

# COMS E6232 - Problem Set #2

Alex Wong (asw2181) - [asw2181@columbia.edu](mailto:asw2181@columbia.edu)

March 4, 2017

## Problem 1

a. Consider an instance where we have two items  $x$  and  $y$  where  $s_x = \epsilon$ ,  $v_x = 2\epsilon$ ,  $s_y = B$ ,  $v_y = B$ , and  $\epsilon \ll 1$ . The  $v_i/s_i$  ratio of item  $x$  is 2 and item  $y$  is 1. Thus, the Greedy algorithm will always pick item  $x$  regardless of how large  $B$  is and regardless how small  $\epsilon$  is. So as  $B$  gets larger and/or  $\epsilon$  gets smaller, the approximation ratio of the algorithm will increase, thus showing that the approximation ratio of Greedy is not bounded by any constant.  $\square$

b. **Theorem 1b** The Modified Greedy algorithm achieves approximation ratio 2.

**Proof:** Let  $OPT$  be the optimal solution that has maximum value where  $v(OPT) = \sum_{i \in OPT} v_i$  subject to  $\sum_{i \in OPT} s_i \leq B$ . We first assume that the items are ordered in a non-increasing fashion according to the ratio  $v_i/s_i$ . Let's call the first item that does not fit in the knapsack using the Greedy algorithm as item  $m$ . We know that item  $m$ 's  $v_i/s_i$  ratio  $\geq$  items  $m+1, \dots, n$ 's  $v_i/s_i$  ratio. Thus, if we are able to fit some fraction of item  $m$  so that it fills up to the capacity of the knapsack, that solution will be  $\geq OPT$ . We can define the fraction of item  $m$  that fits into the knapsack as  $\alpha$  where  $\alpha = (B - \sum_{i=1}^{m-1} v_i)/s_m$ . Thus,  $OPT \leq (\sum_{i=1}^{m-1} v_i) + \alpha v_m$ . Since  $\alpha$  is some fraction  $\leq 1$ , we can also say that:

$$OPT \leq (\sum_{i=1}^{m-1} v_i) + \alpha v_m \leq (\sum_{i=1}^{m-1} v_i) + v_m$$

From the inequality above,  $\sum_{i=1}^{m-1} v_i$  or  $v_m$  must be at least  $OPT/2$ , showing that the Modified Greedy algorithm will always get a solution at least  $OPT/2$ , thus achieving an approximation ratio 2.  $\square$

## Problem 2

- a. The lower bound for OPT is  $\max(\max_i(p_i), \frac{\sum_{i=1}^n p_i}{m})$ .

For  $m = 2$  machines and 5 jobs with processing times 3,3,2,2,2, the LPT algorithm will schedule 3,2,2 on  $m_1$  and 3,2 on  $m_2$ , giving a makespan of 7. For OPT, we know that a lower bound is  $\max = \max(3, 12/2) = 6$ . We can achieve OPT by scheduling 2,2,2 on one machine and 3,3 on the other machine.

For  $m = 3$  machines and 7 jobs with processing times 5,5,4,4,3,3,3, the LPT algorithm will schedule 5,3,3 on  $m_1$ , 5,3 on  $m_2$ , and 4,4 on  $m_3$ , giving a makespan of 11. For OPT, the lower bound is  $\max(5, 27/3) = 9$ . We can achieve OPT by scheduling 3,3,3 on one machine and 5,4 on each of the other two machines.

- b. If  $p_n > OPT/3$ , then  $3p_n > OPT$ , showing that no machine can process more than 2 jobs or else it would be  $> OPT$ , which would contradict OPT being the optimal solution. From this, we know that  $n \leq 2m$ , thus the largest  $m$  jobs will first get scheduled on each of the  $m$  machines and the rest of the  $n - m$  jobs will be assigned to the machine that has the least load at the point of assignment, thus showing that the LPT schedule is optimal.  $\square$

- c. **Theorem 2c** LPT achieves an approximation ratio of  $4/3$ .

**Proof:** Let's define job  $j$  as the job that finishes last in the LPT schedule and  $t_j$  as the span of time from time 0 until the time that job  $j$  starts. We know that in the timespan of  $t_j$ ,  $m \cdot t_j$  amount of processing has been done. This amount of processing can't be

more than the total amount of processing of all jobs, thus  $m \cdot t_j \leq \sum_{i=1}^n p_i \implies t_j \leq \frac{\sum_{i=1}^n p_i}{m}$ .

