Assignment 1

Problem 1. ***The Make-span Problem***

1. Show that the standard greedy algorithm for the make-span problem is an $2-\frac{1}{m}$ approximation where m is the number of machines.

   Proof. Suppose we have machines $h_1, ..., h_m$ and jobs $x_1, ..., x_n$ where job x_i has duration t_i. Further, define C_{OPT} as the optimum make-span and C_i to be the make-span after adding jobs $x_1, ..., x_i$ under the greedy paradigm. Let $S = \left(\sum_{i=1}^{n} t_i\right)/m$ and $D = \max\{\max t_i, S\}$. Observe that $D \leq C_{OPT}$ (1).

   We prove that the greedy algorithm is a $2-\frac{1}{m}$ approximation, by contradicting the statement: $C_i \leq \left(2-\frac{1}{m}\right)\cdot D$ for every $1 \leq i \leq n$. Then by (1),

   $$C_i \leq \left(2-\frac{1}{m}\right)\cdot D \leq \left(2-\frac{1}{m}\right)\cdot C_{OPT}.$$

   In particular, when $i = n$, $C_n \leq \left(2-\frac{1}{m}\right)\cdot C_{OPT}$.

   Let j be the smallest index such that $C_j > \left(2-\frac{1}{m}\right)\cdot D$ and suppose that the greedy algorithm put x_j on machine h. The make-span, $t(h)$, on machine h satisfies $t(h) > \left(2-\frac{1}{m}\right)\cdot D$. Since $t_j \leq \max t_i \leq D$, the time on h before adding x_j must be strictly greater than $(1-1/m)D$. Then make-spans on each of the other machines is strictly less than $(1-1/m)D$ since

   $$t(h) + (m-1)\left(1-\frac{1}{m}\right)D > \left(2-\frac{1}{m}\right)D + \left(m-2+\frac{1}{m}\right)D = mD = S.$$

   By the pigeon hole principle, there exists some machine h' with make-span less than $(1-1/m)D$. This is a contradiction since the greedy algorithm puts x_j on the machine with smallest make-span, but h is not such a machine — h' has strictly smaller make-span. □

2. Argue for $m = 2$ (resp. $m = 3$) machines that *any* (not necessarily greedy) online algorithm would have competitive ration no better than $3/2$ (resp. $5/3$) so that the above bound is tight for any online algorithm.

   Solution. We construct an adversary for any algorithm which forces the output of the algorithm to be $3/2 \cdot C_{OPT}$ where C_{OPT} is the optimum make-span. When $m = 2$, the adversary first gives the algorithm two jobs each requiring one unit of time. Then the adversary looks at the make-span on both machines. If the algorithm put both jobs on the same machine then the adversary terminates. Observe that the $C_{OPT} = 1$ but the algorithm obtain $C_{OBT} = 2$. Otherwise, if the algorithm put one job on each machine then the adversary will give the algorithm one more job which takes two units of time. Observe that $C_{OPT} = 2$, but the algorithm obtained $C_{OBT} = 3$. In both cases the algorithm was unable to achieve a make-span better than $\frac{3}{2} \cdot C_{OPT}$.

   A similar construction is possible in the case with $m = 3$. Assume that the three machines are labeled m_1, m_2, m_3. The following is a sequence of actions taken by the adversary:

   (a) Given the algorithm job j_1 of make-span 1. Without loss of generality (WLOG) the algorithm puts j_1 on m_1.

(b) Give the algorithm job j_2 of make-span 1. If the algorithm puts j_2 on m_1 then terminate. Observe that $C_{OPT} = 1$ and $C_{OBT} = 2 \geq \frac{5}{3} \cdot C_{OPT}$. Thus WLOG we can assume that the algorithm put j_2 on m_2.

(c) Give the algorithm job j_3 of make-span 1. Similar to the reasoning above, the algorithm must put j_3 on m_3.

(d) Give the algorithm job j_4 of make-span 3. Since all three machines have the same make-span, WLOG assume the algorithm put j_4 on m_1.

