# Dynamic Programming (DP)

**MD. Mujahedul Azad (Turin)**
**Hajee Mohammad Danesh Science & Technology University (HSTU)**

# Content

## Edit Distance

Given two strings str1 and str2 and below operations that can performed on str1. Find the minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

1. Insert
2. Remove
3. Replace

**Recursive Method:**

**Complexity : O(n*n)**

**Space: O(n*n)**

```
int dp[mx][mx];
int edr(int i, int j){
   if(i == n) return (m-j);
   if(j == m) return (n-i);

   if(dp[i][j] != -1) return dp[i][j];

   int ans = INF;
   if(s[i] == t[j]) ans = min(ans, edr(i+1, j+1));
   else ans = min(ans, 1+min3(edr(i, j+1), edr(i+1, j), edr(i+1, j+1)));
   return dp[i][j] = ans;
}
```

**Iterative Method:**

**Complexity : O(n*n)**

**Space: O(n*n)**

```
int edi(){
   for(int i=0; i<=n; i++){
      for(int j=0; j<=m; j++){
         if(i == 0) dp[i][j] = j;
         else if(j == 0) dp[i][j] = i;
         else if(s[i-1] == t[j-1])
            dp[i][j] = dp[i-1][j-1];
         else dp[i][j] = 1 + min3(dp[i][j-1], dp[i-1][j], dp[i-1][j-1]);
      }
   }
   return dp[n][m];
}
```

## Egg Drop Puzzle

Finding the minimum number of eggs for find the pivot point on the building

### 1. Complexity : O(n * e)

```
int dp[mx][mx];
int edr(int n, int e){
   if(n <= 1) return n;
   if(e == 1) return n;

   if(dp[n][e] != -1) return dp[n][e];
   int ans = INF;
   for(int x=1; x<=n; x++)
      ans = min(ans, 1+max(edr(n-x, e), edr(x-1, e-1)));
   return dp[n][e] = ans;}
```

### 2. Complexity : O(n * e)

```
int EggDropIte(int n, int e){
   for(int i=0; i<=e; i++)
      dp[0][e] = 0, dp[1][e] = 1;
   for(int i=0; i<=n; i++)
      dp[i][1] = i;
   for(int i=2; i<=n; i++)
   for(int j=2; j<=e; j++){
      dp[i][j] = INF;
      for(int x=1; x<=i; x++)
         dp[i][j] = min(dp[i][j], 1+max(dp[i-x][j], dp[x-1][j-1]));
   }
   return dp[n][e];
}
```

### 3. Complexity : O(e * log n)

```
int sumOf_xCi(int x, int n){
   int sum = 0, prod = 1;
   for(int i=1; i<=n; i++){
      prod *= (x-i+1);
      prod /= i;
      sum += prod;
   }
   return sum;
}
```

```
int EggDropBinary(int n, int e){
    int lw = 1, hh = n;
    while(lw < hh){
        int mid = (lw + hh) / 2;
        if(sumOf_xCi(mid, e) < n)
            lw = mid + 1;
        else hh = mid;
    }
    return lw;
}
```

## Integer Knapsack

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack

```
int n = 5, W = 236;
int wt[] = {10, 20, 30, 40, 50};
int val[] = {25, 59, 48, 36, 78};
```

### 1. Complexity : O(nW), Space : O(nW)

```
int dp[mx][mx];
int ksr(int idx, int w){
    if(idx == n) return 0;
    if(w == 0) return 0;

    if(dp[idx][w] != -1) return dp[idx][w];

    int ans = 0;
    if(w<wt[idx])
        ans = ksr(idx+1, w);
    else ans = max3(ans, ksr(idx+1, w),
                    val[idx]+ksr(idx+1, w-wt[idx]));
    return dp[idx][w] = ans;
}
```

**2. Complexity : O(nW), Space : O(nW)**

```
int knapsackIte(int n, int w){
   for(int i=0; i<=n; i++)
   for(int j=0; j<=w; j++)
      if(i==0 || j==0) dp[i][j] = 0;
      else if(wt[i-1] > j)
         dp[i][j] = dp[i-1][j];
      else dp[i][j] = max(val[i-1] + dp[i-1][j-wt[i-1]],
                          dp[i-1][j]);
   return dp[n][w];}
```

**3. Complexity : O(nW), Space : O(2W)**

```
int dpOpt[2][mx], k=0;
int knapsackIteOptimized(int n, int w){
   for(int i=0; i<=n; i++, k=!k)
   for(int j=0; j<=w; j++)
      if(i==0 || j==0) dp[k][j] = 0;
      else if(wt[i-1] > j)
         dpOpt[k][j] = dpOpt[!k][j];
      else dpOpt[k][j] = max(val[i-1] + dpOpt[!k][j-wt[i-1]],
                          dpOpt[!k][j]);
   return dpOpt[!k][w];
}
```

