

## Lecture #7:

### 0.0.1 Greedy Methods:(Chapter 17)

We now take up the concept of the greedy paradigm. We will consider several illustrative examples.

- Making Change

Suppose we have the following coins: Dollar(100 cents), Quarter(25 cents), Dime(10 cents), Nickel(5 cents), and penny(1 cent). When we to pay some one an amount (say \$2.89=289 cents), we do this with little thinking in a way that in some cases uses the smallest number of coins. [Assume that we have sufficient quantities of each denomination). And the way we do it is known as the **greedy method**. We use 2 dollar coins, three quarters, one dime, and four pennies. If there was a half-dollar (=50 cents) coin, there are two possibilities – the greedy one is 2 dollar coins, one half-dollar coin, one quarter, one dime, and four pennies. So at each step, we use the largest denomination as many times as possible without going over the total. The decision made at each step is never revoked. These are some of the characteristics of greedy methods. **It does not provide the correct solution to every problem!** For example, convince yourself that when the available coins are: {50,25,20,10,5,1}, this method does not always work.

But it is useful in some instances. In some cases, there may be more than one way to solve using greedy ideas. Some may work while others may not.

- Activity Selection Problem (Chapter 17.1)

Suppose we have a set  $S = \{1, 2, \dots, n\}$  of  $n$  activities that wish to use a common resource. Activity  $i$  needs the resource in the time interval  $[s_i, f_i)$  with  $f_i > s_i$ . Two activities  $i$  and  $j$  are said to be *compatible* if their intervals do not overlap – that is if either  $s_i \geq f_j$  or  $s_j \geq f_i$ . The activity selection problem is to select the largest set of mutually compatible activities. Here is the pseudocode from the book:  $s$  and  $f$  are arrays. Assume that the activities are numbered so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .

**GREEDY-ACTIVITY-SELECTOR**( $s, f$ )

$n \leftarrow \text{length}(s)$

$A \leftarrow \{1\}$

$j \leftarrow 1$

**for**  $i = 2$  **to**  $n$

**do if**  $s_i \geq f_j$

```

    then  $A \leftarrow A \cup \{i\}$ 
          $j \leftarrow i$ 
return  $A$ 

```

Clearly the set  $A$  returned by the above algorithm is "feasible" – meaning that the intervals corresponding to activities in  $A$  do not overlap. What remains to be shown is that among such sets  $A$  has the largest size (such sets are called "optimal"). It is the proof of this that we do now.

1. We need to show that 1 should belong to some optimal set.

Proof: Suppose 1 does not belong to any optimal set. Let  $S$  be some optimal set (the existence of such a set is not in question since there are only finitely many feasible sets). Let  $k = \min\{j : j \in S\}$ . This is under the numbering as per an ordering that satisfies  $f_1 \leq f_2 \leq \dots \leq f_n$ . Then  $s_j \geq f_k \geq f_1$  for all  $j \in S - \{k\}$ . Moreover, there is no overlap in the intervals corresponding to activities in  $S - \{k\}$ . Thus  $S' = S - \{k\} + \{1\}$  is also feasible and  $|S'| = |S|$  and hence  $S'$  is also optimal. This contradicts the supposition that 1 does not belong to any optimal set.

2. Since 1 is a subset of some optimal set, eliminating any activity whose interval overlaps with any of the activities 1 does not exclude such optimal sets. Activity  $i \neq 1$  does not overlap with any activity in 1 implies that

$$\begin{aligned} s_i &\geq f_1 \text{ or} \\ s_1 &\geq f_i \end{aligned}$$

Since we know that  $f_i \geq f_1$  it follows that  $f_1 \geq s_i$ . Hence  $i \neq 1$  does not conflict with 1 implies that  $s_i \geq f_1$  and hence this step of the algorithm is correct.

