phi(S_2) = \phi(S_1) = \phi(a_1)\phi(S'_1) = w_iw_{i+1}\ \phi(S'_1).$$

Hence, $\phi(S_2) = w_{i+1}\ \phi(S'_1)$. Total number of words in $S_2$ and $S'_1$ is $k+\ell-1$. Applying the claim inductively on the two words $S_2, S'_1$, we get that there is path from $w_{i+1}$ to $w_j$, where $w_j$ is a character encoding. Combining this path with the edge $(w_i, w_{i+1})$ gives a path from $w_i$ to $w_j$.

(ii) $\phi(a_1)$ is a prefix of $w_i$. Let $w_i = \phi(a_1)w_{i+1}$. Clearly there is an edge from $w_i$ to $w_{i+1}$. If $w_{i+1}$ is a character encoding, then we already have our path. Otherwise define $S'_1 = a_2a_3\cdots a_\ell$. Observe that

$$\phi(a_1)\phi(S'_1) = \phi(S_1) = w_i\ \phi(S_2) = \phi(a_1)w_{i+1}\ \phi(S_2).$$

Hence, $\phi(S'_1) = w_{i+1}\ \phi(S_2)$. Total number of words in $S'_1$ and $S_2$ is $k+\ell-1$. Applying the claim inductively on the two words $S'_1, S_2$, we get that there is path from $w_{i+1}$ to $w_j$, where $w_j$ is a character encoding. Combining this path with the edge $(w_i, w_{i+1})$ gives a path from $w_i$ to $w_j$. □

Finally we argue our main statement using Claim 1.2. Let there be two different words $b_1b_2\cdots b_k$ and $c_1c_2\cdots c_\ell$ which have the same encoding. Without loss of generality, $b_1 \neq c_1$. Define $S_1 = b_1b_2\cdots b_k$, $w_i = \phi(c_1)$, $S_2 = c_2\cdots c_\ell$. Clearly $\phi(S_1) = w_iS_2$. Hence, from Claim 1.2, we have a path from $w_i$ to $w_j$, where $w_i$ and $w_j$ both are character encodings.

**Que 2 [10 marks].** Recall the taxi scheduling problem discussed in the class. Suppose for the given set of bookings, minimum number of taxis required is $k$. Design an algorithm to find a 'bottleneck' of size $k$. That is, a set of $k$ bookings such that no two of them can be scheduled in the same taxi. Equivalently, an independent set of size $k$ in the given directed graph.

Hint: It is not an easy question. You might want to look at the algorithm for taxi scheduling more closely. You might want to first design an algorithm for finding a Hall's block in a bipartite graph (a subset $S$ of left vertices with $|N(S)| = |S| - k$).

**Ans 2.** Let there be $n$ bookings $B_1, B_2, \ldots, B_n$. Let us revisit a construction of a bipartite graph discussed in the class. The left side has $n$ vertices $b_1, b_2, \ldots, b_n$ and the right side has $n$ vertices $b'_1, b'_2, \ldots, b'_n$ (no extra vertices for taxis). We have an edge from $b_i$ to $b'_j$ if and only if booking $B_j$ can be served after booking $B_i$ in the same taxi.

We had claimed that if the minimum number of taxis required is $k$, then the maximum matching size in this bipartite graph is $n-k$. To see this, take any taxi allocation with $k$ taxis. For any $i, j$, if $B_j$ is allocated immediately after $B_i$ in the same taxi, then we include $(b_i, b'_j)$ in the matching. It is easy to see that it is indeed a matching. The size of the matching is $n-k$, because the unmatched vertices on the right side are exactly those bookings which are the first in their taxis.

For the other direction, take a matching with $r$ edges. The unmatched vertices on the right side are allocated to be the first bookings in their taxis. That is, we have $n-r$ taxis. For any $i, j$, If $(b_i, b'_j)$ is in the matching, then $B_j$ is allocated immediately after $B_i$ in the same taxi. This way we allocate all bookings with $n-r$ taxis.

