1. Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x-coordinates $a_1 \dots a_n$ and n cities on the northern bank with x-coordinates $b_1 \dots b_n$. You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. When connecting cities, you are only allowed to connect the the i^{th} city on the northern bank to the i^{th} city on the southern bank.

Solution:

Consider the sequences $A = N(a_1), ..., N(a_n)$ and $B = N(b_1), ..., N(b_n)$ where $N(a_i)$ is the number of the city with x-coordinate a_i. The length of the longest common subsequence of A and B is the maximum number of bridges. Since A and B are non-repeating, you showed in problem set 6 that the length of the LCS for A and B can be calculated in $O(n \log n)$ time. Make sure you understand why that is the case!

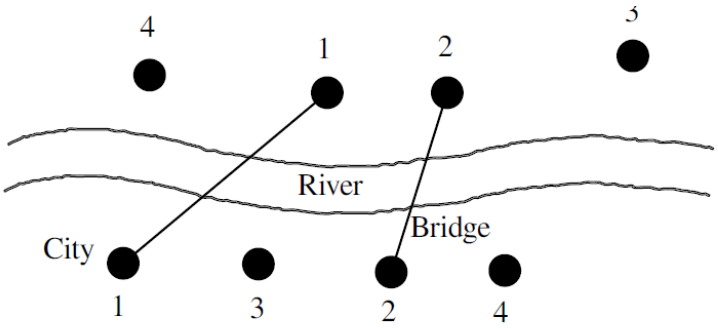


Figure 1: An example of a valid bridge building.

Proof that length of the LCS is the maximum number of bridges: We show that the maximum number of bridges cannot be more than the length of the LCS and that the maximum number of bridges cannot be less than the length of the LCS.

Firstly, assume the length of the LCS is m. Let $c_1, ..., c_m$ be a longest common subsequence of A and B, corresponding to cities $a_{i_1}, ..., a_{i_m}$ in A and $b_{j_1}, ..., b_{j_m}$ in B. Then for $0 < k \leq m$, we can draw a bridge from a_{i_k} to b_{i_k}. None of these bridges intersect. Therefore, we can draw at least as many bridges as the length of the LCS.

Now assume we can draw at most m bridges from cities $C_A = a_{i_1}, ..., a_{i_m}$ to cities $C_B = b_{j_1}, ..., b_{j_m}$ and WLOG assume C_A is ordered by increasing x-coordinate. Then $N(a_{i_k}) = N(b_{j_k})$ since we can draw a bridge between them. Moreover, b_{j_k} must have a higher x-coordinate than any of $b_{j_1}, ...b_{j_{k-1}}$ and a lower x-coordinate than any of $b_{j_{k+1}}, ..., b_{j_m}$ so that none of the bridges cross. Therefore C_A is a subsequence of A and C_B is a subsequence of B and we have found a common subsequence. Thus, the length of the LCS is at least the maxmimum number of bridges.

```java
public static int lis(Pair[] A){
    Arrays.sort(A, new Comparator<Pair>() {
        public int compare(Pair o1, Pair o2) {
            return o1.getX() - o2.getX();
        }
    });

    int n = A.length;
    int max = 0;
    int[] dp = new int[n];
    Arrays.fill(dp, val: 1);

    for(int i=1; i<n; i++){
        for(int j=0; j<i; j++){
            if(A[i].getX() > A[j].getY()){
                dp[i] = Math.max(dp[i], dp[j]+1);
            }
        }
        max = Math.max(max, dp[i]);
    }

    return max;
}
```

3. You are given a set of n types of rectangular boxes, where the i^{th} box has height h_i, width w_i and depth d_i. You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box. (10pts)

Solution:

Recursion: Memoize over $H(j, R)$, the tallest stack of boxes with j on top with rotation R.

$$H(j,R) = \begin{cases} 0 & \text{if } j = 0 \\ \max_{i<j \text{ with } w_i>w_j, d_i>d_j}(H(i,R) + h_j) & \text{if } j > 0 \end{cases} \tag{4}$$

Running Time: The size of H is $O(n|R|)$ where R is the number of possible rotations for a box. For our purposes, $|R| = 3$ (since we only care about which dimension we designate as the "height") so $|H| = O(n)$. Filling in each element of H is also $O(n)$ for a total running time of $O(n^2)$.

