Fibonacci: 斐波那契

Recur: O(2^n); O(n) (consider recursion stack space, if not: O(1))

```
int run(int n){
	if(n<=1) return n;
	return run(n-1)+run(n-2);
}
```

Iteration: O(n); O(1)

```
int run(int n){
	int a, b, c;
	if(!n) return 0;
	a=0, b=1;
	for(int i=2;i<n+1;++i){
		c=a+b;
		a=b;
		b=c;
	}
	return b;
}
```

Matrix: O(logn); O(logn) (consider recursion stack, if not: O(1))

if n%2: f(n)=f(k)*(2*f(k-1)+f(k)), k=n/2

if !(n%2): f(n)=f(k)*f(k)+f(k-1)*f(k-1), k=(n+1)/2

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

```
void multi(int a[2][2], int b[2][2]){
	a[0][0]=a[0][0]*b[0][0]+a[0][1]*b[1][0];
	a[0][1]=a[0][0]*b[0][1]+a[0][1]*b[1][1];
	a[1][0]=a[1][0]*b[0][0]+a[1][1]*b[1][0];
	a[1][1]=a[1][0]*b[0][1]+a[1][1]*b[1][1];
}
void recur(int m[2][2], int n){
	if(!n||n==1) return;
	recur(m, n/2);
	multi(m, m);
	int t[2][2]={{1, 1}, {1, 0}};
```

```
        if(n%2) multi(m, t);
}
int run(int n){
        int m[2][2]={{1, 1}, {1, 0}};
        if(!n) return 0;
        recur(m, n-1);
        return m[0][0];
}
```

Fomula: O(1), O(1)

```
int run(int n){
        return round(pow((1+sqrt(5))/2, n)/sqrt(5));
}
```

Pow(x, n): 快速幂 O(logn)

```
double myPow(double x, int n) {
        double rst=1;
        long p=n;
        if(n<0) p=-p, x=1/x;
        for(;p;x*=x, p>>=1) if(p&1) rst*=x;
        return rst;
    }
```

Task Scheduler: rst=max(s.length(), (maxn-1)*(n+1)+maxc), O(n)

maxn: length of a max task

maxc: number of max tasks

```
int leastInterval(string &s, int n) {
        int mp[26]={0}, mxn=-1, mxc=0;
        for(int i=0;i<s.length();++mp[s[i++]-'A']);
        for(int i=0;i<26;++i)
            if(mxn<mp[i]){
                mxc=1;
                mxn=mp[i];
            }else if(mxn==mp[i]) ++mxc;
        return MAX(s.length(), (mxn-1)*(n+1)+mxc);
    }
```

Maximum size subarray sum equals k: hash sum, O(n)

```
#define MAX(a, b) (a>b?a:b)
    int maxSubArrayLen(vector<int> &v, int k) {
        // Write your code here
        unordered_map<int, int> mp;
        int a=0, rst=0;
        for(int i=0;i<v.size();++i){
            a+=v[i];
            if(a==k) rst=i+1;
            if(!mp.count(a)) mp[a]=i;
            if(mp.count(a-k)) rst=MAX(rst, i-mp[a-k]);
        }
        return rst;
    }
```

One edit distance: string manipulation, char by char, increment ptr of the longer one, O(min(s.length, t.length))

```
bool isOneEditDistance(string &s, string &t) {
        int ss=s.length(), st=t.length(), cs=0, ct=0, c=0;
        if(abs(ss-st)>1) return 0;
        while(cs<ss&&ct<st)
            if(s[cs]==t[ct]) ++cs, ++ct;
            else{
                if(c==1) return 0;
                if(ss>st) ++cs;
                else if(ss<st) ++ct;
                else ++cs, ++ct;
                ++c;
            }
        if(ss-st>1||cs-ct>1) return 0;
        else if(ss>cs||st>ct) ++c;
        return c==1;
    }
```

Edit distance:

Return number of edit needed to transform a string to another.

2d-DP, dp[s1.length()+1][s2.length()+1]

dp[i][j] stores the # of edits to make w1[:i]=w2[:j].

1. **w1[i]=s2[j]: dp[i][j]=dp[i-1][j-1]**
2. **else: dp[i][j]=min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]))+1**

Time complexity: O(w1.length()*w2.length())

```
int minDistance(string word1, string word2) {
        int n1=word1.size(), n2=word2.size(), dp[n1+1][n2 +1];
        for(int i=0;i<= n1;++i) dp[i][0]=i;
        for(int i=0;i<= n2;++i) dp[0][i]=i;
        for(int i=1;i<= n1;++i)
            for(int j=1;j<=n2; ++j)
                if(word1[i-1]==word2[j-1]) dp[i][j] = dp[i-1][j-1];
                else dp[i][j]=min(dp[i-1][j-1], min(dp[i-1][j], dp[i][j-1]))+1;
        return dp[n1][n2];
    }
```

Space-optimization:

https://leetcode.com/problems/edit-distance/discuss/25846/C%2B%2B-O(n)-space-DP

Word Search: DFS, use '#' to indicate a visit, O(n*m*4^s.length) for matrix with size=n*m

Time: O(nm); Space: O(nm)

```
bool exist(vector<vector<char> > &board, string word) {
                m = board.size();
                n = board[0].size();
                for (int x = 0; x<m; x++)
                        for (int y = 0; y<n; y++)
                                if (recur(board, word.c_str(), x, y))
                                        return 1;
                return 0;
        }
        int m, n;
        bool recur(vector<vector<char> > &G, const char* w, int x, int y)
```

```
        {
                if (x<0 || y<0 || x >= m || y >= n || G[x][y] == '\0' || *w !=
G[x][y])
                        return 0;
                if (*(w + 1) == '\0')
                        return 1;
                char t = G[x][y];
                G[x][y] = '\0';
                if (recur(G, w + 1, x - 1, y) ||
           recur(G, w + 1, x + 1, y) ||
           recur(G, w + 1, x, y - 1) ||
           recur(G, w + 1, x, y + 1))
                        return 1;
                G[x][y] = t;
                return 0;
        }
```

2 Sum: Hash O(n)

```
vector<int> twoSum(vector<int> &v, int t) {
        unordered_map<int, int> mp;
        for(int i=0;i<v.size();++i)
           if(mp.count(t-v[i])) return {mp[t-v[i]], i};
           else mp[v[i]]=i;
        return {0, 0};
    }
```

3 Sum: sort, fix an element, 2ptr; O(n^2)

```
        vector<vector<int>> threeSum(vector<int>& v) {
        if(v.size()<3) return {};
        sort(v.begin(), v.end());
        vector<vector<int>> rst;
        int s=v.size();
        for(int i=0;i<s;++i){
           int l=i+1, r=s-1;
           while(l<r)
```

```
            if(v[l]+v[i]+v[r]>0) --r;
            else if(v[l]+v[i]+v[r]<0) ++l;
            else{
                vector<int> buf{v[i], v[l], v[r]};
                rst.push_back(buf);
                for(;l<r&&v[l]==buf[1];++l);
                for(;l<r&&v[r]==buf[2];--r);
            }
        for(;i<v.size()-2&&v[i]==v[i+1];++i);
    }
    return rst;
}
```

Permutation: For a set with 1 element, there is only permutation. For a set with 2, there are {a1 a2} and {a2 a1}, we can easily see that the result is generated by insert a1 in the beginning of a1 and the end of a1.
For a set with 3, there are 6 permutations, and we can also see that a3 is inserted in the between/in front of/after a1 and a2 (both are swapped to generate a new case).
Analysis:
https://www.geeksforgeeks.org/time-complexity-permutations-string/
Time: O(n^2*n!)

```
vector<vector<int> > rst;
void recur(vector<int> &v, int l, int r){
    if(l==r) rst.push_back(v);
    else
        for(int i=l;i<=r;++i){
            swap(v[i], v[l]);
            recur(v, l+1, r);
            swap(v[i], v[l]);
        }
}
public:
```

```
vector<vector<int>> permute(vector<int>& v) {
    recur(v, 0, v.size()-1);
    return rst;
}
```

Permutation II (Permutation with duplicates): Hashmap, permutation.

```
vector<vector<int>> permuteUnique(vector<int>& nums) {
    set<vector<int>> rst;
    permute(nums, 0, rst);
    return vector<vector<int>> (rst.begin(), rst.end());
}
void permute(vector<int>& nums, int a, set<vector<int>>& rst) {
    if(a>=nums.size()) rst.insert(nums);
    for(int i=a;i<nums.size();++i) {
        if(i!=a&&nums[i]==nums[a]) continue;
        swap(nums[i], nums[a]);
        permute(nums, a+ 1, rst);
        swap(nums[i], nums[a]);
    }
}
```

Next Permutation:

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

$1, 2, 3 \rightarrow 1, 3, 2$

$3, 2, 1 \rightarrow 1, 2, 3$

$1, 1, 5 \rightarrow 1, 5, 1$

这道题让我们求下一个排列顺序，有题目中给的例子可以看出来，如果给定数组是降序，则说明是全排列的最后一种情况，则下一个排列就是最初始情况，可以参见之前的博客 Permutations 全排列。我们再来看下面一个例子，有如下的一个数组

1 2 7 4 3 1

下一个排列为：

1 3 1 2 4 7

那么是如何得到的呢，我们通过观察原数组可以发现，如果从末尾往前看，数字逐渐变大，到了2时才减小的，然后我们再从后往前找第一个比2大的数字，是3，那么我们交换2和3，再把此时3后面的所有数字转置一下即可，步骤如下：

1 2 7 4 3 1

1 2 7 4 3 1

1 3 7 4 2 1

1 3 1 2 4 7

Time: O(n); Space: O(1)

```
void nextPermutation(vector<int>& nums){
        int n=nums.size(), i=n-2, j=n-1;
        for(;i>=0&&nums[i]>=nums[i+1];--i);
        if(i>=0){
            for(;nums[j]<nums[i];--j);
            swap(nums[i], nums[j]);
        }
        reverse(nums.begin()+i+1, nums.end());
    }
```

Subsets || (Combination with duplication): sort, ;O(n!) for time O(n^2) for space

```
vector<vector<int>> subsetsWithDup(vector<int> &v) {
        if(v.empty()) return {{}};
        int n=v.size(), bs=1, l=v[0];
        vector<vector<int> > rst(1);
        sort(v.begin(), v.end());
        for(int i=0;i<n;++i){
            if(l!=v[i]){
                l=v[i];
                bs=rst.size();
```

```
            }
            int ts=rst.size();
            for(int j=ts-bs;j<ts;++j){
                rst.push_back(rst[j]);
                rst.back().push_back(v[i]);
            }
        }
        return rst;
    }
```

Subsets (Combination): O(2^n) for time and space

```
vector<vector<int>> subsets(vector<int> &v) {
        vector<vector<int>> rst(1);
        int n=v.size();
        if(!n) return rst;
        sort(v.begin(), v.end());
        for(int i=0;i<n;++i){
            int rs=rst.size();
            for(int j=0;j<rs;++j){
                rst.push_back(rst[j]);
                rst.back().push_back(v[i]);
            }
        }
        return rst;
    }
//official solution: recursion
void run(vector<vector<int> > &rst, vector<int> &ts, vector<int> &v, int s){
        rst.push_back(ts);
        for (int i=s;i<v.size();i++) {
            ts.push_back(v[i]);
            run(rst, ts, v, i+1);
            ts.pop_back();
        }
    }
 public:
```

```
    vector<vector<int> > subsets(vector<int> &v) {
        vector<vector<int> > rst;
        vector<int> ts;
        sort(nums.begin(), nums.end());
        run(rst, ts, v, 0);
        return rst;
    }
```

Subset given size:

DFS

```
vector<vector<string>> rst;
void recur(vector<string> v, int sz, int ind,
                vector<string> &tmp, int i){
        if(ind==sz){
                vector<string> t(tmp);
                rst.append(t);
                return;
        }
        //end of v
        if(i>=v.size()) return;
        //include current element
        rst[ind]=v[i];
        tmp[ind]=v[i];
        recur(v, sz, ind+1, tmp, i+1);
        //skip current element, replace it with next element.
        recur(v, sz, ind, tmp, i+1);
}
vector<vector<string>> run(vector<string> v, int sz){
        vector<string> tmp;
        recur(v, sz, 0, tmp, 0);
        return rst;
}
```

NPC-Subset sum true false/determine subset sum/subset sum exist:

DP: we can use a 2d dp array, dp[i][j]: whether array[0:j-1] can form a subset with sum equal to i
Time: O(k*n), Space O(k*n)

```
bool run(vector<int> a, int k){
        int n=a.size();
        bool dp[a+1][k+1]={0};
        for(int i=0;i<n;dp[i++][0]=1);
        for(int i=1;i<=n;++i)
                for(int j=1;j<=k;++j){
                        if(j<a[i-1]) dp[i][j]=dp[i-1][j];
                        if(j>=a[i-1]) dp[i][j]=dp[i-1][j]||dp[i-1][j-a[i-1]];
                }
        return dp[n][k];
}
```

