Algorithms and Program Design

MoSIG 2014–2015 Exam — Semester 1 — Session 1

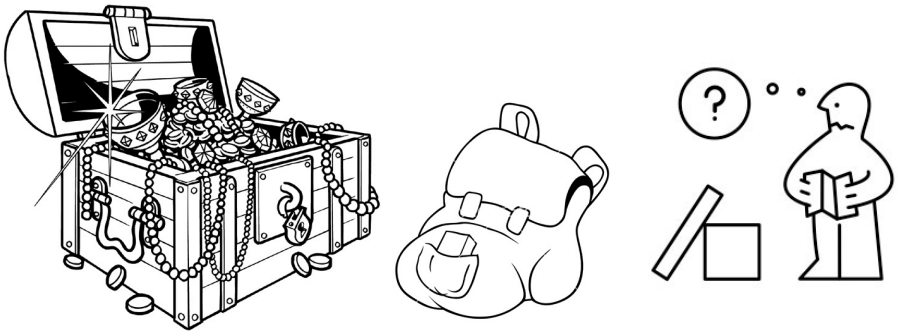
December 15th, 2014

This exam has 2 exercises, for a total of 22½ 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 [11½ 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]

Solution:

```
order I by value
S = empty
C = 0
R = W /* remaining weight */
```

```
for i = 1 to n do
  if w_i <= R
    add i to S
    C = C + v_i
    R = R - w_i
return C
```

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

Solution: $W = 1$, $v_1 = 2$, $w_1 = 1$, $v_2 = 1$, $w_2 = \frac{1}{3}$, $v_3 = 1$, $w_3 = \frac{1}{3}$, $v_4 = 1$, $w_4 = \frac{1}{3}$,

2. Give the **definition** of an α-approximation algorithm for a **maximization** problem. [½ pt]

Solution: An algorithm is an α-approximation if, for every instance I of the problem $C \geqslant \frac{C^*}{\alpha}$, or $\frac{C^*}{C} \leqslant \alpha$.

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 $C = 1$.

Solution: One item x of value $v_x = 1$, and $2\alpha + 2$ items of value $\frac{1}{2}$ and weight $\frac{W}{2\alpha+2}$.

(**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]

Solution: No. $W = 10$, $v_1 = 2$, $w_1 = 1$, $v_2 = 10$, $w_2 = 10$.

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]

Solution: We change the value of x to $v_x = \alpha + 1$. Now I contains only x and another item y: $v_y = 1$ and $w_y = \frac{W}{\alpha+2}$. So $r_x = \frac{\alpha+1}{W}$ and $r_y = \frac{\alpha+2}{W}$. Since $r_y > r_x$, y will be chosen first, but then x cannot fit in the bag so $C = v_y = 1$, but the optimal is choosing y: $C^* = v_y = \alpha + 1$. So $\frac{C^*}{C} = \alpha + 1 > \alpha$.

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]

Solution:

Data: a list of items with weights w_i and values v_i
Result: the number of times each item is selected
foreach *item* o_i **do**
  compute a value/weight ratio: $\frac{v_i}{w_i}$;
end
sort all items in decreasing ratios order;
let R be the remaining free weight in the knapsack;
initially $R = W$;
foreach *item* i *in the sorted list* **do**
  put $\left\lfloor \frac{R}{w_i} \right\rfloor$ times i in the knapsack;
  $R = R - \left\lfloor \frac{R}{w_i} \right\rfloor \times w_i$;
end
return *the selected items*

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]

Solution: $w_x = W$ and $r_x = 1$, and $w_y = \frac{W}{2} + 1$ and $r_y = 1.1$.

(**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]

Solution: We reuse again x and y from the previous counter example. $w_x = W$ and $r_x = 1$, $w_y = \frac{W+\epsilon}{2}$ and $r_y = 1 + \epsilon$.

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 $C \leqslant \frac{C^*}{2} + \epsilon$.

