Introduction to Algorithms **Problem Set 10 (3 problems)**
CS 4820 Fall 2018

(1) Approximate Knapsack. *(10 points, with option for 10 additional bonus points)* The KNAPSACK problem discussed in class on Friday, November 17th is given by n items each with a value $v_i \geq 0$ and weight $w_i \geq 0$, and a weight limit W. The problem is to find a subset I of items maximizing the total value $\sum_{i \in I} v_i$ while obeying the weight limit $\sum_{i \in I} w_i \leq W$. You may assume that $w_i \leq W$ for all i, that is, each item individually fits in the knapsack.

This coding problem is asking you to design an efficient algorithm to approximately solve this problem. For full credit, your algorithm needs to find a solution that obeys the weight limit with total value of at least 1/2 of the maximum possible. You can get bonus points (which will count towards your homework total) if you pass *every* test with an even better ratio. In particular, for a given test case, let $\beta = \frac{\text{optimal solution}}{\text{your solution}}$. You will get 13 points on this problem if you achieve $\beta \leq 1.2$ on every test case, 18 points if $\beta \leq 1.1$, and 20 points if $\beta \leq 1.01$. Random (or typical) instances of knapsack problems tend to be rather easy to solve to very close to optimal values. Here we are using a library of hard instances to test your program.

We will use the following input sizes: $n \leq 10,000$, $v_i, w_i \leq 100,000$, $W \leq 10,000,000$. You should make sure that your program runs in less than 1 second on every test case provided. Because we will use a powerful desktop to test your program, this should be sufficient to avoid TimeLimitExceeded errors.

Since this is an NP-complete problem, we don't have specific algorithm requirements for you. You can use any algorithm that you like - just be careful with the time limit. You may consider reading Section 11.8 of the book, which will not be covered in class; but note that the algorithm discussed there is slow and needs tons of space (as dynamic programs usually do). You may be better off with simple heuristics, while thinking about the following idea: consider a greedy algorithm that orders the items in some way, and adds all items until it reaches one that does not fit. At this point, let $1, 2, \ldots, k$ be the items in the knapsack, and let $k+1$ be the first item that does not fit. If item $k+1$ has relatively small value, the total value of items $1, 2, \ldots, k$ must be close to optimal. If this is not the case, maybe a better solution arises by somehow forcing item $k+1$ to be part of the solution.

You must write your code using the framework we provide on CMS (`Framework.java`). You do not need to code anything to handle inputs or outputs; you should only write your code between the comments // YOUR CODE STARTS HERE and // YOUR CODE ENDS HERE. Do not add code anywhere else (including package statements), and delete all print statements before submitting. Please read the `Framework.java` file carefully before starting - to make this easier, see the bold sections below on *complete data structures we provide* and *data structures we provide that you must fill in.*

You can test your code with the test cases provided on CMS. In particular, `Framework.java` takes two command line arguments: the first one is the name of the input file, and the second is the name of the output file, e.g.:

```
javac Framework.java; java Framework SampleTest_1 MyOutput_1
```

The input file should be in the same folder as your compiled Java code. After you compile and run your code, the output file will also be in the same folder. In order to test your code with the provided test cases, copy the test cases in the folder in which you have compiled your code, and set the input file (in the command line) to be the name of one of the sample inputs (*SampleTest_i.txt* in which $0 \leq i \leq 5$), and set the output file (in the command line) however desired. You can then compare your output with the provided (optimal) sample outputs (*SampleOutput_i.txt* in which $0 \leq i \leq 5$). To calculate β (as defined above), compare the first line of the provided output file with the first line of your output file.

Complete data structures we provide:

- n, an int representing the number of items that are given. Below, we consider the items to be labeled with indices $\{0, 1, \ldots, n-1\}$.
- values, a *0-indexed* int array holding the value of each item; i.e., values[0] holds the value of the first item.
- weights, a *0-indexed* int array holding the weight of each item; i.e., weights[0] holds the weight of the first item.
- weight_limit, an int representing the weight limit of the knapsack.

Data structures we provide that you must fill in:

- picked, a *0-indexed* boolean array representing which items you select to put in your knapsack; i.e., picked[i] should be set to true if and only if you wish to put item i in your knapsack (recall that items are labeled with indices $\{0, 1, \ldots, n-1\}$). Recall that **your picked items must not exceed the weight limit**, and should obtain $\geq 1/2$ of the max possible value without exceeding the weight limit.