NPC-subset sum all/all subset sum/Find all subsets sum up to k:
DP. We need a 2d dp array, dp[i][j]: whether sum j is possible to get with array from 0 to i.
After done with the dp array, we traverse the path to build the result set. For the element traversed, we need to store path before reaching it and we also need to consider whether we should abandon it.
Time: O(2^n); Space: O(2^n+) (considering the recursion stack)

```
#include <bits/stdc++.h>
using namespace std;
//find all subsets sum to k
vector<vector<bool>> dp;
vector<vector<int>> rst;
void recur(vector<int> a, int i, int k, vector<int> &t) {
    //we reach the end, valid sum, a[0]==k (dp[0][k]==1)
    if (!i&&k&&dp[0][k]) {
        t.push_back(a[i]);
        rst.push_back(t);
        return;
    }
```

```
    //k is 0
    if (!i && !k) {
        rst.push_back(t);
        return;
    }
    //ignore the current element
    if (dp[i - 1][k]) {
        vector<int> tt = t;
        recur(a, i - 1, k, tt);
    }
    //consider current element
    if (k >= a[i] && dp[i - 1][k - a[i]]) {
        t.push_back(a[i]);
        recur(a, i - 1, k - a[i], t);
    }
}
vector<vector<int>> run(vector<int> a, int k) {
    int n = a.size();
    if (!n || k < 0) return {};
    dp.resize(n);
    for (int i = 0; i < n; ++i) {
        dp[i].resize(k + 1, 0);
        dp[i][0] = 1;
    }
    if (a[0] <= k) dp[0][a[0]] = 1;
    for (int i = 1; i < n; ++i)
        for (int j = 0; j < k + 1; ++j)
            dp[i][j] = a[i] <= j ? dp[i - 1][j] || dp[i - 1][j - a[i]] : dp[i - 1][j];
    if (!dp[n - 1][k]) return {};
    vector<int> t;
    recur(a, n - 1, k, t);
    return rst;
}
int main(){
    vector<int> a = { 1, 2, 3, 4, 5 };
```

```
    vector<vector<int>> ttt = run(a, 10);
    for (int i = 0; i < ttt.size(); ++i) {
        for (int j = 0; j < ttt[i].size(); ++j) cout << ttt[i][j] << " ";
        cout << endl;
    }
}
```

Regex matching || regular expression matching: dp,

1. **if(p[j-1]!='*'&&s[i-1]==p[j-1]||p[j-1]=='.') dp[i][j]=dp[i-1][j-1]**
2. **if(p[j-1]=='*'&&pattern repeats 0 times) dp[i][j]=dp[i][j-2]**
3. **if(p[j-1]=='*'&&pattern repeats >=1 times):**
   dp[i][j]=dp[i-1][j]&&(s[i-1]==[p-2]||p[j-2]=='.')

O(m*n)

```
bool isMatch(string &s, string &p) {
        int s1=s.size(), s2=p.size();
        bool dp[++s1][++s2]={0};
        dp[0][0]=1;
        for(int i=0;i<s1;++i) for(int j=1;j<s2;++j)
            if(p[j-1]-'*') dp[i][j]=i&&dp[i-1][j-1]&&(s[i-1]==p[j-1]||p[j-1]=='.');
            else dp[i][j]=dp[i][j-2]||(i&&dp[i-1][j]&&(s[i-1]==p[j-2]||p[j-2]=='.'));
        return dp[s1-1][s2-1];
    }
```

Wildcard matching: i for ptr(s), j for ptr(p), t1 for star-matching in s, t2 for star-position in p.

if(i<s.length()) start the loop:

1. **s[i]=p[j], or p[j]=='?': ++i, ++j**
2. **p[j]='*': t2=j, ++j, t1=i**
3. **s[i]!=p[j]: if there is a star, revert j back to t2+1, increment i&t1.**
4. **return false**

When the for loop ends, check whether j can get the end of p.

Time complexity O(n)

```
bool isMatch(string &s, string &p) {
        int s1=s.length(), s2=p.length(), t1=-1, t2=-1, i, j;
        for(i=0, j=0;i<s1;)
```

```
            if(s[i]==p[j]||p[j]=='?') ++i, ++j;
            else if(p[j]=='*') t1=i, t2=j++;
            else if(t1>=0) i=++t1, j=t2+1;
            else return 0;
        for(;p[j]=='*';++j);
        return j==s2;
    }
```

dp: bool dp[m+1][n+1] (+1 to consider (0,0)), dp[i][j]: s[0:i] match p[0:j]? dp[0][0]=1 (both empty). If s is empty and p only has“continuous *”, also return 1.

1. **p[j]=’*’: dp[i][j]=dp[i][j-1]||dp[i-1][j] (* matches null || * matches any char)**
2. **p[j]!=’*’: dp[i][j]=(s[i-1]==p[j-1]||p[j-1]=='?')&&dp[i-1][j-1] (same char at this position, or p[j]=? can match any single char; and previous chars are matched)**

Time complexity: O(n*m)

```
bool isMatch(string s, string p) {
        int m=s.size(), n=p.size();
        vector<vector<bool>> dp(m+1, vector<bool>(n+1, 0));
        dp[0][0]=1;
        for(int i=1;i<=n;++i) if(p[i-1] == '*') dp[0][i]=dp[0][i-1];
        for(int i=1;i<=m;++i)
            for(int j=1;j<=n;++j)
                if(p[j-1]=='*') dp[i][j]=dp[i-1][j]||dp[i][j-1];
                else dp[i][j]=(s[i-1]==p[j-1]||p[j-1]=='?')&&dp[i-1][j-1];
        return dp[m][n];
    }
```

Reverse integer: Trivial

```
int reverse(int x) {
        int a=0;
        for(int i=x;i;i/=10) a=a*10+i%10;
        return a;
    }
```

Reverse linked list:
http://www.cnblogs.com/grandyang/p/4478820.html

Reverse linked list II:
http://www.cnblogs.com/grandyang/p/4306611.html

Reverse nodes in k-group: get the length of the linked list, if it>=k, start to reverse (k=2: swap twice, k=3, swap 3 times ⇒ for loop goes from 1), update ptr "p" (pre) and cnt, end the swap when cnt<k; O(n)

```
class ListNode{ int val; ListNode* next; ListNode(int val){ this->val=val;
this->next=NULL; } }
ListNode * reverseKGroup(ListNode * head, int k) {
        ListNode *buf=new ListNode(-1), *p=buf, *c=p;
        int cnt=0;
        buf->next=head;
        for(;c=c->next;++cnt);
        for(;cnt>=k;p=c, cnt-=k){
            c=p->next;
            for(int i=1;i<k;++i){
                ListNode *t=c->next;
                c->next=t->next;
                t->next=p->next;
                p->next=t;
            }
        }
        return buf->next;
    }
```

Set Matrix zeros: scan all the elements, when there is a zero at (x, y), set M[x][0]=0, M[0][y]=0. In the end, scan M[0:][0] and M[0][0:] to start replacing.
Time complexity: O(m*n); Space: O(1)

```
void setZeroes(vector<vector<int> > &matrix) {
        if (matrix.empty()||matrix[0].empty()) return;
```

```
        int m=matrix.size(), n=matrix[0].size();
        bool rz=0, cz=0;
        for(int i=0;i<m;++i) if(!matrix[i][0]) cz=1;
        for(int i=0;i<n;++i) if (!matrix[0][i]) rz=true;
        for(int i=1;i<m;++i)
            for(int j = 1; j < n; ++j)
                if(!matrix[i][j]) matrix[0][j]=matrix[i][0]=0;
        for(int i=1;i<m;++i)
            for(int j=1;j<n;++j)
                if(!matrix[0][j]||!matrix[i][0]) matrix[i][j]=0;
        if(rz) for(int i=0;i<n;++i) matrix[0][i]=0;
        if (cz) for(int i=0;i<m;++i) matrix[i][0]=0;
    }
```

Best time to buy and sell stock: linear scan, use a tmp variable to store possible buying option. If there is revenue, do a transaction.
Time: O(n), Space: O(1)

```
#define INF 0x3f3f3f3f
int maxProfit(vector<int>& v) {
        if(!v.size()) return 0;
        int rst=-INF, t=INF, s=v.size();
        for(int i=0;i<s;++i){
            t=t>v[i]?v[i]:t;
            rst=rst<v[i]-t?v[i]-t:rst;
        }
        return rst;
    }
```

Best time to buy and sell stock II: linear scan, if there is revenue, do the transaction.
Time: O(n), Space: O(1)

```
int maxProfit(vector<int>& V) {
        int rst=0, n=V.size();
        for(int i=0;i<n-1;++i)
            if(V[i]<V[i+1]) rst+=V[i+1]-V[i];
```

```
        return rst;
    }
```

Best time to buy and sell stock III: DP, use 2 recurrence equation to maintain 2 variable, one records local, the other records global.
dl[i][j]: max revenue produced by "ith day with at most jth transactions, and the last selling happens at the last day".
dg[i][j]: max revenue produced by "ith day with at most jth transactions".
dl[i][j]=max(dg[i-1][j-1]+max(t, 0), dl[i-1][j]+t)
dg[i][j]=max(dl[i][j], dg[i-1][j])
*** ith day local maxima is selected between the revenue produced by "dg[i-1][j-1]+revenue (if there is)" (前一天少交易一次的全局最优加上该次交易(如果有收益)), and "dl[i-1][j]+t" (前一天的局部最优加上该次交易), t: difference**
Time: O(n), space: O(n)

```
int maxProfit(vector<int> &prices) {
        if (prices.empty()) return 0;
        int n=prices.size(), dg[n][3]={0}, dl[n][3]={0};
        for (int i=1;i<prices.size();++i){
            int t=prices[i]-prices[i-1];
            for (int j=1;j<=2;++j)
                dl[i][j]=max(dg[i-1][j-1]+max(t, 0), dl[i-1][j]+t), dg[i][j]=max(dl[i][j],
dg[i-1][j]);
        }
        return dg[n-1][2];
    }
```

We only need j from 2 to 1, we can use constant space to solve it.
Optimization on space: O(1)

```
#define INF 0x3f3f3f3f
#define MAX(a, b) (a>b?a:b)
int maxProfit(vector<int>& v) {
        if(!v.size()) return 0;
        int n=v.size(), dg[3]={0}, dl[3]={0};
        for(int i=0;i<n-1;++i){
```

```
            int t=v[i+1]-v[i];
            for(int j=2;j>=1;--j)
                dl[j]=MAX(dg[j-1]+MAX(t, 0), dl[j]+t), dg[j]=MAX(dl[j], dg[j]);
        }
        return dg[2];
    }
```

Can be simplified further: dl[i][j] is ith day with jth transaction and sell in last day, there are only 3 cases:

1. **bought today: dl[i][j]=dg[i-1][j-1], effectively do nothing**
2. **bought yesterday: dl[i][j]=dg[i-1][j-1]+t, dg[i-1][j-1] plus the transaction today.**
3. **bought earlier: dl[i][j]=dl[i-1][j]+t, plan to sell yesterday becomes plan to sell today.**

And case 1 is unneeded as it is useless, can be included in dg[i-1][j-1], and we dont need max(0, t) anymore.

dl[i][j]=max(dg[i-1][j-1], dl[i-1][j])+t; dg[i][j]=max(dl[i][j], dg[i-1][j])

dl[j]=MAX(dg[j-1], dl[j])+t; dg[j]=MAX(dl[j], dg[j]);

Best time to buy and sell stock IV: DP, more general then III.

