# Algorithms Basic Training

Note: If not stated otherwise. The name of the function written is “FUNC”

## Sorting

## Dynamic Programming

### 15.4-5

Input: a sequence of n numbers. Output: longest subsequence of monotonically increasing subsequence of the input.

1. X <- input, Y<-quicksort(input)
2. \_,b<-LCS-LENGTH(X,Y) # as defined on IA p. 394
3. PRINT-LCS(b,X, X.length, Y.length)

Explanation: PRINT-LCS will return the longest common sequence of X and Y -> the longest common sequence of input and sorted input -> the longest sequence that is monotonically increasing and appears in input -> longest monotonically increasing subsequence of input. The most expensive operation in the algorithm is building b, which takes .

### 15.4-6

**Wrong solution (running time)**

Input: a sequence of n number (X), current answer (ans), current largest number (t)

Output: longest subsequence of monotonically increasing subsequence of the input.

1. if len(X)=0 then return ans # finished iterating over input
2. if X[0] < t then return FUNC(x[1:], ans, t) # Can’t add current value, move to the next one and return best solution for it
3. option1 = FUNC(X[1:], ans.append(X[0]), X[0]) # Add current value to ans update threshold and find best solution for the subsequence
4. option2 = FUNC(X[1:], ans, t) # Find best solution when not adding current value to ans.\
5. if len(option1)>len(option2) return option1 else return option2

Input: a sequence of n number (X)

Output: longest subsequence of monotonically increasing subsequence of the input.

1. Thresholds<- array of length X.length+1. Init value = inf
2. SubAns<- array of length X.length+1. Init calue = empty list
3. For i=1…X.length:
   1. j<- KindOfBinarySearch(X[i]-eps) # Find the index of the smallest value in B that is smaller than X[i] – eps is metaphorical.
   2. Thresholds[j+1] = X[i]
   3. SubAns[j+1]<-Subans[j].append(X[i])
4. Return SubAns[k] where k is the largest value with a non-empty list in the corresponding SubAns list.

Explanation: In each iteration, our invariant is that in each cell at Thresholds, we hold the largest value of the corresponding list in SubAns. At the start of the algorithm, it happens naturally because we haven’t had any input. After one iteration, the value of j is 0, meaning that in the end of the iteration: Threshold[1]=X[1] and SubAns[1]=list(X[1]). At a given iteration, for X[m], we will receive the value of the smallest subans that can’t use this number (meaning the next one can) and add X[m] to the following length. The invariant is hasn’t changed for all values except Threshold[j+1] ad SubAns[j+1]. It is also good for the latter because now, after adding X[m] to the solution, we used the updated subsol of length j (which is good from invariant). The running time is because we have n iterations and the loop takes because finding j can be done sing a search algorithm on a sorted array.

## Greedy Algorithms

### 16.1

#### A

Input: value (n)

Output: optimal change: number of quarters, dimes, nickels and pennies that sum to n (q,d,n,p)

1. (q,d,n,p)<-(0,0,0,0), s<-n
2. While s>0:
   1. If s>=q.value then q++ and restart loop
   2. If s>=d.value then d++ and restart loop
   3. If s>=n.value then n++ and restart loop
   4. If s>=p.value then p++ and restart loop
3. Return (q,d,n,p

Explanation: assume the solution is not optimal, meaning there is a solution that uses less coins. This happens only if Opt.q>Func.q or (Opt.q=Func.q and Opt.d>Func.d) etc. The situation is prevented from the structure of the loop: as long as we can increase the number of coins with larger values we will do so.

#### B

Note: the algorithm is the same as before but with the given coins (so it examines the coins from the largest to the smallest based on value). Proof similar to the explanations from A.

#### C

Assuming the coins are in the values of {1,4,5}. The algorithm will not yield an optimal solution. It will return (3,0,1) although the optimal solution is (0,2,0).

#### D

Input: value (n), current solution (c1,c2,c3…) # current solution is the number of coins per coin type. Init with zeros

1. If n = 0 then return (c1,c2,c3…)
2. If n<0 then return (inf, …)
3. bestSol = null, bestCost=inf
4. for i=1…k:
   1. opt = Func(n-ci.value, (c1,c2…,ci+1,…,ck))
   2. if bestCost > sum(opt) then bestSol=opt, bestCost = sum(opt)
5. return bestSol

Explanation: it keeps the invariant that for each m<n it returns an optimal solution. The base case (m=0) is trivial. For any m, it returns the best solution among the optimal solutions of m-ck for each possible k and adds 1 to it. The running time is because the inner loop runs k times and the recursion happens n times (each time n is decreased by at least 1).

## Graphs