Input file format (just FYI; only work with the data structures above):

- The first line has two numbers, n, W, which will be stored as n and weight_limit.
- In the i-th line (let $i = 0, \cdots, n-1$) of the next n lines, there are two numbers v_i, w_i, which will be stored as values[i], weights[i], respectively.

Output file format (just FYI; only work with the data structures above):

- The first line has one number, V, which is the total value obtained by your picked array (we calculate this for you).
- In the i-th line (let $i = 0, \cdots, n-1$) of the next n lines, an int representation of picked[i] is outputted: i.e., 1 means you picked item i and 0 means you didn't pick that item.

Solution: See the lecture notes `http://www.cs.cornell.edu/courses/cs4820/2016sp/lectures/approx-online.pdf` for a 2-approximation of Knapsack and an explanation of why some natural greedy algorithms do not work (i.e., common mistakes).

(2) Move-It-Quik You have been enlisted by the Move-It-Quik moving company to evaluate how efficient their operations are. Move-It-Quik specializes in large moves, such as when companies change offices. Oftentimes, these moves require numerous semi trucks to fit everything. However, the CEO of Move-It-Quik thinks they might be using more trucks than they need for each move. She has asked you to figure out how inefficient the movers are being with loading the trucks.

You visit a site to see how the movers load the trucks. Each truck has the same weight capacity of K pounds. Typically, a loading site only has room for one truck to be loaded at a time. So, rather than specifically planning for each truck to have different items in it, the movers just load one item at a time until the next item wouldn't fit, then move on to the next truck. The movers don't ask for any plan in advance.

The pseudocode for the truck-loading algorithm is as follows:

```
TRUCK LOADING:
    c = K is the capacity remaining for the current truck
    m = 1 is the number of trucks used so far.
    While there is an item i with weight w_i left to load:
        If c ≥ w_i:
            // The item fits; load item i on the current truck
            Set c := c − w_i
        Else:
            // The item won't fit; send off the current truck
            // and get a new one to load item i
            Set m+ = 1
            Set c = K − w_i
        Endif
    Endwhile
    Return m as the total number of trucks used.
```

Suppose at the end of one of these moving jobs, n items with weights $w_1, w_2, \ldots, w_n$ are packed up in that order. You agree that the CEO is right, that this is not the most efficient that the truck loading could be, but you want to quantify how much worse than optimal this process could be.

(a) (6 points) Show that the TRUCK LOADING algorithm is 2-optimal: that is, it will never use more than 2 times the optimal number of trucks needed.

Solution: First, we upper-bound how well the optimal packing OPT can get. We know that the optimal solution cannot pack trucks beyond their weight capacity. Let $W = \sum_i w_i$: any optimal solution must use at least $\lceil \frac{W}{K} \rceil$ trucks to pack all the items.

Next, we show that for any pair of consecutive trucks, the combined load of the two trucks must be greater than K. Suppose by way of contradiction that truck j has some load l_j and truck $j+1$ has load l_{j+1}, where $l_j + l_{j+1} \leq K$. In this case, all of the items in these two trucks together could have fit on the same truck, so this implies a contradiction: there was still room to load the first item on truck $j+1$ on truck j. Thus their combined load must be more than K.

Because every pair of trucks is more than half full, we can consider this summed over all possible trucks: if the number of trucks used is even, we get that on average (averaging truck $2j-1$ and truck $2j$), every truck must have at a load of more than $K/2$. This means that the total number of trucks used must be less than $W/\frac{K}{2} = 2W/K$, i.e. $2W/K > ALG \geq OPT \geq W/K$. Rewriting, we can use this to find that $\frac{ALG}{OPT} < \frac{2W/K}{W/K} = 2$, or that the ratio of trucks used between the mover's algorithm and the optimal solution is upper-bounded by 2. If the number of trucks uses is odd, say $ALG = 2m+1$, the first $2m$ trucks have total weight greater than mK, which means we need at least $m+1$ trucks, so we get $OPT \geq m+1$ and $ALG = 2m+1$. In this case we also have $\frac{2m+1}{m+1} = ALG/OPT \leq 2 = \frac{2m+2}{m+1}$ as well.