We need 2 recurrence eq to update local and global best.

dl[i][j]=max(dg[i-1][j-1], dl[i-1][j])+t

dg[i][j]=max(dl[i][j], dg[i-1][j])

*** ith day local maxima is selected between the revenue produced by "dg[i-1][j-1]+revenue (if there is)" (前一天少交易一次的全局最优加上该次交易(如果有收益)), and "dl[i-1][j]+t" (前一天的局部最优加上该次交易), t: difference**

*** It can be optimized if k>n/2. we can use the same algorithm as II: "if there is revenue, do the transaction"**

```
int maxProfit(int k, vector<int> &prices){
        if(prices.empty()) return 0;
        if(k>=prices.size()) return greedy(prices);
        int g[k+1]={0}, l[k+1]={0};
        for(int i=0;i<prices.size()-1;++i){
            int t=prices[i+1]-prices[i];
            for(int j=k;j>=1;--j){
```

```
                l[j]=max(g[j-1], l[j])+t;
                g[j]=max(g[j], l[j]);
            }
        }
        return g[k];
    }
    int greedy(vector<int> &v){
        int rst=0;
        for(int i=1;i<v.size();++i) if(v[i]-v[i-1]>0) rst+=v[i]-v[i-1];
        return rst;
    }
int maxProfit(int k, vector<int> &v) {
        int n=v.size(), rst=0, l=0, g=0;
        priority_queue<int> pq;
        stack<pair<int, int> > stk;
        for(;g<n;stk.push(pair<int, int>(l, g))){
            for(l=g;l<n-1&&v[l]>=v[l+1];++l);
            for(g=l+1;g<n&&v[g]>=v[g-1];++g);
            for(;!stk.empty()&&v[l]<v[stk.top().first];stk.pop())
                pq.push(v[stk.top().second-1]-v[stk.top().first]);
            for(;!stk.empty()&&v[g-1]>=v[stk.top().second-1];stk.pop()){
                pq.push(v[stk.top().second-1]-v[l]);
                l=stk.top().first;
            }
        }
        for(;!stk.empty();stk.pop())
            pq.push(v[stk.top().second-1]-v[stk.top().first]);
        for(int i=0;i<k&&!pq.empty();++i){
            rst+=pq.top();
            pq.pop();
        }
        return rst;
    }
```

Best time to buy and sell stock with cooldown: DP

buy[i]: max revenue where last operation before ith day is buying
sell[i]: max revenue where last operation before ith day is selling
rest[i]: max revenue where last operation before ith day is resting
buy[i]=max(rest[i-1]-price, buy[i-1])
sell[i]=max(buy[i-1]+price, sell[i-1])
rest[i]=max(sell[i-1], buy[i-1], rest[i-1])
Since buy[i]<=rest[i] and existence of cooldown, we have
rest[i]=max(sell[i-1], rest[i-1])=sell[i-1]
we then have:
buy[i]=max(sell[i-2]-price, buy[i-1])
sell[i]=max(buy[i-1]+price, sell[i-1])
Time: O(n), Space: i only depends on i-1 and i-2, we can have O(1)

```
int maxProfit(vector<int>& prices) {
        int b=INT_MIN, pb=0, s=0, ps=0;
        for (int price:prices){
            pb=b;
            b=max(ps-price, pb);
            ps=s;
            s=max(pb+price, ps);
        }
        return sell;
    }
```

Best time to buy and sell stock with transaction fee: DP
sold[i]: max revenue where "sell stock at ith day"
hold[i]: max revenue where "hold stock at ith day"
At ith day:

1. **sold: total_rev=max(rev_hold_yesterday+(sell_revenue-fee), rev_sell_yesterday)**
2. **hold: total_rev=max(rev_sold_yesterday+buy_price, rev_hold_yesterday)**

sold[i]=max(sold[i-1], hold[i-1]+price[i]-fee)
hold[i]=max(hold[i-1], sold[i-1]-price[i])
i only depends on i-1, we can solve with constant space.

```
int maxProfit(vector<int>& prices, int fee) {
```

```
        int s=0, h=-prices[0];
        for (int price:prices) {
            int t=s;
            s=max(s, h+ price - fee);
            h=max(h, t - price);
        }
        return s;
    }
```

Valid parentheses: stack

Time: On; Space: On

```
bool isValid(string s) {
        stack<char> st;
        int sz=s.length();
        for(int i=0;i<sz;++i){
            if(s[i]=='('||s[i]=='{'||s[i]=='[') st.push(s[i]);
            else{
                if(st.empty()) return 0;
                char t=st.top();
                st.pop();
                if(abs(s[i]-t)>2) return 0;
                //ASCII of parentheses only differ within 2
            }
        }
        return st.empty();
    }
```

Valid parentheses string:

Traverse twice, 1st treats all * as '(' and count '(', 2nd treats all * as ')' and count ')', if both count>=0, return 1, else return 0.

Time: O(n); Space: O(1)

```
bool checkValidString(string s) {
        int n=s.length(), cnt=0;
        for(int i=0;i<n;++i){
            if(s[i]=='('||s[i]=='*') ++cnt;
```

```
            else --cnt;
            if(cnt<0) return 0;
        }
        //if(!cnt) return 1;
        cnt=0;
        for(int i=n-1;i>=0;--i){
            if(s[i]==')'||s[i]=='*') ++cnt;
            else --cnt;
            if(cnt<0) return 0;
        }
        return 1;
    }
```

Can be done with 2 counters.

t1: the number of '(' when "there is '(', the number of '*' being treated as ')'"

t2: the number of ')' when "there is '(', the number of '*' being treated as '('"

traverse S:

1. **s[i]='(': ++t1, ++t2**
2. **s[i]=')': --t2, if(t1>0): --t1 (avoid t1<0)**
3. **s[i]='*': ++t2, if(t1>0): --t1 (avoid t1<0). if(t2<0): return 0 (too many ')'s)**

return t1==0

Remove Invalid Parentheses: BFS

Push S into the queue and validate it, if legal then return, else traverse it. For the char encountered, remove it to generate a new string. If the string is not presented before (check by hash), push it into the queue, and if the string is legal, record it. If the queue is empty and valid string is not found, return null set.

Analysis: In BFS we handle the states level by level, in the worst case, we need to handle all the levels, we can analyze the time complexity level by level and add them up to get the final complexity.

On the first level, there's only one string which is the input string s, let's say the length of it is n, to check whether it's valid, we need O(n) time. On the second level, we remove one (or) from the first level, so there are C(n, n-1) new strings, each of them has n-1 characters, and for each string, we need to check whether it's valid or not, thus the total time complexity on this level is (n-1)*C(n, n-1). Come to the third level, total time complexity is (n-2)*C(n, n-2)... Therefore, T(n)=n*C(n, n)+(n-1)*C(n, n-1)+...+1*C(n, 1)=n*2^(n-1). Time: O(n*2^n); Space: O(n*2^n), the space needed for visited set/queue

```
vector<string> removeInvalidParentheses(string s) {
        vector<string> rst;
        unordered_set<string> visited{{s}};
        queue<string> q{{s}};
        bool found=false;
        while(!q.empty()){
            string t=q.front();
            q.pop();
            if (isValid(t)){
                rst.push_back(t);
                found=1;
            }
            if(found) continue;
            for(int i=0;i<t.size();++i) {
                if(t[i]!='('&&t[i] != ')') continue;
                string str=t.substr(0, i)+t.substr(i+1);
                if(!visited.count(str)){
                    q.push(str);
                    visited.insert(str);
                }
            }
        }
        return rst;
    }
    bool isValid(string t){
```

```
        int cnt=0;
        for(int i=0;i<t.size();++i)
            if(t[i]=='(') ++cnt;
            else if(t[i]== ')'&&--cnt<0) return 0;
        return !cnt;
    }
```

Remove Invalid Parentheses, return only 1 solution: Stack
Time: On; Space: On

```
string run(string s){
        stack<int> stk;
        for(int i=0;i<s.length();++i)
                if(s[i]==’(‘) stk.push(i);
                else{
                        if(stk.empty()){
                                s[i]=’#’
                                continue;
                        }
                        stk.pop();
                }
        }
        while(!stk.empty()){
                s[stk.top()]=’#’;
                stk.pop();
        }
        string rst=””;
        for(int i=0;i<s.length();++i)
                if(s[i]!=’#’) rst+=s[i];
        return rst;
}
```

Longest Valid Parentheses: stack; use a variable start to record the starting pos of the legal substr. Traverse the string:

1. **s[i]==‘(’: stk.push(i)**

2. **s[i]==')': if(stk.empty()){ start=i+1 }else{ stk.pop()**
   rst=stk.empty()?max(rst, i-start+1):max(rst, i-stk.top())}

Time: O(n), Space: O(n)

```
int longestValidParentheses(string s) {
        int rst=0, start=0;
        stack<int> m;
        for(int i=0;i<s.size();++i)
            if(s[i]=='(') m.push(i);
            else if(s[i]==')'){
                if(m.empty()) start=i+1;
                else{
                    m.pop();
                    rst=m.empty()?max(rst, i-start+1):max(rst, i-m.top());
                }
            }
        return rst;
    }
```

Generate Parentheses:

Recursion with 2 ptr, left and right to record remaining left and right parentheses.

1. **If left>right, the string is invalid, can be returned directly**
2. **else if left=right=0, satisfied, record the string**
3. **else if left>0 || right>0: call the recursion function.**

Time: O(2^n); Space: O(2^n)

```
vector<string> generateParenthesis(int n) {
        vector<string> rst;
        dfs(n, n, "", rst);
        return rst;
    }
void dfs(int l, int r, string s, vector<string> &rst) {
        if(l>r) return;
        if(!l&&!r) rst.push_back(s);
        else{
            if (l>0) dfs(l-1, r, s+'(', rst);
```

```
        if (r>0) dfs(l, r-1, s+')', rst);
    }
}
```

String manipulation: find '(', if there is one, add a ')' behind it and '()' in the beginning. May exist duplication, use set to avoid.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n}\frac{n+k}{k} \qquad \text{for } n \geq 0.$$

So time complexity should be O(C(2n,n)/(n+1)).

Time: O(C(2n, n)/(n+1))

```
vector<string> generateParenthesis(int n) {
    set<string> t;
    if (!n) t.insert("");
    else{
        vector<string> pre=generateParenthesis(n-1);
        for (auto a:pre) {
            for (int i=0;i<a.size();++i) {
                if (a[i]=='('){
                    a.insert(a.begin()+i+1, '(');
                    a.insert(a.begin()+i+2, ')');
                    t.insert(a);
                    a.erase(a.begin()+i+1, a.begin()+i+3);
                }
            }
            t.insert("()" + a);
        }
    }
    return vector<string>(t.begin(), t.end());
}
```

Different ways to add parentheses: Recursion, partition left/right subtree and construct with recursion.

Analysis: use nested loop to add the final result into the answer set. Note that when the base case does not have operator, we can add it into the set. Time complexity should be catalan number.

Time: O(2^n); Space: O(2^n)

```
vector<int> diffWaysToCompute(string input) {
        if(!input.length()) return {};
        vector<int> rst;
        for (int i=0;i<input.size();++i) {
            if(input[i]=='+'||input[i]=='-'||input[i]=='*'){
                vector<int> left=diffWaysToCompute(input.substr(0, i));
                vector<int> right=diffWaysToCompute(input.substr(i+1));
                for(int j=0;j<left.size();++j)
                    for(int k=0;k<right.size();++k)
                        if(input[i]=='+') rst.push_back(left[j]+right[k]);
                        else if(input[i]=='-') rst.push_back(left[j]-right[k]);
                        else rst.push_back(left[j]*right[k]);
            }
        }
        if(rst.empty()) rst.push_back(atoi(input.c_str()));
        return rst;
    }
```

Special Binary String: Recursion (the swapped substr must be valid as well). We can find that to reach the desired goal, we should move as many 1 as possible to the left of the string. Therefore, we can extract the substr in the middle of the string and sort it, and put it back. And recursion is suitable and easy to write here.
Use counter: '1': ++; '0': --; when counter=0, it is a valid substr, and we can put it into an array, and use i to indicate the position of the start point of the substr. As we need to sort the substr, and we need to use recursion upon S (not necessarily on the whole substr's as the start and end pos is certain, either 1 or 0). We only need recursion in the middle and then update i=j+1. After this, we sort the string vector produced previously and produce the largest swapped string by connecting all the substr's.
Time analysis:

We can draw a recursion tree to help analyze the complexity.
Let the branch factor `b` be unique to simplify analysis.
b then represents how many sub sbs S contains, so each sub sbs has length = `N/b`.

1. At level 1: We need to sort b sub sbs, each sub sbs has length = `N/b`, then we combine them.
   so it takes `N/b * b log b + N = N log b + N`
   That is term1 * term2 + term3, where:
   term1 = `N/b`, which is the time it takes for comparison of each pair of sub sbs's.
   term2 = `b log b`, which is the number of comparions to be done to sort b items.
   term3 = `N`, which is the time it takes to combine/build a length of N string.
2. At level 2: each sub_sbs branches to b subsub_sbs, each subsub_sbs has length = `N/b/b`.
   To sort all subsub_sbs's within a sub_sbs and combine them, it takes `N/b/b * b log b + N/b = N/b logb + N/b`
   But note, there are b sub_sbs's at this level, so in total it takes: `(N/b logb + N/b) * b = N log b + N`,
   which is the same as level 1.

That means each level takes the same time = `N log b + N`.
How many recursion levels/depths do we need?
The answer is `log(b)N`, pronounced as log-base-b of N. (If you don't see why, please read any textbook to look for mergesort or recursion analysis.)
So, roughly, the running time should be `O(log(b)N * (N log b + N))` which is quite fast.

(https://leetcode.com/problems/special-binary-string/discuss/113211/Easy-and-Concise-Solution-with-Explanation-C%2B%2BJavaPython)

```
string makeLargestSpecial(string S) {
        int cnt=0, i=0;
        vector<string> v;
        string rst="";
        for (int j=0;j<S.size();++j) {
            cnt+=S[j]=='1'?1:-1;
            if(!cnt){
                v.push_back('1'+makeLargestSpecial(S.substr(i+1, j-i-1))+'0');
                i=j+1;
            }
        }
        sort(v.begin(), v.end(), greater<string>());
        for(int i=0;i<v.size();++i) rst+=v[i];
        return rst;
    }
```

Minimum Window Substring: Hash&Sliding window. Hash to setup mapping relationship between char and #_of_occurrence. Use hashmap to record the occurrences of a char.