As we saw earlier in part a,  $\frac{\sum_{i=1}^n p_i}{m}$  is essentially a lower bound for OPT, thus we can say that  $t_j \leq OPT$ . Then by adding  $p_j$  to  $t_j$ , we get the makespan of the machine that has scheduled job  $j$ , and we can say that  $t_j + p_j \leq OPT + p_j$ . What the inequality tells us is that the processing time of job  $j$  indicates how well the LPT algorithm performs. Referring to part b, if  $p_j > OPT/3$ , the LPT schedule is optimal. But in the worst case, if  $p_j = OPT/3$ , then LPT will give an  $OPT + p_j = OPT + OPT/3 = 4/3OPT$  makespan, thus showing that LPT achieves an approximation ratio of  $4/3$ .  $\square$

- d. **Theorem 2d** The ratio of  $4/3$  of LPT is asymptotically tight as  $m \rightarrow \infty$ .

**Proof:** We generalize the examples of part a: given  $m$  machines, we have 3 jobs that have a processing time of  $m$ , and 2 jobs for each processing time  $2m-1, \dots, m+1$  (if  $2m-1 = m+1$ , then there are only 2 jobs for both  $2m-1$  and  $m+1$ ) giving us a total

of  $2m+1$  jobs. With LPT scheduling, we find that every machine will get scheduled two jobs with a total processing time of  $3m - 1$  with exception of the first machine that will get scheduled with 3 jobs with a total processing time of  $3m - 1 + m = 4m - 1$ . Thus, the makespan with LPT scheduling is  $4m - 1$ . For an optimal makespan  $OPT$ , we first schedule all 3 jobs with processing time  $m$  on the first machine, and then use LPT scheduling for the rest of the jobs. This will give each machine a total processing time of  $3m$ , thus making the makespan of  $OPT = 3m$ . Without loss of generality, the figure below illustrates the makespan of LPT and  $OPT$  of the example from part *a* when  $m = 3$ :

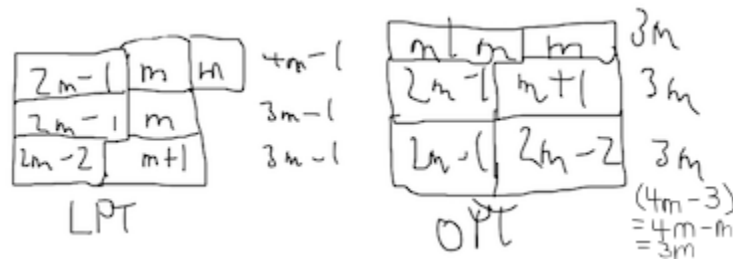


Figure 1: Makespan of LPT =  $4m - 1$ , Makespan of OPT =  $3m$

Thus, we see the ratio of  $LPT/OPT = (4m - 1)/3m = \frac{4m}{3m} - \frac{1}{3m} = \frac{4}{3} - \frac{1}{3m}$ , showing that the approximation ratio  $4/3$  is asymptotically tight as  $m \rightarrow \infty$ .  $\square$

## Problem 3

a. **Proof:** An optimal schedule will have all on-time jobs complete before all late jobs because the objective is to schedule jobs to maximize the total weight of the jobs that complete by their due date; any late jobs' weight will not count towards the total weight, essentially having a weight of 0 in the optimal schedule since they are late. We can also equally reframe the objective of maximizing weight of on-time jobs to minimizing weight of late jobs. By ordering jobs in an earliest due date order, we can achieve minimizing weight of late jobs, which means maximizing the total weight of on-time jobs, thus showing that an optimal schedule has all on-time jobs complete before late jobs and on-time jobs complete in an earliest due date order.  $\square$

b. There are  $n$  jobs where job  $j$  has processing time  $p_j$ , a weight  $w_j$ , and a due date  $d_j$ , where  $j = 1, \dots, n$ . Let  $W = \sum_j w_j$  which signifies the maximum possible value an optimal schedule can achieve, where every job completes on time. We use the following Dynamic Programming algorithm to compute the optimal schedule with a maximum total weight of jobs that complete by their due date:

### DP Scheduling Algorithm

**Preprocessing:** We first order the jobs by nondecreasing due date order, which takes  $O(n \log n)$  time, where  $n$  is the number of jobs to be scheduled. It will not affect the overall runtime since the overall runtime of the algorithm is  $O(nW)$  where  $W$  has a lower bound of  $n$  since each job has weight is a positive integer, making  $O(nW) \geq O(n^2)$ , which is  $> O(n \log n)$ .