Solution: Now $v_x = W$ (so that $r_x = 1$), $w_y = \frac{W+\epsilon}{2}$ and $v_y = \frac{W}{2} + \epsilon$. Then $r_y = \frac{W+2\epsilon}{W+\epsilon} > 1 = r_x$. So y will be chosen first, and only one can be inside the bag has $w_y > \frac{W}{2}$. Then $C = v_y$ but the optimal would be exactly one item x with $C^* = v_x = W$. Hence $C = v_y = \frac{W}{2} + \epsilon = \frac{C^*}{2} + \epsilon$. □

(**b**) Conclude.

Solution: The algorithm cannot be an α-approximation with $\alpha < 2$. Proof: by contradiction let us take $\alpha < 2$. Then we need to prove that $\alpha \geqslant \frac{C^*}{C}$.
But, given the previous question, we have an family of examples where

$$\alpha \geqslant \frac{C^*}{\frac{C^*}{2} + \epsilon} = \frac{2}{1 + \frac{2\epsilon}{C^*}} \quad (1)$$

$$= \frac{2}{1 + \frac{2\epsilon}{W}} \quad (2)$$

(3)

Since W and ϵ are independent, we can make $\frac{2\epsilon}{W}$ as small as we want make making ϵ sufficiently small, and so force α to be infinitely close to 2.

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]

Solution: Let us consider item 1, which has the highest ration r_1. If $w_1 > \frac{W}{2}$, then the proof is over. Otherwise, suppose the algorithm has put as much item 1 as possible, and the knapsack is still not half-filled. The space remaining is then at least $\frac{W}{2}$, which is enough for another item 1... □

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

Solution: No: if the knapsack is exactly half-filled, we can just add again exactly the same items and end up with a knapsack exactly filled.

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

Solution: $\frac{W}{2} \times r_1$, where r_1 is the best ratio (the items are ordered in decreasing ratio).

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

Solution:

$$C^* = W * r_1$$

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

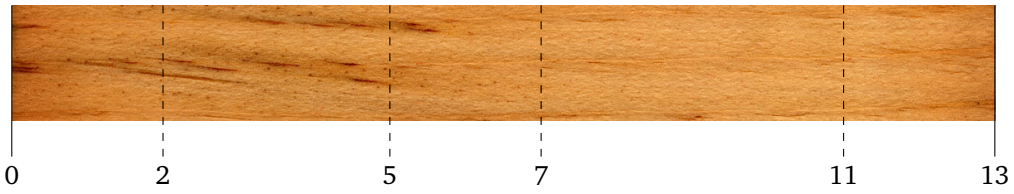
Solution: We already have $C \geqslant \frac{W}{2} \times r_1 \geqslant \frac{C^*}{2}$. So the algorithm is a 2-approximation. □

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.

> **Solution:** Same order: cost = 13 + 11 + 8 + 6 = 38.
>
> Best order: $\{7, 2, 5, 11\}$, cost = 13 + 7 + 5 + 6 = 31.
>
> Worst example: cuts = $\{1, 2\}$. Cost = $l + (l - 1)$ while best = $l + 2$. in general if cuts are k_1 and k_2, cost = $l + (l - k_1)$ while best = $l + k_2$. To maximize cost/best, we want cost to be as large as possible (so k_1 small) while best is as small as possible (so k_2 small); obviously, if we allow only integer cuts, is to choose $k_1 = 1$ and $k_2 = 2$.

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.

> **Solution:** Example, $l = 8$ and cuts = $\{2, 4, 6\}$.
>
> Counter example: $l = 100$ with cuts $\{49, 50, 51\}$.

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]

> **Solution:** By convention, $c_0 = 0$ and $c_k = l$. And $\hat{C}(\mathcal{C}_{0,k+1})$ is the minimum cost to cut the whole board.

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

Solution: 0

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

Solution: $c_{i+2} - c_i$, the position of c_{i+1} does not matter.