**Bottleneck from Hall's block:** Once we have established this, it's easy to get a bottleneck from a Hall's block. We postpone the discussion on how to find a Hall's block. Hall's block: if in a bipartite graph, the maximum matching size is $n-k$, then there exists a set $S$ of left side vertices such that its neighborhood set $N(S)$ has only $|S| - k$ vertices. Let us propose the following bottleneck set $\mathcal{B}$ of bookings:

$$\mathcal{B} = \{B_i : b_i \in S, \text{ but } b'_i \notin N(S)\}.$$

Why is it a bottleneck set? Consider two bookings $B_i$ and $B_j$ in $\mathcal{B}$. We claim that $B_i$ and $B_j$ cannot be served by the same taxi. Because if $B_j$ can be served after $B_i$ in the same taxi then $b'_j \in N(b_i) \subseteq N(S)$. But, $b'_j$ cannot be in $N(S)$ by definition.

What is the size of the bottleneck $\mathcal{B}$? Observe that its size is at least $|S| - |N(S)|$ (because we are including $S$ and excluding $N(S)$). But, we know that this quantity is $k$.

**Construction of Hall's block:** Now, we describe how to construct Hall's block in a bipartite graph and that will finish the algorithm's description. We run the augmenting path algorithm on the bipartite graph to construct a maximum matching. Let its size be $n-k$. The left and right side both will have $k$ unmatched vertices each, let these sets be $U_L$ and $U_R$.

As usual, direct all the matching edges from right to left and direct all the non-matching edges from left to right. Start a BFS from all the vertices in $U_L$ and collect all the vertices that are reachable from any vertex in $U_L$. Let the set of reachable vertices on the left side be $Q_L$ (excluding $U_L$) and on the right side be $Q_R$. Note that $Q_R$ cannot contain any unmatched vertices, otherwise we would get an augmenting path from left to right. An augmenting path is not possible because we are already at a maximum matching. Hence, $Q_R$ only has matched vertices. Since matching edges go from right to left, all the matched partners of $Q_R$ are all reachable, i.e., they are in $Q_L$. But, anything in $Q_L$ can only be reached via their matched partners. Hence, we conclude $|Q_L| = |Q_R|$. Now, our set $S$ can be defined as $Q_L \cup U_L$. From the above discussion $N(S) = Q_R$. Hence, $|N(S)| = |S| - |U_L| = |S| - k$.

**Que 3 [10 marks].** Given a set of intervals, you need to assign a color to each interval such that any two intersecting intervals should have different colors. Consider the following algorithm for this problem.

1. Initialize $c \leftarrow 1$.
2. Find a largest set of disjoint intervals (can be done via the interval scheduling algorithm).
3. Assign color $c$ to each of these intervals and remove them.
4. If there are any intervals left then update $c \leftarrow c + 1$ and go to line 2.

Give an example where this algorithm fails to color with minimum possible number of colors. To convince the reader, please show the number of colors used by this algorithm on your example and also a better way of coloring.

**Ans 3.** Consider the example $(1,3),(2,9),(4,6),(7,15),(10,12),(14,17)$. We can color it with two colors: $(1,3),(4,6),(7,15)$ get one color and $(2,9),(10,12),(14,17)$ get the second color.

Now, let us see what will the algorithm give. First we want to find a largest set of disjoint intervals. For this, let's use the greedy algorithm for interval scheduling. First we need to sort the intervals in increasing order of ending times. We get $(1,3),(4,6),(2,9),(10,12),(14,17),(7,15)$. Now, greedily selecting a disjoint set of intervals, we get $(1,3),(4,6),(10,12),(14,17)$. So, these four intervals get color 1.

We are left with two intervals $(2,9),(7,15)$. We again find the largest set of disjoint intervals from these, which will simply have one interval $(2,9)$. The interval $(2,9)$ will get color 2. Finally, we are left with $(7,15)$, which will get color 3.

To conclude, this is an example where the algorithm uses 3 colors, but there is another coloring scheme with just 2 colors. Hence, the algorithm fails to give an optimal solution.

**Que 4 [10 marks].** There is an election with $N$ voters and two candidates. To predict the election result, you select a sample set of $k$ voters as follows:

---

$S \leftarrow$ set of voters
for $i = 1$ to $k$
  Choose a voter from $S$ uniformly randomly (i.e., each voter has probability $1/|S|$ of being chosen).
  Remove the chosen voter from $S$.

