# 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

### 5.13

#### A

Input: Tree (V,E)

Output: minimal V’ such that for each e=(u,v) in E, v or u in V’.

1. Parent<-array of size |V|. Calculate values using BFS so Parent[v] <- parent of v in the graph
2. G’=(V,E) # copy input
3. While G’.E not empty:
   1. For each leaf l in G’.v:
      1. Insert Parent[l] to V’
      2. Remove l, Parent[l] and their corresponding edges from G’.
4. Return V’.

Explanation: The solution is valid because we always take at least one vertex from each edge (V or its parent). If the solution is not optimal, this means we have a vertex to remove while keeping the solution valid. Since for each (u,v) in E, v or u is in V’, both of them were in V’ to begin with. Assuming v is the parent of u, and u has no other son, this means either we examined v as a leaf of some G’ or that we examined a son of v. either way, we take u (and not v) or v (and not u). If u has another son, this must mean at some iteration we examined that other son, k, as a leaf, took u and also took v because it is a parent. In this case, we can’t remove u because the (u,k) won’t be in the graph and can’t remove v because the edge with it son won’t be in the graph.

#### B

Same as A, but in this case we take the leaves and not the parents because the weight of n leaves is n and the weight of one parent is n+1. The solution holds using the same explanation.

#### C

???

### 5.17

#### A

Input: undirected graph (V,E)  
Output: True if G contains cycle of length 3, False otherwise

1. For each u,v,t in V:
   1. If (u,v), (u,t)m (v,t) in E return True
2. Return False

#### B

Input: undirected graph (V,E)  
Output: True if G contains cycle of length 3, False otherwise

1. M <- matrix of |V|x|V| where M[I,j]=1 if (I,j) in E and 0 otherwise
2. for each u,v in V:
   1. commonNeighbor <- M[,u] & M[,v] # calculate vertex that are neighbors of v and u
   2. if commonNeighbor>0 and (u,v) in E then return True
3. return False

Explanation: A cycle of size 3 means u and v have a common neighbor and an edge between them. Calculating the matrix costs, finding commonNeighbor can be done in because we can use bitwise ‘and’. The loop itself is done.

## MST

### 23.2-7

Input: Graph (V,E) and it’s corresponding spanning tree (S), new vertex v and its corresponding edges E’

1. Run DFS with the new vertex as root in order to find all cycles with the new vertex
2. From each cycle, remove the heaviest edge until we have n-1 edges (n being the new number of vertex).

Explanation: removing edges from each cycle and having n-1 edges, we will end up with a spanning tree. Assuming it won’t be minimal, it means we can replace an edge with another and getting a lighter spanning tree. Looking at this edge and adding it to our tree, we will have a cycle. If the new vertex is not in this cycle, it contradicts the fact we had an MST to begin with. If the new vertex is in this cycle, it contradicts the fact that for each cycle, we dropped the heaviest edge.

## Shortest Path

#### 24.3-8

Input: Weighted, directed graph (V,E) with nonnegative weight function E->{0,…,W}, node s in V.

Output: Shortest paths from s.

1. Arr<- array of |V|\*W+1 empty lists # Arr[i] will hold the vertices reachable from s in i steps
2. Arr[0].append(s)
3. Arr[|V|\*W+1].append(|V|\{S}) # initialize distance to be max
4. Dist<- array of |V|, init with inf # distances
5. Prev<- array of |V|, init with null # parents
6. Q<-|V|
7. Dist[s]<-0
8. lastIndex=0
9. While Q not empty:
   1. For i=lastIndex,…,|V|\*W: if Arr[i] is not empty then break # Find next candidate
   2. U<- Arr[i][0] # Take the first vertex with distance d from s
   3. Remove u from Q and from Arr[i]
   4. For each v neighbor of u:
      1. If Dist[u]+w(u,v) < Dist(v):
         1. Arr[Dist[v]].remove(v)
         2. Dist[v] = Dist(u)+w(u,v)
         3. Prev[v] = u
         4. Arr[Dist[v]].add(v)

Explanation: The algorithm remains the same but we find the minimum using an array where cell j holds the vertices whose current distance from s is j. In this way, we can find the next candidate faster. In particular, since the value of i is always increasing, the loop takes in total. The rest is as in the original version of Dijkstra: in total.

## Linear Programming

### 29.2-3

Goal: maximize

Subject to:

## Computational Geometry

### 33.2-5

Input: 2 polygons with n vertices total.

Output: True if they intersect, False otherwise

1. E<- all edges (u1,v1) in polygon1 and (u2,v2) in polygon2
2. If polygon1 and polygon2 share a vertex then return True
3. If any e in E is vertical, rotate in so none will be
4. Ans<- ANY-SEGMENTS –INTERSECT(E) but ignore intersections that occur when a vertex appears in two different edges
5. Return ans

Explanation: We have n vertices in total meaning we have edges in total. We use operations and an algorithm that runs in and also proves correctness.

### 33.3-6

## Heuristic Methods