(e) Given the algorithm job j_5 of make-span 3. If the algorithm puts j_5 on m_1 then terminate. Observe that $C_{OPT} = 4$ and $C_{OBT} = 7 \geq \frac{5}{3} \cdot C_{OPT}$. Thus WLOG we can assume that the algorithm put j_5 on m_2.

(f) Give the algorithm job j_6 of make-span 3. Similar to the reasoning above, the algorithm must put j_6 on m_3.

(g) Finally give the algorithm job j_7 of make-span 6. Observe that $C_{OPT} = 6$ and $C_{OBT} = 10 \geq \frac{5}{3} \cdot C_{OPT}$. Since each step was forced, it is impossible for any algorithm to do better than a $\frac{5}{3}$-approximation on three machines. Thus the greedy algorithm is optimal here.

3. Consider the greedy online algorithm for the make-span problem in the random order model (ROM). Show that for any $\epsilon > 0$, there exists a sufficiently large m such that the (expected) approximation ratio is

$$\frac{E[C_{Greedy}]}{C_{OPT}} \geq 2 - \epsilon$$

where the expectation is with respect to the uniform distribution on input arrival order.

Solution. We choose an m such that $\frac{3}{2m} < \epsilon$ and define $M = 2m(2m-1)+1$. The adversary will give the algorithm a set of M objects such that the algorithm will achieve an $\left(\frac{3}{2} - \epsilon\right)$ expected approximation. The adversary will give the greedy online algorithm items $\{t_1, ..., t_{M-1}\}$, each with make-span one, and one item t_M of make-span $2m$. Observe that if item t_M came last then the greedy online algorithm would obtain a value $C_{OBT} = 4m - 1$ while the optimum value would be $C_{OPT} = 2m$. This would be a $(2 - 1/m)$ approximation. Next we consider the expected value.

Observe that if item t_M was the i^{th} item in the random sequence then the minimum make-span C_{OBT} obtained by the greedy algorithm satisfies:

$$C_{OBT} \geq 2m + \frac{i-1}{2m} - 1$$

since the shortest stack after adding $i-1$ items of make-span one is

$$\left\lfloor \frac{i-1}{2m} \right\rfloor \geq \frac{i-1}{2m} - 1$$

Now that we have a lower bound on the minimum make-span when t_M is the i^{th} item added

we can calculated the expected minimum make-span over all $1 \leq i \leq M$:

$$\mathbb{E}[C_{OBT}] \geq \frac{1}{M} \sum_{i=0}^{M-1} \left(2m + \frac{i}{2m} - 1\right) \tag{1}$$

$$= 2m - 1 + \frac{1}{2mM} \left(\sum_{i=0}^{M-1} i\right) \tag{2}$$

$$= 2m - 1 + \frac{1}{2mM} \frac{M(M-1)}{2} \tag{3}$$

$$= 2m - 1 + \frac{2m(2m-1)}{4m} \tag{4}$$

$$= \left(\frac{3}{2} - \frac{3}{2m}\right) \cdot 2m = \left(\frac{3}{2} - \frac{3}{2m}\right) \cdot C_{OPT} \tag{5}$$

Thus the expected make-span is greater than or equal to $\frac{3}{2} - \frac{3}{2m} \geq \frac{3}{2} - \epsilon$ times C_{OPT}.

4. Prove that the LPT algorithm for the make-span problem is an $(\frac{4}{3} - \frac{1}{3m})$-approximation.
Proof. Let C_{OPT} be the the optimum make-span. Without loss of generality, the job causing the make-span of $> (4/3 - 1/(3m)) \cdot C_{OPT}$ is p_r, the job having minimum processing cost.