---

You assume that each chosen voter tells you their voting preference correctly and thus, you predict the candidate with the majority vote from the sample set as the winner.

Suppose $\epsilon N$ is the winning margin for the winner candidate in the actual election. Prove that if you want your prediction to be correct with probability at least $1 - \delta$, then it suffices to take $k = O(\frac{1}{\epsilon^2}\log(1/\delta))$.

**Ans 4.** Let us say $W$ is the set of voters who voted for the winner candidate in the actual election and $L$ is the set of voters who voted for the loser candidate. When we randomly sample $k$ voters as given in the algorithm, the probability that exactly $j$ sampled voters come from $L$ is

$$\frac{1}{N}\frac{1}{N-1}\cdots\frac{1}{N-k+1} \times \binom{|L|}{j}\binom{|W|}{k-j} \times k!.$$

Here the first product term is probability that a particular given sequence of $k$ voters is sampled. It follows by the number of ways to choose $j$ voters from the $L$, number of ways to choose $k-j$ voters from $W$ and the number of ways to arrange these $k$ voters. The probability is same as

$$\frac{\binom{|L|}{j}\binom{|W|}{k-j}}{\binom{N}{k}}.$$

Now, let us think about the case when our prediction is wrong. That will happen when $k/2$ or more sample candidates are from $L$. The probability of this happening is

$$\sum_{j=k/2}^{k} \frac{\binom{|L|}{j}\binom{|W|}{k-j}}{\binom{N}{k}}.$$

As the winning margin is $\epsilon N$, we have $|W| = N(1+\epsilon)/2$ and $|L| = N(1-\epsilon)/2$. The probability expression becomes

$$\sum_{j=k/2}^{k} \frac{\binom{N(1-\epsilon/2)}{j}\binom{N(1+\epsilon/2)}{k-j}}{\binom{N}{k}}.$$

On Moodle, it was said that this quantity can be taken as bounded by $e^{-\epsilon^2 k/2}$ (for a proof, see this `https://www.cse.iitb.ac.in/~rgurjar/CS601/HypergeometricTail.pdf`).

Hence we have that the probability of making a wrong prediction is at most $e^{-\epsilon^2 k/2}$. The question asks for prediction to be correct with probability at least $1-\delta$. That is, the probability of wrong prediction should be at most $\delta$. This can be ensured by choosing $k$ such that

$$e^{-\epsilon^2 k/2} \leq \delta.$$

This implies that $k \geq 2\frac{1}{\epsilon^2}\log(1/\delta)$ is good enough for us.

**Que 5 [10 marks].** There is a processor and $n$ jobs $\{J_1, J_2, \dots, J_n\}$ which can be potentially scheduled on it. For $1 \le i \le n$, the job $J_i$ has a processing time $t_i$ and a deadline $d_i$. If you schedule the job $J_i$ to start at time $t$ then it will finish on time $t + t_i$. The jobs can be processed only one at a time. Your goal is to maximize the number of jobs which can be scheduled before their deadlines.

**Note:** Clearly, a job should either be finished before its deadline or not scheduled at all. If there is a subset of jobs which is schedulable, then their schedule can be simply in increasing order of deadlines.

Design an efficient algorithm which takes $n$, processing times $\{t_i\}$, deadlines $\{d_i\}$ as input and outputs the maximum number of jobs schedulable before their deadlines.

The values of processing times and deadlines are quite large, so, it is undesirable to have the algorithm running time proportional to these values. Let $T = \sum_i t_i$ and let $D$ be the maximum deadline. Zero marks if the running time dependence is linear on $T$ or $D$. For full marks, your running time should be polynomial in $(n, \log T, \log D)$. Essentially, it means that you can add/compare the processing times and deadlines. But, you cannot run a loop with $D$ or $T$ iterations. If you think such an algorithm is not possible, then you can write that.

**Ans 5.** This question turned to be much more amazing then I thought earlier. We will discuss four algorithms for this problem, two DP and two greedy.