**4. Complexity : O(nW), Space : O(W)**

```
int dpOpt2[mx];
int knapsackIteOptimized2(int n, int w){
   memset(dpOpt2, 0, sizeof dpOpt2);
   for(int i=0; i<n; i++)
   for(int j=w; j>=wt[i]; j--)
      dpOpt2[j] = max(dpOpt2[j], val[i]+dpOpt2[j - wt[i]]);
   return dpOpt2[w];
}
```

```
/// required knapsackIte()
void printKS(){
    int res = knapsackIte(n, W), w = W;
    for(int i=n; i>0 && res>0; i--)
        if(res == dp[i-1][w]) continue;
        else{
            /// do something, cout << wt[i-1] << " ";
            res -= val[i-1]; w -= wt[i-1];
        }
    cout nl;
}

// moderate : if there are double bag
int dbDp[mx][mx][mx];
int doubleKS(int idx, int w1, int w2){
    if(idx == n) return 0;
    if(dbDp[idx][w1][w2] != -1)
        return dbDp[idx][w1][w2];

    int ans=0, ans1=0, ans2=0;
    if(w1 >= wt[idx])
        ans1 = val[idx] + doubleKS(idx+1, w1-wt[idx], w2);
    if(w2 >= wt[idx])
        ans2 = val[idx] + doubleKS(idx+1, w1, w2-wt[idx]);
    ans = doubleKS(idx+1, w1, w2);
    ans = max3(ans, ans1, ans2);
    return dbDp[idx][w1][w2] = ans;
}

/// if multiple instances allowed
int unboundedKSR(int idx, int w){
    if(idx == n) return 0;

    if(dp[idx][w] != -1) return dp[idx][w];

    int ans = 0;
    if(w >= wt[idx])
        ans = max(ans, val[idx]+unboundedKSR(idx, w-wt[idx]));
    ans = max(ans, unboundedKSR(idx+1, w));
    return dp[idx][w] = ans;
}
```

### /// Memory Optimized unbounded KS

```
int unboundDp[mx];
int unboundedKSI(int n, int w){
    for(int i=0; i<=w; i++)
    for(int j=0; j<n; j++)
        if(wt[j] <= i)
            unboundDp[i] = max(unboundDp[i], unboundDp[i - wt[j]]+val[j]);
    return unboundDp[w];
}
```

## Largest Independent Set

Given a Binary Tree, find the size of the **L**argest **I**ndependent **S**et(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

```
int LIS(int u){
    if(dp[u] != -1) return dp[u];

    int ans1 = 0, ans2 = 1;
    for(int i=0; i<adj[u].size(); i++){
        ans1 += LIS(adj[u][i]);
        int v = adj[u][i];
        for(int j=0; j<adj[v].size(); j++)
            ans2 += LIS(adj[v][j]);
    }
    return dp[u] = max(ans1, ans2);
}
```

## Longest Increasing Subsequence

### 1. Complexity : O(n*n), Space : O(n)

```
int dp[mx];
int lis(){
    dp[0] = 1;
    for(int i=1; i<n; i++){
        dp[i] = 1;
        for(int j=0; j<i; j++)
            if(ara[j]<ara[i])
                dp[i] = max(dp[i], dp[j]+1);
    }
    return *max_element(dp, dp+n);
}
```

```
void printLis(){
   auto it = max_element(dp, dp+n) - dp;
   vi ans; ans.pb(it); it--;
   while(it>=0){
      if(dp[it]+1 == dp[ans.back()])
         ans.pb(it);
      it--;
   }
   reverse(all(ans));
   for(int i=0; i<ans.size(); i++)
      cout << ara[ans[i]] << " ";
   cout nl;}
```

**2. Complexity : O(n * log n), Space : O(n)**

```
int tail[mx], len;
int findPos(int x){
   int lw=0, hh=len-1, pos=0;
   while(lw<=hh){
      int mid = (lw+hh)/2;
      if(tail[mid] >= x)
         pos = mid, hh=mid-1;
      else lw=mid+1;
   }
   return pos;
}

int lis2(){
   len = 1; tail[0] = ara[0];
   for(int i=1; i<n; i++){
      if(ara[i] < tail[0]) tail[0] = ara[i];
      else if(ara[i] > tail[len-1]) tail[len++] = ara[i];
      else tail[findPos(ara[i])] = ara[i];
   }
   return len;
}
```

