ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Exercise VI, Algorithms 2013-2014

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

For more exercises on dynamic programming and video explanations of solutions, see *http://people.csail.mit.edu/bdean/6.046/dp/*

Rod Cutting

1 (Exercise 15.1-2) Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

Solution: Assume that the price function is given as:

$$p_i = \begin{cases} 1 & \text{if } i = 1, \\ 5 & \text{if } i = 2, \\ 7 & \text{if } i = 3 \end{cases}$$

The density function is then:

$$d_i = p_i/i = \begin{cases} 1 & \text{if } i = 1, \\ 2.5 & \text{if } i = 2, \\ 2.33 & \text{if } i = 3 \end{cases}$$

The greedy strategy will cut the rod of length 3 into pieces of size 2 and 1 for the total price of 6, while the optimal solution is to leave the rod of size 3 with the price of 7.

2 (Based on Exercise 15.1-3) Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a cost of c. The revenue associated with a solution is now the sum of prices of the pieces minus the costs of making the cuts.

2a Write the optimal revenue r_n of a rod of length n in terms of optimal revenues from shorter rods. In other words, give a recursive formulation of the optimal revenue.

Solution: The optimal solution is either the rod of length n or a combination of two optimal solutions whose length totals to n. Assuming that we view the decomposition as a piece of length i and the remainder of length $n - i$, the formula is:

$$r_n = \max(p_n, \max_{1 \leq i \leq n-1} (p_i + r_{n-i} - c))$$

2b Describe how to calculate r_n using the "top-down with memoization" approach. In addition, analyze asymptotically your algorithm's time and space complexity.

Solution:

```
MEMOIZED-CUT-ROD(p, n, c)
1  let r[0..n] be a new array
2  r[0] = 0
3  for i = 1 to n
4     r[i] = -∞
5  return MEMOIZED-CUT-ROD-AUX(p, n, c, r)

MEMOIZED-CUT-ROD-AUX(p, n, c, r)
1  if r[n] ≥ 0
2     return r[n]
3  q = p[n]
4  for i = 1 to n - 1
5     q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, c, r) - c)
6  r[n] = q
7  return r[n]
```

The solution solves each subproblem only once. Therefore the number of iterations of the for loop forms an arithmetic series, and the running time is $\Theta(n^2)$ iterations. The amount of extra space used is $\Theta(n)$ for the array r.

2c Describe how to calculate r_n using the "bottom-up" approach. In addition, analyze asymptotically your algorithm's time and space complexity.

Solution:

```
BOTTOM-UP-CUT-ROD(p, n, c)
1  let r[0..n] be a new array
2  r[0] = 0
3  for i = 1 to n
4     q = p[i]
5     for j = 1 to i - 1
6        q = max(q, p[j] + r[i - j] - c)
7     r[i] = q
8  return r[n]
```

The number of iterations of the inner loop forms an arithmetic series, so the running time is again $\Theta(n^2)$. The space complexity is again $\Theta(n)$.

2d How would you modify the algorithms to return not only the value but the actual solution too?

Solution: In order to be able to return the actual solution, along with the maximal revenue $r[i]$ for every length i, we need to also store $s[i]$, the size of the first piece to cut off.

```
BOTTOM-UP-CUT-ROD(p, n, c)
1  let r[0..n] and s[0..n] be new arrays
```

```
2  r[0] = 0
3  for i = 1 to n
4     q = p[i]
5     for j = 1 to i − 1
6        if q < p[j] + r[i − j] − c then
7           q = p[j] + r[i − j] − c
8           s[i] = j
9     r[i] = q
10 return r and s
```

```
CUT-ROD-PRINT-SOLUTION(p, n, c)
1  (r, s) = BOTTOM-UP-CUT-ROD(p, n, c)
2  while n > 0
3     print s[n]
4     n = n − s[n]
```

Matrix-Chain Multiplication

3 (Based on Exercise 15.2-1) Consider the dynamic programming algorithm for matrix-chain multiplication taught in class. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6\rangle$. Find the answer (and explain how the algorithm proceeds) by filling in the "memory" table in the same way as the algorithm.

