CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 13

March 3, 2022

Instructor: Sepehr Assadi

1 Job Scheduling on a Single Machine

Let us consider another standard problem that admits a greedy solution.

Problem 1 (Job Scheduling on a Single Machine). Suppose we have a collection of n (computing) jobs and a single machine to process these jobs. For each job $1 \leq i \leq n$, we know that the *length* of processing the i-th job on the machine is some positive number $L[i]$ (think of the input being given as an array $L[1:n]$). We assume that all the lengths are *distinct*.

We want to find an ordering π (a schedule) to process these jobs on the machine: first process job $\pi(1)$, then job $\pi(2)$, and so on, until we process job $\pi(n)$ and finish. For any ordering π of jobs and any job i, we define the *delay* of job i as the time it took from the beginning until we finished processing job i. Our goal is to *minimize* the total delay of all jobs, denoted by $delay(\pi)$, which is

$$delay(\pi) = \sum_{k=1}^{n} \sum_{j=1}^{k} L[\pi(j)].$$

(This formula reflects the fact that for the first job, we have to "pay" $L[\pi(1)]$ in delay, for the second job, we pay $L[\pi(1)] + L[\pi(2)]$ in delay, and so on and so forth).

Example: Suppose we have 5 jobs with lengths $L = [3, 2, 5, 6, 1]$. Let us consider three different schedules for this problem:

- Schedule 1: Let us simply process the job in the same order they are presented, i.e., pick $\pi = (1, 2, 3, 4, 5)$. This would give us the total delay:

  $$delay(\pi) = 3 + 5 + 10 + 16 + 17 = 51.$$

  (the first job finishes at time 3 so has delay 3, the second job at time 5, so has delay 5 and so on).

- Schedule 2: We now consider a different schedule which switch the order of the last two jobs only, i.e., pick $\pi = (1, 2, 3, 5, 4)$ (this means that at the end we first run job 5 with length 1 and then run job 4 with length 6). This would give us the total delay:

  $$delay(\pi) = 3 + 5 + 10 + 11 + 17 = 46.$$

- Schedule 3: Finally, we consider the schedule which processes the job in the increasing order of their lengths, i.e., pick $\pi = (5, 2, 1, 3, 4)$ (here $\pi(i) < \pi(i+1)$ for all i). This would give us the total delay:

  $$delay(\pi) = 1 + 3 + 6 + 11 + 17 = 38.$$

As these examples suggest, processing the shorter jobs first seems to be the best strategy for this problem. We use this intuition to design our greedy algorithm.

Algorithm: The algorithm is simply as follows:

1. Sort the jobs in L based on their length in the increasing order (recall that $L[i]$'s are distinct).
2. Let π (i.e., the schedule) be this sorted ordering of the jobs.

Proof of Correctness: Let π^O denote an arbitrary optimal schedule and π^G denote the schedule output by our greedy algorithm. If $\pi^O = \pi^G$ we are done, so let us assume $\pi^O \neq \pi^G$.

The assumption $\pi^O \neq \pi^G$ implies that there exists an index i such that $L[\pi^O(i)] > L[\pi^O(i+1)]$ (if such an index does not exist, π^O would be sorted in the increasing order and since L-values are unique, it will be equal to π^G but we just assumed they are different). We can pick any such index i to act as our "first" difference between the optimal and the greedy solutions. After picking index i, we simply *flip* the place of $\pi^O(i)$ and $\pi^O(i+1)$ to obtain the new schedule O'. Formally, O' is defined such that $\pi^{O'}(j) = \pi^O(j)$ for all $j \notin \{i, i+1\}$ and $\pi^{O'}(i) = \pi^O(i+1)$ and $\pi^{O'}(i+1) = \pi^O(i)$. See Figure 1 for an illustration[1].

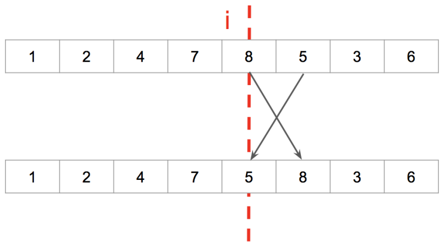

Figure 1: An example with 8 jobs with lengths in $\{1, 2, \ldots, 8\}$ (top schedule is O and the bottom one is O').