3. Let  $T$  represent the set of activities which does not include any of activity 1 or any activity whose interval overlaps with that of 1. No activity in  $T$  can conflict with 1. Thus, an optimal set  $S$  that includes 1 has the property (by induction hypothesis) that  $S - \{1\}$  is optimal for  $T$ . And the above algorithm applied to  $T$  clearly produces  $S - \{1\}$ . Hence the algorithm "works".

{23}' Alternate: Step 1 is the same. Now suppose that the algorithm correctly produces  $A$  of some step – meaning that this  $A$  is a subset of some optimal set. Then excluding any activity that conflicts with any activity in  $A$  will not exclude such optimal sets. It is easy to verify that an activity (that has not already been excluded)  $i \notin A$  conflicts with some activity in  $A$  iff  $s_i \geq f_j$  where  $j$  is the last activity added to  $A$ . Thus the algorithm's next addition is also correct by reasoning similar to that in step 1.

- Huffman Codes (Chapter 17.3)

HUFFMAN( $C$ )

1.  $n \leftarrow |C|$
2.  $Q \leftarrow C$
3. **for**  $i \leftarrow 1$  to  $n - 1$
4.     **do** allocate a new node  $z$
5.          $left[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$
6.          $right[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$
7.          $f[z] \leftarrow f[x] + f[y]$
8.         INSERT( $Q, z$ )
9.     **return** EXTRACT-MIN( $Q$ )

- Minimum/Maximum Spanning Tree Problem
- Dijkstra's Algorithm in Shortest Paths

The discussion of the last two problems also comes under the category Graph Algorithms.

- Several Scheduling Problems

**Example 1** *A single server (such as a processor, a cashier in a bank, etc.) has  $n$  customers to serve. The service time required by each customer is known in advance: customer  $i$  will require  $t_i$  time units ( $1 \leq i \leq n$ ). We want to minimize the average time a customer spends in the system.*

Since the number of customers is fixed, this is equivalent to minimizing the total time spent by all customers. So we want to minimize  $\sum_{i=1}^n C_i$  where  $C_i$  is the time at which customer  $i$  completes service. Suppose  $n = 3$  and  $t_1 = 3; t_2 = 10; t_3 = 5$  the table below summarizes the values for all the ways in which we could render the service:

Order	$\sum_{i=1}^n C_i$
1, 2, 3	34
1, 3, 2	29 Optimal
2, 3, 1	43
2, 1, 3	41
3, 1, 2	31
3, 2, 1	38

Let  $P = p_1 p_2 \dots p_n$  be a permutation of  $\{1, 2, \dots, n\}$  and let  $s_i = t_{p_i}$ . If customers are served in the order corresponding to  $P$ ,

$$\begin{aligned} \sum_{i=1}^n C_i &= s_1 + (s_1 + s_2) + \dots + (s_1 + \dots + s_n) \\ &= \sum_{k=1}^n (n - k + 1) s_k \end{aligned}$$

This is minimized by the permutation that has  $s_1 \leq s_2 \leq \dots \leq s_n$ . This corresponds to the greedy strategy of processing the customer with the least time required first.

**Example 2** We have  $n$  jobs to execute, each one of which takes a unit time to process. At any time instant we can do only one job. Doing job  $i$  earns a profit  $p_i$ . The deadline for job  $i$  is  $d_i$ .

Suppose  $n = 4; p = [50, 10, 15, 30]; d = [2, 1, 2, 1]$ . It should be clear that we can process no more than two jobs by their respective deadlines. The set of feasible sequences are:

Sequence	Profit
1	50
2	10
3	15
4	30
1, 3	65
2, 1	60
2, 3	25
3, 1	65
4, 1	80 optimal
4, 3	45

A set of jobs is feasible if there is a sequence that is feasible for this set. The greedy method chooses jobs in decreasing order of their profits. If the inclusion of the next job does not affect feasibility, we do so. Else we do not include the last job that was considered. We need to show that this algorithm provides an optimal solution. We also need to consider the implementation that is most efficient. Some results given below help in this.