1. **Record count of char in t; for(char chr:t) ++mp[chr];**
2. **Traverse s, expand the window from the right side; for(char chr:s): --mp[chr];**
    a. **If mp[c]>0, it is a char in t, we need a counter to record matching status**
    b. **When the counter=t.length(), the window has all the char needed in t, now record the length and the substr.**
3. **Shrink the window from the left side; for(char chr:s): ++mp[chr];**
    a. **If mp[chr]>0, we lose a char in t, and need to decrease counter, and move left boundary.**
    b. **Note: for char not in t, decrease-mp stage will not increase counter, increase-mp stage will not decrease counter.**

Time: O(n); Space: O(n)

```
string minWindow(string s, string t) {
        string rst="";
        vector<int> mp(128, 0);
        int left=0, cnt=0, minLen=INT_MAX;
        for(char c:t) ++mp[c];
        for(int i=0;i<s.size();++i){
            if(--letterCnt[s[i]]>=0) ++cnt;
            while(cnt==t.size()){
                if(minLen>i-left+1){
                    minLen=i-left+1;
                    rst=s.substr(left, minLen);
                }
                if(++letterCnt[s[left]]>0) --cnt;
                ++left;
            }
        }
        return rst;
    }
```

Merge 2 sorted linked list:

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(!l1) return l2;
        if(!l2) return l1;
        ListNode *t1=l1, *t2=l2, *rst, *t;
        if(t1->val<t2->val){
            rst=t1;
            t1=t1->next;
        }else{
            rst=t2;
            t2=t2->next;
        }
        t=rst;
        while(t1&&t2){
            if(t1->val<t2->val){
                t->next=t1;
                t1=t1->next;
            }else{
                t->next=t2;
                t2=t2->next;
            }
            t=t->next;
        }
        t->next=t1?t1:t2;
        return rst;
    }
```

Merge K sorted lists/merge lists/merge k lists: DQ.
Eg, to merge 10 lists, we merge 0&5, 1&6, 2&7, 3&8, 4&9, then merge 2 with another shorter one.
Note that k=(n+1)/2 is to make sure when n is an odd number, k can still point to one of the lists in the second half.
Eg cont. from above, when n=5, we merge 0&3, 1&4, with 2 left; when n=3, we merge 0&2 with 1 left, when n=2, we merge 0&1, and the 10 lists are then merged completely.
Time: O(logn*m) m: # of nodes in a list.

Note: the problem can be solved easily with priority queue.

```
ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.empty()) return NULL;
        int n=lists.size();
        for(int k=((n+1)>>1);n>1;n=k, k=((n+1)>>1)
            for(int i=0;i<n/2;++i) lists[i]=mergeTwoLists(lists[i], lists[i+k]);
        return lists[0];
    }
//Merge 2 sorted lists/merge 2 lists
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(!l1) return l2;
        if(!l2) return l1;
        ListNode *t1=l1, *t2=l2, *rst, *t;
        if(t1->val<t2->val){
            rst=t1;
            t1=t1->next;
        }else{
            rst=t2;
            t2=t2->next;
        }
        t=rst;
        while(t1&&t2){
            if(t1->val<t2->val){
                t->next=t1;
                t1=t1->next;
            }else{
                t->next=t2;
                t2=t2->next;
            }
            t=t->next;
        }
        t->next=t1?t1:t2;
        return rst;
    }
```

Remove duplicates from sorted array/duplicate array: Trivial
Time: O(n); Space: O(1)

```
int removeDuplicates(vector<int>& v) {
        if(!v.size()) return 0;
        int a=v[0], rst=1, sz=v.size(), c=1;
        for(int i=1;i<sz;++i)
            if(a!=v[i]){
                a=v[i];
                v[c++]=a;
                ++rst;
            }
        return rst;
    }
```

Remove duplicates from sorted array II/duplicate array II: Trivial, use a buffer variable to record current number. A ptr is used as we dont know the distribution of the data.
Time: O(n); Space: O(1)

```
int removeDuplicates(vector<int>& v) {
        if(!v.size()) return 0;
        int a=v[0], rst=1, *ba=NULL, c=1, sz=v.size();
        for(int i=1;i<sz;++i)
            if(a==v[i]&&!ba){
                ba=&v[i];
                v[c++]=v[i];
                ++rst;
            }else if(a==v[i]){
                ba=NULL;
                for(;v[i]==v[i+1];++i);
            }
            else{
                a=v[i];
                ba=NULL;
                v[c++]=v[i];
                ++rst;
```

```
        }
    return rst;
}
```

Remove Duplicates from sorted List: **Trivial**

Time: On; Space O1

```
ListNode* deleteDuplicates(ListNode* head) {
    ListNode *c=head;
    while(c&&c->next)
        if(c->next->val==c>val) c->next=c->next->next;
        else c=c->next;
    return head;
}
```

Remove Duplicates from sorted List II: **Need a new node dummy for the case where the head node is a duplicated node. We define 2 ptrs, a pre, a cur. For everytime pre points to a new node, cur keeps going down until a different node is met, and let pre->next=cur; then move the pre to next node and repeat the process.**

Time: O(n); Space: O(1)

```
ListNode* deleteDuplicates(ListNode* head) {
    if(!head||!head->next) return head;
    ListNode *dummy=new ListNode(-1), *pre=dummy;
    dummy->next=head;
    while(pre->next){
        ListNode *cur=pre->next;
        for(;cur->next&&cur->next->val==cur->val;cur=cur->next);
        if(cur!=pre->next) pre->next=cur->next;
        else pre=pre->next;
    }
    return dummy->next;
}
```

The barycentre of the trees: 树的重心. mass center. barycenter

http://hehejun.blogspot.com/2018/05/leetcodethe-barycentre-of-trees.html

Binary Tree level order traversal:
http://www.cnblogs.com/grandyang/p/4051321.html

Binary Tree level order traversal II:
http://www.cnblogs.com/grandyang/p/4051326.html

Average of levels in binary tree:
http://www.cnblogs.com/grandyang/p/7259209.html

Binary tree zigzag order traversal:
http://www.cnblogs.com/grandyang/p/4297009.html

Minimum depth of binary tree:
http://www.cnblogs.com/grandyang/p/4042168.html

Maximum depth of binary tree:
http://www.cnblogs.com/grandyang/p/4051348.html

Binary tree vertical order traversal:
http://www.cnblogs.com/grandyang/p/5278930.html

N-ary tree level order traversal:
https://www.cnblogs.com/grandyang/p/9672233.html

Graph valid tree: **Union find.**
Traverse the nodes, if 2 nodes are connected, we update their root. We init root array as -1, and calling find() for both nodes in a pair, if we get the same value, there is a cycle, return 0, else, we union their root values.

```
bool validTree(int n, vector<pair<int, int>>& edges) {
        vector<int> roots(n, -1);
        for(auto a:edges) {
```

```
            int x=find(roots, a.first), y = find(roots, a.second);
            if(x==y) return 0;
            roots[x]=y;
        }
        return edges.size()==n-1;
    }
    int find(vector<int> &roots, int i){
        for(;roots[i]!=-1;i=roots[i]);
        return i;
    }
```

We can also use BFS. Use a queue for traverse, and a set for recording visit status. Note that we need to delete the node visited for an adjacency list.

```
bool validTree(int n, vector<pair<int, int>>& edges) {
        vector<unordered_set<int>> g(n, unordered_set<int>());
        unordered_set<int> s{{0}};
        queue<int> q{{0}};
        for (auto e:edges) {
            g[e.first].insert(e.second);
            g[e.second].insert(e.first);
        }
        while (!q.empty()) {
            int t=q.front();
            q.pop();
            for(auto a:g[t]) {
                if(s.count(a)) return 0;
                s.insert(a);
                q.push(a);
                g[a].erase(t);
            }
        }
        return s.size()==n;
    }
```

Validate BST Binary Search Tree: Inorder traversal, and check ascending order of the array. Note: duplicate is not allowed.
Time: O(n); Space: O(n)

```
vector<int> v;
    void run(TreeNode *rt){
        if(!rt) return;
        run(rt->left);
        v.push_back(rt->val);
        run(rt->right);
    }
    bool isValidBST(TreeNode* root) {
        run(root);
        int n=v.size();
        bool f=0;
        for(int i=0;i<n-1&&!f;++i)
            if(v[i]>=v[i+1]) f=1;
        return !f;
    }
```

kth smallest element in BST binary search tree:
Inorder traverse, return v[k]: Time: O(n); Space: O(n)
DQ: Time: O(n); Space: O(1) (if not considering recursion stack)

```
int kthSmallest(TreeNode* root, int k) {
        int cnt=count(root->left);
        if(k<=cnt) return kthSmallest(root->left, k);
        else if(k>cnt+1) return kthSmallest(root->right, k-cnt-1);
        return root->val;
    }
    int count(TreeNode* node) {
        if(!node) return 0;
        return 1+count(node->left)+count(node->right);
    }
```

If the modification & query(k) is often, to optimize, use DQ algo; and we need to change the structure of a node, to record the count number of the subtree starting from that node.

```
struct MyTreeNode {
        int val;
        int count;
        MyTreeNode *left;
        MyTreeNode *right;
        MyTreeNode(int x):val(x), count(1), left(NULL), right(NULL) {}
    };
    MyTreeNode* build(TreeNode* root){
        if(!root) return NULL;
        MyTreeNode *node=new MyTreeNode(root->val);
        node->left=build(root->left);
        node->right=build(root->right);
        if(node->left) node->count+=node->left->count;
        if(node->right) node->count+=node->right->count;
        return node;
    }
    int kthSmallest(TreeNode* root, int k){
        MyTreeNode *node=build(root);
        return helper(node, k);
    }
    int helper(MyTreeNode* node, int k){
        if (node->left){
            int cnt=node->left->count;
            if(k<=cnt) return helper(node->left, k);
            else if(k>cnt+1) return helper(node->right, k-1-cnt);
            return node->val;
        } else {
            if(k==1) return node->val;
            return helper(node->right, k-1);
        }
    }
```

2nd Minimum node in binary tree/2nd min in BST:

Recursion. We know that the root must be the smallest node, called "first"; we only need to find "second". We init second=INT_MAX;

and call recursion upon the root with parameter=(node, first, second). In the recursion:

1. **Node is null, return**
2. **Node!=first, node must be larger than first, we check whether smaller than second, and update if necessary.**
3. **Call recursion upon left/right node of current node.**

```
int findSecondMinimumValue(TreeNode* root) {
        int first=root->val, second=INT_MAX;
        helper(root, first, second);
        return (second==first||second==INT_MAX)?-1:second;
    }
    void helper(TreeNode* node, int& first, int& second){
        if(!node) return;
        if(node->val!=first&&node->val<second) second=node->val;
        helper(node->left, first, second);
        helper(node->right, first, second);
    }
```

Iteration, traverse layer by layer

```
int findSecondMinimumValue(TreeNode* root) {
        int first=root->val, second=INT_MAX;
        queue<TreeNode*> q{{root}};
        while(!q.empty()){
            auto t=q.front();
            q.pop();
            if(t->val!=first&&t->val<second) second=t->val;
            if(t->left) q.push(t->left);
            if(t->right) q.push(t->right);
        }
        return (second==first||second==INT_MAX)?-1:second;
    }
```

Unique Binary Search trees I: Catalan number. Since null tree is a tree, we set f(0)=1, f(1)=f(0)*f(0)=1*1=1.
When n=2, 1 and 2 are eligible to be a root: f(2)=f(0)*f(1)+f(1)*f(0).
Similarly, f(3)=f(0)*f(2)+f(1)*f(1)+f(2)*f(0). (root=1, root=2, root=3)

From this, we can get the recurrence equation as: f(0)=1, f(n+1)=Sum(i=0, n)(f(i)*f(n-i))
Time: O(n^2), space: O(n)

```
int numTrees(int n) {
        int dp[n+1]={0};
        dp[0]=dp[1]=1;
        for(int i=2;i<=n;++i) for(int j=0;j<i;++j) dp[i]+=dp[j]*dp[i-j-1];
        return dp[n];
    }
```

* Optimization: f(n)=(2n)!/((n+1)!*n!)=Multiply(k=2, n)((n+k)/k). With the formula, can calculate the number as On

Unique Binary Search trees II: Recursion, partition left and right subtree, construct with recursion. Use ptr to avoid variable copy.