**DP 1 (zero marks for this solution).** As mentioned in the question, any subset of jobs that is schedulable, can be scheduled in increasing order of deadlines. So, let us sort the jobs in increasing order of deadlines and assume $d_1 \le d_2 \le \cdots \le d_n$. Define $f(i, t)$ to be the maximum number of jobs from first $i$ jobs $\{J_1, J_2, \dots, J_i\}$ that we can schedule and finish within time $t$. Let's try to define a recurrence for this function.

$$f(i,t) = \max \begin{cases} f(i-1,t) \\ f(i-1, t-t_i) + 1 \quad & \text{(consider only if } t \ge d_i) \end{cases}$$

Here we have partitioned the possible solutions into two classes: ones which contain $J_i$ and others which don't contain $J_i$. When $J_i$ is not included then we simply need to find maximum number of jobs from $\{J_1, J_2, \dots, J_{i-1}\}$ that can be finished within time $t$. When $J_i$ is included, then it has to be scheduled at the end because it has the highest deadline. This possibility should be consider only if $t \ge d_i$. Then we need to find the maximum number of jobs from $\{J_1, J_2, \dots, J_{i-1}\}$ that can finished within time $t - t_i$.

Base cases are left for you to fill up. Clearly, $f(n, D)$ is our final answer. The running time of the algorithm is proportional to $n \times D$, which is not desirable.

**DP 2.** Again sort the jobs in increasing order of deadlines and assume $d_1 \le d_2 \le \cdots \le d_n$. For $k \le i$, let $h(i, k)$ be the minimum total time taken by any $k$ schedulable jobs from the first $i$ jobs. Let's look at the following example.

| | | | | | | |
|---|---|---|---|---|---|---|
| Deadlines | 3 | 7 | 8 | 13 | 15 | 16 |
| Processing times | 2 | 6 | 2 | 4 | 3 | 1 |

In the example given above take $i = 4$ and $k = 3$. There are four subsets of size 3:

- $\{J_1, J_2, J_3\}$ is not schedulable because $t_1 + t_2 = 8$ is more than $d_2 = 7$.
- $\{J_1, J_2, J_4\}$ is not schedulable because $t_1 + t_2 = 8$ is more than $d_2 = 7$.
- $\{J_1, J_3, J_4\}$ is schedulable and total time taken is $t_1 + t_3 + t_4 = 8$.
- $\{J_2, J_3, J_4\}$ is schedulable and total time taken is $t_2 + t_3 + t_4 = 12$.

So, among the schedulable subsets the minimum time taken is 8. Hence, $h(4, 3)$ is 8. If no subset of size $k$ from first $i$ is schedulable then define $h(i, k) = \infty$.

Let's try to define a recurrence relation for this function.

$$h(i,k) = \min \begin{cases} h(i-1,k) & \text{(consider only if } k \le i-1) \\ h(i-1,k-1) + t_i & \text{(consider only if } h(i-1,k-1) + t_i \le d_i) \end{cases}$$

Here we have partitioned the possible solutions into two classes: ones which contain $J_i$ and other which don't contain $J_i$. When $J_i$ is not included, then we simply need to find the minimum time for a set of $k$ schedulable jobs from first $i-1$. When $J_i$ is included, then it has to be scheduled at the end because it has the highest deadline. Then we need to consider minimum time over all sets of $k-1$ schedulable jobs from first $i-1$. and add $t_i$ to it. Note that this possibility can be considered only if $J_i$ can be finished within its deadline, i.e., $h(i-1, k-1) + t_i \leq d_i$.

*Initialization:* $h(i, 0) = 0$ for each $1 \leq i \leq n$. All other entries are initialized to infinity.

We will compute $h(n, k)$ for each $1 \leq k \leq n$. Output the maximum value of $k$ such that $h(n, k)$ is finite.

**Greedy 1.** The algorithm uses a subroutine to test if a set of jobs is schedulable, which is described below.

---

Sort the jobs in increasing order of their processing times and assume $t_1 \leq t_2 \leq \cdots \leq t_n$.
Maintain a set $S$ of schedulable jobs (initially empty).
for ($i = 1$ to $n$)
  if $\{S \cup J_i\}$ is **schedulable** then insert $J_i$ in $S$.