Lemma 1. *The make-span of p_r is greater than $C_{OPT}/3$.*

Proof. Suppose for a contradiction that $p_r = C_{OPT} \cdot k < C_{OPT}/3$. Then machine h processing p_r must have had make-span $> (4/3 - k - 1/(3m)) \cdot C_{OPT}$ before adding p_r. Since the greedy algorithm put incoming jobs on a machine with the least make-span, all other machines must have make-span $> (4/3 - k - 1/(3m)) \cdot C_{OPT}$ as well. Then the total make-span on all the machines including the make-span of p_r is greater than:

$$mC_{OPT} \cdot \left(\frac{4}{3} - k - \frac{1}{3m}\right) + C_{OPT} \cdot k = \left(m + \frac{m}{3} - mk - \frac{1}{3} + k\right) \cdot C_{OPT}$$

$$= mC_{OPT} + (m-1)\left(\frac{1}{3} - k\right) C_{OPT}.$$

Since $k < 1/3$ and $m \geq 1$, the total make-span on all machines is greater than mC_{OPT}. This is impossible since $C_{OPT} \geq (\sum_i p_i)/m$ where i ranges over the index of all jobs. □

By the above lemma we see that an optimum solution can schedule at most two jobs per machine; if three jobs are scheduled on the same machine then the make-span on that machine would be greater than C_{OPT}.

Let p_f be the first job that the optimum solution schedules differently from the greedy algorithm. That is, the greedy algorithm puts p_f on machine h and the optimum solution puts p_f on another machine h'. Either both machines were empty before adding p_f — in which case we can relabel the machines— or h' contained some job p'. If h was initially empty, moving p_r to h in the optimum solution does not increase the make-span. Otherwise h contained some other job p. Since the greedy algorithm puts incoming job on the machine with least make-span, the make-span of p is less than that of p'. By swapping jobs p and p' the new make-span is less than or equal to the original. By iterating though this process for all jobs, we have transformed the optimum schedule to the LPT schedule. □

Problem 2. ***The Knapsack problem.***

Consider the knapsack problem with input items $\{(v_1, s_1), ..., (v_n, s_n)\}$ and capacity C. Without loss of generality suppose $s_j < C$ for all j. Consider the following "natural" greedy algorithms which initially sort the input set and then schedules greedily. For each algorithm provide an input instance which show that the algorithm cannot achieve a c-approximation for any constant c.

Solution. We will describe the input sequences as follows:

1. *Greedy by value:* let the input items be $\{(v_0, s_0), ..., (v_{2c}, s_{2c})\}$ such that $(v_0, s_0) = (1, 2c)$ and $(v_i, s_i) = (0.5, 1)$ for all $1 \leq i \leq 2c$. Let $C = 2c$. The optimal solution would take items $1, ..., 2c$ and obtain a value of c. However the greedy by value algorithm would take item 0 and obtain value 1.

2. *Greedy by size:* let the input items be $\{(v_0, s_0), (v_1, s_1)\}$ where $(v_0, s_0) = (1, 1)$ and $(v_1, s_1) = (c, c)$. Let $C = c$. The optimal solution would take item 1, obtaining a value of c. The greedy by increasing size algorithm will take item 0 and obtain a value of 1.

3. *Greedy by value-density:* let the input items be $\{(v_0, s_0), (v_1, s_1)\}$ where $(v_0, s_0) = (1, 1)$ and $(v_1, s_1) = (c, 2c)$. Let $C = 2c$. The optimal solution would take item 1, obtaining a value of c. The greedy by increasing value-density algorithm will take item 0 since its density is 1 which is greater than the density of 0.5 for item 1. It obtains a value of 1.

Problem 3. ***More Knapsack Problems.***

For the knapsack problem, consider the algorithm that returns the maximum of greedy by value *and* greedy by value-density *as defined in the previous question. Show that this algorithm is a* 2*-approximation for the knapsack problem.*

Proof. Let C be the capacity bound, V_{OPT} the optimum value, V_{DOBT} the value obtained by the greedy by density algorithm, V_{VOBT} the value obtained by the greedy by value algorithm, and let $\{t_1, ..., t_n\}$ be the set of items where item t_i has value v_i and weight s_i. Suppose without loss of generality that $s_i \leq C$ for all items.