**Lemma 3** Let  $S$  be a set of jobs. Suppose that the jobs are numbered so that  $d_1 \leq d_2 \leq d_3 \leq \dots$ . (This is known as the Earliest Due Date order – EDD for short). The set  $S$  is feasible if and only if the sequence  $\{1, 2, \dots\}$  is feasible.

**Proof.** The "if" part is clear. To show that "only if" part we use proof by contradiction. Suppose that the sequence  $\{1, 2, \dots\}$  is NOT feasible. Let the first job in the sequence that misses its deadline be job  $r$ . This means that the deadline for job  $r$ ,  $d_r \leq r - 1$ . Since our sequence is EDD, the first  $r$  jobs all have deadlines  $\leq (r - 1)$ . However, these jobs are scheduled, at least one will not meet its deadline. Hence there is no feasible sequence for this set and hence for  $S$ . ■

This is good news – since we don't have to check all the permutations of this set to check feasibility.

**Theorem 4** The greedy algorithms solves the problem.

**Proof.** Suppose the greedy algorithm chooses the set  $I$  of jobs and suppose the set  $J$  is optimal. We want to show that the two have same profit and hence  $I$  is also optimal. Suppose that corresponding sequences are  $S_J$  and  $S_I$  respectively. These two sequences may have gaps – time slots where there are no jobs. By rearranging these sequences, we can get sequences  $S'_J$  and  $S'_I$  in which the jobs common to the sets  $J$  and  $I$  are scheduled in the same time slots as shown below:

p	y	q	x	r				$S_I$
r	s	t	p	u	v	q	w	$S_J$
x	y		p	r		q		$S'_I$
u	s	t	p	r	v	q	w	$S'_J$

These sequences are still feasible since both the previous ones were. This performs at most one exchange for each job that is common to the two sets. If the new sequences are not the same we may have one of the following possibilities:

- There is a job  $a$  in  $S'_I$  opposite a gap in  $S'_J$  : This means that  $a \notin J$ . But  $J \cup \{a\}$  is also feasible because we can put  $a$  in this gap and this would be more profitable than  $J$ . But this can not happen since  $J$  is optimal.
- There is a job  $b$  in  $S'_J$  opposite a gap in  $S'_I$  : This means that  $b \notin I$ . But  $I \cup \{b\}$  is also feasible because we can put  $b$  in this gap. This implies that the greedy algorithm would have included the job  $b$  as well. This is impossible since  $b$  was not included in the greedy schedule.

The only remaining possibility is that some job  $b$  is scheduled in  $S'_I$  opposite some job  $a$  in  $S'_J$ . In this case  $b \notin S'_J$  and  $a \notin S'_I$ . There are three cases to consider:

1.  $p_a > p_b$ : One could substitute  $a$  by  $b$  in  $J$  and we would get a higher profit – contradiction to optimality of  $J$ .
2.  $p_b > p_a$ : The greedy algorithm would have selected  $b$  before selecting  $a$  – contradicting the outcome of greedy method.

3.  $p_a = p_b$ : In this case the values are the same

Hence, both schedules have equal values jobs in each time slot and hence the total values are equal. Hence  $I$  is also optimal.

■

The following lemma helps in faster implementation of the greedy algorithm.

**Lemma 5** *A set  $J$  of  $n$  jobs is feasible if and only if we can construct a feasible sequence as follows: Start with an empty schedule with  $n$  time slots. For each job  $i \in J$ , schedule  $i$  at time slot  $t$ , where  $t$  is the largest integer such that  $1 \leq t \leq \min[n, d_i]$  and time slot  $t$  is free.*

In other words, schedule the next job as late as possible without violating its deadline. If a job can not be scheduled before its deadline under this algorithm, then there is no feasible sequence for the set  $J$ .