**Print in O(n * log n)**

```
int pre[mx];
int getPos(int x){
    int lw=-1, hh=len-1, ans=0;
    while(hh-lw>1){
        int m = lw + (hh-lw)/2;
        if(ara[tail[m]] >= x)
            hh = m;
        else lw = m;
    }
    return hh;
}

int lis2Construct(){
    mem(tail, 0); mem(pre, -1); len = 1;
    for(int i=1; i<n; i++){
        if(ara[i] < ara[tail[0]]) tail[0] = i;
        else if(ara[i] > ara[tail[len-1]]){
            pre[i] = tail[len-1];
            tail[len++] = i;
        }
        else{
            int p = getPos(ara[i]);
            pre[i] = tail[p-1];
            tail[p] = i;
        }
    }
    return len;
}

void lis2Print(){
    /// in reverse order
    for(int i=tail[len-1]; i>=0; i=pre[i])
        cout << ara[i] << " ";
    cout << endl;
}
```

## Longest Common Subsequence

```
string a = "helloworld";
string b = "missworld";
int n = a.size(), m = b.size();

/// Recursive Method
/// Complexity : O(nm)
/// Space : O(mn)

int dp[mx][mx];
int lcsRec(int i, int j){
    if(i==n || j==m) return 0;

    if(dp[i][j] != -1)
        return dp[i][j];

    int ans = 0;
    if(a[i] == b[j])
        ans = 1+lcsRec(i+1, j+1);
    else ans = max(lcsRec(i+1, j), lcsRec(i, j+1));
    return dp[i][j] = ans;
}

/// Recursive Method
/// Complexity : O(nm)
/// Space : O(mn)

int lcsIte(){
    for(int i=0; i<=n; i++)
    for(int j=0; j<=m; j++)
        if(min(i,j)==0) dp[i][j] = 0;
        else if(a[i-1] == b[j-1])
            dp[i][j] = 1+dp[i-1][j-1];
        else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    return dp[n][m];
}
```

**/// LCS Print Function**
**/// Required lcsIte function**
**/// Complexity : Linear**

```
string ans = "";
void lcsPrint(){
    int i = n, j = m;
    while(min(i,j)>0){
        if(a[i-1] == b[j-1]){
            ans += a[i-1];
            i--; j--;
        }
        else if(dp[i-1][j] > dp[i][j-1]) i--;
        else j--;
    }
    reverse(all(ans));
    cout << ans nl;
}
```

**/// Memory Optimized LCS**
**/// Space : O(m)**

```
int lcs[2][mx], k=0;
int lcsIteOp(){
    for(int i=0; i<=n; i++, k=!k)
    for(int j=0; j<=m; j++)
        if(min(i,j)==0) dp[k][j] = 0;
        else if(a[i-1] == b[j-1])
            dp[k][j] = 1+dp[!k][j-1];
        else dp[k][j] = max(dp[!k][j], dp[k][j-1]);
    return dp[!k][m];
}
```

## Longest Common Increasing Subsequence
## Complexity : O(nm)

```
int n = 4, m = 7;
int a[] = {3, 4, 9, 1};
int b[] = {5, 3, 8, 9, 10, 2, 1};

int ans[mx];
int LCIS(){
   for(int i=0; i<n; i++){
      int cur = 0;
      for(int j=0; j<m; j++){
         if(a[i] == b[j])
            ans[j] = max(ans[j], cur+1);
         else if(a[i] > b[j])
            cur = max(cur, ans[j]);
      }
   }
   return *max_element(ans, ans+m);
}
```

### Print Function

```
void printLCIS(){
   int p = max_element(ans, ans+m)-ans;

   vi dis; dis.pb(p);
   for(int i=p-1; i>=0; i--)
      if(ans[i] && ans[i]+1 == ans[dis.back()])
         dis.pb(i);

   reverse(all(dis));
   for(auto u:dis)
      cout << b[u] << " ";
   cout nl;
}
```

## Longest Common Subsequence with at most k changes allowed

Given two sequence P and Q of numbers. The task is to find Longest Common Subsequence of two sequence if we are allowed to change at most k elements in first sequence to any value.

**Complexity** : O(nmk)

```
int n = 5, m = 5;
int a[] = { 1, 2, 3, 4, 5 }, b[] = { 5, 3, 1, 4, 2 };

int dp[mx][mx][mx];
int lcsAtMostK(int i, int j, int k){
   if(i==n || j==m) return 0;

   if(dp[i][j][k] != -1)
      return dp[i][j][k];

   int ans = 0;
   if(a[i] == b[j]) ans = 1+lcsAtMostK(i+1, j+1, k);
   else {
      ans = max(lcsAtMostK(i+1, j, k), lcsAtMostK(i, j+1, k));
      if(k) ans = max(ans, 1+lcsAtMostK(i+1, j+1, k-1));
   }
   return dp[i][j][k] = ans;
}
```

## Consider a variation of edit distance

where we are allowed only two operations insert and delete, find edit distance in this variation.

