Introduction to Algorithms — **Problem Set 10 (3 problems)**
CS 4820 Fall 2018 — **Due 11:59pm Thursday, November 29**

Your homework submissions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas.

The solution to each question need to be uploaded to CMS as a separate pdf file. To help provide anonymity in your grading, do not write your name on the homework (CMS will know it's your submission). (Questions with multiple parts need to be uploaded as a single file.)

(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 $1, 2, \dots, k$ 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 $1, 2, \dots, k$ 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.

(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 $c \geq w_i$:
            // 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.

(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.*

(3) Randomized approximate SAT algorithm.

(3) *(10 points)* 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.

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).

(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 $0.75m$.

(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.

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.