**DP Initialization:** Let  $A$  be a table that stores values of minimum completion times. Let  $A(w, j)$  represent the minimum completion time of a subset of jobs  $1, \dots, j$  that are on-time and adding all on-time jobs' weights gives the total weight of  $w$ . If there is no feasible subset, then  $A(w, j) = \infty$ . We initialize  $A(0, j) = 0$ , for all  $j = 0, \dots, n$  because to get a total weight of 0, we do not need to have any on-time jobs, thus having a minimum completion time of 0. We initialize  $A(w, 0) = \infty$ , for all  $w = 1, \dots, W$  where  $W = \sum_j w_j$  because it is not feasible to get some total weight  $> 0$  without having a subset of jobs to schedule. Because we are doing constant work on  $n + W$  entries of table  $A$ , DP initialization has a runtime of  $O(n + W)$ .

### DP Recursion:

For  $1 \leq w \leq W$ ,  $1 \leq j \leq n$ , if job  $j$  is not an on-time job, we know then that the subset of jobs  $1, \dots, j - 1$  offers the minimum completion time at that weight  $w$ . If job  $j$  is an on-time job, as in  $A(w - w_j, j - 1) + p_j \leq d_j$ , then we take the minimum completion time between the minimum completion time of subset of jobs  $1, \dots, j - 1$  at weight  $w$  or the minimum completion time of subset of jobs  $1, \dots, j - 1$  at weight  $w - w_j$  since

jobs  $1, \dots, j-1$  will have at least  $w - w_j$  weight, added with the processing time  $p_j$  of job  $j$ . The following equation shows the recursion:

For  $1 \leq w \leq W, 1 \leq j \leq n$ :

$$A(w, j) = \begin{cases} \min(A(w, j-1), A(w - w_j, j-1) + p_j) & \text{if } w \geq w_j \text{ and } A(w - w_j, j-1) + p_j \leq d_j \\ A(w, j-1) & \text{otherwise} \end{cases}$$

For the runtime of DP recursion, we see that we do a constant amount of work for  $n \cdot W$  entries of table  $A$ , thus the runtime of DP recursion is  $O(nW)$ .

### DP Answer:

Let  $OPT$  be the maximum total weight of jobs that complete by their due date. After filling out table  $A$ , the maximum weight of on-time jobs is:

$$OPT = \max\{w \mid A(w, n) \neq \infty\}$$

which takes  $O(W)$  running time. Thus, the total running time is

$$O(n \log n) + O(n + W) + O(nW) + O(W) = O(nW)$$

□

### c. FPTAS for Scheduling Problem:

- Let's first define  $W' = \max_j(w_j)$ , where  $j = 1, \dots, n$ .
- Since  $W = \sum_{j=1}^n w_j$ , we can say that  $W \leq nW'$ .
- Assume, without loss of generality, that  $\epsilon \leq 1$ .
- Let  $K = \epsilon W' / 2n$ .
- Now, for each job  $j$ , let  $w'_j = \lfloor \frac{w_j}{K} \rfloor$
- Then  $W'' = \max_j(w'_j) = \lfloor \frac{W'}{K} \rfloor = \lfloor \frac{W'}{\epsilon W' / 2n} \rfloor = \lfloor \frac{2n}{\epsilon} \rfloor$

### Runtime:

Applying the DP algorithm from part *b* with these new weights  $w'_j$  and a new total max weight of  $nW''$ , with everything else remaining the same, we get a runtime of  $O(n \cdot (nW'')) = O(n \cdot (n \cdot \frac{2n}{\epsilon})) = O(n^3/\epsilon)$  which is a runtime that is both polynomial in  $n$  and  $1/\epsilon$ .

### Approximation Ratio:

Let's define  $Q$  as an optimal solution using the original weights and  $S$  as an optimal

solution using the new weights  $w'_j$ .

For each job  $j$ , we know that:  $\frac{w_j}{K} - 1 < w'_j \leq \frac{w_j}{K}$ . This implies that:

$$\frac{\sum_{j \in S} w_j}{K} - |S| < \sum_{j \in S} w'_j \leq \frac{\sum_{j \in S} w_j}{K}$$

For OPT, we know that:

$$OPT = w(Q) = \sum_{j \in Q} w_j \text{ which tells us } \frac{w(Q)}{K} - |Q| < w'(Q) \leq w'(S)$$

$$\implies w(Q) \leq K \cdot w'(S) + K \cdot |Q| \leq w(S) + K \cdot n$$

$$\implies w(Q) \leq w(S) + \frac{\epsilon}{2} W' \leq w(S) + \frac{\epsilon}{2} w(Q), \text{ because } W' \leq w(Q)$$