**Proof.** The "if" is obvious. For the "only if" part: Since there are only  $n$  jobs in all, no job needs to be scheduled later than time slot  $n$ . Thus, we can reset the deadlines to  $\min[n, d_i]$  for job  $i$ . When we try to add a job, there is always a "gap". Suppose we are unable to add a job whose deadline is  $d$ . This happens because all slots from  $t = 1$  to  $t = r$  (where  $r = \min[n, d]$ ) are taken by previous jobs. Let  $s > r$  be the smallest integer such that time slot  $t = s$  is free. Thus the schedule built up so far includes  $s - 1$  jobs whose deadlines are earlier than  $s$ , no job with deadline  $s$ , and possibly others with deadlines later than  $s$ . The job we are trying to add also has a deadline less than  $s$ . Thus,  $J$  includes at least  $s$  jobs with deadlines less than or equal to  $s - 1$ . Whatever be the schedule, at least one of these will be late. ■

This lemma suggests that we should take up the jobs in decreasing order of their profit values and schedule each job one by one to positions in a sequence of length  $n$ . For any position  $t$  define  $n_t = \max\{k \leq t : \text{position } k \text{ is free}\}$ . We want the next job  $i$  to be assigned to position  $n_t$  if  $t = \min[n, d_i]$ . We need some other data structures to do this right. We will take this up later when we do minimum spanning trees in Graph algorithms.

**Example 6** *We have  $n$  jobs to execute. We can do only one job at any time instant. Job  $i$  requires a duration  $t_i$ . The deadline for job  $i$  is  $d_i$ . We want to maximize the number of jobs done on or before their deadlines.*

Let the jobs be ordered in EDD order – numbered so that  $d_1 \leq d_2 \leq d_3 \leq \dots \leq d_n$ . Define sets  $I_k = \{1, 2, \dots, k\}$  with this numbering of jobs. If  $I_n$  is feasible, then it, clearly, is the solution. If not, let  $k_1$  be the minimum  $k$  for which  $I_k$  is not feasible. At least one of these jobs can not be included in the final solution. The most promising candidate is again found by the greedy method: it is the job  $k_2$  in the set  $I_k$  for which  $t_{k_2}$  is the largest. Job  $k_2$  is removed from the list and we repeat this process.

**Proof.** Suppose the optimal solution has the set  $J$  of jobs and let  $k_2 \in I = J \cap I_k$ . So our algorithm removed job  $k_2$  but the optimal solution has included that job. Clearly,  $I \neq I_k$  since this set of jobs is not feasible and hence any

set that properly contains this set can not be feasible as well. Let  $l \in I_k \setminus I$ . Consider the set  $J' = J \cup \{l\} \setminus \{k_2\}$ . Since  $t_l \leq t_{k_2}$ , the set  $J'$  is also feasible. ■

- A Loading Problem

**Example 7** Consider a train that travels from point 1 to point  $n$  with intermediate stops at points  $2, 3, \dots, (n - 1)$ . We have  $p_{i,j}$  packages that need to be delivered from point  $i$  to point  $j$  where  $1 \leq i < j \leq n$ . Packages have the same size. The maximum number at any point in the train must not exceed its capacity  $C$ . We want to deliver as many packages as possible.

For example, the following table gives data for  $n = 6$  with  $C = 8$ :

$\frac{\text{TO} \rightarrow}{\text{FROM} \downarrow}$	1	2	3	4	5	6
1		3	3	2	2	2
2			4	1	2	2
3				4	4	1
4					2	2
5						3
6						

We use the greedy idea twice in the solution of this problem. We describe the algorithm informally. At point 1 load packages in increasing order of their destination till capacity is reached. When the train arrives at point 2, (conceptually) unload and reload according to the same principle as above. If some packages that were picked up at point 1 get left at point 2, do not load them at point 1 in the first place! Here is the evolution of the algorithm for this example:

