Homework 6 (Shun Qiao 0524407)

A Greedy Algorithms

A.1 Bill Exchange

Let S be a set of integers representing bill values. For example, for US dollars we have $S_{us} = \{1, 5, 10, 20, 50, 100\}$. For a given amount k, your task is to find the minimum number of bills that can compose k dollars.

1. What is the optimal solution for 379 dollars on S_{us}?

2. Give a greedy algorithm that solves the task on S_{us}.

```python

def bill_exchange_iteratively(k):
    bi = len(bills) - 1
    while k:
        if k >= bills[bi]:
            k -= bills[bi]
            result[0] += 1
            if k == 0:
                break
        else:
            bi -= 1
            result.insert(0, 0)

    print bills
    while len(result) < len(bills):
        result.insert(0, 0)
    print result


if __name__ == '__main__':
    bill_exchange_iteratively(379)

```

3. Show that the greedy algorithm doesn't work if the set S is arbitrarily chosen, by providing a counterexample.

A.2 Lights on the Road

We model a road as a numeric range $[0, L]$ on the x axis, where L is the length of the road. You are given a set S of coordinates within $[0, L]$, which are places on the road that need light. Each light can light up a consecutive range of length K to the right of it. That is, if a light is installed at coordinate x, it can light up the range $[x, x+K]$. Your task is to find the minimum number of lights required so that all the places in S can get light.

For example, let $L = 10, S = \{1, 2, 3, 8, 9\}, K = 2$, the answer is 2. We can install two lights at $x = 1$, covering $[1, 3]$ and $x = 7$, covering $[7, 9]$.

```python
def lights_on_the_road(L, S, K):
    result = []
    need_new_light = True
    k = 0

    for i in range(0, len(S) - 1):
        cur = S[i]
        nex = S[i + 1]
        if need_new_light:
            result.append(cur)
            k = K
            need_new_light = False

        if cur + k < nex:
            need_new_light = True
            # If the last spot is not covered
            if i == len(S) - 2:
                result.append(nex)
        else:
            k -= nex - cur

    print result


if __name__ == '__main__':
    L = (x for x in range(0, 11))
    S = [2, 4, 6, 7, 9]
    K = 3
    lights_on_the_road(L, S, K)
```

```
Solution:  [2, 6]
[Finished in 0.0s]
```

A.3 Assigning Agents

There are n agents and m tasks ($n \geq m$). Denote an agent as A_i (i is the agent id from 1 to n) and a task as T_j (j is the task id from 1 to m). Each agent has a capability $A_i.capability$. Each task has a difficulty $T_j.difficulty$. Agent A_i can solve task T_j if $A_i \geq T_j$. Each agent can solve at most one task.

Now your job is to assign the agents so that all tasks can be solved. If you assign one task to agent A_i, you need to pay $A_i.capability$ dollars. However, you have only a limited budget of k dollars. Design an algorithm that does the assignment, given all the agent and task information, along with your budget k.

```
def assigning_agents(agents, tasks, k):
    agents = sorted(agents)
    tasks = sorted(tasks)
    result = []
    # n   -  current cost
    # ai  -  index of the agents array
    # ti  -  index of the tasks array
    n = ai = ti = 0
    possible = True

    while True:
        if tasks[ti] > agents[ai]:
            ai += 1
        else:
            result.append(agents[ai])

            n += agents[ai]
            ai += 1
            ti += 1

        if ti == len(tasks):
            possible = True
            break
        if ai == len(agents):
            possible = False
            break

    return n <= k and possible, result


if __name__ == '__main__':
    A = [1, 2, 2, 3, 5, 7, 15, 20]
    T = [2, 4, 3, 6, 2]
    k = 20

    is_possible, res = assigning_agents(A, T, k)
    if is_possible:
        print 'Possible', res
    else:
        print 'Impossible Assignment'

```

```
Possible [2, 2, 3, 5, 7]
[Finished in 0.0s]
```

```
if __name__ == '__main__':
    A = [1, 2, 2, 3, 5, 7, 15, 20]
    T = [2, 4, 3, 6, 2]
    k = 18
```

```
Impossible Assignment
[Finished in 0.0s]
```