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]

Solution:

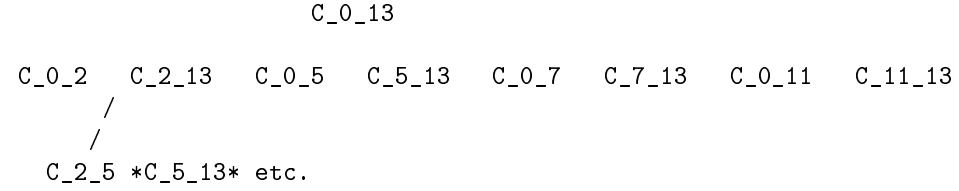
$$\hat{C}(\mathcal{C}_{ij}) = (c_j - c_i) + \min_{i<x<j} \left(\hat{C}(\mathcal{C}_{ix}) + \hat{C}(\mathcal{C}_{xj}) \right)$$

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

Solution: $O(k^k)$

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

Solution:



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

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

Solution: Naive solution: A matrix M (or triangular matrix) of size $(l+1) \times (l+1)$, where $M[i][j]$ contains $\hat{C}(\{c_x \text{ where } i < x < j\})$.
Much better solution: A matrix M (or triangular matrix) of size $(k+2) \times (k+2)$, where $M[i][j]$ contains $\hat{C}(\mathcal{C}_{ij})$.
We can even trim the matrix of the first diagonal line if we want ($M[i][i]$ is meaningless)...

(**b**) Where in this storage will you find the optimal value?

Solution: In $M[0][l+1]$ or $M[0][k+1]$ depending on the choice.

(**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.

Solution: In the naive solution, clearly a top-down as many cells will never need to be computed.

In the preferred solution, a bottom-up starting from one line above the diagonal (the diagonal is 0, one line above is $c_{i+1} - c_i$). As all cells will be computed, so we can compute iteratively with no loss of computation time in function calls or check that values are already computed.
It is important in that case to compute diagonal by diagonal, since we know about dependency vectors that $\hat{C}(\mathcal{C}_{ij})$ only depends on $\hat{C}(\mathcal{C}_{xy})$ where $y - x < j - i$. (And for a fixed $y - x$, they are all on the same diagonal.)
(Actually, to be precise, $M[i][j]$ seems to depend only on all $M[i][x < j]$ and $M[x > i][j]$, so *in theory* it seems to be possible to order computation column by column left to right, going up in each column, but I'm not 100% sure...)

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.)

Solution:

```
if k < 1  return 0
if k == 1 return l

/* otherwise we do something */
make_cache M[k+2][k+2]
/* fill first two interesting diagonals */
for i = 0 to k-1
  M[i][i+1] = 0
  M[i][i+2] = c_i+2 - c_i
-- C[i][i+2] = i+1 --
M[k][k+1] = 0

for d = 3 to k+1  /* the diagonals */
  for e = 0 to (k+1)-d /* ''index'' of cells in the diagonal */
    i = e
    j = d+e

    best = M[i][i+1] + M[i+1][j]
--  cut = i+1 --
    for x = i+2 to j-1
      best = min (best, M[i][x] + M[x][j])
   -- if best updated: cut = x --

    M[i][j] = best + (c_j - c_i)
--  C[i][j] = cut --

return M[0][k+1]
```

Note that it is not necessary to fill diagonal 2 before the big loop.

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

Solution: $O(k^3)$

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

Solution: Add the lines in `-- --` to the above program, then before returning:

```
function show_cut (i, j) =
  if (j-i < 2) return
  cut = C[i][j]
  print cut
  show_cut (i, cut)  /* order between those  */
  show_cut (cut, j)  /* lines is unimportant */

in
show_cut (0, k+1)
```

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

Solution: We must prove that, given an optimal solution S^*, with c_{i_0} being the first cut, then the order S_1^* and S_2^* of the cuts c_j, $j < i_0$ and c_j, $j > i_0$ are optimal solution to subproblems $\mathcal{C}_{0i_0}$ and $\mathcal{C}_{i_0k+1}$.

Suppose these (or one of these) are not optimal solution, then let us consider optimal orders S_1 and S_2 for these subproblems.

Then we can construct the order $c_{i_0} \circ S_1 \circ S_2$, whose cost is $k+1+\hat{C}(S_1)+\hat{C}(S_2) > \hat{C}(S_1^*)+\hat{C}(S_2^*)$. This is not possible as S^* was optimal. □