(b) (4 points) In part (a), you showed that if OPT is the number of trucks used in the optimal solution, $2 \cdot OPT$ is an upper bound on the number of trucks used by this algorithm. In general, an upper bound is called "tight" if there is an instance where the upper-bound is actually achieved: in this case, we know there can't possibly be a smaller upper bound. Show that this 2-optimal bound is tight: describe how to construct a sequence of objects for any value of OPT that will ensure that while the optimal algorithm uses OPT trucks, the TRUCK LOADING algorithm uses at least $2OPT - 2$ trucks. *An example that works for some single value of OPT but not for any OPT can get 2/4 points.*

Solution: Consider a loading problem with truck capacity $K = 1$ and the alternating sequence $w_1 = 1/2, w_2 = 2/n, w_3 = 1/2, \ldots w_{n-1} = 1/2, w_n = 2/n$. This gives us $n/2$ items of weight $1/2$, which can be loaded into $n/4$ trucks, completely filling the trucks. The remaining $n/2$ items all have weight $2/n$ and thus can fit in a truck together. In total, we use $OPT = n/4 + 1$ trucks in this optimal solution, and every truck is full. The algorithm, however, will always load every even item with weight $2/n$ on with the previous odd item of weight $1/2$, which will force the movers to start a new truck every other item and using a total of $n/2 = 2OPT - 2$ trucks.

Common Mistakes

- In part (a) some students assume that ALG is always even and argue that on average each truck will have more than $K/2$ weight. This is the right idea, but ALG may not always be even, and it is not true that the average truck load needs to be at least $K/2$, see for example the sequence ϵ, K, ϵ. ALG will use 3 trucks, and the average load is just above $K/3$.
- In part (b) some students came up with an example that doesn't scale with OPT. We are looking for a family of examples such that if you are given any OPT, there is an example (with n, $w_1, \ldots, w_n$ all possibly depend on OPT) sucht that the algorithm uses exactly $ALG = 2OPT - 2$ trucks. This is because we want to say that asymptotically as OPT increases, the greedy algorithm is a 2-approximation.

(3) Randomized approximate SAT algorithm.

We can use the linear programming based approximation technique discussed in class on Monday, November 26th, for approximately solving SAT problems. Let Φ be a SAT formula with variables $x_1, \ldots, x_n$ in the usual conjunctive normal form. Suppose Φ has m clauses $C_1, \ldots, C_m$, where C_j contains the set $P_j = \{i \text{ such that } x_i \in C_j\}$ (variables x_i occur positively in C_j) and the set $N_j = \{i \text{ such that } \bar{x}_i \in C_j\}$ (variables x_i occur negatively in C_j).

In this problem, we consider the following linear programming problem.

$$\begin{aligned} y_i, z_i &\geq 0 \text{ for all } i = 1, \ldots, n \\ y_i + z_i &= 1 \text{ for all } i = 1, \ldots, n \\ \sum_{i \in P_j} y_i + \sum_{i \in N_j} z_i &\geq 1 \text{ for all } j = 1, \ldots, m \end{aligned}$$

(a.) *(2 points)*) Show that if Φ has a solution, then this inequality system is also satisfiable.

Solution: Consider a satisfying assignment for formula Φ and set $y_i = 1$ if $x_i = true$ and $z_i = 1$ if $x_i = false$. We need to prove that Φ is satisfiable if and only if the above linear program has an integer solution.

- If Φ is satisfiable, let x be a satisfying assignment, and set y and z to 0 or 1 as stated above. We claim this satisfies the inequalities. We set one of y_i and z_i to 1, so indeed $y_i + z_i = 1$. Consider an inequality corresponding to a clause $\sum_{i \in P_j} y_i + \sum_{i \in N_j} z_i \geq 1$. With y_i and z_i is 0 or 1, this inequality is true if at least one of the variables involved is 1, and that happens exactly if one of the variables in the clause j are true.
- If the linear program has an integer solution, we use the above to set $x_i = true$ if $y_i = 1$ and set $x_i = false$ if $y_i = 0$ (and hence $z_i = 0$ as we have $x_i + y_i = 1$). Now we need to argue that all clauses are satisfied. To show this, we need to show that one of the literals in clause j is true, and that follows as y, z satisfy the corresponding constraint $\sum_{i \in P_j} y_i + \sum_{i \in N_j} z_i \geq 1$.

For the rest of this problem, we consider the case when the inequality system has a solution y, z. We will try to turn this solution into a truth assignment that satisfies many clauses.

By the first two sets of constraints, y_i and z_i are non-negative and sum to 1, so we can use them together as probabilities. So, we will use y_i and z_i as probabilities for the assignment of x_i: independently for each variable x_i, set x_i true with probability y_i and set x_i false with probability z_i.