$\frac{\text{TO} \rightarrow}{\text{FROM} \downarrow}$	1	2	3	4	5	6
1		3	3	2		
2						
3						
4						
5						
6						

After Loading at Point 1

$\frac{\text{TO} \rightarrow}{\text{FROM} \downarrow}$	1	2	3	4	5	6
1		3	3	1		
2			4			
3						
4						
5						
6						

After Loading at Point 2

$\frac{TO \rightarrow}{FROM \downarrow}$	1	2	3	4	5	6
1		3	3	1		
			4			
				4	3	

After Loading at Point 3

$\frac{TO \rightarrow}{FROM \downarrow}$	1	2	3	4	5	6
1		3	3	1		
2			4			
3				4	3	
4					2	2
5						
6						

After Loading at Point 4

$\frac{TO \rightarrow}{FROM \downarrow}$	1	2	3	4	5	6
1		3	3	1		
2			4			
3				4	3	
4					2	2
5						3
6						

After Loading at Point 5

I will leave the proof to you as an exercise.

- Two Machine Scheduling Problem (S. Johnson)

**Example 8** Consider the following scheduling problem: We have two machines  $A$  and  $B$  (they perform different operations) and a set  $\{1, 2, \dots, n\}$  of  $n$  jobs. All jobs are processed by machine  $A$  first and then by machine  $B$ . Job  $i$  requires a duration  $A_i$  on machine  $A$  and  $B_i$  on machine  $B$ . We wish to minimize the time required to process all jobs and the corresponding schedule. S.M. Johnson gave a greedy algorithms to solve this problem.

- **Johnson Algorithm:**

Partition the jobs into two sets  $S_1 = \{i : A_i < B_i\}$  and  $S_2 = \{i : A_i \geq B_i\}$ . Schedule the jobs in  $S_1$  first in increasing order of their  $A_i$  values and then the jobs in  $S_2$  in decreasing order of their  $B_i$  values. Note that this can

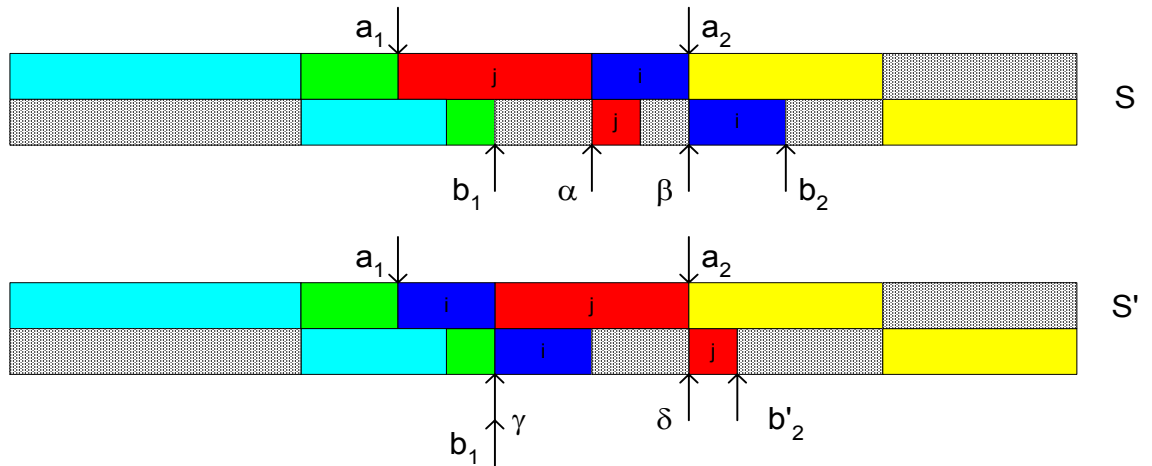
also be viewed as looking for the smallest of all durations and if this is an  $A$  type schedule the job first and if it is a  $B$  type schedule the job last. In this sense, this is a greedy method. The effort is  $\Theta(n \lg n)$ . To prove that this produces an optimal schedule, we use proof by contradiction.