**Complexity : O(nm)**

```
string a = "cat", b="cut";
int n=a.size(), m=b.size();
```

### Recursive Method

```
int dp[mx][mx];
int editDistLCS(int i, int j){
    if(i==n) return (m-j);
    if(j==m) return (n-i);

    if(dp[i][j] != -1)
        return dp[i][j];

    int ans = INF;
    if(a[i] == b[j])
        ans = editDistLCS(i+1, j+1);
    else{
        ans = min(ans, 1+editDistLCS(i+1, j));
        ans = min(ans, 1+editDistLCS(i, j+1));
        ans = min(ans, 2+editDistLCS(i+1, j+1));
    }
    return dp[i][j] = ans;
}
```

### Iterative Method

```
int editDistLCSIte(){
    for(int i=0; i<=n; i++)
    for(int j=0; j<=m; j++)
        if(min(i,j) == 0) dp[i][j] = 0;
        else if(a[i-1]==b[j-1]) dp[i][j] = 1+dp[i-1][j-1];
        else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    return n+m-2*dp[n][m];
}
```

## Longest Common Substring

```
string a = "OldSite:GeeksforGeeks.org";
string b = "NewSite:GeeksQuiz.com";
int n = a.size(), m = b.size();
```

**/// Iterative Method**

```
int dp[mx][mx], r, c, len;
int lcSubstrIte(){
    int ans = 0;
    for(int i=0; i<=n; i++)
    for(int j=0; j<=m; j++)
        if(min(i,j) == 0) dp[i][j] = 0;
        else if(a[i-1]==b[j-1]){
            dp[i][j] = 1+dp[i-1][j-1];
            if(ans < dp[i][j]){
                ans = dp[i][j];
                r = i; c = j;
            }
        }
        else dp[i][j] = 0;
    return len = ans;
}
```

**/// Recursive Method**

```
int dp2[mx][mx][mx];
int lcSubstrRec(int i, int j, int c){
    if(i == n) return c;
    if(j == m) return c;

    if(dp2[i][j][c] != -1)
        return dp2[i][j][c];

    int ans = c;
    if(a[i] == b[j])
        ans = max(ans, lcSubstrRec(i+1, j+1, c+1));
    ans = max3(ans, lcSubstrRec(i+1, j, 0), lcSubstrRec(i, j+1, 0));
    return dp2[i][j][c] = ans;
}
```

**/// Print Method**

**/// Required lcsSubstrIte Method**

```
void printLCSub(){
    if(len == 0) cout << "No Common Sub\n";
    else{
        string ans = "";
        while(dp[r][c]){
            ans += a[r-1];
            r--; c--;
        }
        reverse(all(ans));
        cout << ans nl;
    }
}
```

**Memory Optimized**

```
int dp3[2][mx];
int lcSubstrIteOp(){
    int ans = 0, k=0;
    for(int i=0; i<=n; i++, k=!k)
    for(int j=0; j<=m; j++)
        if(min(i,j) == 0) dp[k][j] = 0;
        else if(a[i-1]==b[j-1]){
            dp[k][j] = 1+dp[!k][j-1];
            if(ans < dp[k][j]){
                ans = dp[k][j];
                r = i; c = j;
            }
        }
        else dp[k][j] = 0;
    return len = ans;
}
```

## Maximum Subarray and Sub-sequence

Complexity : O(n), Space : O(1)

```
int a[] = {-1, 7, 8, -6, 4, -21, 12, 3};
int n = sizeof(a)/4;

/// Kadane's Algorithm, maximum sub-array
int maxSubArraySum(){
    int maxAns = INT_MIN, maxCur = 0;
    for(int i=0; i<n; i++){
        maxAns = max(maxAns, maxCur+a[i]);
        maxCur = max(0, maxCur+a[i]);
    }
    return maxAns;
}

/// normal implementation, maximum subsequence array
int maxSubSeSum(){
    int ans = 0, cnt = 0, maxV = INT_MIN;
    for(int i=0; i<n; i++){
        maxV = max(maxV, a[i]);
        ans = max(ans, ans+a[i]);
    }
    if(maxV<0) ans = maxV;
    return ans;
}
```

## Total palindromic subsequence

Finding the total number of palindromic substring can be found in a string

```
string s = "abcba";
int n = s.size();
```

