Programming Homework: The Knapsack Problem

M1 MOSIG: Algorithms and Program Design

November 23, 2009

1 Problem Definition

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.
In the following, we have n items, 1 through n. Each item j has a value p_j and a weight w_j. We assume that all values and weights are nonnegative. The maximum weight that we can carry in the bag is W. We want to find a subset I of items such that the total value in the bag is as large as possible

$$\max_I \sum_{i \in I} p_i \tag{1}$$

subject to the constraint that all items must fit in the bag

$$\sum_{i \in I} w_i \leq W. \tag{2}$$

2 Library provided

2.1 Generate instances

We provide you a C program from David Pisinger to generate instances of the knapsack problem.

To compile the code use:
`gcc -O3 -march=native -o generator generator.c`.
To run the code use:
`generator n r type i S`
where

n: number of items

r: range of coefficients p_i and w_i

type: corresponds to various types of instances, we will use type 2

i: instance number ($i \in [1, \ldots, S]$)

S: number of instances

The output will be written to the file `test.in`.
For example, to generate 100 instances with $n = 10$ items, each of them having a weigth w and a profit p between 1 and 20, you can use the following shell script.

```
for i in ‘seq 1 100‘
do
./generator 10 20 2 $i 100
mv test.in instance_10_20_$i
done
```

This script will generate 100 files from `instance_10_20_1` to `instance_10_20_100`.

2.2 Read an instance

The previous script will generate instance 65 in file `instance_10_20_65`:

```
10
    1     5     5
    2    16    14
    3    10     8
    4     3     5
    5    14    14
    6     3     3
    7    15    15
```

```
    8     15     13
    9     17     19
   10     17     16
72
```

On the first line we have the number of items $n = 10$, then one item per line with i the item number, p the profit and w the weight. Finally the last line contains the capacity of the bag $W = 72$.
The provided C program `read_instance.c` reads an instance from such a file, prints the instance and frees the allocated memory.
You can compile and run it with the commands

```
gcc -O3 -march=native -o read_instance read_instance.c -lrt
./read_instance instance_10_20_65
```

2.3 Measure the running time of your implementation

The `read_instance.c` file uses macros provided in `timer.h` to measure the running time of the function print_instance.
To measure the running time of your algorithm, you need to include the file `timer.h` and put the macros BEGIN_MAIN and END_MAIN at the beginning and end of your main program. You can then time each of your algorithms by calling the corresponding function inside the two macros BEGIN_EXPERIMENT and END_EXPERIMENT.
Here is the main function of `read_instance.c`.

```
int main ( int argc, char** argv ) {
BEGIN_MAIN
        if ( argc != 2 ) {
                printf ("./read_instance instance_file\n");
                return 0 ;
        }
        instance_t instance ;
        read_instance ( argv[1], &instance ) ;
BEGIN_EXPERIMENT
        print_instance ( &instance ) ;
END_EXPERIMENT
        free_instance ( &instance ) ;
END_MAIN
```

```
        return 0 ;
}
```

3 Questions

3.1 Brute-force algorithm

Implement an algorithm `BruteForce` that tries all possible subsets of items, check if the subset fits in the bag using equation 2 and report the best possible subset, the one that maximizes equation 1.

Question 1 *What is the worst case complexity of* `BruteForce` *in* Θ *notation?*

3.2 Greedy algorithm

Implement an algorithm `Greedy` that sorts the items in decreasing order of value per unit of weight p_i/w_i and inserts items in that order in the bag if they fit. If an item does not fit, try the next one until there are no more items or the bag is full.

Question 2 *What is the worst case complexity of* `Greedy`*?*

Question 3 *Does algorithm* `Greedy` *always give the best solution? If yes, provide a proof, if not give an instance of the problem where algorithm* `Greedy` *does not yield the optimal solution.*

3.3 Experimental comparison

Question 4 *For each algorithm, draw the graph of the experimental running time of your algorithm (using the macros discribed in section 2.3) with varying instance sizes. Carefully explain what are your measures, which instances do you use, etc.*

Question 5 *When does algorithm* `Greedy` *outperform algorithm* `BruteForce`*?*

Question 6 *What is the maximum size of an instance you can solve under 30 seconds for each algorithm?*

Question 7 *Which algorithm would you recommend in practice and why?*

3.4 Using Dynamic Programming

Question 8 *Give a recursive algorithm* `RecDP` *to compute the value of the optimal solution using dynamic programming. What is the running time of your algorithm? How much space do you use? Make sure to describe the recursive formulation you use.*

Question 9 *Give a sequential algorithm* `SeqDP` *to compute the value of the optimal solution using dynamic programming. What is the running time of your algorithm? How much space do you use?*

Question 10 *How can you recover a solution yielding the optimal value? Can you recover a solution without using additional storage space?*

Implement both algorithms `RecDP` and `SeqDP`.

Question 11 *How does* `RecDP` *and* `SeqDP` *compare to* `BruteFroce` *and* `Greedy` *in practice? What is the maximum size of an instance you can solve under 30 seconds with your fastest dynamic programming algorithm?*

Question 12 *Does the* `RecDP` *algorithm always use all the space in the array caching intermediate solutions? To check that, you can initialized all cells of the array with* -1 *and count at the end of the algorithm how much cells remained at* -1*. What does this mean in term of number of subproblems evaluated by* `RecDP` *compared to* `SeqDP`*?*

Question 13 *Using the previous question, try to come up with a family of instances where* `RecDP` *is a lot faster than* `SeqDP`*.*