CS222 Homework 3

Exercises for Algorithm Design and Analysis by Li Jiang, 2016 Autumn Semester

5140309507 林禹臣 yuchenlin@sjtu.edu.cn

1. You are given coins of different denominations and a total amount of money amount.

   Write a function to compute the fewest number of coins that you need to make up that amount.

   If that amount of money cannot be made up by any combination of the coins, return -1.

   Input:

   int coins[];

   int n: length of coins[];

   int amount;

   Output:

   int num;

   Solution. It is a typical Knapsack Problem, which can be easily solved with a dynamic programming strategy.
   We can define a state i of this problem means that the answer when the $amount = i$. Then, we can simply fill the state space and return the state when $amount = amount$. The time complexity of this simple Algorithm 1 is $O(amount * n)$.

Algorithm 1: Compute the fewest number of coins.

Input: int *coins*[],int n, int *amount*
Output: int the minimum of the number of coins

```
1 Initialize all state[] is INF. state[0] = 0;
2 for 1 ≤ i ≤ amount do
3     for each c in coins which ≤ i do
4         state[i] = min{state[i], state[i − c] + 1};
5 Change state[amount] to be -1 if it is INF.
6 return state[amount];
```

□

2. Given a string s, partition s such that every substring of the partition is a palindrome.

   Return the minimum cuts needed for a palindrome partitioning of s.

   For example, given s = "aab",

   Return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

   Input:

   string s;

   Output:

   int cuts;

   Solution. Obviously, this problem is similar to the RNA problem we talked about in the course. This is a typical Interval Dynamic Programming problem, in which we usually have to expand our state space into a 2-dimensional space. This problem is not that simple, however, we can still use an 1-dimensional array to save the states we need and use a 2-dimensional array to save the middle result so that we can reuse them.

   We define that $Res[i]$ means the result from i to n, and thus we can simply return the $Res[0]$ as the final result. Meanwhile, we can define a boolean array $isP[i, j]$ to indicate whether $s[i]...s[j]$ is a palindrome, which we may reuse more than once.

   Here 2 is the detailed algorithm and its time complexity is $O(n^2)$ evidently.

Algorithm 2: Compute the mini cut number to build all palindrome.

Input: String s, int n (the length of s)
Output: int the minimum of the number of cuts

```
1  Initialize all res[n] is 0.
2  Initialize all isP[n,n] is False.
3  state[0] = 0;
4  for n − 1 ≥ i ≥ 0 do
5      res[i] = n − 1 − i;
6      for i ≤ j ≤ n − 1 do
7          if (s[i]==s[j]) AND ( isP[j + 1, i − 1] OR i − j ≤ 1) then
8              isP[i, j] = True;
9              if j == n − 1 then
10                 res[i] = 0;
11             else
12                 res[i] = min{ res[i], res[j + 1] + 1 };
13 return res[0];
```

□

3. Given two arrays of length m and n with digits 0-9 representing two numbers.

   Create the maximum number of length $k \leq m + n$ from digits of the two.

   The relative order of the digits from the same array must be preserved.

   Return an array of the k digits.

   You should try to optimize your time and space complexity.

   Input:

   int nums1[], int m;

   int nums2[], int n;

   int k;

   Output:

   int nums[];

   Solution. Actually, this problem can be seen as a select-and-merge process. Every possible solution is just select x from m elements and $k - x$ from n elements, and then we just merge x and $k - x$ without breaking the original ordering.

   But how should we select x, y form m, n elements so that we can build a maximum result? I could not find a smart way, but the simple way is to try every possible combination. This is to say, we can iterate x from 0 to $min(m, k)$. In each selecting step, we can just find a non-ascending subsequence of $num1[]$ or $num2[]$.
   Here 3 is the algorithm and the time complexity of it is $O(n^2)$.
   Since the inner merge and getSub function is up to $O(n)$ and the outer loop executes $O(n)$ times.
   Note, here we can compare the two array in a python way, in which nums1>nums2 means the number combined by nums1 is greater than that by nums2.

Algorithm 3: Compute the k-maximum from the two array.

Input: int $nums1[], nums2[], m, n, k$
Output: int max

```
1 res = [];
2 for max(0, k − n) ≤ x ≤ min(k, m) do
3     tmp = merge( getSub(nums1, x), getSub(nums2, k − x) );
4     Update res if tmp is greater;
5 }
6 return res;
```

Algorithm 4: Get the non-ascending x-sized subsequence of an n-sized array

Input: int $arr[], n, x$
Output: int $sub[]$

```
1 sub = new stack of int;
2 for 0 ≤ t ≤ n − 1 do
3     while (not sub.empty()) AND sub.size()+(n-t)>x AND arr[t]>sub.top() do
4         sub.pop();
5     sub.push(arr[t]);
6 }
7 return sub;
```

Algorithm 5: Merge the two sub-sequences

Input: int $sub1[], sub2[], m, n$
Output: int $merged[]$

```
1 merged = [];
2 while sub1 is not empty OR sub2 is not empty do
3     if sub1 > sub2 then
4         merged.append(sub1[0]) sub1.delete(0)
5     else
6         merged.append(sub2[0]) sub2.delete(0)
7 return merged;
```

□