### Recursive Method: Complexity : O(n*n), Space : O(n*n)

```
int dp[mx][mx];
int countPS(int i, int j){
    if(i>j) return 0;
    if(dp[i][j] != -1)
        return dp[i][j];

    if(i==j) return 1;
    if(i+1==j)
        return s[i]==s[j]?3:2;

    int ans = 0;
    if(s[i]==s[j])
        ans = countPS(i+1, j)+countPS(i, j-1)+1;
    else ans = countPS(i+1, j)+countPS(i, j-1)-countPS(i+1,j-1);
    return dp[i][j] = ans;
}
```

### Iterative Method: Complexity : O(n*n), Space : O(n*n)

```
int countPSIte(){
    for(int i=0; i<n; i++) dp[i][i] = 1;
    for(int i=2; i<=n; i++)
    for(int j=0; j<n; j++){
        int k = i + j - 1;
        if(s[j] == s[k]) dp[j][k] = dp[j][k-1]+dp[j+1][k]+1;
        else dp[j][k] = dp[j][k-1]+dp[j+1][k]-dp[j+1][k-1];
    }
    return dp[0][n-1];
}
```

## longest biotonic subsequence

a subsequence of arr[] is called Bitonic if it is first increasing, then decreasing

```
int a[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15};
int n = sizeof(a)/sizeof(a[0]);
```

### Iterative Method : Complexity : O(n*n), Space : O(n)

```
int dp[2][mx];
int lbs(){
    for(int i=0; i<n; i++){
        dp[0][i] = 1;
        for(int j=0; j<i; j++)
            if(a[j] < a[i])
                dp[0][i] = max(dp[0][i], dp[0][j]+1);
    }

    for(int i=n-1; i>=0; i--){
        dp[1][i] = 1;
        for(int j=n-1; j>i; j--)
            if(a[i] > a[j])
                dp[1][i] = max(dp[1][i], dp[1][j]+1);
    }

    int ans = 0, p=0;
    for(int i=0; i<n; i++)
        if(ans < dp[0][i]+dp[1][i]-1)
            ans = dp[0][i]+dp[1][i]-1, p = i;

    return ans;
}
```

**Print Function of longest biotonic subsequence**
**Required lbs() function**

```
void lbsPrint(){
    int ans = 0, p=0;
    for(int i=0; i<n; i++)
        if(ans < dp[0][i]+dp[1][i]-1)
            ans = dp[0][i]+dp[1][i]-1, p = i;

    vi dis; dis.pb(p);
    for(int i=p-1; i>=0; i--)
        if(dp[0][i]+1 == dp[0][dis.back()])
            dis.pb(i);
    Sort(dis);

    for(int i=p+1; i<n; i++)
        if(dp[1][i]+1 == dp[1][dis.back()])
            dis.pb(i);

    for(int i=0; i<ans; i++)
        cout << a[dis[i]] << " ";
    cout nl;
}
```

**Binary Search Method : Complexity : O(n log n)**

```
int inc[2][mx], tail[2][mx], in;
int getPos(int x, int p){
    int lw=0, hh=in-1, ans=-1;
    while(lw<=hh){
        int m = (lw+hh)/2;
        if(inc[p][m]>=x)
            ans=m, hh=m-1;
        else lw=m+1;
    }
    return ans;
}
```

```
int lis(int p){
    inc[p][0] = a[0]; in = 1; tail[p][0] = 0;
    for(int i=1; i<n; i++)
        if(a[i] < inc[p][i])
            tail[p][0] = 0, inc[p][0] = a[i];
        else if(inc[p][in-1] < a[i])
            tail[p][i] = in, inc[p][in++] = a[i];
        else{
            int pp = getPos(a[i], p);
            tail[p][i] = pp, inc[p][pp] = a[i];
        }
    return in;
}

int lbs2(){
    lis(0); reverse(a, a+n); lis(1);

    reverse(a, a+n);
    reverse(tail[1], tail[1]+n);

    int ans = 0;
    for(int i=0; i<n; i++)
        ans = max(ans, tail[0][i]+tail[1][i]+1);
    return ans;
}
```

**Print Function For Binary Search Method:**
**Required lbs2() function**

```
void lbs2Print(){
    int ans = 0, p=0;
    for(int i=0; i<n; i++)
        if(ans < tail[0][1]+tail[1][i]+1)
            ans = tail[0][1]+tail[1][i]+1, p = i;
    vi dis; dis.pb(p);
    for(int i=p-1; i>=0; i--)
        if(tail[0][i]+1 == tail[0][dis.back()])
            dis.pb(i);
    Sort(dis);

    for(int i=p+1; i<n; i++)
        if(tail[1][i]+1 == tail[1][dis.back()])
            dis.pb(i);

    for(auto u:dis)
        cout << a[u] << " ";
    cout nl;
}
```