For example, if Φ has the clause $(x_i \vee x_j)$ with $i \neq j$, and our linear programming solution is represented by vectors y and z of length n, then our probabilistic setting of x will make x_i true with probability y_i and false with probability z_i, and x_j will be true with probability y_j and false with probability z_j. Because the choices for i and j are independent, with probability $z_i z_j$ neither is true (and hence the clause is not satisfied).

(b.) *(2 points)* Consider a clause C with 2 variables. Show that the probability that C is satisfied by the above algorithm is at least 3/4 (assuming (y, z) is a solution to the linear program with clause C one of the clauses).

Solution: Let a and b be the probabilities of the two literals (variables or negated variables in the clause) being true each. By the inequality we know that $a + b \geq 1$. The formula is not satisfied with probability $(1 - a)(1 - b)$. The worst case occurs when $a + b = 1$, as otherwise we could lower either a or b until $a + b = 1$ and it would strictly increase the value of $(1 - a)(1 - b)$. Using this, we get the probability that the clause is not satisfied is $(1 - a)a = a - a^2$. This expression is maximized when the derivative is 0, that is when $1 - 2a = 0$, so $a = 1/2$, and in this case the

probability of not satisfying the clause is exactly 1/4, so the clause is true with probability at least 3/4.

(c.) *(3 points)* Consider a formula with m clauses where all clauses have only 1 or 2 terms. Show that the expected number of clauses satisfied by the above method is at least $0.75m$.

Solution: By linearity of expectation. Clauses with 1 literal are deterministically true to the corresponding inequality. Clauses with two literals are true with probability at least 3/4 by (b). Let p_j denote the probability that clause C_j is satisfied. Now the expected number of satisfied clauses is $\sum_j p_j$ by linearity of expectation, and hence at least $0.75m$, as claimed.

Common Mistakes

- Some students don't mention linearity of expectation in the explanation.

(d.) *(3 points)* Show that with probability at least 1/2 the above method satisfies at least 1/2 of the clauses in any formula where all clauses have either 1 or 2 terms.

Solution 1: Let S be a random variable representing the number of satisfied clauses. Suppose for contradiction that $x = P(S \geq m/2) < 1/2$. We can write $P(S < m/2)$ as $1 - x$. The expected number of clauses is then at most:

$$\begin{aligned} E(S) &= \sum_i i * P(S = i) \\ &< \frac{m}{2} \sum_{i < \frac{m}{2}} P(S = i) + m \sum_{i \geq \frac{m}{2}} P(S = i) \\ &= \frac{m}{2} * (1 - x) + mx \end{aligned}$$

Since by assumption, $x < \frac{1}{2}$, this is at most $\frac{3}{4}m$, which contradicts part b.

A less formal way to write this solution (without adding notation) is that the worst case is that whenever more than $m/2$ clauses are satisfied then all m are satisfied, and otherwise just barely less than $m/2$, and then do the calculation saying that in this case the expectation is $mP(S \geq m/2) + \frac{m}{2}(1 - P(S \geq m/2)$.

Solution 2: Using Markov inequality Let $\hat{K}$ be the number of not-satisfied clauses. We have that $\hat{K} = m - K$, so by (b) the expectation $E(\hat{K}) \leq m/4$. Now using Markov's inequality that for any nonnegative variable $Pr(X \geq \alpha E(X)) \leq 1/\alpha$ with $\alpha = 2$ we get

$$Pr(\hat{K} \geq 2m/4) \leq 1/2$$

and $Pr(\hat{K} \geq m/2)$ is the same as $Pr(K \leq m/2)$.

Common Mistakes

- Some students try to use Markov inequality directly on K above. I don't see how to get anything useful from that.
- Some students attempt to argue that with probability $(3/4)^n$ all clauses are satisfied, with probability $\binom{n}{1}(1/4)(3/4)^{n-1}$ we have $(n-1)$ clauses satisfied, etc. There are two issues with this attempt. Most notable the clauses are not independent, so not true that with probability $(3/4)^n$ all are satisfied, etc. Many students also don't know how to sum these numbers.

Comment: For a formula with m clauses without a limit on number of variables, one can prove that the expected number of clauses satisfied by the above method is at least $(1 - 1/e)m$. It isn't part of the homework to prove this, but just a fun fact for interested students.