B Dynamic Programming

B.1 Longest Common Subsequence

In the dynamic programming algorithm for the longest common substring problem, the dynamic programming table $c[i, j]$ stores the longest common subsequence found so far to index pair (i, j). For the strings "abccba" and "bcbca", fill in the dynamic programming table c as shown below.

	a	b	c	c	b	a
b						
c						
b						
c						
a						

B.2 Rod Cutting

Given an initial rod of integer length m and a table that gives the values of rods with integer lengths from 1 to m, you can solve the rod cutting problem (computing the maximum value you can get after cutting) using dynamic programming. Now, suppose cutting the rod introduces a non-negative cost (you lose value if you cut at that position). The cost is different if you cut at different positions. That is, you are given another table that tells you the cost of cutting at each integer position x respectively $(1 \leq x \leq m - 1)$. The goal is still to get the maximum value after cutting. How would you modify your dynamic programming algorithm to solve the new rod cutting problem?

For example, suppose $m = 4$, the values of rods with length $1, 2, 3, 4$ are $0, 10, 1, 0$ respectively and the cutting costs at position $1, 2, 3$ are $0, 20, 0$ respectively. Then instead of cutting the rod into two rods of length 2 (get value 2×10 but cost 20 for cutting, so you get nothing at last), you shall cut at $x = 1$ or $x = 3$ (get value $0 + 1 = 1$ but cost 0 for cutting, so you get 1 at last).

```python
def rod_cutting(prices, cost, n):
    len_p = len(prices)
    dp = [0 for i in xrange(len_p)]
    result = [0 for i in xrange(len_p)]

    rod_cutting_iteratively(prices, cost, n, dp, result)

    k = n - 1 # convert length to index


def rod_cutting_iteratively(prices, cost, n, dp, result):
    for i in xrange(n):
        q = float('-inf')
        for j in xrange(i + 1):
            c = cost[i - j - 1] if j != i else 0
            v = prices[j] + dp[i - j - 1] - c
            # print 'v: ', v, ' cost: ', c, ' i: ', i + 1, 'j: ', j + 1
            if q < v:
                q = v
                result[i] = j + 1
        dp[i] = q


if __name__ == '__main__':
    p = [1, 5, 8, 9, 10, 17, 17, 20, 24, 30]
    c = [1, 1, 2, 0,  2,  7,  5,  2,  1, 0]

    # p = [0, 10, 1, 0]
    # c = [1, 20, 1]

    value, cuts = rod_cutting(p, c, 7)

    print 'total value: ', value
    print 'cuts(cut point / length / value / cost): ', cuts

```

```
dp:     [1, 5, 8, 9, 12, 17, 17, 0, 0, 0]
result: [1, 2, 3, 2, 3, 6, 3, 0, 0, 0]


total value:  17
cuts(cut point / length / value / cost):  [(4, 3, 8, 0), (2, 2, 5, 1), (0, 2, 5, 0)]
[Finished in 0.0s]
```

B.3 Bill Exchange

Recall that the greedy bill exchange algorithm doesn't give the optimal solution with arbitrarily chosen S. Design a dynamic programming algorithm that computes the minimum number of bills required to compose exactly k dollars. The bill value set S only contains positive integers. Give the algorithm and analyze its worst-case running time in terms of k, $|S|$.

```python
import math

def bill_exchange_iteratively_DP(bills, k):
    dp = [0 for i in xrange(k + 2)]
    dp_result = [() for i in xrange(k + 2)]
    bills = bills[::-1]

    for i in xrange(1, k + 1):
        dp[i] = float('+inf')
        for j in xrange(len(bills)):
            if i >= bills[j] and 1 + dp[i - bills[j]] < dp[i]:
                dp[i] = 1 + dp[i - bills[j]]
                dp_result[i] = bills[j]

    print dp[k - 1]
    return dp_result

if __name__ == '__main__':
    S = [1, 4, 5, 9, 10, 20, 50, 90, 94, 100]
    k = 283 # optimal solution: 94 x 3, 1 x 1

    tmp_k = k
    dp_result = bill_exchange_iteratively_DP(S, k)
    while tmp_k > 0:
        print dp_result[tmp_k],
        tmp_k -= dp_result[tmp_k]

    print '\n'

```

```
3
94 94 94 1

[Finished in 0.0s]
```