From here, we can say that:

$$w(Q) \leq w(S) + \frac{\epsilon}{2} w(Q)$$

$$\implies w(Q) - \frac{\epsilon}{2} w(Q) \leq w(S)$$

$$\implies w(Q) \cdot (1 - \frac{\epsilon}{2}) \leq w(S)$$

And since we are looking for the approximation ratio:

$$\frac{OPT}{w(S)} = \frac{w(Q)}{w(S)} \leq \frac{w(Q)}{w(Q) \cdot (1 - \frac{\epsilon}{2})} = \frac{1}{1 - \frac{\epsilon}{2}} \leq 1 + \epsilon, \text{ since } \epsilon \leq 1$$

Thus showing that the approximation ratio is  $1 + \epsilon$ .

**Conclusion:** We have provided a FPTAS by scaling and rounding the weights of each job, which resulted in a runtime polynomial in both  $n$  and  $1/\epsilon$  with an approximation ratio of  $1 + \epsilon$ .  $\square$

## Problem 4

a. To get a pseudopolynomial algorithm for the Assembly problem, we utilize dynamic programming.

### DP Assembly Algorithm

**Preprocessing:** We first define  $n$  as the number of all choices of all components, which equates to  $\sum_{i=1}^k |S(i)|$ . We then define a max cost  $C$  for our DP algorithm to run on by summing up the max cost of a choice from each set  $S(i)$  choices where  $i = 1, \dots, k$ . Thus,  $C = \sum_{i=1}^k \max_x (S(i)_x)$  for  $x = 1, \dots, |S(i)|$ . This takes  $O(n)$  runtime since we have to iterate through all choices of all components, which we have defined as  $n$ .

**DP Initialization:** Let  $A$  be a table that stores values of minimum weights. Let  $A(c, i)$  represent the minimum weight of a subset of components  $1, \dots, i$  that have a weight below  $W$  and adding all subset of components' costs gives the total cost of  $c$ . If there is no feasible subset, then  $A(c, i) = \infty$ . We first initialize  $A(0, 0) = 0$  because having 0 components with 0 costs means a 0 weight. We then initialize  $A(0, i) = \infty$ , for all  $i = 1, \dots, k$  because to we can not get a total cost of 0 for any number of components  $> 0$  since costs are positive integers. We initialize  $A(c, 0) = \infty$ , for all  $c = 1, \dots, C$  because it is not feasible to get some total cost  $> 0$  without having some subset of components to work with. Because we are doing constant work on  $k + C$  entries of table  $A$ , and  $k \leq n$  since each set of choices  $S(i)$  must contain at least one choice, DP initialization has a runtime of  $O(n + C)$ .

### DP Recursion:

For  $1 \leq c \leq C$ ,  $1 \leq i \leq k$ , we iterate through the set of choices  $S(i)$ , where each choice will be denoted  $S(i)_x$ ,  $x = 1, \dots, |S(i)|$ , to find the minimum weight choice that doesn't exceed the upper bound weight  $W$  when the weight  $w_{S(i)_x}$  is added to the subset of components  $1, \dots, i - 1$  and where the cost of the subset of components  $1, \dots, i - 1$  has cost  $c - c_{S(i)_x}$  so that if choice  $S(i)_x$  is included, the cost will equal  $c$ . The last constraint is that  $A(c - c_{S(i)_x}, i - 1) \neq \infty$  because that signifies that there wasn't a feasible subset from components  $1, \dots, i - 1$  so there is no point to add component  $S(i)_x$  since there wasn't any feasible subset before. If no choice  $S(i)_x$  meets the constraints, then  $A(c, i) = \infty$ . The following equation shows the recursion:



For  $1 \leq c \leq C$ ,  $1 \leq i \leq k$ :

$$A(c, i) = \min \begin{cases} A(c - c_{S(i)_1}, i - 1) + w_{S(i)_1} & \text{if } c \geq c_{S(i)_1}, \\ \cdot & A(c - c_{S(i)_1}, i - 1) \neq \infty, \\ \cdot & \text{and } A(c - c_{S(i)_1}, i - 1) + w_{S(i)_1} \leq W \\ \cdot & \\ \cdot & \\ A(c - c_{S(i)_{|S(i)|}}, i - 1) + w_{S(i)_{|S(i)|}} & \text{if } c \geq c_{S(i)_{|S(i)|}}, \\ & A(c - c_{S(i)_{|S(i)|}}, i - 1) \neq \infty \\ & \text{and } A(c - c_{S(i)_{|S(i)|}}, i - 1) + w_{S(i)_{|S(i)|}} \leq W \\ \infty & \text{otherwise} \end{cases}$$

For the runtime of DP recursion, we see that we do  $\sum_{i=1}^k |S(i)| = n$  amount of work for  $C$  number of rows of table  $A$ , thus the runtime of DP recursion is  $O(nC)$ .