Lemma 2. *Suppose that $v_1/s_1 \geq \cdots \geq v_n/s_n$ and let t_c be the first item rejected by the greedy by value density algorithm such that $\sum_{i=1}^{c} s_i > C$. Then $\sum_{i=1}^{c} v_i \geq V_{OPT}$.*

Proof. Let $T = \{t_1, ..., t_c\}$ and suppose that $T_{OPT} = \{t_{k_1}, ..., t_{k_m}\}$ is the set of items taken by the optimum solution sorted by decreasing density. We will show that a disjoint — possibly fractional — subset of $\{t_1, ..., t_c\}$ is more valuable than t_{k_i} for each i. Consider item t_{k_1} with capacity s_{k_1}. Since $s_{k_1} \leq C < \sum_{i=1}^{c} s_i$ there exist an index j such that $s_{k_1} \leq \sum_{i=1}^{j} s_i$. Split item t_j into two items t'_j and t''_j with the same density — possibly with one trivial item of zero weight and zero value — such that $s_{k_1} = \sum_{i=1}^{j-1} s_i + s'_j$. Since the items are greedy by density we know that the density of items $t_1, ..., t_{j-1}, t_j$ is greater than or equal to the density of t_{k_1}. Since the weights of t_{k_1} and $\{t_1, ..., t_{j-1}, t'_j\}$ is equal, $v_{k_1} \leq \sum_{i=1}^{j-1} v_i + v'_j$. Remove t_{k_1} from T_{OPT}. Remove $\{t_1, ..., t_j\}$ and add t''_j to T. Observe that the invariant: *the total capacity of T_{OPT} is less than the total capacity of T* is maintain. Repeat the above argument for items $t_{k_2}, ..., t_{k_m}$. Thus the total value of T_{OPT} is less than or equal to the total value of T. □

Since $\sum_{i=1}^{c-1} v_i + v_c \geq V_{OPT}$ by lemma (2), either $\sum_{i=1}^{c-1} v_i \geq V_{OPT}/2$ or $v_c \geq V_{OPT}/2$ by the pigeon hole principle. If the former is the case then the greedy by density algorithm produces a 2-approximation. If the latter is the case then the greedy by value algorithm produces a 2-approximation since $V_{VOBT} \geq v_c$. □

Problem 4. *Consider the following knapsack type problem. We need to place a subset of n items in a railroad car of integral length C; items have different integral lengths and types R and B. Further item I_i has value v_i. Note that between two items of type R must be an item of type B. Provide a dynamic programming algorithm with time complexity polynomial in n and C which maximizes the values of the chosen items. Specify the array used and the recursive definition used to compute the entries of the array. Indicate how the desired output is obtained.*

Solution. The input is specified as follows: there are s items of type R and t items of type B. The lengths of these items are $r_1, ..., r_s$ and $b_1, ..., b_t$ respectively. Further, the item with length r_j has value v_{r_j} and the item with length b_i has value v_{b_i}. Suppose C is the length of the rail car.

We use two arrays for the dynamic program each with three dimensions $\langle i, j, c\rangle$; $1 \leq i \leq t$ specifies the index of the B type items, $1 \leq j \leq s$ specifies the index of the R type items, and c specifies the remaining capacity. The recurrence relations are as follows:

$$V_B(i,j,c) = \begin{cases} \max \begin{cases} V_B(i-1,j,c), \\ V_B(i-1,j,c-b_i) + v_{b_i}, \\ V_R(i-1,j,c-b_i) + v_{b_i} \end{cases} & \text{if } i \geq j \geq 0 \text{ and } c > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$V_R(i,j,c) = \begin{cases} \max \begin{cases} V_R(i,j-1,c), \\ V_B(i,j-1,c-r_j) + v_{r_j} \end{cases} & \text{if } i-1 \geq j \geq 0 \text{ and } c > 0 \\ 0 & \text{otherwise} \end{cases}$$

with base case:

$$V_{\{B,R\}}(0,0,c) = V_{\{B,R\}}(i,j,0) = 0.$$

$V_B(i,j,c)$ represents the optimum value that can be obtained considering lengths $b_1, ..., b_i$ and $r_1, ..., r_j$ with maximum length constrained to be $\leq c$ if a type B object must be picked. $V_R(i,j,c)$ is similar though we require that a type R object needs to be picked. Observe that each array has $stC \in O(n^2C)$ cells where $n = s + t$. Calculating each cell takes constant time since we only need to look up a constant number of previously calculated values. Thus the total time needed to construct the two arrays is $O(n^2C)$ which is polynomial in n and C. To obtain the optimum value for the problem, calculate: $V_{OPT} = \max\{V_R(s,t,C), V_B(s,t,C)\}$.