The adapted recurrence is: D[i] = maximum height we can obtain if the last tower must be i.

```
D[1] = h(1);
D[i] = h(i) + max(D[j] | j < i, we can put block i on top of block j)

Answer is the max element of D.
```

4. You are traveling by a canoe down a river and there are n trading posts along the way. Before starting your journey, you are given for each $1 \leq i < j \leq n$ the fee $F(i, j)$ for renting a canoe from post i to post j. These fees are arbitrary. For example it is possible that $F(1, 3) = 10$ and $F(1, 4) = 5$. You begin at trading post 1 and must end at trading post n (using rented canoes). Your goal is to design an efficient algorithms which produces the sequence of trading posts where you change your canoe which minimises the total rental cost. (10pt)

Solution. First we will define a function *Cheap* that gives the cost of cheapest sequences of canoe rentals, and then we will find a recurrence for *Cheap*.

We define, for $1 \leq k \leq n$, the function *Cheap*(k) to be the cheapest cost of canoe rentals from trading post 1 to k. $R(i,j)$ represents the cost of renting a canoe at node i and return it at j, with no intermediate rentals.

$$\text{Cheap}(k) = \begin{cases} 0 & \text{if } k = 1 \\ \min_{1 \leq i < k}\{\text{Cheap}(i) + R(i,k)\} & \text{otherwise} \end{cases} \quad (1)$$

This recurrence can be justified by considering an optimal (cheapest) sequence of canoe rentals ending at some post $k > 1$. This sequence must contain a last canoe rental from some other post i to post k. The cost of this optimal sequence is then the cost of an optimal rental sequence from post 1 to post i, plus the cost of the last rental form post i to post j. Since all possibilities for this last rental are considered, the recurrence correctly computes the costs of optimal rental sequences.

Note that it would be an error to define Cheap(k) using $\min_{1 \leq i \leq k}\{\text{Cheap}(i) + R(i,k)\}$ as this would be defining Cheap(k) in terms of itself.

Second, describe an appropriate table for remembering previously computed values in order to reduce the work done by the straightforward recursive algorithm for computing the values of *cheap*.

We will keep an one-dimensional array $C[1 \ldots n]$ where $C[k] = Cheap(k)$.

Third, we now outline an iterative procedure for filling in the table.

$C[1 \ldots n]$ can be filled in in the following fashion:

```
procedure Cost(R[1...n][1...n])
begin
(1)  C[1]=0
(2)  for i=2 to n do            /*  compute C[i] */
(3)    C[i] = C[i-1] + R(i-1,i)
(4)    for j=i-2 down to 1 do
(5)      if C[j] + R(j,i) < C[i] then
(6)        C[i] = C[j] + R(j,i)
(7)      end do
(8)    end do
(9)  end do
end
```

lines (3)–(8) compute the *minimum* over last rentals in the recurrence.

Another version

Let $m[i]$ be the rental cost for the best solution to go from post i to post n for $1 \leq i \leq n$. The final answer is in $m[1]$, and $m[n] = 0$. The optimal cost for traveling starting at post i is given by $m[i] = \min_{j:i<j\leq n}(m[j] + f_{i,j})$.

The canoe must be rented starting at post i (the starting location) and then returned next at a station among $i+1, \ldots, n$. We try all possibilities for the next (with j being the station where the canoe is next returned). For the time complexity there are n subproblems to be solved each of which takes $O(n)$ time. These subproblems can be computed in the order $m[n], m[n1], \ldots, m[1]$. Hence the overall time complexity is $O(n^2)$.

Extras

Exercise: Use a similar reasoning to solve the following problem: Given a sequence of n numbers find the longest increasing subsequence **in nlog n steps**.

```
class LIS
{
    // Binary search (note boundaries in the caller)
    // A[] is ceilIndex in the caller
    static int CeilIndex(int A[], int l, int r, int key)
    {
        while (r - l > 1)
        {
            int m = l + (r - l)/2;
            if (A[m]>=key)
                r = m;
            else
                l = m;
        }

        return r;
    }

    static int LongestIncreasingSubsequenceLength(int A[], int size)
    {
        // Add boundary case, when array size is one

        int[] tailTable   = new int[size];
        int len; // always points empty slot

        tailTable[0] = A[0];
        len = 1;
        for (int i = 1; i < size; i++)
        {
            if (A[i] < tailTable[0])
                // new smallest value
                tailTable[0] = A[i];

            else if (A[i] > tailTable[len-1])
                // A[i] wants to extend largest subsequence
                tailTable[len++] = A[i];

            else
                // A[i] wants to be current end candidate of an existing
                // subsequence. It will replace ceil value in tailTable
                tailTable[CeilIndex(tailTable, -1, len-1, A[i])] = A[i];
        }

        return len;
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int A[] = { 2, 5, 3, 7, 11, 8, 10, 13, 6 };
        int n = A.length;
        System.out.println("Length of Longest Increasing Subsequence is "+
                           LongestIncreasingSubsequenceLength(A, n));
    }
}
```

