CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 12

October 17, 2019

Instructor: Sepehr Assadi

1 The Exchange Arguments: Recap

Let us recall the pattern in proving the correctness of a greedy algorithm, namely, the exchange argument:

(1) Fix an optimal solution O that is different from the greedy solution G.

(2) Find the "first" difference between this solution O and the greedy solution G.

(3) Show that exchanging the choice of the optimal solution with the choice of the greedy solution does not hurt the value of the solution, namely, the resulting solution O' would also be optimal.

(4) We will be done at this point because we can repeat this argument with O' and so on until the optimal solution is entirely exchanged with the greedy solution G – this means that G has to be optimal also since in every step of the way, we maintained that the intermediate solutions were optimal.

One rather tricky aspect of this argument is what should we choose as the "first" difference in Line (2). In general, there is no real recipe for this (exactly like there is no recipe for algorithm design or writing a proof!). However, typically the algorithm we design itself will suggest a way to do this. Remember that the only important thing in this step is to make sure by exchanging this particular difference between O and G, the resulting solution O' is *strictly more "similar"* to G than O; this will then ensure that by repeatedly applying the same argument again and again, we get more and more similar to G and will eventually end up with G itself.

2 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].

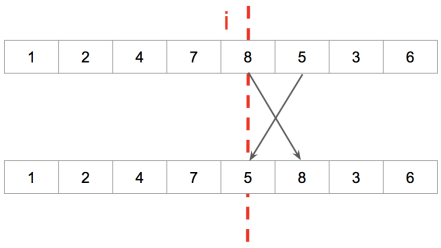

Figure 1: An example with 8 jobs with lengths in $\{1, 2, \ldots, 8\}$ (top schedule is O and the bottom one is O').

Let us now examine what happens to $delay(\pi^{O'})$ compared to $delay(\pi^O)$. 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

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

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*).

3 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.

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

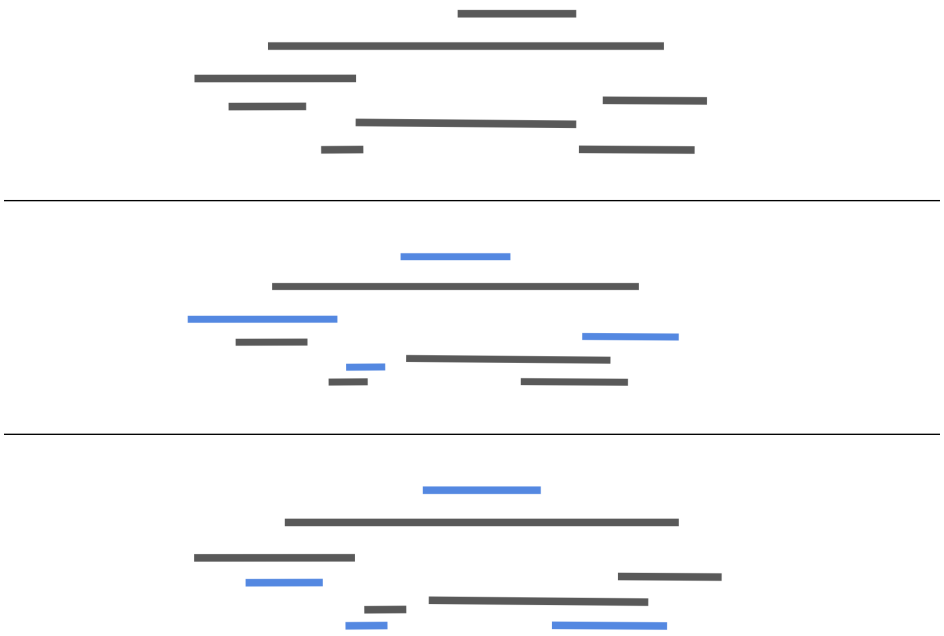

Figure 2: Note that for the purpose of clarity, each interval is drawn on a different level of the y-axis, even though in the problem, all intervals are defined over a single line (there is no y-axis in the problem).

We will go over an algorithm for this problem in the next lecture.