## Maximum Product Increasing Subsequence

Given an array of numbers, find the maximum product formed by multiplying numbers of an increasing subsequence of that array

```
ll a[] = {3, 100, 4, 5, 150, 6};
ll n = sizeof(a)/sizeof(ll);
```

**Iterative Method : complexity : O(n*n), space : O(n)**

```
ll dp[mx];
ll maxProIncSubIte(){
    for(int i=0; i<n; i++) dp[i] = a[i];
    for(int i=1; i<n; i++)
    for(int j=0; j<i; j++)
        if(a[i] > a[j])
            dp[i] = max(dp[i], dp[j]*a[i]);
    return *max_element(dp, dp+n);
}
```

## Maximum Sum Increasing Subsequence

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then the output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then the output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then the output should be 10.

```
int a[] = {1, 101, 2, 3, 100, 4, 5};
int n = sizeof(a)/4;
```

### Iterative Method : Complexity : O(n*n), Space : O(n)

```
int dp[mx];
int maxSumIncSubIte(){
   for(int i=0; i<n; i++) dp[i] = a[i];
   for(int i=1; i<n; i++)
   for(int j=0; j<i; j++)
      if(a[i] > a[j])
         dp[i] = max(dp[i], dp[j]+a[i]);
   return *max_element(dp, dp+n);
}
```

**For printing values**

```
vector< ll > vec[mx];
ll findSum(int p){
   ll sum = 0;
   for(auto x:vec[p])
      sum += x;
   return sum;
}
```

```
ll sum[mx];
void printMaxSumIncSub(){
    vec[0].pb(a[0]); sum[0] = findSum(0);
    for(int i=1; i<n; i++){
        for(int j=0; j<i; j++)
            if(a[i]>a[j] && sum[i] < sum[j])
                vec[i] = vec[j], sum[i] = sum[j];
        vec[i].pb(a[i]);
        sum[i] += a[i];
    }

    int p = 0;
    for(int i=1; i<n; i++)
        if(sum[p] < sum[i])
            p = i;

    for(auto x:vec[p])
        cout << x << " ";
    cout nl;
}
```

## Subset Sum Problem

Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

```
int a[] = {1, 2, 3, 4, 5};
int n = sizeof(a) / sizeof(a[0]);
```

### Recursive Method: Complexity : O(nS), Space : O(nS)

```
int dp[mx][mx];
int subSetSum(int idx, int s){
    if(s <= 0) return s == 0;
    if(idx == n) return 0;
    if(dp[idx][s] != -1)
        return dp[idx][s];
    return dp[idx][s] = subSetSum(idx+1, s) || subSetSum(idx+1, s-a[idx]);
}
```

**Iterative Method : Complexity : O(nS), Space : O(nS)**

```
int subSetSumIte(int s){
    mem(dp, 0);
    for(int i=0; i<=n; i++) dp[i][0] = 1;
    for(int i=1; i<=s; i++) dp[0][i] = 0;
    for(int i=1; i<=n; i++)
    for(int j=1; j<=s; j++)
        if(j<a[i-1]) dp[i][j] = dp[i-1][j];
        else dp[i][j] = dp[i-1][j] || dp[i-1][j-a[i-1]];
    return dp[n][s];
}
```

**Printing all subsets with given sum**

```
int subSetSumPrint(int s){
    for(int i=0; i<n; i++) dp[i][0] = 1;
    if(a[0] <= s) dp[0][a[0]] = 1;
    for(int i=1; i<n; i++)
    for(int j=0; j<=s; j++)
        if(a[i] <= j) dp[i][j] = dp[i-1][j] || dp[i-1][j-a[i]];
        else dp[i][j] = dp[i-1][j];
    return dp[n-1][s];
}

void PrintUtill(int i, int s, vi &p){
    if(!i && s && dp[0][s]){
        p.pb(a[i]); cout << p nl;
        return;
    }
    if(!i && !s){cout << p nl; return;}
    if(dp[i-1][s]){vi b = p; PrintUtill(i-1, s, b);}
    if(s>=a[i] && dp[i-1][s-a[i]]){
        p.pb(a[i]); PrintUtill(i-1, s-a[i], p);}
}

void subSetPrint(int s){
    /// must call subSetSumPrint(s);
    subSetSumPrint(s); vi p;
    cout << "All subset sum for --> " << s nl;
    PrintUtill(n-1, s, p);
}
```