18. (Escape Problem) This is a problem which is faced by people who design printed circuit boards. An $n \times n$ grid is an undirected graph consisting of n rows, each row containing n vertices, with vertices connected to all of their immediate neighbours (2 at all of the 4 corners, 3 at all of the 4 sides and 4 in the interior of the grid. The escape problem is, given $m \leq n^2$ many vertices in the grid, connect them with m many distinct vertices located on the 4 sides by non intersecting paths or return "impossible" when there is no such a solution.

Sometimes, some of the information is actually spurious, as in the following example (do you really need the capacities of edges?)

Solution: To use max-flow algorithm, we need to add a source node s and a destination node t to the grid. We connect s to each of the m starting points and draw a link from every boundary point to t. We replace grid edges with two directed edges and give a capacity of one to all edges in the graph. Because all capacities are integral, the max flow will be integral, so we may find feasible edge-disjoint paths. If the size of maximum-flow in the resulting graph is m then there exists m edge-disjoint paths from each of the m starting points to a boundary point.

Note that to ensure correctness, we need to consider the case when both directed edges between a pair of grid nodes have flow in the output max flow. In this case, the output flow is not edge disjoint. However, we may remove the flow on both these edges without

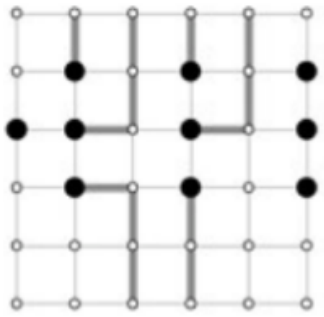


Figure 1: Grid for the escape problem. Starting points are black, and other grid vertices are white.

changing any feasibility and without affecting the size of the flow. Therefore, these cases do not affect the size of the maximum flow. To actually find the paths, we first need to remove all these bi-directional flows between grid nodes.

To find the m edge-disjoint paths, we start from s and follow the edges with positive flow to reach t, call this path P. P is clearly an escape path from one of the starting points to a boundary point. We remove edges of P from the flow network, it is easy to observe that the remaining flow is still feasible. We can do this $m-1$ more times to retrieve all the m escape paths.

Max Flow

Dining Model

•*n* families go out to a dinner together to a restaurant. Assume that the number of members in the families is given by a(1), a(2), a(3), … a*(n)* members respectively.

•At the restaurant there are *m* tables, and that the seating capacity at the tables are given by c(1), c(2), c(3), … c*(m)* respectively.

•To increase their social interaction, they would like to sit at tables *so that no more than k members of the same family are seated at the same table*. Hence, if $k = 1$, there is atmost one member form the same family at a table.

•**Objective:** Is it possible to obtain a seating arrangement that satisfies this requirement ? If so, describe it.

In practical applications,

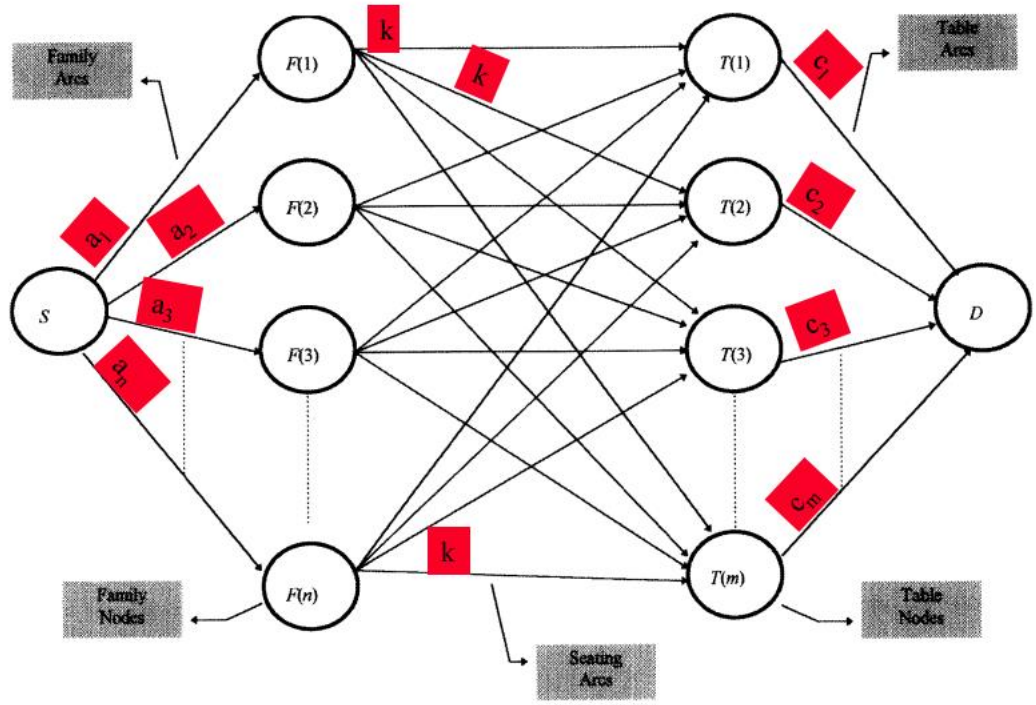
Families are employees with different backgrounds