```
vector<TreeNode *> generateTrees(int n) {
        if(!n) return {};
        return *dfs(1, n);
    }
    vector<TreeNode*> *dfs(int start, int end){
        vector<TreeNode*> *rst=new vector<TreeNode*>();
        if(start>end) rst->push_back(NULL);
        else{
            for(int i=start;i<=end;++i){
                vector<TreeNode*> *l=dfs(start, i-1),*r=dfs(i+1, end);
                for(int j=0;j<l>size();++j)
                    for(int k=0;k<r>size();++k) {
                        TreeNode *node=new TreeNode(i);
                        node->l=(*l)[j];
                        node->r=(*r)[k];
                        rst->push_back(node);
                    }
            }
        }
        return rst;
```

```
}
```

Binary trees with factors:

Count the number of BST can form from numbers given. DP, hash.
We can first try a 1d dp array, dp[i] represents the number of trees when the value of root is i. With this approach, we can get the result by summing all the elements in the array. For the bellman equation, as it is required that the root value must be product of left and right child, dp[root]=dp[left(root)]*dp[right(root)].
Since we need to find a value with a certain key (the key of the node) quickly, we can use hashmap into the algorithm. We can first set one key, and then find the other in the hashmap quickly. With this modification, we can use a hashmap as our dp container.
Since all the numbers are greater than 1, a child node must not be greater than its parent, and we can sort the array to traverse the small values first and build large values later.
Given the root, a child, to determine another child, we can first determine whether they are divisible, if yes, then calculate the quotient and check the hashmap. When all the dp values are calculated, we can sum them up.

```
int numFactoredBinaryTrees(vector<int>& A) {
        long rst=0, M=1e9+7;
        unordered_map<int, long> dp;
        sort(A.begin(), A.end());
        for(int i=0;i<A.size();++i) {
            dp[A[i]]=1;
            for(int j=0;j<i;++j){
                int t=A[i]/A[j];
                if(A[i]%A[j]&&dp.count(t)) dp[A[i]]=(dp[A[i]]+dp[A[j]]*dp[t])%M;
            }
        }
        for(auto a:dp) rst=(rst+a.second)%M;
        return rst;
    }
```

Serialize and deserialize binary tree:

Layer order traverse, iteration. BFS with queue

```
string serialize(TreeNode * rt) {
        string rst="";
        queue<TreeNode*> Q;
        if(rt) Q.push(rt);
        while(!Q.empty()){
            TreeNode *t=Q.front();
            Q.pop();
            if(t){
                rst+=to_string(t->val)+" ";
                Q.push(t->left);
                Q.push(t->right);
            }else rst+="# ";
        }
        return rst;
    }
TreeNode * deserialize(string &s) {
        // write your code here
        if(s.empty()) return nullptr;
        stringstream ss(s);
        queue<TreeNode*> Q;
        string t;
        ss>>t;
        TreeNode *rst=new TreeNode(stoi(t)), *c=rst;
        Q.push(c);
        while(!Q.empty()){
            TreeNode *tn=Q.front();
            Q.pop();
            if(!(ss>>t)) break;
            if(t!="#"){
                c=new TreeNode(stoi(t));
                Q.push(c);
                tn->left=c;
            }
```

```
            if(!(ss>>t)) break;
            if(t!="#"){
                c=new TreeNode(stoi(t));
                Q.push(c);
                tn->right=c;
            }
        }
        return rst;
    }
```

Inorder recursion:

```
public:
    string serialize(TreeNode* root) {
        string out;
        rs(root, out);
        return out;
    }
    TreeNode* deserialize(string data) {
        istringstream in(data);
        return rd(in);
    }
private:
    void rs(TreeNode *rt, string &oss) {
        if (rt) {
            oss+=to_string(rt->val)+' ';
            rs(rt->left, oss);
            rs(rt->right, oss);
        } else oss+="# ";
    }
    TreeNode* rd(istringstream &iss) {
        string val;
        iss>>val;
        if (val == "#") return nullptr;
        TreeNode *rt=new TreeNode(stoi(val));
        rt->left=rd(iss);
        rt->right=rd(iss);
```

```
        return rt;
    }
```

N queens:DFS, backtracking.

Scan layer by layer, we can use a array arr; arr[i] record the position of the queen in ith row, with initialization to -1. For each row, we traverse the cells(columns), and determine whether there is a conflict. When we reach the last row, one solution is generated.

```
class Solution {
public:
    vector<vector<string> > solveNQueens(int n) {
        vector<vector<string> > rst;
        vector<int> pos(n, -1);
        dfs(pos, 0, rst);
        return rst;
    }
    void dfs(vector<int> &pos, int row, vector<vector<string> > &rst) {
        int n=pos.size();
        if (row==n){
            vector<string> out(n, string(n, '.'));
            for (int i=0;i<n;++i) out[i][pos[i]]='Q';
            rst.push_back(out);
        }else
            for(int col=0;col<n;++col)
                if (isValid(pos, row, col)){
                    pos[row]=col;
                    dfs(pos, row+1, rst);
                    pos[row]=-1;
                }
    }
    bool isValid(vector<int> &pos, int row, int col){
        for(int i=0;i<row;++i)
            if(col==pos[i]||abs(row-i)==abs(col-pos[i])) return 0;
        return 1;
    }
```

};
Optimization: state compression.
Time: O(n^2), Space: O(n)
with analysis: https://segmentfault.com/a/1190000003762668
multi algo: http://www.cnblogs.com/TenosDoIt/p/3801621.html

```
class Solution {
private:
    vector<vector<string> > rst;
    int upperlim;
public:
    vector<vector<string> > solveNQueens(int n) {
        upperlim=(1<<n)-1;
        vector<string> cur(n, string(n, '.'));
        run(0, 0, 0, cur, 0);
        return res;
    }
    void run(int row, int ld, int rd, vector<string>&cur, int index){
        int pos, p;
        if (row!=upperlim){
            pos=upperlim&(~(row|ld|rd));
                //pos中二进制为1的位，表示可以在当前行的对应列放皇后
                //和upperlim与运算，主要是ld在上一层是通过左移位得到的
，它的高位可能有无效的1存在，这样会清除ld高位无效的1
            while(pos){
                p=pos&(~pos+1);
                //获取pos最右边的1,例如pos=010110，则p=000010
                pos=pos-p;
                //pos最右边的1清0
                setQueen(cur, index, p, 'Q');
                //在当前行，p中1对应的列放置皇后
                run(row|p, (ld|p)<<1, (rd|p)>>1, cur, index+1);
                //设置下一行
                setQueen(cur, index, p, '.');
            }
        }
```

```
        else res.push_back(cur);
                //找到一个解
    }
    //第row行，第loc1(p)列的位置放置一个queen或者清空queen，loc1(p)表示p中二进制1的位置
    void setQueen(vector<string>&cur, const int row, int p, char val){
        int col = 0;
        while(!(p & 1)){
            p >>= 1;
            col++;
        }
        cur[row][col] = val;
    }
};
```

N queens II: Simplification of N Queens, use a 1d array to record status of current queen (int state[n]). state[i]: the column of ith queen. To place a queen in a new row k:

1. **Check whether column is conflicted by checking state[k]==state[0:k-1]?conflict:no_conflict**
2. **Check whether diagonal is conflicted by checking |row1-row2|==|column1-column2|?conflict:no_conflict, 1st queen: (row1, column1), 2nd queen: (row2, column2)**

Time: O(n^2), Space: O(n)

https://segmentfault.com/a/1190000003762668

```
class Solution {
private:
    int rst;
public:
    int totalNQueens(int n) {
        vector<int> pos(n, -1);
        int rst=0;
        dfs(pos, rst, 0);
        return rst;
    }
```

```
    void dfs(vector<int> &pos, int &rst, int row){//放置第row行的皇后
        int n=pos.size();
        if(row==n){
            ++rst;
            return;
        }
        for(int col=0;col<n;++col)
            if(isValid(pos, row, col)){
                pos[row]=col;
                dfs(pos, rst, row+1);
                pos[row]=-1;
            }
    }
    //判断在row行col列位置放一个皇后，是否是合法的状态
    //已经保证了每行一个皇后，只需要判断列是否合法以及对角线是否合法。
    bool isValid(vector<int> &pos, int row, int col){
        for(int i=0;i<row;++i)//只需要判断row前面的行，因为后面的行还没有放置
            if(pos[i]==col||abs(row-i)==abs(col-pos[i])) return 0;
        return 1;
    }
};
```

Maximum sum of 3 non-overlapping subarrays:
http://www.cnblogs.com/grandyang/p/8453386.html

Continguous Array: Augmented sum/counter, Hash.
use a counter, and traverse the array:

1. **if v[i]==1: counter++**
2. **else v[i]==0: counter--**
3. **if mp.count(counter): rst=max(rst, i-mp[counter])**
4. **else mp[counter]=i**

when the iteration ends, return rst.
Time: O(n); Space: O(n)

```
int findMaxLength(vector<int>& v) {
        int n=v.size(), rst=0, cnt=0;
        unordered_map<int, int> mp{{0, -1}};
        for(int i=0;i<n;++i){
            if(v[i]) ++cnt;
            else --cnt;
            //if can be replaced by:
            //cnt+=(v[i]<<1)-1;
            if(!mp.count(cnt)) mp[cnt]=i;
            else rst=max(rst, i-mp[cnt]);
        }
        return rst;
    }
```

Peeking iterator:

Given an Iterator class interface with methods: next() and hasNext(), design and implement a PeekingIterator that support the peek() operation -- it essentially peek() at the element that will be returned by the next call to next().

Trivial, add peek() and hasNext() to an iterator. Peek() does not move the pointer, only next() does. We can use a buffer to store next value, and a flag variable to indicate whether there is a valid number in the buffer.

```
class PeekingIterator : public Iterator {
    int buf;
    bool flag;
public:
        PeekingIterator(const vector<int>& nums) : Iterator(nums) {
        flag=0;
    }
         int peek() {
        if(!flag) buf=Iterator::next();
        flag=1;
        return buf;
         }
```

```
        int next() {
        if(!flag) return Iterator::next();
        flag=0;
        return buf;
        }
        bool hasNext() const {
        return flag||Iterator::hasNext();
        }
};
```

For the peek(), we can create a copy and call its next(), the copy will be destroyed after use, and there is no memory issue for that.

```
class PeekingIterator : public Iterator {
public:
        PeekingIterator(const vector<int>& nums) : Iterator(nums) { }
        int peek() {
        return Iterator(*this).next();
        }
        int next() {
            return Iterator::next();
        }
        bool hasNext() const {
            return Iterator::hasNext();
        }
};
```

Follow-up: Generic and work with all types: Use template

```
template <class T>
class PeekingIterator: public Iterator {
   T buf;
   bool flag;
public:
        PeekingIterator(const vector<T>& nums) : Iterator(nums) {
                flag=0;
        }
        T peek() {
                if(!flag) buf=Iterator::next();
```

```
            flag=1;
            return buf;
        }
        T next() {
            if(!flag) return Iterator::next();
            flag=0;
            return buf;
        }
        bool hasNext() const {
            return flag||Iterator::hasNext();
        }
};
```

LRU Cache: Hashmap

get:

1. **If exist: move the element to the top by using splice (moving one or more nodes in a linked list to a specific position, here we only move the iterator of the key to the top of our structure), then return element**
2. **else: return -1**

put:

1. **If exist: remove original one**
2. **General: insert a new one on the top, check whether there is overflow, if yes, remove the end element.**

```
class LRUCache{
public:
    LRUCache(int capacity) {
        cap = capacity;
    }
    int get(int key) {
        auto it = m.find(key);
        if (it == m.end()) return -1;
        lst.splice(lst.begin(), lst, it->second);
        return it->second->second;
    }
```

```
    void put(int key, int value) {
        auto it = m.find(key);
        if (it != m.end()) lst.erase(it->second);
        lst.push_front(make_pair(key, value));
        m[key] = lst.begin();
        if (m.size() > cap) {
            int k = lst.rbegin()->first;
            lst.pop_back();
            m.erase(k);
        }
    }
private:
    int cap;
    list<pair<int, int>> lst;
    unordered_map<int, list<pair<int, int>>::iterator> m;
};
```

Largest Rectangle in Histogram: Monotone Stack.

1. **Create a stack stk.**
2. **Traverse the bar array:**
    a. **If(stk.empty or bar[i]>bar[stk.top]): stk.push(i)**
    b. **If(bar[i]<bar[stk.top]): keep pop while bar[stk.top]>bar[i]. Let the removed bar be bar[tp], calculate the area with bar[tp] as smallest bar. For bar[tp], the left index is the previous item in the stack next to tp, and right index is current i.**
3. **If(!stk.empty): remove all bars one by one and do 2-b for every removal.**

Time: O(n); Space: O(n). Every bar is pushed and poped only once.