Let us now examine what happens to *delay*$(\pi^{O'})$ compared to *delay*(π^O). For concreteness, let $a = \pi^O(i)$ and $b = \pi^O(i+1)$ (so $a = \pi^{O'}(i+1)$ and $b = \pi^{O'}(i)$ also). An important observation is that O and O' differ from each other only in indices i, and $i+1$ which are flipped: this means that for every job other than a and b, the delay incurred by the job is exactly the same under both schedules. Moreover, in the new schedule O', job a is now postponed to after b, hence incurring *additional* delay of $L[b]$, while b is now done before a, hence, *reducing* its delay by $L[a]$. Since $L[a] > L[b]$ (recall the definition of the index i), this implies that the *total* delay of O' is now smaller than the delay of O by $L[a] - L[b]$.

At this point, we are done because we found a way to make O' more similar to the greedy solution G than O, without increasing the total delay. More formally, by repeating this process of finding a proper index i as above and flipping the jobs at positions i and $i+1$, we can get closer and closer to the sorted order of jobs and eventually sort the entire jobs in increasing order of their lengths as in G. Thus, our exchange argument proves the optimality of G.

However, it is worth mentioning that at this point (for this particular problem), we do not even need a full exchange argument: by the above argument, *delay*(O') is *strictly smaller* than *delay*(O); but this is a *contradiction* since we assumed O is optimal! This means that we should not have been able to find such an index i in the first place in the optimal solution O; this immediately means that O was in fact in the sorted order and hence was equal to G (this is often called *proof by contradiction*).

2 Maximum Disjoint Intervals

We now consider yet another standard problem that admits a greedy algorithm.

Problem 2 (Maximum Disjoint Intervals). We are given a collection of n intervals specified by their two endpoints: $[a_1, b_1], [a_2, b_2], \cdots, [a_n, b_n]$. We say that two intervals $[a_i, b_i]$ and $[a_j, b_j]$ are *disjoint* if there is no point that belongs to both of them. Our goal in this problem is to find the *maximum* number of intervals in the input that are all disjoint from each other.

[1]For the purpose of this example, let us ignore the fact that O is actually not optimal here.

Example: Figure 2 shows an example of an input and two different optimal solution to the problem.

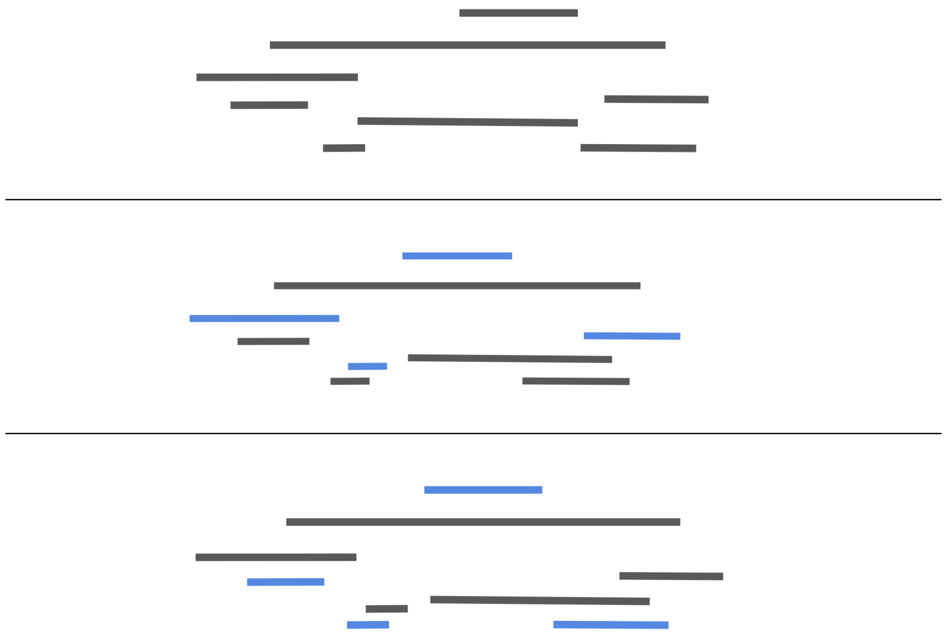

Figure 2: Note that for the purpose of clarity, each interval is drawn on a different level of the y-axis, even though in the problem, all intervals are defined over a single line (there is no y-axis in the problem).