---

A set of jobs is said to be schedulable, if all the jobs in the set can be scheduled to finish within their deadlines. To test this, sort the jobs in increasing order of their deadlines. Say, the deadlines in this order are $\delta_1, \delta_2, \ldots, \delta_k$. Let the corresponding processing times be $\tau_1, \tau_2, \ldots, \tau_k$. The set of jobs is schedulable if and only if

$$\text{for each } 1 \leq j \leq k, \quad \tau_1 + \tau_2 + \cdots + \tau_j \leq \delta_j.$$

This works because if a set of jobs is schedulable then they can be scheduled in increasing order of their deadlines. The condition is just checking if the $j$-job finishes within its deadline.

**Proof of correctness for Greedy 1.** This proof was given by Aniruddha Joshi.

Recall that the jobs are sorted in increasing order of processing times i.e., $t_1 \leq t_2 \leq \cdots \leq t_n$. Let $S$ be the set of jobs computed by the algorithm. Let $O$ be an optimal size set of schedulable jobs. Suppose $S$ and $O$ agree on first $r-1$ jobs. That is,

$$S \cap \{J_1, J_2, \ldots, J_{r-1}\} = O \cap \{J_1, J_2, \ldots, J_{r-1}\}.$$

We consider two possibilities for the $r$-th job.

(i) $J_r \notin S$ and $J_r \in O$. Note that the algorithm will decide to not put $J_r$ into $S$ only if it was not schedulable together with the jobs selected so far. But, $O$ and $S$ agree on the jobs selected so far. Hence, $J_r$ cannot be in $O$.

This leaves us with the only possibility (ii) $J_r \notin O$ and $J_r \in S$. Now, let's try to insert $J_r$ in $O$. Since $O$ is an optimal size set, $O \cup \{J_r\}$ should not be schedulable. So, let us remove a job $J_k$ from $O \cup \{J_r\}$ which has $t_k > t_r$ and has the minimum deadline among such jobs. We claim that the new set $O' = O \cup \{J_r\} - \{J_k\}$ is again schedulable. Recall that for any given set of jobs, the best arrangement is in increasing order of deadlines. Arrange the jobs of $O$ in increasing order of deadlines. When we remove $J_k$ and add $J_r$, all the jobs having deadlines after $J_k$ will only see a decrement in their finish times (as $J_k$ is longer than $J_r$). So, we only need to worry about jobs in $O$ having deadlines before $J_k$. By choice of $J_k$, all these jobs are actually shorter than $J_r$. Recall that $S$ and $O$ agree on jobs shorter than $J_r$. And $J_r \in S$ means that $J_r$ was schedulable with all the jobs in $S$ which are shorter than $J_r$. So, these jobs will not violate their deadlines when we insert $J_r$. Hence $O'$ is schedulable.

To conclude, we constructed a new optimal size set $O'$ which agrees with $S$ on one more step. Repeating this argument again and again, we will get an optimal size set that agrees with $S$ on all jobs.

**Greedy 2.** This algorithm was told to me by Aniruddha. It is also known as Moore's algorithm.

For a set $S$ of jobs, $t(S)$ will denote the sum of processing times of the jobs in $S$.

---

Sort the jobs in increasing order of deadlines and assume $d_1 \leq d_2 \leq \cdots \leq d_n$.
for $i = 1$ to $n$:
  If $t(S) + t_i \leq d_i$
    then $S \leftarrow S \cup \{J_i\}$.
  Else
    Let $J_j$ be the job with maximum processing time in $S_i$.
    If $t_j > t_i$
      then $S \leftarrow S - \{J_j\} \cup \{J_i\}$.

---

To summarize, if $J_i$ can be inserted in $S$ without any deadline violation we do it. Otherwise we try to remove a job with maximum processing time from $S$, which is also larger than $J_i$ and replace it with $J_i$. Intuitively, $S$ is maintained to be the maximum size set of schedulable jobs that minimizes the total processing time.

**Proof of correctness for Greedy 2.** For a schedulable subset $S$ of jobs, let $t(S)$ denote its total processing time. For a given set of jobs, let $O_h$ be defined as the subset with exactly $h$ jobs, that is schedulable and minimizes the total processing time. That is, $O_h$ is the set of jobs such that

$$t(O_h) = \min_O \{t(O) : |O| = h \text{ and } O \text{ is schedulable}\}.$$

We first prove the following claim.

