**P1. M09043BB**

1. Greedy method: A greedy algorithm is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) that follows the [problem solving](http://en.wikipedia.org/wiki/Problem_solving) [heuristic](http://en.wikipedia.org/wiki/Heuristic_(computer_science)) of making the locally optimal choice at each stage with the hope of finding a global optimum.
2. Longest common subsequence (LCS) problem: The longest common subsequence (LCS) problem is to find the longest [subsequence](http://en.wikipedia.org/wiki/Subsequence) common to all sequences in a set of sequences.
3. Optimal substructure property: a problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming and greedy algorithms for a problem.
4. A [spanning tree](http://en.wikipedia.org/wiki/Spanning_tree_(mathematics)) of that graph is a [subgraph](http://en.wikipedia.org/wiki/Glossary_of_graph_theory#Subgraphs) that is a [tree](http://en.wikipedia.org/wiki/Tree_graph) and connects all the [vertices](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) together. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its [connected components](http://en.wikipedia.org/wiki/Connected_component_(graph_theory)).

**3.**

**(a) What is dynamic programming?**

**Dynamic programming** is typically applied to optimization problems. In such problem there can be many solutions. Each solution has a value, and we wish to find a solution with the optimal value.

**The development of a dynamic programming algorithm** can be broken into a sequence of four steps:

**1.** Characterize the structure of an optimal solution.

**2.** Recursively define the value of an optimal solution.

**3.** Compute the value of an optimal solution in a bottom up fashion.

**4.** Construct an optimal solution from computed information.

**(b) Write a dynamic-programming algorithm to calculate C (n, k), the number of k-combinations (i.e., k-element subsets) of an n-element set. Use the formulas:**

C (n, k) = C (n - 1, k - 1) + C (n - 1, k)

valid for 1 <= k <= n - 1, and C (n, n) = 1 = C (n, 0) valid for n >= 0.



Combine(n, k)

for i = 0 to n do

C[i, i] = 1

C[i, 0] = 1

for i = 2 to n do

for j = 1 to n – 1 do

C[i, j] = C[i – 1, j – 1] + C[i – 1, j]

return C[n, k]

Note that C[n, k] is a symmetric matrix, so we can modify code as follows:

Combine(n, k)

m = k

if (m > n / 2) then m= n / 2

for i = 0 to n do

C[i, 0] = 1

for i = 0 to m do

C[i, i] = 1

for i = 2 to n do

for j = 1 to n – 1 do

if (j <= m)

C[i, j] = C[i – 1, j – 1] + C[i – 1, j]

if (k > n – k) then

return C[n, n – k]

return C[n, k]

**(c) The worst-case time of this algorithm is O (n2)**

**(d) Trace your algorithm for C (7, 5):**



**P2. M053AC2A**

**2.**

1. **Asymptotically upper bound the function f (n) = n2− 2n + 1 by the O notation. Justify your answer by demonstrating the constants.**

We have f(n) = (n – 1)2 <= n2 if n > 0 => f(n) = O(n2)

1. **Asymptotically lower bound the function f (n) = n2− 2n + 1 by the Ω notation. Justify your answer by demonstrating the constants.**

We have f(n) = (n – 1)2 >= n2 – 4n + 4 = (n-2)2 if n > 0 => f(n) = Ω (n2)

**3.**

**(a) Find a longest increasing subsequence of {3, 7, 5, 9, 2, 6, 4}.**

A longest increasing subsequence of this array is {3, 7, 9} with length is 3.

**(b) Describe an algorithm that solves the longest increasing subsequence problem in O(n2) time.**

**LongestIncreasingSubsequence(A)**

// A[] is the input array (a sequence)

// M[*i*] indicates the longest length from first element to the *i*th element, default M[*i*] = 1

// T[*i*] indicates the index of back element in the subsequence, default T[*i*] = i

// At the A[*i*], we find A[*j*], a previous element A[*i*], so that A[*j*] < A[*i*] and M[*j*] >= M[*i*]

// then increase M[*i*] by 1.

max = 1 // indicates the last index of the longest increasing subsequence

for i = 1 to n do

M[i] = 1

T[i] = i

for j = i - 1 down to 1 do

if A[j] < A[i] and M[j] >= M[i] then

M[i] = M[j]

T[i] = j;

M[i] = M[i] + 1

if M[i] > M[max] then

max = i

// Print LIS

do

print A[max]

max = T[max]

while T[max] <> max