Tables are projects.

And the objective is to assign people to projects such that each team is as "heterogeneous" as possible with regards to the backgrounds of its family members.

This problem can be formulated as a Max Flow Problem

Application of Max-Flow Model to Assignment Problems



Application of Max-Flow Model to Assignment Problems

•In the graph, the nodes are as follows:

1. The source node S and the destinations/sink node D
2. n Family nodes F(1), F(2), … F(n) where each node F(i) represents family i.
4. M Table nodes T(1). T(2), T(3), …, T(m), where each node T(j) represents table j.

•In the graph, the arcs are as follows:

1. The family arcs go from source node S to each of the family nodes F(i). The capacity of a family arc is assumed to be equal to the number of members in that family. Hence, the capacity of arc (S, F(1)) is a(1).
2. The table arcs go from each table node T(j) to the destination node D. The capacity of a table arc is assumed to be equal to the seating capacity of that table. Hence, the capacity of the table arc (T(1), D) is c(1).
3. The seating arcs that go form each family node F(i) to each table node T(j). The capacity of each of these arcs is set at k, which is the maximum number of members form each family that are permitted to be at the same table.

•As one can see, a flow of x units on the seating arc (F(i), T(j)) indicates that x members form family i should be seated at table j.

•Now solve the Max.-Flow problem form S to D. If the max flow is equal to (a(1) + a(2) + a(3) + … + a(n)), then we are satisfied that all family members can be seated with the above constraint.

•Else, relax the parameters (increase value of k or increase the number of tables, or increase capacity of the tables etc.) and solve the problem again.

5. Given a sequence of n real numbers $A_1 \ldots A_n$, determine *in linear time* a contiguous subsequence $A_i \ldots A_j$ for which the sum of elements in the subsequence is maximised. (10pts)

```
public int solve(int [] a){
    int [] solution = new int[a.length+1];
    solution[0] = 0;

    for (int j = 1; j <solution.length ; j++) {
        solution[j] = Math.max(solution[j-1]+a[j-1],a[j-1);
    }
    //this will print the solution matrix
    //System.out.println(Arrays.toString(solution));
    //now return the maximum in the solution array
    int result = solution[0];
    for (int j = 1; j <solution.length ; j++) {
        if(result<solution[j])
            result = solution[j];
    }

    return result;
}
```

Recursion: We recurse on the maximum value subsequence ending at j:

$$M(j) = \begin{cases} a_j & \text{if } j = 0 \\ \max(M(j-1) + a_j, a_j) & \text{else} \end{cases} \quad (1)$$

With each element of M, you also keep the starting element of the sum (the same as for $M(j-1)$ or j if you restart). At the end, you scan M for the maximum value and return it and the starting and ending indexes. Alternatively, you could keep track of the maximum value as you create M.

Running Time: M is size n and evaluating each element of M takes $O(1)$ time for $O(n)$ time to create M. Scanning M also takes $O(n)$ time for a total time of $O(n)$.

6. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation and, or, xor between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true* and *false* xor *true* such that it evaluates to true.