Algorithm:

1. Sort the intervals based on their *last point* (i.e., b_i for interval $[a_i, b_i]$) in the non-decreasing order.
2. Pick the first interval in the solution. Go over this ordering until you find the next interval that does not intersect the previously chosen interval and pick it in the solution. Continue like this until you iterate over all the intervals.

Proof of Correctness: We have to prove both that the output of the algorithm is a valid solution (the intervals it outputs are all disjoint) and that it is also optimal (it contains the largest number of disjoint intervals).

Let $G = \{g_1, \ldots, g_\ell\}$ be the set of intervals chosen by the algorithm. By construction of the algorithm, we know that each g_i is disjoint from g_{i-1}. Thus, we only need to show that g_i is also disjoint from g_j for $j < i$ (we do not need to worry about $j > i$; do you see why?). This is true because g_i's starting point is larger than g_{i-1}'s endpoint, while g_{i-1}'s endpoint is larger than the endpoint of all intervals g_j for $j < i$; thus g_i is also disjoint from all g_j.

We now prove the optimality of the algorithm by an exchange argument. Let $O = \{o_1, \ldots, o_k\}$ be any optimal solution (we do not know a priori whether $\ell = k$ or not – in fact, this is what we want to prove).

If $G = O$ we are already done since it means the greedy algorithm is also optimal. So let us assume that $G \neq O$. We are going to also sort the intervals in O based on their endpoint so that $b_{o_1} \leq b_{o_2}, \ldots, b_{o_k}$; this clearly does not change the solution O and its size. Let j be the index where $g_1 = o_1, \ldots, g_{j-1} = o_{j-1}$ but $g_j \neq o_j$: note that such an index always exist since $G \neq O$ and also $j \leq \ell$ because if G and O are equal in their first ℓ choices, it cannot be the case that size of O is larger than ℓ because if there was an interval disjoint from $g_1, \ldots, g_j$ (in particular o_{j+1}), we would have picked it in G also.

We now define a new "solution" $O' = \{o'_1, \ldots, o'_k\}$ obtained from O by exchanging o_j with g_j, i.e., for all $i \neq j$, $o'_i = o_i$ but $o'_j = g_j$ instead. We call O' "solution" at this point because it is not clear yet whether it is indeed a valid solution to the problem or not: we need to ensure that all intervals in O' are disjoint. To prove this, note that since we only changed o_j to g_j, we need to prove $o'_j = g_j$ does not intersect with any other interval in O'. To prove this we have:

- g_j does not intersect with $g_1, \ldots, g_{j-1}$ by definition of the greedy algorithm.
- g_j has the smallest endpoint among all intervals that do *not* intersect with $g_1, \ldots, g_{j-1}$; this is again by definition of the greedy algorithm after we sort the intervals based on their endpoint.
- o_j do not intersect with $g_1, \ldots, g_{j-1}$ because O is a solution to the problem and $\{o_1, \ldots, o_{j-1}, o_j\} = \{g_1, \ldots, g_{j-1}, o_j\}$, so o_j cannot intersect with the previous intervals chosen in O.
- The two points above imply that endpoint of g_j can only be smaller than the endpoint of o_j. On the other hand, starting point of any interval $o_{j+1}, \ldots, o_k$ can only be larger than the starting point of o_j (otherwise they will intersect). This implies that g_j is also disjoint from $o_{j+1}, \ldots, o_k$.

The last two points above imply that O' is also a valid solution to the problem indeed. Now, since size of O and O' are the same, we have that O' is another optimal solution. But now O' has its first j indices equal to G (hence O' is "more similar" to G that O was). We are done now as we can repeat this argument again and again (exactly as before) until we end up with G itself (by changing all the ℓ indices of the solution to become equal to G): this proves the correctness of the algorithm.

Runtime Analysis: The algorithm involves sorting n intervals by their endpoints which can be done in $O(n \log n)$ time and then iterating over intervals in $O(n)$ time. Hence, the total runtime is $O(n \log n)$.