Algorithms and Program Design

MoSIG 2014–2015 Exam — Semester 1 — Session 1

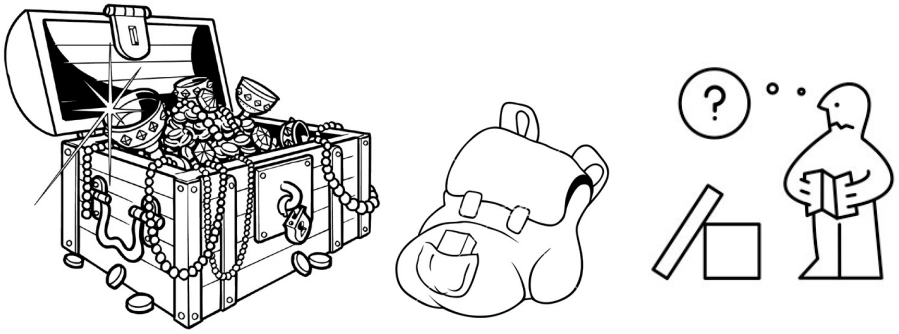
December 15th, 2014

This exam has 2 exercises, for a total of $22\frac{1}{2}$ points. The marking scheme is there to help you in discriminating the important questions from the easy ones and is subject to slight changes. All three exercises are independent and can be solved in any order. Bonus questions should be answered only if you have answered all the regular questions of the exercise first.

Allowed documents: hand-written and printed lecture notes.
All other materials forbidden (including: books and electronic devices).

Exam duration: 3h.

I Approximation algorithm [$11\frac{1}{2}$ points]



You travel with a knapsack and come across a treasure. However, your knapsack can only hold so much before it breaks so you will have to choose wisely what to keep of the loot. As input to this problem, there is the maximum weight W your knapsack can hold, and a set of n items $I = \{1, \ldots, n\}$; Each item i has a value v_i and a weight w_i. You can suppose that $w_i \leqslant W$ for all i.

We consider first the **finite** version of the problem, i.e., there is only a fixed number of each item available. To simplify, we consider two identical items to have numbers $i \neq j$ but $v_i = v_j$ and $w_i = w_j$, so you can take only **one** time item i. The goal is then to choose a subset $S \subset I$ that maximizes the total value of the sack $C = \sum_{i \in S} v_i$ under the constraint $\sum_{i \in S} w_i \leqslant W$.

Being as always an ultra-greedy thief, you first consider the following "ultra-greedy" algorithm: first take the item of highest value and try to put it in the bag; Then continue with the item of second highest value and so on, until the bag is full or there is no more items to try.

1. (**a**) Write the pseudo-code of the ultra-greedy algorithm. [1 pt]

(**b**) Give a counter-example which proves this algorithm is not optimal.

2. Give the **definition** of an α-approximation algorithm for a **maximization** problem. [$\frac{1}{2}$ pt]

3. We take $\alpha > 1$ a constant. Consider an item x such that $w_x = W$. [1 pt]

(**a**) Give a instance I of the items including x (you must choose v_x) and such that the optimal value of the sack is $C^* = \alpha + 1$ while the value found by the ultra-greedy algorithm is $C = 1$.

(**b**) Is this algorithm an α-approximation with α constant?

After a moment of reflexion, you reconsider your initial impulse and come up with a wiser (yet still greedy) algorithm: you first compute for each i a ratio $r_i = v_i/w_i$ ("value per weight") and now consider every item in decreasing ratio order, i.e., trying first to put the item with highest r_i, then the item of second highest r_i, and so on.

4. Is the algorithm optimal now? (Justify your answer.) [½ pt]

5. Re-using the item x from question **3** (and potentially changing v_x), prove this algorithm is still not an α-approximation with α constant. [1½ pts]

Feeling demoralized, you give up on the treasure, taking nothing in your knapsack... and that is when you stumble upon Aladdin's cave of wonders! Your heart is pure and you are allowed to enter. Now things are much simpler as each item is in **infinite** supply.
We denote by x_i the number of times item i is taken in the knapsack (x_i can be 0); the goal is then to maximize the total value of the sack $\sum_i x_i \times v_i$ under the constraint $\sum_i x_i \times w_i \leqslant W$.

6. Adapt the previous algorithm (the one that uses ratios) to the problem with infinite supply. Give the pseudo-code for this algorithm. [1 pt]

We will first show that this algorithm cannot be better than a 2-approximation algorithm, i.e., it is not an α-approximation with $\alpha < 2$.

7. We want to force the algorithm to make bad choices. First, let us consider only the **weights and ratios** of items, and try to force the algorithm to leave as much empty space as possible in the knapsack.