#### DP Answer:

Let  $OPT$  be the minimum cost to assemble a product that meets the upper bound weight  $W$ . After filling out table  $A$ , the minimum cost to assemble a product is:

$$OPT = \min\{c \mid A(c, i)\}$$

If we find that  $OPT = \infty$ , then that means there was no combination of components that could make a product with a weight  $\leq W$ . The running time to find  $OPT$  takes  $O(C)$ . Thus, the total running time is

$$O(n) + O(n + C) + O(nC) + O(C) = O(nC)$$

which shows that the DP algorithm is a pseudopolynomial algorithm for the Assembly problem.  $\square$

#### b. FPTAS for Assembly Problem:

- Let's first define  $C'$  as the maximum cost of any choice of any component  $S(i)_x$ , where  $i = 1, \dots, k$  and  $x = 1, \dots, |S(i)|$ . Thus  $C' = \max_i(\max_x(c_{S(i)_x}))$ .
- Since  $C = \sum_{i=1}^k \max_x(c_{S(i)_x})$  for  $x = 1, \dots, |S(i)|$ , we can say that  $C \leq kC'$ .
- Assume, without loss of generality, that  $\epsilon \leq 1$ .
- Let  $K = \epsilon C' / 2k$ .

- Now, for each choice  $S(i)_x$ , let  $c'_{S(i)_x} = \lfloor \frac{c_{S(i)_x}}{K} \rfloor$
- Then  $WC' = \max_j(w'_j) = \lfloor \frac{W'}{K} \rfloor = \lfloor \frac{W'}{\epsilon W'/2n} \rfloor = \lfloor \frac{2n}{\epsilon} \rfloor$

**Runtime:**

Applying the DP algorithm from part *b* with these new weights  $w'_j$  and a new total max weight of  $nW''$ , with everything else remaining the same, we get a runtime of  $O(n \cdot (nW'')) = O(n \cdot (n \cdot \frac{2n}{\epsilon})) = O(n^3/\epsilon)$  which is a runtime that is both polynomial in  $n$  and  $1/\epsilon$ .

**Approximation Ratio:**

Let's define  $Q$  as an optimal solution using the original weights and  $S$  as an optimal solution using the new weights  $w'_j$ .

For each job  $j$ , we know that:  $\frac{w_j}{K} - 1 < w'_j \leq \frac{w_j}{K}$ . This implies that:

$$\frac{\sum_{j \in S} w_j}{K} - |S| < \sum_{j \in S} w'_j \leq \frac{\sum_{j \in S} w_j}{K}$$

For OPT, we know that:

$$OPT = w(Q) = \sum_{j \in Q} w_j \text{ which tells us } \frac{w(Q)}{K} - |Q| < w'(Q) \leq w'(S)$$

$$\implies w(Q) \leq K \cdot w'(S) + K \cdot |Q| \leq w(S) + K \cdot n$$

$$\implies w(Q) \leq w(S) + \frac{\epsilon}{2} W' \leq w(S) + \frac{\epsilon}{2} w(Q), \text{ because } W' \leq w(Q)$$

From here, we can say that:

$$w(Q) \leq w(S) + \frac{\epsilon}{2} w(Q)$$

$$\implies w(Q) - \frac{\epsilon}{2} w(Q) \leq w(S)$$

$$\implies w(Q) \cdot (1 - \frac{\epsilon}{2}) \leq w(S)$$

And since we are looking for the approximation ratio:

$$\frac{OPT}{w(S)} = \frac{w(Q)}{w(S)} \leq \frac{w(Q)}{w(Q) \cdot (1 - \frac{\epsilon}{2})} = \frac{1}{1 - \frac{\epsilon}{2}} \leq 1 + \epsilon, \text{ since } \epsilon \leq 1$$

Thus showing that the approximation ratio is  $1 + \epsilon$ .

**Conclusion:** We have provided a FPTAS by scaling and rounding the weights of each job, which resulted in a runtime polynomial in both  $n$  and  $1/\epsilon$  with an approximation ratio of  $1 + \epsilon$ .  $\square$