Solution: From the sequence of dimensions given we can identify six matrices with the following dimensions:

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	5×10	10×3	3×12	12×5	5×50	50×6

The algorithm proceeds using the following formula:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ max_{i \le k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{i<j.} \end{cases}$$

The table is computed starting with the diagonal of the matrix, i.e. it computes first $m[i,i] = 0$ for $i = 1, 2, ..., n$. Then this information is used to compute $m[i, i+1]$ for $i = 1, 2, ..., n-1$ so on and so forth. The resulting tables, m and s, will be filled as follows:

m=

i \ j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

s=

i \ j	2	3	4	5	6
1	1	2	2	4	2
2		2	2	2	2
3			3	4	4
4				4	4
5					5

Therefore the optimal parenthesization is $((A_1A_2)((A_3A_4)A_5A_6))$.

4 (Exercise 15.2-6) Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.
(Hint: use induction on the number of elements)

Solution: Recall the definition of fully parenthesized matrices: A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrices surrounded by a parentheses.

Therefore by induction

Base case:

1. if $n \leftarrow 1$, there are no parentheses ($n - 1 = 0$ parentheses)

Assume that this relationship holds for $n-1$ matrices, i.e., $n-1$ matrices have $n-2$ pairs of parentheses.

So the product of n matrices can be viewed as multiplying 1 matrix (which is already fully parenthesized) by the $n-1$ fully parenthesized matrices (which according to the previous step should have $n-2$ pairs of parenthesis). According to the definition the product of 2 fully parenthesized matrices is fully parenthesized by surrounding them by a parentheses resulting in a total of $n-1$ pairs of parentheses.

Solve New Problems Using Dynamic Programming

5 (half *, Problem 15-1) **Longest simple path in a directed graph.**

Suppose that we are given a directed acyclic graph $G = (V, E)$ with real valued edge weights and two distinguished vertices s and t. Describe a dynamic programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

Solution: First of all, note that all paths in DAG(directed acyclic graph) are simple. If there are two parts of the path then they cannot visit the same vertex, except when this vertex is the end and the beginning of the first and the second subpaths respectively.

The following observation gives an idea to the algorithm. Let p be the longest path from s to t and let v be the next vertex on this path after s. Then the longest path p' from v to t is a part of p. If not, let p'' be longer than p', then the path from s, which goes to v and then continues with p'', is the path from s to t, which is longer than p.

Let $dist[s]$ denote the weight of the longest path from s to t.

$$dist(s) = \begin{cases} 0 & \text{if } s = t, \\ max_{(s,v)\in E}(w(s,v) + dist[v]) & \text{otherwise.} \end{cases}$$

To solve the problem the following algorithm can be used. *next* is an array, which stores the next vertex on the longest path. Also, if there is a value, then the subproblem for this vertex has been already solved. Before it we have to initialise both arrays: *dist* with 0 and *next* with *null* values.

```
LONGEST-PATH(G,S,T, DIST, NEXT)
1   if s == t
2      dist[s] = 0
3      return (dist, next)
4   elseif next[s] == null
5      return (dist, next)
6   else
7      for each vertex v ∈ G.Adj[s] (adjacent vertex)
8         (dist, next) = LONGEST-PATH(G,V,T,DIST,NEXT)
9         if w(s, v) + dist[v] ≥ dist[s]
10             dist[s] = w(s, v) + dist[v]
11             next[s] = v
12     return (dist, next)
```

At the end $dist[s]$ stores the weight of the longest path from s to t. To print the path we need only to go by the links of the array *next* starting from s until we reach t.

The running time of this algorithm is $O(V)$ to initialize both arrays, $O(E)$ for the $Longest - Path$ part (we call it once for each edge of the vertex, thus we call it no more than E times) and $O(V)$ to restore the path. Thus, in sum it requires $O(V + E)$.

6 (*) **Knapsack Problem** Suppose that you are going on a beautiful hike in the Swiss Alps. As always, you are faced with the following problem: your knapsack is too small to fit all the items that you wish to bring with you. As items are of different importances and have different sizes, you would like to maximize the total value of the items that you can bring with you. Formally, we can define the "packing" problem as follows:

INPUT: A knapsack of capacity C and n items where item $i = 1, 2, \ldots, n$ has value $v_i \geq 0$ and size $s_i \geq 0$.