(**a**) Show a bad case example where the knapsack is not completely full, i.e., $\sum_i x_i \times w_i < W$, but the optimal is a full knapsack. *Reason with only the w_i and r_i (and not v_i).* [½ pt]

(**b**) Exhibit a worst-case family of examples, where the knapsack is as close as possible from being half empty; i.e., given any $\epsilon > 0$, construct an instance I such that $\sum_i x_i \times w_i < \frac{W}{2} + \epsilon$. [1½ pts]

8. Now we want to reason again with the **value** of the items and knapsack: $C = \sum_i x_i \times v_i$. [1 pt]

(**a**) Modify your example so that given any $\epsilon > 0$, you construct an instance I such that $C \leqslant \frac{C^*}{2} + \epsilon$.

(**b**) Conclude.

What remains to do now is proving the algorithm is a 2-approximation.

9. (**a**) Prove the knapsack will always be at least half-filled (i.e., $\sum_i x_i \times w_i \geqslant \frac{W}{2}$). [1½ pts]

(**b**) Can the knapsack be exactly half-filled?

(**c**) What is the exact value of the half-filled part of the knapsack?

10. Find an upper bound on the optimal solution $C^* = \sum_i x_i^* v_i$. *(Hint: consider only the best ratio.)* [1 pt]

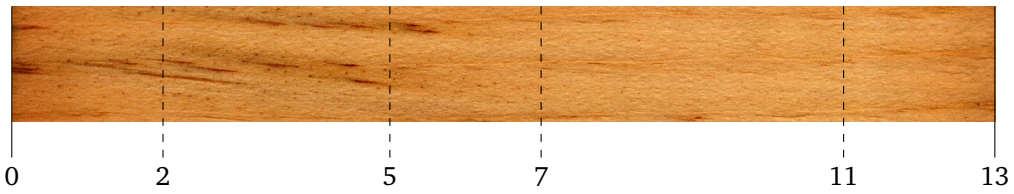
11. Finish the proof that the algorithm is a 2-approximation. [½ pt]

II Dynamic programing [11 points + 1 bonus]

Cutting a wooden board is cumbersome when the board is long, that's why your sawmill charges by length to cut each board. For instance, if your board is 7 feet in length, it will cost you 7€ to make one cut anywhere on the board.

So the cost of cutting a single board into multiple smaller boards will depend on the order of the cuts. Initially, you know the length l of your board, which is marked with k locations to cut $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$. You can assume all numbers are integers.

Warning: this problem is really about the **positions** of the cuts (and not the lengths of the final small pieces), you are not allowed to change them.



12. Consider the above example where $l = 13$, and the positions where to cut are $\mathcal{C} = \{2, 5, 7, 11\}$. [1 pt]

- **(a)** How much would it cost you to make them cut in this order (left-to-right)?
- **(b)** Which order would be the best?
- **(c)** Find the worst counter-example possible for the left-to-right order with $k = 2$ cuts and l fixed.

13. We propose the following: always cut a piece as close as possible to its center. [1 pt]

- **(a)** Show an example with $k = 3$ cuts where this algorithm yield the optimal solution.
- **(b)** Show this algorithm is not optimal in the general case.

Let us define for $1 \leqslant i < j \leqslant k$ the following sets $\mathcal{C}_{ij} = \{c_i, \ldots, c_j\}$. Supposing the original board has already been cut at position c_i and c_j, then we denote $\hat{C}(\mathcal{C}_{ij})$ the minimum cost to finish cutting the board between c_i and c_j.

14. **(a)** Extend the above definition to $0 \leqslant i < j \leqslant k + 1$. What is the meaning of $\hat{C}(\mathcal{C}_{0,k+1})$? [1 pt]

- **(b)** For $0 \leqslant i \leqslant k$, how much is $\hat{C}(\mathcal{C}_{i,i+1})$?
- **(c)** For $0 \leqslant i < k$, how much is $\hat{C}(\mathcal{C}_{i,i+2})$?

15. Write the recursive equation that defines $\hat{C}(\mathcal{C}_{ij})$ when $i < j - 2$, depending on $\hat{C}(\mathcal{C}_{xy})$ with $i \leqslant x < y \leqslant j$ well chosen. [1½ pts]

16. **(a)** What would be the complexity of the algorithm that directly tries to compute this equation? [1 pt]

- **(b)** Draw part of the tree of recursive calls of the formula on the above example, until you find some redundant computations (highlight them).

The goal is now to give a solution using dynamic programming.

17. **(a)** Propose a storage solution to avoid re-computations. [2 pts]

- **(b)** Where in this storage will you find the optimal value?
- **(c)** Is it preferable to use a bottom-up (i.e., from the leaves of the recursive call tree to the root) or a top-down (i.e., from the root to the leaves) approach? Justify your answer.

18. Write a dynamic programming algorithm that computes and returns the optimal cost. [2 pts]
(Bonus point: bottom-up approach with correct justification of the order of computation.)

19. What is the complexity of this dynamic program? [½ pt]

20. Modify your program so that it also gives the order in which to cut the board. [1 pt]

21. *Bonus: Prove that the algorithm finds the optimal cost.* [1 pt (bonus)]