**Claim 1.3.** *Let $h \geq 2$. If the set $O_h$ exists then the set $O_{h-1}$ can be obtained by removing the largest processing time job from $O_h$.*

*Proof.* Let $J_p$ be the job with largest processing time in $O_h$. We want to show that $O_{h-1} = O_h - \{J_p\}$. For the sake of contradiction, let us assume that the two sets are different. Let $J_r$ be shortest job where the sets $O_{h-1}$ and $O_h$ disagree, i.e., $J_r$ is present in one but not in the other and $J_r \neq J_p$. Clearly, $t_r \leq t_p$.

Case (i): $J_r \notin O_{h-1}$ and $J_r \in O_h$. Let's try to insert $J_r$ in $O_{h-1}$. The union might not remain schedulable. So, let us remove another job $J_k$ from $O_{h-1} \cup \{J_r\}$ which has $t_k \geq t_r$ and has the minimum deadline among such jobs. We claim that the new set $O'_{h-1} = O_{h-1} \cup \{J_r\} - \{J_k\}$ is schedulable. Recall that for any given set of jobs, the best arrangement is in increasing order of deadlines. Arrange the jobs of $O_{h-1}$ in increasing order of deadlines. When we remove $J_k$ and add $J_r$, all the jobs having deadlines after $J_k$ will only see a decrement (or no change) in their finish times (as $J_k$ is longer than $J_r$). So, we only need to worry about jobs in $O_{h-1}$ having deadlines before $J_k$. By choice of $J_k$, all these jobs are actually shorter than $J_r$. Recall that $O_{h-1}$ and $O_h$ agree on jobs shorter than $J_r$. And $J_r \in O_h$ means that $J_r$ was schedulable with all the jobs in $O_{h-1}$ which are shorter than $J_r$. So, these jobs will not violate their deadlines when we insert $J_r$. Hence $O'_{h-1}$ is schedulable. But $t(O'_{h-1}) \leq t(O_{h-1})$. Hence $O'_{h-1}$ is also an optimal set of size $h-1$ and agrees with $O_h$ on $J_r$ .

Case (ii): $J_r \in O_{h-1}$ and $J_r \notin O_h$. By exactly the same arguments as above, we can find another job $J_k$ with $t_k \geq t_r$ such that $O'_h := O_h \cup \{J_r\} - \{J_k\}$ is schedulable. $O'_h$ is also an optimal set of size $h$ and agrees with $O_{h-1}$ on $J_r$ .

To conclude, we can transform $O_{h-1}$ and $O_h$ so that they remain optimal and agree on $J_r$. By repeatedly applying this argument, we can make $O_{h-1}$ and $O_h$ agree on all jobs other than $J_p$.

□

Now, using Claim 1.3, we can argue that algorithm Greedy 2 gives us an optimal solution. Let $S_i$ denote the maximum size set of schedulable set of jobs from the first $i$ jobs (sorted w.r.t. deadlines) which minimizes the total processing time. We inductively argue that after the $i$-th iteration, the algorithm has computed $S_i$.

If $|S_i| = |S_{i-1}| + 1 = h$, then clearly $J_i \in S_i$. Since $J_i$ has the largest deadline, it will be scheduled last in $S_i$. If $J_i$ can be included with some set of $h-1$ jobs from first $i-1$ jobs, then clearly it can be included with the optimal set of $h-1$ jobs, which is $S_{i-1}$. And, $S_i = S_{i-1} \cup \{J_i\}$. This is what the first condition in the algorithm checks.

Now, consider the case when $|S_i| = |S_{i-1}| = h$. There are two possibilities, $J_i$ might or might not be in $S_i$. If $J_i$ is not in $S_i$ then clearly $S_i = S_{i-1}$. Consider the possibility that $J_i \in S_i$. If $J_i$ can be included with some set of $h-1$ jobs from first $i-1$ jobs, then clearly it can be (and should be) included with the optimal set of $h-1$ jobs from the first $i-1$ jobs. Using Claim 1.3, the optimal set of $h-1$ jobs can be obtained by removing the largest job from the optimal set of $h$ jobs, which is $S_{i-1}$. To conclude, $S_i$ can be obtained by removing the largest job from $S_{i-1}$ and adding $J_i$.