Solution:

Recursion: Let our expression consist of n atomic elements (TRUE, FALSE) and $n-1$ operators (AND, OR, XOR). The ith atomic element is a_i and the ith operator is o_i so that our expression is $a_1 o_1 a_2 o_2 ... a_{n-1} o_{n-1} a_n$. We keep $T(i,j)$, the number of parenthesizations that makes the expression between a_i and a_j true, and $F(i,j)$, the number of parenthesizations that makes the expression between a_i and a_j false. The recursion for each is:

$$T(i,j) = \sum_{k=i}^{j} \begin{cases} T(i,k)T(k+1,j) & \text{if } o_k = \text{and} \\ (T(i,k)+F(i,k))(T(k+1,j)+F(k+1,j)) - F(i,k)F(k+1,j) & \text{if } o_k = \text{or} \\ T(i,k)F(k+1,j) + F(i,k)T(k+1,j) & \text{if } o_k = \text{xor} \end{cases} \quad (6)$$

$$F(i,j) = \sum_{k=i}^{j} \begin{cases} (T(i,k)+F(i,k))(T(k+1,j)+F(k+1,j)) - T(i,k)T(k+1,j) & \text{if } o_k = \text{and} \\ F(i,k)F(k+1,j) & \text{if } o_k = \text{or} \\ F(i,k)F(k+1,j) + T(i,k)T(k+1,j) & \text{if } o_k = \text{xor} \end{cases}$$

Running Time: There are $O(n^2)$ elements in each of T and F and filling in each element takes $O(n)$ time for a total running time of $O(n^3)$.

```
int n = operands.length;

int T[n][n];
int F[n][n];

for (int i=0; i<n; i++)
{
    F[i][i] = (symb[i] == 'F')? 1: 0;
    T[i][i] = (symb[i] == 'T')? 1: 0;
}

for (int gap=1; gap<n; gap++) // loop required to fill whole of the matrices T and F
{
    for (int i=0, j=gap; j<n; i++, j++) // vary i,j from 0 to n
    {
        T[i][j] = F[i][j] = 0;
        for (int k=i; k<j; k++) // vary k from i to j
        {
            // above equations
        }
    }
}

return T[0][n-1];
```

7. You have n_1 items of size s_1, n_2 items of size s_2, and n_3 items of size s_3. You would like to pack all of these items into bins, each of capacity C, using as few bins as possible.

DP method Suppose Bin(i, j) gives the min total number of bins used, then Bin(i, j) = min{Bin(i', j') + Bin(i − i',j − j')} where i + j > i' + j' > 0. There will be n^2 − 2 different (i',j') combinations and one pair of (n1,n2) combination. So the complexity is about O(n^2).

Complexity O(n^2)

Example: Let s1 = 3, n1 = 2, s2 = 2, n2 = 2, C = 4. Find the min bins needed, i.e., b.

```
<pre>
i j b
- - -
0 1 1
0 2 2
1 0 1
1 1 1
1 2 2
2 0 2
2 1 3
2 2 3  -> (n1,n2) pair
</pre>
```

So as you can see, 3 bins are needed.

```
<pre>
Note that Bin(2,2) = min{
  Bin(2,1) + Bin(0,1),
  Bin(2,0) + Bin(0,2),
  Bin(1,2) + Bin(1,0),
  Bin(1,1) + Bin(1,1)}
= min{3, 4}
= 3
</pre>
```

15. You are given an ordered sequence of n cities, and the distances between every pair of cities. You must partition the cities into two subsequences (not necessarily contiguous) such that person A visits all cities in the first subsequence (in order), person B visits all cities in the second subsequence (in order), and such that the sum of the total distances travelled by A and B is minimised. (30pt)

Solution:

Recursion Relation: We recurse on $C(i, j)$, the minimum distance traveled if person A ends at city i and person B ends at city j. Assume WLOG $i < j$. The relation is:

$$C(i,j) = \begin{cases} \sum_{k=1}^{j-1} d(k, k+1) & \text{if } i = 0 \\ \min_{0<k<i}(C(k,j) + d(k,i)) & \text{else} \end{cases} \tag{10}$$

where $d(i, j)$ is the distance between cities i and j.

Running Time: There are n^2 entries in $C(i, j)$ and filling in each entry takes $O(n)$ for a total of $O(n^3)$.

11. A palindrome is a sequence of at least three letters which reads equally from left to right and from right to left.

    (a) Given a sequence of letters S, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters. (20pt)

    (b) Find the total number of occurrences of all subsequences of S which are palindromes of any length ≥ 3. (25pt)

```
// Returns the length of the longest palindromic subsequence in seq
static int lps(String seq)
{
   int n = seq.length();
   int i, j, cl;
   int L[][] = new int[n][n];  // Create a table to store results of subproblems

   // Strings of length 1 are palindrome of lentgh 1
   for (i = 0; i < n; i++)
       L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
        {
            j = i+cl-1;
            if (seq.charAt(i) == seq.charAt(j) && cl == 2)
               L[i][j] = 2;
            else if (seq.charAt(i) == seq.charAt(j))
               L[i][j] = L[i+1][j-1] + 2;
            else
               L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}
```