Lemma 3. *The dynamic algorithm just described outputs the optimum value for the knapsack type problem.*

Proof. The proof is by induction. We show the value in each cell of the array represents the optimum solution for the associated subproblem. Consider the base cases: $V_{\{B,R\}}(0,0,c)$ and $V_{\{B,R\}}(i,j,0)$. In the former we have no items and in the latter there is no more space. In either case no more

value can be obtained. Suppose for the induction hypothesis that the arrays are filled up correctly up to, but not including, cells $V_B(i,j,c)$ and $V_R(i,j,c)$. For $V_B(i,j,c)$, if the capacity is positive and there are at least as many type B objects as there are type R objects, we consider the most value if we: (1) don't choose b_i and insist that the current item is of type B, (2) chose b_i and insist that the previous item is of type B, and (3) chose b_i and insist that the previous item is of type R. For $V_R(i,j,c)$, again we make the necessary checks, then consider the most value if we: (1) don't choose r_j and insist that the current item is of type R, (2) chose r_j and insist that the previous item is of type B. By the induction hypothesis, these subproblems are solved optimally so under these considerations we obtain the optimum solution for $V_{\{B,R\}}(i,j,c)$. Thus the dynamic program outputs the optimum solution in general. □

Problem 5. *Consider the following scheduling problem. Each job J_i is described by a tuple (d_i, p_i, v_i) where d_i is the deadline, p_i is the processing time, and v_i is the value of the job if scheduled so that it finishes processing by its deadline. The goal is to schedule jobs without any overlap so as to maximize the value for the items scheduled. Specify in words the array you are using and the associated recursive definition for computing the entries in this array. Indicate how the desired output is obtained.*

Solution. Let the input be a set of n jobs $J_1, ..., J_n$ with associated triple (d_i, p_i, v_i) as described in the problem for each J_i. Sort the jobs by increasing latest starting times such that $d_1 - p_1 \leq d_2 - p_2 \leq \cdots \leq d_n - p_n$.

Our dynamic algorithm has an associated 2-dimensional array with dimensions $\langle i, v\rangle$ where i indicates the job we are considering and v indicates the value we already obtained after considering objects $J_1, ..., J_{i-1}$. The recursion used to fill out the array is as follows:

$$V(i,v) = \begin{cases} \max \begin{cases} V(i+1, v), \\ V(i+\alpha(i), v+v_i) \end{cases} & \text{if } i \leq n \\ v & \text{otherwise} \end{cases}$$

where $\alpha(i)$ returns the smallest value k such that $d_{i+k} - p_{i+k} \geq d_i$. This indicates that job $J_{i+\alpha(i)}$ can be scheduled after job i. The base case is $V(n+1, v) = v$ for all v. Let $v_{\max}$ be the largest value. When there are n jobs in total, v is at most $\sum_{i=1}^{n} v_i \leq n \cdot v_{\max}$. Thus there are at most $n^2 v_{\max}$ cells in the array. Filling in each cell takes constant time since it only requires accessing two previously calculated values, thus the total running time is $O(n^2 v_{\max})$ which is polynomial in n and $v_{\max}$. To obtain the optimum value, calculate: $V_{OPT} = V(1, 0)$.

Lemma 4. *This dynamic algorithm outputs the optimum value for the scheduling problem.*

Proof. The proof is by induction. We show that the value in each cell of the array represents the optimum solution for the associated subproblem. Consider the base case: $V(n+1, v)$. Since there are no more jobs to consider the value obtained is just the current value. Suppose that the array is filled out correctly up to but not including $V(i, v)$. For cell $V(i, v)$ we consider the subproblem when we don't take job i and when we do. Since these subproblems are solved optimally by the induction hypothesis, maximizing over the two cases obtains the optimum solution for $V(i, v)$ as well. Thus the dynamic algorithm presents outputs the optimum solution in general. □