**Proof.** Suppose this algorithm does not work for some problem instance. Let the optimal schedule for this be  $S$ . Since the algorithm does not work, there are two consecutive jobs say  $j$  followed by  $i$  in  $S$  such that one of the following conditions holds:

1.  $j \in S_2$  and  $i \in S_1$
2.  $j \in S_1; i \in S_1$  and  $A_j > A_i$
3.  $j \in S_2; i \in S_2$  and  $B_j < B_i$

■

It suffices to show that in any of these cases, if we exchange the positions of these jobs, we get a schedule that is at least as good as  $S$ . By repeated applications of these exchanges we get a schedule that would be obtained by the algorithm. This would be a contradiction to the statement that the algorithm does not work. See below:



It suffices to show that  $b'_2 \leq b_2$  to show that the schedule  $S'$  is at least as good as schedule  $S$ . Please note that the jobs after  $i$  in  $S$  can start at the same time or earlier on machine  $B$  in  $S'$  in this case. Let the job before  $j$  in  $S$  be  $k$

and the job after  $i$  be  $m$ .

$$\begin{aligned}
\alpha &= \max[b_1, (A_j + a_1)] \\
\beta &= \max[a_2, (B_j + \alpha)] \\
b_2 &= B_i + \beta \\
&= B_i + \max[a_2, B_j + \max[b_1, (A_j + a_1)]] \\
&= B_i + \max[a_2, B_j + b_1, B_j + A_j + a_1] \\
&\quad \max[a_1 + A_j + A_i + B_i, B_j + B_i + b_1, B_j + B_i + A_j + a_1] \\
\gamma &= \max[b_1, (A_i + a_1)] \\
\delta &= \max[a_2, (B_i + \gamma)] \\
b'_2 &= B_j + \delta \\
&= B_j + \max[a_2, B_i + \max[b_1, (A_i + a_1)]] \\
&= B_j + \max[a_2, B_i + b_1, B_i + A_i + a_1] \\
&= \max[a_1 + A_j + A_i + B_j, B_i + B_j + b_1, B_i + B_j + A_i + a_1]
\end{aligned}$$

The middle terms in  $b_2$  and  $b'_2$  are the same. Now we consider each of our cases above:

1. First term in  $b_2 = a_1 + A_j + A_i + B_i \geq B_i + B_j + A_i + a =$  Third term in  $b'_2$  because  $j \in S_2 \implies A_j \geq B_j$   
Third term in  $b_2 = B_j + B_i + A_j + a_1 \geq a_1 + A_j + A_i + B_j =$  First term in  $b'_2$  because  $i \in S_1 \implies A_i < B_i$ .  
Hence  $b_2 \geq b'_2$  in this case.
2. Third term in  $b_2 = B_j + B_i + A_j + a_1 > B_i + B_j + A_i + a =$  Third term in  $b'_2$  because  $A_j > A_i$   
Third term in  $b_2 = B_j + B_i + A_j + a_1 > a_1 + A_j + A_i + B_j =$  First term in  $b'_2$  because  $A_i < B_i$  since  $i \in S_1$   
Hence  $b_2 \geq b'_2$  in this case.
3. First term in  $b_2 = a_1 + A_j + A_i + B_i \geq B_i + B_j + A_i + a =$  Third term in  $b'_2$  because  $j \in S_2 \implies A_j \geq B_j$   
First term in  $b_2 = a_1 + A_j + A_i + B_i > a_1 + A_j + A_i + B_j =$  First term in  $b'_2$  because  $B_j < B_i$   
Hence  $b_2 \geq b'_2$  in this case.

By a similar argument, we can also show that if there are two possible schedules that could be obtained by the algorithm, they are equally good. This completes the proof that the algorithm works.

We will revisit greedy algorithms when we discuss graph algorithms.