```
int getMaxArea(int hist[], int n){
    // Create an empty stack. The stack holds indexes
    // of hist[] array. The bars stored in stack are
    // always in increasing order of their heights.
    stack<int> s;
    int max_area = 0; // Initalize max area
```

```
    int tp;  // To store top of stack
    int area_with_top; // To store area with top bar as the smallest bar
    // Run through all bars of given histogram
    int i=0;
    while(i<n){
        //If this bar is higher than the bar on top
        //stack, push it to stack
        if (s.empty()||hist[s.top()] <= hist[i]) s.push(i++);
        //If this bar is lower than top of stack,
        //then calculate area of rectangle with stack
        //top as the smallest (or minimum height) bar.
        //'i' is 'right index' for the top and element
        //before top in stack is 'left index'
        else{
            tp=s.top();  //store the top index
            s.pop();  //pop the top
            //Calculate the area with hist[tp] stack
            //as smallest bar
            area_with_top=hist[tp]*(s.empty()?i:i-s.top()-1);
            //update max area, if needed
            if(max_area<area_with_top) max_area=area_with_top;
        }
    }
    // Now pop the remaining bars from stack and calculate
    // area with every popped bar as the smallest bar
    while(!s.empty()){
        tp=s.top();
        s.pop();
        area_with_top=hist[tp]*(s.empty()?i:i - s.top() - 1);
        if(max_area<area_with_top) max_area=area_with_top;
    }
    return max_area;
}
```

Maximal Rectangle: similar to largest rectangle in histogram

Update each column of a row with corresponding column of previous row, and find largest histogram area for that row.

1. **Traverse all the rows:**
   for each col in that row:
   if(M[row][col]==1): A[row][col]+=A[row-1][col]
   find area for this row and update max area.

Time: O(row*col); Space:

```
int maximalRectangle(vector<vector<char> > &matrix) {
    if(matrix.empty()) return 0;
    int rst=0;
    vector<int> height(matrix[0].size(), 0);
    for(int i=0;i<matrix.size();++i){
        for(int j=0;j<matrix[0].size();++j)
            if(matrix[i][j]=='0') height[j]=0;
            else ++height[j];
        rst=max(rst, largestRectangleArea(height));
    }
    return rst;
}
int largestRectangleArea(vector<int> &height) {
    stack<int> s;
    height.push_back(0);
    int maxa=0;
    for(int i = 0;i<height.size();++i){
        if(s.empty()||height[i]>=height[s.top()]) s.push(i);
        else{
            int tmp=height[s.top()];
            s.pop();
            maxa=max(maxa, tmp*(s.empty()?i:i-1-s.top()));
            --i;
        }
    }
    return maxa;
}
```

Maximal Square: DP, dp[][].

dp[i][j]: the max length of edge of the square when with a point as (i,j).

We first consider the boundary, i=0||j=0: max area can not be larger than 1.

And for dp[i][j], we only need to consider left/up/left&up. (note that when V[i][j]=0, dp[i][j] is 0 as well). If V[i][j]=1, we need to check dp[i-1][j-1], dp[i][j-1], dp[i-1][j], and find the min, and then +1, we will get dp[i][j]. This is because there cannot be a 0 in the square, we will only taken the unioned value of the 3 statuses.

Time: O(m*n); Space: O(m*n)

```
int maximalSquare(vector<vector<char>>& matrix) {
        if(matrix.empty()||matrix[0].empty()) return 0;
        int m=matrix.size(), n=matrix[0].size(), rst=0, dp[m][n]={0};
        for(int i=0;i<m;++i) {
            for(int j=0;j<n;++j) {
                if(!i||!j) dp[i][j]=matrix[i][j]-'0';
                else if(matrix[i][j]=='1') dp[i][j]=min(dp[i-1][j-1], min(dp[i][j-1],
dp[i-1][j]))+1;
                rst=max(rst, dp[i][j]);
            }
        }
        return rst*rst;
    }
```

Space optimization: Since for dp[i][j], we only need dp[i-1][j], dp[i-1][j-1], dp[i][j-1], we can use a buffer variable pre, and a 1D dp array to solve the problem.

Space O(n)

```
int maximalSquare(vector<vector<char>>& matrix) {
        if(matrix.empty()) return 0;
        int m=matrix.size(), n=matrix[0].size(), rst=0, pre, dp[n]={0};
        for(int i=0;i<m;++i) {
            for(int j=0;j<n;++j){
                int tmp=dp[j];
                if(!i||!j||matrix[i][j]=='0') dp[j]=matrix[i][j]-'0';
```

```
                else dp[j]=min(pre, min(dp[j], dp[j - 1]))+1;
                rst=max(dp[j], rst);
                pre=tmp;
            }
        }
        return rst*rst;
    }
```

Number of Islands: DFS. Use a visited array to record visit, we need recursion of 4 directions.
Time: O(m*n); Space: O(m*n)

```
int numIslands(vector<vector<char> > &grid) {
        if(grid.empty()||grid[0].empty()) return 0;
        int m=grid.size(), n=grid[0].size(), rst=0;
        vector<vector<bool> > visited(m, vector<bool>(n, 0));
        for(int i=0;i<m;++i){
            for(int j=0;j<n;++j){
                if(grid[i][j]=='1'&&!visited[i][j]){
                    dfs(grid, visited, i, j);
                    ++rst;
                }
            }
        }
        return rst;
    }
    void dfs(vector<vector<char>> &G, vector<vector<bool>> &V, int x, int
y){
        if(x<0||x>=G.size()) return;
        if(y<0||y>=G[0].size()) return;
        if(G[x][y] != '1' || V[x][y]) return;
        V[x][y]=1;
        dfs(G, V, x-1, y);
        dfs(G, V, x+1, y);
        dfs(G, V, x, y-1);
        dfs(G, V, x, y+1);
```

```
}
```

Number of Islands II/Add land: Union find

We first validate whether a cell is part of an island(not water). And we need a array roots[m*n] to record which location belongs to which island. Suppose every location is a single island, we can use coordinate to represent the island. In the init stage, we set all values to be -1 so that we can easily know what part is not converted into island yet. Then we traverse the island array, set its label as its coordinate, ++island_cnt, then traverse 4 directions, and skip water, then we use getRoot to find adjacent islands' label. If the returned value is the same, do nothing, otherwise, merge the 2 island, --island_cnt. Optimized with path Compression.

Time: O(k*α(n*m)), Space: O(m*n); α: inverse Ackermann function

```
vector<int> numIslands2(int m, int n, vector<Point>& positions) {
    vector<int> rst, r(m * n, -1);
    int cnt=0;
    vector<vector<int>> dirs{{0, -1}, {-1, 0}, {0, 1}, {1, 0}};
    for(auto a : positions){
        int id=n*a.x+a.y;
        if(r[id]==-1){
            r[id]=id;
            ++cnt;
        }
        for (auto dir:dirs){
            int x=a.x+dir[0], y=a.y+dir[1], cur_id=n*x+y;
            if(x<0||x>=m||y<0||y>=n||roots[cur_id]==-1) continue;
            int p=findRoot(r, cur_id), q=findRoot(r, id);
            if (p!=q){
                roots[p] = q;
                --cnt;
            }
        }
        rst.push_back(cnt);
    }
```

```
        return rst;
    }
    int rt(vector<int>& r, int id) {
        if(id==r[id]) return id;
        else r[id]=findRoot(r, r[id]);
        return r[id];
    }
```

Max area of island/largest area of island/max island/largest island:

Time: O(n*m)

BFS with queue

```
vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int m=grid.size(), n=grid[0].size(), rst=0;
        for(int i=0;i<m;++i){
            for(int j=0;j<n;++j){
                if(!grid[i][j]) continue;
                int cnt=0;
                queue<pair<int, int>> q{{{i, j}}};
                grid[i][j]*=-1;
                while(!q.empty()){
                    pair<int, int> t= q.front();
                    q.pop();
                    rst=max(rst, ++cnt);
                    for(auto dir:dirs) {
                        int x=t.first+dir[0], y=t.second+dir[1];
                        if(x<0||x>=m||y<0||y>=n||grid[x][y]<=0) continue;
                        grid[x][y]*=-1;
                        q.push({x, y});
                    }
                }
            }
        }
        return rst;
    }
```

Time: O(m*n), Space: O(1)
Recur:

```
int maxAreaOfIsland(vector<vector<int>>& grid) {
        int n=grid.size(), m=grid[0].size(), rst=0;
        if(!n||!m) return 0;
        for(int i=0;i<n;++i)
            for(int j=0;j<m;++j)
                rst=max(rst, area(grid, j, i, m, n));
        return rst;
    }
    int area(vector<vector<int>>& G, int x, int y, int m, int n) {
        if(x<0||y<0||x>=m||y>=n||!G[y][x]) return 0;
        G[y][x]=0;
        return area(G, x-1, y, m, n)+area(G, x+1, y, m, n)+area(G, x, y-1, m,
n)+area(G, x, y+1, m, n)+1;
    }
```

Number of distinct Islands: DFS, bit compression/relative coordinate, hash
http://www.cnblogs.com/grandyang/p/7698778.html

Number of distinct Islands II: DFS
https://blog.csdn.net/magicbean2/article/details/79282937

Basic Calculator: stack. If encounter a number, may be multiple digit, need to read them all.

```
int calculate(string s) {
        stack <int> nums, ops;
        int num=0, rst=0, sign=1;
        for (char c:s)
            if (c>='0'&&c<='9') num=num*10+c-'0';
            else{
                rst+=sign*num;
                num=0;
                if(c=='+') sign=1;
```

```
            if(c=='-') sign=-1;
            if(c=='('){
                nums.push(rst);
                ops.push(sign);
                rst=0;
                sign=1;
            }
            if(c==')'&&ops.size()){
                rst=ops.top()*rst+nums.top();
                ops.pop();
                nums.pop();
            }
        }
        rst+=sign*num;
        return rst;
    }
```

Excel sheet column title: positional notation

Time: O(logn); Space: O(1)

```
string convertToTitle(int n) {
        string t, rst;
        for(;n>0;n/=26){
            --n;
            t+=(char)((n%26)+'A');
        }
        int l=t.length();
        for(int i=l-1;i>=0;rst+=t[i--]);
        return rst;
    }
```

Letter case permutation: Trivial

We let result be an empty set, if there is a number, simply append it, if there is a char, we append both lower and upper case. For the appending, we append the char to all elements in the set.

Time: O(n!); Space: O(n!)

```
vector<string> letterCasePermutation(string &s) {
        // write your code here
        vector<string> rst;
        int n=s.length();
        rst.push_back("");
        if(!n) return rst;
        for(char c : s){
            int l=rst.size();
            if(c>='0'&&c<='9') for(string &str : rst) str+=c;
            else
                for(int i=0;i<l;++i){
                    rst.push_back(rst[i]);
                    rst[i]+=tolower(c);
                    rst[i+l]+=toupper(c);
                }
        }
        return rst;
    }
```

Merge Intervals: sort, linear scan

Time: O(nlogn) (Considering delete in array, it takes O(n^2)

```
def merge(self, V):
        if(len(V)<2): return V
        V=sorted(V, key=lambda x: (x.start))
        s=len(V)-1
        i=0
        while(i!=s):
            if V[i].end>=V[i+1].start:
                V[i].end=max(V[i].end, V[i+1].end)
                del V[i+1]
                s-=1
            else:
                i+=1
        return V
```

Time: O(nlogn)

```
vector<Interval> merge(vector<Interval>& intervals) {
        int n = intervals.size();
        vector<Interval> rst;
        vector<int> l, r;
        for (int i = 0; i < n; ++i) {
            l.push_back(intervals[i].start);
            r.push_back(intervals[i].end);
        }
        sort(l.begin(), l.end());
        sort(r.begin(), r.end());
        for (int i = 0, j = 0; i < n; ++i) {
            if (i == n - 1 || r[i + 1] > l[i]) {
                rst.push_back(Interval(l[j], r[i]));
                j = i + 1;
            }
        }
        return rst;
    }
```

Integer to roman: Trivial, positional notation, list all cases

```
string intToRoman(int num) {
        string v1[4]={"", "M", "MM", "MMM"}, v2[10]={"", "C", "CC", "CCC",
"CD", "D", "DC", "DCC", "DCCC", "CM"}, v3[10]={"", "X", "XX", "XXX",
"XL", "L", "LX", "LXX", "LXXX", "XC"}, v4[10]={"", "I", "II", "III", "IV", "V",
"VI", "VII", "VIII", "IX"};
        return v1[num/1000]+v2[(num %
1000)/100]+v3[(num%100)/10]+v4[num%10];
    }
```

Roman to Integer: Map. There are 2 cases to consider, if current char is the last char, or the following char is smaller or equal to it, we add it; else we minus it.