**/// Memory Optimized Subset Sum**
**/// Complexity : O(nS)**
**/// Space : O(S)**

```
int dpOp[2][mx], k=0;
int subSetSumOpt(int s){
   for(int i=0; i<=n; i++, k=!k)
   for(int j=0; j<=s; j++)
      if(j == 0) dpOp[k][j] = 1;
      else if(i==0) dpOp[k][j] = 0;
      else if(j < a[i-1]) dpOp[k][j] = dpOp[!k][j];
      else dpOp[k][j] = dpOp[!k][j] || dpOp[!k][j-a[i-1]];
   return dpOp[!k][s];
}
```

## Longest Palindromic Subsequence

Finding a longest subsequence string that is palindromic

```
string a, b, s = "GEEKSFORGEEKS";
int n = s.size();
```

**Recursive Method : Complexity : O(n * n), Space : O(n * n)**

```
int dp[mx][mx];
int lpsRec(int i, int j){
   if(i == j) return 1;
   if(i+1 == j)
      return (s[i]==s[j]) ? 2 : 1;

   if(dp[i][j] != -1)
      return dp[i][j];

   int ans = 0;
   if(s[i] == s[j]) ans = 2+lpsRec(i+1, j-1);
   else ans = max(lpsRec(i+1, j), lpsRec(i, j-1));
   return dp[i][j] = ans;
}
```

**Iterative Method: Complexity : O(n * n), Space : O(n * n)**

```
int lpsIte(){
   for(int i=0; i<n; i++)
      dp[i][i] = 1;
   for(int k=2; k<=n; k++)
   for(int i=0; i<n-k+1; i++){
      int j = i + k - 1;
      if(s[i] == s[j] && k==2)
         dp[i][j] = 2;
      else if(s[i] == s[j])
         dp[i][j] = 2+dp[i+1][j-1];
      else dp[i][j] = max(dp[i+1][j], dp[i][j-1]);
   }
   return dp[0][n-1];
}
```

**For Printing, we can use two strings of s where a = from starting, b = in reverse, and finding a common subsequence method for finding length and values**

```
int lcsIte(){
   for(int i=0; i<=n; i++)
   for(int j=0; j<=n; j++)
      if(min(i,j)==0) dp[i][j] = 0;
      else if(a[i-1] == b[j-1])
         dp[i][j] = 1+dp[i-1][j-1];
      else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
   return dp[n][n];
}

string lpsPrint(){ string ans = "";
   a = s; b = s; reverse(all(b)); lcsIte();
   int i = n, j = n;
   while(min(i,j)>0){
      if(a[i-1] == b[j-1]){
         ans += a[i-1];
         i--; j--;
      }
      else if(dp[i-1][j] > dp[i][j-1]) i--;
      else j--;
   }
   reverse(all(ans));
   return ans;}
```

**Memory Optimized Method**

```
int dpOp[mx];
int lpsOp(){
   for(int i=n-1; i>=0; i--){
      dpOp[i] = 1;
      int tmp = 0, tmp2;
      for(int j=i+1; j<n; j++)
         if(s[i] == s[j]){
            tmp2 = dpOp[j];
            dpOp[j] = tmp + 2;
            tmp = tmp2;
         }
         else{
            tmp = dpOp[j];
            dpOp[j] = max(dpOp[j-1], dpOp[j]);
         }
   }
   return dpOp[n-1];
}
```

## Minimum and Maximum values of an expression with * and +

Given an expression which contains numbers and two operators '+' and '*', we need to find maximum and minimum value which can be obtained by evaluating this expression by different parenthesization

**Complexity : O(n*n*n), Space : O(n*n)**

```
string s = "0*2+2*0+2", tmp = "";
vector<int> num;
vector<int> opr;

int len;
ll minVal[mx][mx];
ll maxVal[mx][mx];

void printMinMaxValueExp(){
   auto isOpt = [](char a){
      return a=='+' || a=='*';};
```

```
    for(int i=0; s[i]; i++)
        if(isOpt(s[i])){
            opr.pb(s[i]);
            num.pb(stringToT(tmp, 1));
            tmp = "";
        }
        else tmp += s[i];
    num.pb(stringToT(tmp, 1)); tmp = ""; // Here, StringToT is String to Integer

    len = num.size();
    for(int i=0; i<len; i++)
    for(int j=0; j<len; j++){
        minVal[i][j] = INT_MAX;
        maxVal[i][j] = 0;
        if(i == j)
            minVal[i][j] = maxVal[i][j] = num[i];
    }

    for(int l=2; l<=len; l++)
    for(int i=0; i<len-l+1; i++){
        int j = i + l - 1;
        for(int k=i; k<j; k++){
            ll mintmp = 0, maxtmp = 0;
            if(opr[k] == '+'){
                mintmp = minVal[i][k] + minVal[k+1][j];
                maxtmp = maxVal[i][k] + maxVal[k+1][j];
            }
            else if(opr[k] == '*'){
                mintmp = minVal[i][k] * minVal[k+1][j];
                maxtmp = maxVal[i][k] * maxVal[k+1][j];
            }

            minVal[i][j] = min(minVal[i][j], mintmp);
            maxVal[i][j] = max(maxVal[i][j], maxtmp);
        }
    }

    cout << minVal[0][len-1] gf maxVal[0][len-1] nl;
}
```

## Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications

```
int n = 5;
int p[mx] = {40, 20, 30, 10, 30};
ll dp[mx][mx];
```

### Recursive Method : Complexity : O(n * n), Space : O(n * n)

```
ll MatrixChain(int i, int j){
   if(i == j) return 0;
   if(dp[i][j] != -1)
      return dp[i][j];
   ll ans = LLONG_MAX, tmp;
   for(int k=i; k<j; k++){
      tmp = (ll)p[i-1]*p[k]*p[j] + MatrixChain(i, k) + MatrixChain(k+1, j);
      if(tmp < ans){
         ans = tmp;
         dp[j][i] = k;
      }
   }

   return dp[i][j] = ans;
}
```

**For Printing the multiplication way of the matrix**

```
char ch = 'A';
void printUtil(int i, int j){
   if(i == j){cout << ch++; return;}

   cout << "(";
   printUtil(i, dp[j][i]);
   printUtil(dp[j][i]+1, j);
   cout << ")";
}

void printMatrixChain(){
   /// must call MatrixChain
   printUtil(1, n-1); cout nl;
}
```

**Iterative Method : Complexity : O(n * n), Space : O(n * n)**

```
ll MatrixChain(){
   for(int i=1; i<n; i++) dp[i][i] = 0;
   for(int l=2; l<n; l++)
   for(int i=1; i<n-l+1; i++){
      int j = i + l - 1; dp[i][j] = LLONG_MAX;
      for(int k=i; k<j; k++)
         dp[i][j] = min(dp[i][j], (ll)p[i-1]*p[k]*p[j] +
                              dp[i][k] + dp[k+1][j]);
   }
   return dp[1][n-1];
}
```

**Memory Optimized Method**

```
ll MatrixChainOpt(){
   for(int i=1; i<n; i++) dp[i][i] = 0;
   for(int l=1; l<n-1; l++)
   for(int i=1; i<n-l; i++)
      dp[i][i+l] = min(p[i-1]*p[i]*p[i+l] + dp[i+1][i+l],
                  p[i-1]*p[i+l-1]*p[i+l] + dp[i][i+l-1]);
   return dp[1][n-1];
}
```

## Longest Substring Without repeated characters, Complexity : O(n)

```
int ara[26];
string s = "geeksforgeeks";
string longestSubstring(){
   mem(ara, -1); int ans = 0, st = 0, start = 0;
   for(int i=0; s[i]; i++){
      int p = s[i] - 'a';
      if(ara[p] == -1) ara[p] = i;
      else {
         if(ara[p] >= st){
            if(ans < i-st) ans = i-st, start = st;
            st = ara[p] + 1;
         }
         ara[p] = i;
      }
   }
   return s.substr(start, ans);}
```

## Longest Palindromic substring (Manacher's Algorithm)

**/// Complexity : O(n), Space : O(n)**

```
int p[mx<<1];
string Converter(string s){
    string ans = "@";
    for(int i=0; s[i]; i++)
        ans += "#" + s.substr(i, 1);
    ans += "#$"; return ans;
}

string manacharPalindrom(string s){
    string ss = Converter(s); int c=0, r=0;
    for(int i=1; ss[i]; i++){
        int j = c - (i - c);
        if(r > i) p[i] = min(r-i, p[j]);
        while(ss[i+1+p[i]] == ss[i-1-p[i]])
            p[i]++;
        if(i + p[i] > r) c = i, r = i + p[i];
    }

    int mxp = 0, ci = 0;
    for(int i=1; ss[i]; i++)
        if(mxp < p[i])
            mxp = p[i], ci = i;
    return s.substr((ci-1-mxp)/2, mxp);
}
```

## Lexicographically Largest Palindromic Subsequence

```
string s = "asvaweoinczzoihefnosjidfzoeitjh";
int n = s.size();

string fun(){
    string ans = ""; char ch = 'a';
    for(int i=0; s[i]; i++)
        ch = max(ch, s[i]);
    for(int i=0; s[i]; i++)
        if(ch == s[i])
            ans += ch;
    return ans;
}
```