OUTPUT: A subset of items S that maximizes $\sum_{i \in S} v_i$ (the total value) subject to $\sum_{i \in S} s_i \leq C$ (the total size of the packed items is at most the capacity).

6a (The Fractional Knapsack Problem) Suppose that your items are divisible, i.e., you can pack a fraction $f \in [0, 1]$ of an item i and in that case it will give you a profit of $f \cdot v_i$ and occupy a space of $f \cdot s_i$ in the knapsack. Give a greedy algorithm for this case that runs in time $O(n \log n)$.

Solution: In Fractional Knapsack Problem we can pack a part of an item. This gives a simple greedy solution. To maximize the total value of the knapsack we simply need to take items, which value of a unit of weight are the biggest ones. Let rank each item i by the value of the unit of weight v_i/s_i and fill in the knapsack with items which have greater rank until there are no items left or a knapsack is full.

```
FRACTIONAL-KNAPSACK(V,C,C)
1  Sort items by v_i/s_i. Assume that v_i/s_i ≥ v_{i+1}/s_{i+1} for all i
2  load = 0
3  i = 1
4  while i ≤ n and load ≤ C
5    if C − load ≥ s_i
6      then take the whole item i and load = load + s_i
7      else take C − load/s_i of item i and load = C
8    i = i + 1
```

Time of the algorithm is $O(n\ lg\ n)$ for sorting and $O(n)$ for the second part of the algorithm.

6b Show that the greedy algorithm fails when the items are indivisible. How bad can the profit of the solution returned by the greedy algorithm be compared to an optimal solution?

Solution: In Knapsack Problem (non-fractional) greedy algorithm can leave a lot of empty space in the knapsack. Assume there are 2 items: first one has a size 1 and a cost v, and the second one with size C and a cost $C * (v - 1)$. The greedy algorithm will choose the first item and there will not be enough space for the second one. The optimal solution is to take the second item (here it is assumed that v and C are big).

In the worst case the greedy algorithm leaves practically the whole knapsack empty and it can be up to C times worse than the optimal algorithm.

6c Design a dynamic programming algorithm to solve the Knapsack Problem. Your algorithm should run in time $O(nC)$.

Solution: We can recursively define the value of an optimal solution by the following formula:

$$c(i, s) = \begin{cases} 0 & \text{if } i = 0 \text{ or } s = 0, \\ c(i-1, s) & \text{if } s_i \geq s, \\ max(v_i + c(i-1, s-s_i), c(i-1, s)) & \text{otherwise,} \end{cases}$$

where $c(i, s)$ is an optimal solution for the set of items $\{1...i\}$ and the size of a knapsack s. The last case says that the optimal solution can either take item i and items from the optimal solution

$c(i-1, s-s_i)$, or don't take item i, and in this case it is the same as the optimal solution for $c(i-1, s)$.

The algorithm fills the table $c[0..n, 0..C]$, each cell stores the best value. The table is filled row by row. At the end the cell $c[n, C]$ has the optimal value for the problem. To find the set of items we need to take, we trace this table back from $c[n, C]$. If $c[i, s] = c[i-1, s]$ then we do not take item i and continue tracing from $c[i-1, s]$. Otherwise we take item i and continue from $c[i-1, s-s_i]$. Here is the algorithm:

```
DYNAMIC-KNAPSACK(N,V,S,C)
1   for j = 0 to C
2     c[0, j] = 0
3   for i = 1 to n
4     c[i, 0] = 0
5   for i = 1 to n
6     for j = 1 to C
7       if s_i ≤ j
8         then c[i, j] = max(v_i + c[i − 1, j − s_i], c[i − 1, j])
9         else c[i, j] = c[i − 1, j]
10  i = n
11  j = C
12  while i > 0
13    if c[i − 1, j] ≠ c[i, j]
14      then printi
15      j = j − s_i
16    i = i − 1
```

The algorithm takes $\Theta(nC)$ time to fill the table and $\Theta(n)$ time to trace the solution.