```
int romanToInt(string s) {
        int rst=0;
```

```
        unordered_map<char, int> m{{'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C',
100}, {'D', 500}, {'M', 1000}};
        for(int i= 0;i<s.size();++i) {
            int val=m[s[i]];
            if(m[s[i+1]]<=m[s[i]]||i==s.size()-1) rst+=val;
            else rst-=val;
        }
        return rst;
    }
```

Anagrams: Hash, bit compression. Use a 26-bit int array to record time of occurrences of a char, and hash it.
More problems: http://www.cnblogs.com/grandyang/p/4385822.html

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> rst;
        unordered_map<string, vector<string>> m;
        for (string str:strs) {
            vector<int> cnt(26, 0);
            string t="";
            for(char c:str) ++cnt[c-'a'];
            for(int d:cnt) t+=to_string(d)+"-";
            m[t].push_back(str);
        }
        for (auto a:m) rst.push_back(a.second);
        return rst;
    }
```

Word Break: DP, hash.
Hash the dictionary vector first.
Use 1d dp array. dp[i]: whether string ranging in [0, i) can be separated. Since we need to handle null string, we need to add size of dp by 1: dp[s.length()+1], and we need to initialize dp[0] as 1. Then, we start to traverse s. We need a nested loop, as we need to check all the substrings. We use to separate [0, i) in to [0, j) and [j, i), and dp[j] represent [0, j), [j, i) will be s.substr(j, i-j). Since dp[j] is

calculated in previous process, we need need to check existence in dictionary of s.substr(j, i-j). If both are true, dp[i]=true, and we can break the loop as [0, i) is separable, and there is no need to use j to break [0, i). In the end, we return dp[s.length()] to indicate the result.

Time: O(n^2), Space: O(n) or O(sizeof(dictionary))

```
bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size()+1, 0);
        dp[0]=1;
        for(int i=0;i<dp.size();++i)
            for(int j=0;j<i;++j)
                if(dp[j]&&wordSet.count(s.substr(j, i-j))){
                    dp[i]=1;
                    break;
                }
        return dp.back();
    }
```

Word Ladder: BFS, hashmap.

Use BFS to traverse possible edition for a word, use a hashmap to setup the end word and its iteration length, and the started word should be mapped as 1. We also need a queue for the BFS, we push the starting words into the queue and then start to traverse. For every word, we substitute letter by letter with other alphabets, and check whether it equals to the end word, and we can return the end word, and plus 1 in the hashmap. If the substituted word not exist in the hashmap, we need to push it into the queue, and +1 in the value keyed by previous end word from the hashmap. If the search is finished, we return 0.

```
int ladderLength(string bw, string ew, vector<string>& wlst) {
        unordered_set<string> wset(wlst.begin(), wlst.end());
        if(!wset.count(ew)) return 0;
        unordered_map<string, int> mp{{{bw, 1}}};
        queue<string> q{{bw}};
```

```
        while(!q.empty()){
            string word=q.front();
            q.pop();
            for(int i=0;i<word.size();++i){
                string nw=word;
                for(char ch='a';ch<='z';++ch){
                    nw[i]=ch;
                    if (wset.count(nw)&&nw==ew) return mp[word]+1;
                    if (wset.count(nw)&&!mp.count(nw)){
                        q.push(nw);
                        mp[nw]=mp[word] + 1;
                    }
                }
            }
        }
        return 0;
    }
```

Word ladder II: BFS to find all of the path.

We setup a set of path to record them. For the starting path p, we record the starting word, and use 2 int var to record:

1. **len: current length of the path in the iteration**
2. **minLen: minimum length of the path recorded.**

By doing this, we can efficiently prune some path if they are larger than the currently known best. In addition, we also need a set to record words we encountered during the search. Then we need to specify our BFS approach. We need to traverse the path we established, and pop the front path, if it has a path longer than variable "len", we know that there are some words already recorded in the path, if it appears more than 1 times, the path is definitely not the shortest path, and we need to delete such words in the dictionary and clear the word set, and pruning the iteration. Then we pop the last word of current path, substitute every char to check whether dictionary has the substituted word. If it has, we added into the word set, and add the path to generate this word in

the original path we traversed. If this word is the end word, we added into the result, and update the minLen, else, we add the new path and continue our search.

```
vector<vector<string>> findLadders(string bw, string ew, vector<string>&
wlst) {
        vector<vector<string>> rst;
        unordered_set<string> dict(wlst.begin(), wlst.end());
        vector<string> p{bw};
        queue<vector<string>> paths;
        paths.push(p);
        int len=1, minLen=INT_MAX;
        unordered_set<string> wset;
        while(!paths.empty()){
            auto t=paths.front();
            paths.pop();
            if(t.size()>len){
                for(string w:wset) dict.erase(w);
                wset.clear();
                len=t.size();
                if(len>minLen) break;
            }
            string last=t.back();
            for(int i=0;i<last.size();++i){
                string tmp=last;
                for(char ch='a';ch<='z';++ch){
                    tmp[i]=ch;
                    if(!dict.count(tmp)) continue;
                    words.insert(tmp);
                    vector<string> newPath=t;
                    newPath.push_back(tmp);
                    if(tmp==ew){
                        rst.push_back(newPath);
                        minLen=len;
                    } else paths.push(newPath);
                }
```

```
        }
    }
    return rst;
}
```

Text justification: String manipulation

Handle the string row by row. We first determine the number of words that each row can hold, that is, to compare (n_words+(n-1)_whitespace) with given L, and then calculate the number of white spaces in this line. Then we need to append each word with the spaces. Since the last line has different settings with the others, we need to handle it separately. And we should be careful about remainder of division while distributing the white spaces. For instance, if there is a line containing 3 words (which has 2 spaces in between) with number of whitespaces not divisible by 2, the left space should have more whitespace than the right one.

```
vector<string> fullJustify(vector<string> &words, int L) {
    vector<string> rst;
    int i=0;
    while(i<words.size()){
        int j=i, len=0;
        while(j<words.size()&&len+words[j].size()+j-i<=L)
            len+=words[j++].size();
        string tr;
        int sp=L-len;
        for(int k=i;k<j;++k) {
            tr+=words[k];
            if(sp>0){
                int tmp;
                if(j==words.size()) tmp=(j-k)==1?sp:1;
                else
                    if(j-k-1>0) tmp=sp/(j-k-1)+(!(sp%(j-k-1))?0:1);
                    else tmp=sp;
                tr.append(tmp, ' ');
```

```
                sp-=tmp;
            }
        }
        rst.push_back(tr);
        i=j;
    }
    return rst;
}
```

Letter Combinations of a phone number: Dictionary. Use a dictionary to record chars represented by a number. We traverse the digits to get all the chars, and traverse strings in the current result, to append the chars, and save the result.
For the dictionary, since 1 represents no char, we set it as “”.
Time: O(digits.length*4^n);Space: O(digits.length*4^n)

```
vector<string> letterCombinations(string digits) {
    if(digits.empty()) return {};
    vector<string> rst{""};
    string dict[9]={"", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"};
    for(int i=0;i<digits.length();++i){
        vector<string> t;
        string str=dict[digits[i]-'0'-1];
        for(int j=0;j<str.length();++j)
            for(string s:rst) t.push_back(s + str[j]);
        rst=t;
    }
    return rst;
}
```

Trapping Rain water/trap water/接雨题:
pre-compute highest bar on the left and right of every given bar. Then use the values to find the amount of water in every unit “bar”.
Time: O(n); Space: O(n)

```
int trap(vector<int>& height) {
```

```
        if(!height.size()) return 0;
        int s=height.size(), l[s], r[s], rst=0;
        l[0]=height[0], r[s-1]=height[s-1];
        for(int i=1;i<s;++i) l[i]=max(l[i-1], height[i]);
        for(int i=s-2;i>=0;--i) r[i]=max(r[i+1], height[i]);
        for(int i=0;i<s;++i) rst+=min(l[i], r[i])-height[i];
        return rst;
    }
```

Trapping rain water II: BFS. Bar on the boundary cannot trap any water, we can push all such points into the queue and they will be the starting point of BFS.
To trap water, we need to make sure there is a notch whose trapping amount will be the smallest height of bar around it. To solve the problem, we can simulate the process of sea level ascending. We need to maintain a sea level height, mx=-INF, and start to increase it from 1. Then the BFS will need to start to traverse from the shortest bar; to achieve this, we can change the queue in our BFS to priority queue, and place the shortest bar in the front to start BFS from this bar.
To simulate the process, we need to set the condition of BFS that it cannot go to another node higher than the current node. However, for the nodes that is attempted to visit while being held by this condition, we can add it into the priority queen.
For each iteration finished, we increase the sea level to next height stored in the priority queue and do the BFS from these nodes. And for each node lower than current node, we can store water in it, and update the result.
When the BFS ends, we will get the total water amount.
Time: O(n^2) Space: O(n); Note that n=width*height

```
int trapRainWater(vector<vector<int>>& heightMap) {
        if(heightMap.empty()) return 0;
        int m=heightMap.size(), n=heightMap[0].size(), res=0,
mx=INT_MIN, dir[4][2]={{0,-1},{-1,0},{0,1},{1,0}};
```

```
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> q;
        vector<vector<bool>> visited(m, vector<bool>(n, false));
        for(int i=0;i<m;++i)
            for(int j=0;j<n;++j)
                if(!i||i==m-1||!j||j==n-1){
                    q.push({heightMap[i][j], i*n+j});
                    visited[i][j]=1;
                }
        while(!q.empty()){
            auto t=q.top();
            q.pop();
            int h=t.first, r=t.second/n, c=t.second%n;
            mx=max(mx, h);
            for (int i=0;i<dir.size();++i){
                int x=r+dir[i][0], y=c+dir[i][1];
                if(x<0||x>=m||y<0||y>=n||visited[x][y]) continue;
                visited[x][y]=1;
                if (heightMap[x][y]<mx) rst+=mx-heightMap[x][y];
                q.push({heightMap[x][y], x*n+y});
            }
        }
        return rst;
    }
```

First Bad version: **binary search**

```
int firstBadVersion(int n) {
        int l=0, r=n-1;
        for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1)
            if(isBadVersion(m+1)) r=m;
            else l=m+1;
        return r+1;
    }
```

First position of target: **binary search**

```
int binarySearch(vector<int> &array, int target) {
    if(!array.size()) return -1;
    int l=0, r=array.size()-1;
    for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1))
        if(array[m]==target) r=m;
        else if(array[m]<target) l=m+1;
        else r=m-1;
    return array[l]==target?l:-1;
  }
```

Last position of target: binary search

```
int binarySearch(vector<int> &array, int target) {
    if(!array.size()) return -1;
    int l=0, r=array.size()-1;
    for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1))
        if(array[m]==target) l=m;
        else if(array[m]<target) l=m+1;
        else r=m-1;
    return array[r]==target?r:-1;
  }
```

Find first and last element: Binary search

Time: O(logn)

Combination of first and last.

Search for a range: Binary search

Binary search twice, first find left boundary, then find the right one.

Time: O(logn)

```
vector<int> searchRange(vector<int>& nums, int target) {
      vector<int> rst(2, -1);
      int l=0, r=nums.size()-1;
      for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1))
          if(nums[m]<target) l=m+1;
          else r=m;
      if(nums[r]!=target) return rst;
```

```
        rst[0]=r;
        r=nums.size();
        for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1))
            if(nums[m]<=target) l=m+1;
            else r=m;
        rst[1]=l-1;
        return rst;
    }
```

Longest palindrome:

We need to find the number of characters that occurs odd number of times which can be done with a set. The final result will be s.length-max(0, set.size()-1). The reason is that, if there is no odd number char, all char can be used, and we will compare 0 and -1 which give us 0 which will return s.length(), a correct answer; if there is odd number char, we will need to minus it by 1, as there is at most 1 char that cannot be paired thoroughly in a palindrome. Time: O(n); Space: O(1) (there are limited number of char can exist)

```
int longestPalindrome(string s) {
        unordered_set<char> st;
        for(char c:s)
            if(!st.count(c)) st.insert(c);
            else st.erase(c);
        return s.length()-max(0, (int)(st.size()-1));
    }
```

Palindrome Permutation II:

http://www.cnblogs.com/grandyang/p/5315227.html

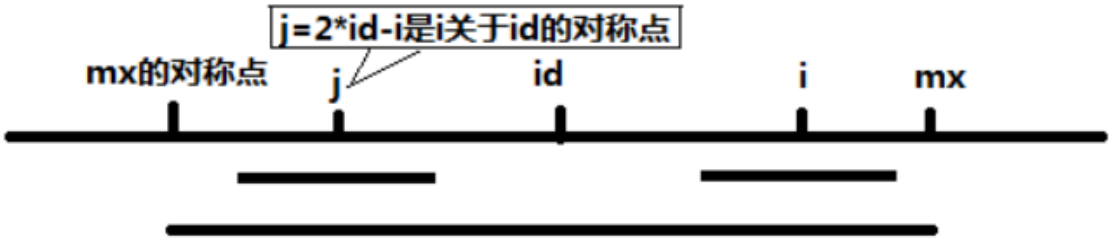
Palindrome Permutation:

http://www.cnblogs.com/grandyang/p/5223238.html

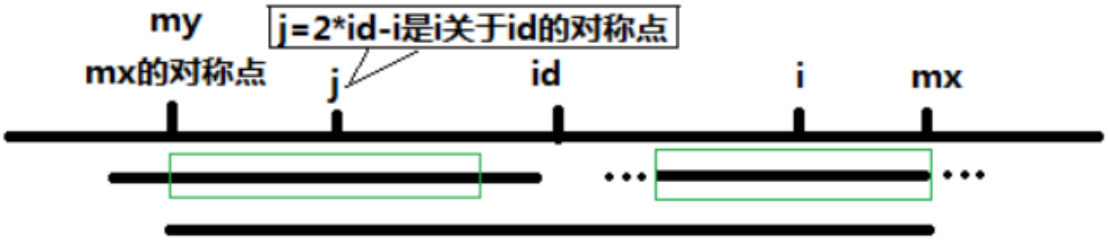
Longest Palindrome substring: Manacher

Add a non-existing char '#' into the string to make it has odd number of chars without affecting already existing palindromes, name it t.
To solve the problem , we can then find the longest palindrome radius and its center. And to deal with possible negative index of the substring index, we add another non-existing char '@' in front of the modified char. eg("#1#2#1#")
We use an array p with same length as modified string, p[i] represents the radius of palindrome with center as t[i], if p[i]=1, the palindrome is t[i] itself, a '#' char.
To calculate p[i], we need 2 more variables for expansion. one (r) records the palindrome's expansion to the longest position in the right side; one (rc) records the center of palindrome.
When r>i, p[i]=min(p[2*rc-i], r-i); otherwise, p[i]=1.
This is because:

1. **r-i>p[2*rc-i], the palindrome centered with s[2*rc-i] is included in the palindrome centered with s[rc], as i and 2*rc-i is symmetric about rc, we have p[i]=p[2*rc-i]**



2. **When p[2*rc-i]>=r-i, the palindrome centered with s[2*rc-i] may not be included in the palindrome centered with s[rc]. However, based on the symmetricity of palindromes, we know that the palindrome centered with s[i] is able to expand to s[r], meaning p[i]=r-i, and for further substring, we cannot make assumptions and have to do further matching.**



3. **For r<=i, we cannot make any assumption upon p[i], and we can only set it as 1 for further matching.**

Thus, p[i]=r>i?min(p[2*rc-i], r-i):1;

Time: O(n); Space: O(n)

```
string longestPalindrome(string s) {
        string t="$#";
        for(int i=0;i<s.size();t+=s[i++], t+='#');
        int r=0, rc=0, rst=0, rstl=0, p[t.size()]={0};
        for(int i=1;i<t.size();++i){
            for(p[i]=r>i?min(p[(rc<<1)-i], r-i):1;t[i+p[i]]==t[i-p[i]];++p[i]);
            if(r<i+p[i]){
                r=i+p[i];
                rc=i;
            }
            if(rstl<p[i]){
                rstl=p[i];
                rst=i;
            }
        }
        return s.substr((rst-rstl)/2, rstl-1);
    }
```

Ruassian Doll envelopes: binary search, LIS

```
int maxEnvelopes(vector<pair<int, int>>& envelopes) {
        vector<int> dp;
        sort(envelopes.begin(), envelopes.end(), [](const pair<int, int> &a,
const pair<int, int> &b){
            if (a.first == b.first) return a.second > b.second;
            return a.first < b.first;
```

```
        });
        for (int i = 0; i < envelopes.size(); ++i) {
            int left = 0, right = dp.size(), t= envelopes[i].second;
            while (left < right) {
                int mid = left + (right - left) / 2;
                if (dp[mid] < t) left = mid + 1;
                else right = mid;
            }
            if (right >= dp.size()) dp.push_back(t);
            else dp[right] = t;
        }
        return dp.size();
    }
```

Longest increasing subsequence: LIS, DP, binary search

http://www.cnblogs.com/grandyang/p/4938187.html

```
int lengthOfLIS(vector<int>& nums) {
        vector<int> dp;
        for (int i=0;i<nums.size();++i) {
            int l=0, r=dp.size();
            for(int m=((r-l)>>1);l<r;m=l+((r-l)>>1)
                if(dp[m]<nums[i]) l=m+1;
                else r=m;
            if(r>=dp.size()) dp.push_back(nums[i]);
            else dp[r]=nums[i];
        }
        return dp.size();
    }
```

Russian doll envelopes: Binary search, LIS, dp

http://www.cnblogs.com/grandyang/p/5568818.html

```
int maxEnvelopes(vector<pair<int, int>>& envelopes) {
        vector<int> dp;
        sort(envelopes.begin(), envelopes.end(), [](const pair<int, int> &a,
const pair<int, int> &b){
```

```
            if (a.first == b.first) return a.second > b.second;
            return a.first < b.first;
        });
        for (int i = 0; i < envelopes.size(); ++i) {
            int left = 0, right = dp.size(), t= envelopes[i].second;
            while (left < right) {
                int mid = left + (right - left) / 2;
                if (dp[mid] < t) left = mid + 1;
                else right = mid;
            }
            if (right >= dp.size()) dp.push_back(t);
            else dp[right] = t;
        }
        return dp.size();
    }
```

Paint House:
http://www.cnblogs.com/grandyang/p/5319384.html

Paint House II:
http://www.cnblogs.com/grandyang/p/5322870.html

Maximum subarray IV/xor array/xor subarray:
Brute Force: O(n^2)

```
int maxSubarrayXOR(int arr[], int n){
    int rst=-INF;     // Initialize result
    // Pick starting points of subarrays
    for(int i=0;i<n;++i){
        int t=0; // to store xor of current subarray
        // Pick ending points of subarrays starting with i
        for (int j=i;j<n;++j){
            t^=arr[j];
            rst=max(rst, t);
        }
    }
```

```
    return rst;
}
```

maximum subarray xor, Trie data structure

Time: O(n)

1. **Create a Trie. Every node of Trie will have two children, either 0 or 1.**
2. **Initialize a variable pre=0 and insert into the Trie.**
3. **Initialize rst=-INF**
4. **We then traverse the array:**
    a. **pre=pre^arr[i]; pre now contains the xor result of elements from arr[0] to arr[i].**
    b. **We then can search the max xor value ending with arr[i] in the trie.**
    c. **We update result if the queried value in previous step is larger than rst.**

How does 4.b work?

We can observe from above algorithm that we build a Trie that contains XOR of all prefixes of given array. To find the maximum XOR subarray ending with arr[i], there may be two cases.

i) The prefix itself has the maximum XOR value ending with arr[i]. For example if i=2 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[2] is the whole prefix.

ii) We need to remove some prefix (ending at index from 0 to i-1). For example if i=3 in {8, 2, 1, 12}, then the maximum subarray xor ending with arr[3] starts with arr[1] and we need to remove arr[0].

To find the prefix to be removed, we find the entry in Trie that has maximum XOR value with current prefix. If we do XOR of such previous prefix with current prefix, we get the maximum XOR value ending with arr[i]. If there is no prefix to be removed (case i), then we return 0 (that's why we inserted 0 in Trie).

https://www.geeksforgeeks.org/find-the-maximum-subarray-xor-in-a-given-array/

```
#include<bits/stdc++.h>
using namespace std;
```

```
// Assumed int size
// A Trie Node
struct TrieNode {
    int value;  // Only used in leaf nodes
    TrieNode *arr[2];
};
// Utility function tp create a Trie node
TrieNode *newNode() {
    TrieNode *temp = new TrieNode;
    temp->value = 0;
    temp->arr[0] = temp->arr[1] = NULL;
    return temp;
}
// Inserts pre_xor to trie with given root
void insert(TrieNode *root, int pre_xor) {
    TrieNode *temp = root;
    // Start from the msb, insert all bits of
    // pre_xor into Trie
    for (int i=INT_SIZE-1; i>=0; i--) {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);
        // Create a new node if needed
        if (temp->arr[val] == NULL)
            temp->arr[val] = newNode();
        temp = temp->arr[val];
    }
    // Store value at leaf node
    temp->value = pre_xor;
}
// Finds the maximum XOR ending with last number in
// prefix XOR 'pre_xor' and returns the XOR of this maximum
// with pre_xor which is maximum XOR ending with last element
// of pre_xor.
int query(TrieNode *root, int pre_xor)
{
```

```
    TrieNode *temp = root;
    for (int i=INT_SIZE-1; i>=0; i--) {
        // Find current bit in given prefix
        bool val = pre_xor & (1<<i);
        // Traverse Trie, first look for a
        // prefix that has opposite bit
        if (temp->arr[1-val]!=NULL)
            temp = temp->arr[1-val];

        // If there is no prefix with opposite
        // bit, then look for same bit.
        else if (temp->arr[val] != NULL)
            temp = temp->arr[val];
    }
    return pre_xor^(temp->value);
}
// Returns maximum XOR value of a subarray in arr[0..n-1]
int maxSubarrayXOR(int arr[], int n) {
    // Create a Trie and insert 0 into it
    TrieNode *root = newNode();
    insert(root, 0);
    // Initialize answer and xor of current prefix
    int result = INT_MIN, pre_xor =0;
    // Traverse all input array element
    for (int i=0; i<n; i++) {
        // update current prefix xor and insert it into Trie
        pre_xor = pre_xor^arr[i];
        insert(root, pre_xor);
        // Query for current prefix xor in Trie and update
        // result if required
        result = max(result, query(root, pre_xor));
    }
    return result;
}
```

Median of two sorted arrays/find median of two sorted arrays:

DQ. We can partition 2 arrays into 2 halves with the same size and the median can be easily found then.

We first consider find the median in 1 array. To deal with odd number of size, we can copy the right middle element, like [1 2 3], we can partition it as [1 2] [2 3], to do this, we can set left ptr l as (n-1)/2, and right ptr r as n/2 and the median can be represented as (v[(n-1)/2]+v[n/2])/2. As there are 2*n+1 positions for a cut in one array, and index of l is (n-1)/2, index of r is n/2, we can represent l as (cut-1)/2, r as cut/2.

With the 2 arrays problem, we should set up the 2 partitions as "any number in the left should not be greater than any number in the right". As there are 2*n+2*m+2 cutting positions, there are n+m cutting positions on one side of the cut, and 2 is occupied by the cut itself.

Therefore, if we cut at index i in array b, the cut in a should be n+m-i. With such cut strategy, we can have 2 left numbers and 2 right numbers: l1=a[(c1-1)/2]], r1=a[c1/2]; l2=b[(c2-1)/2], r2=b[c2/2].

We now need to check whether the cut is the final cut. After the cut, we need to determine whether this is the final cut. As l1, l2 are the largest numbers on the left and r1, r2 are the smallest number on the right, we only need to make sure that l#<=r#.

Based on this idea, we can implement a DQ to calculate the result.

1. **l1>r2: there are too many large numbers on the left half of v1, we need to move the cut in v1 to the left.**
2. **l2>r1: there are too many large numbers on the left half of v2, we need to move the cut in v2 to the left.**
3. **valid cut: return (max(l1, l2)+min(r1, r2))/2**

Note:

- **as cut in an array can determine the cut in another, we can just move one of the two cuts. And we should move the one in shorter array, as all cut positions are possible for the valid cut. While in the longer array, there are positions illegal for a valid cut (too small/too large). For example, given [1], [1 2 3 4 5 6], the position between 1 and 2 is impossible as there are**

too little element on the left half. Therefore, we can use the long array as the basis for the first cut which will needs a check for the range. By cutting on the shorter array, we can reach time complexity as O(log(min(a.size(), b.size())))

- **The general idea is done, and we now need to think about some edge cases. One is that when a cut falls on the first or last position of the array, like cut2=2*b.size(), then r2=b[2*b.size()/2]=b[b.size()], which is an invalid index. To deal with this, we can use 2 extra element, as we can set -INF at a[-1] and INF at b[b.size()]. This will not affect the answer, and make it much easier to implement.**

Time: O(log(n+m)); Space: O(1)

```
#define INF 0x3f3f3f3f
double findMedianSortedArrays(vector<int>& a, vector<int>& b) {
    int n=a.size(), m=b.size();
    if(n<m) return findMedianSortedArrays(b, a);     //let b be shorter
    int l=0, r=m*2;
    while(l<=r){
        int ms=(l+r)/2, ml=n+m-ms;//cut
        double l1=!ml?-INF:a[(ml-1)/2];
        double l2=!ms?-INF:b[(ms-1)/2];
        double r1=ml==n*2?INF:a[ml/2];
        double r2=ms==m*2?INF:b[ms/2];
        if(l1>r2) l=ms+1;        //a's lower half is too big, move c1 left
        else if(l2>r1) r=ms-1; //b's lower half is too big, move c2 left
        else return (max(l1, l2)+min(r1, r2))/2;  //valid cut
    }
    return -INF; //invalid case
}
```

Find the maximum element in an array which is first increasing and then decreasing/升序降序/increase decrease find max:

Binary search. Based on the original binary search:

1. **mid>both its neighbors: mid is max**
2. **mid>next&&mid<pre: max is on the left**

3. **mid<next&&mid>pre: max is on the right**

Time: O(logn); Space: O(1)

```
int recur(vector<int> a, int l, int r){
        //only 1 element now
        if(l==r) return a[l];
        //only 2 elements:
        //1. first is greater, then the first element is maximum
        //2. second is greater, then the 2nd is maximum
        if(r==l+1) return (arr[l]>arr[r]?arr[l]:arr[r]);
        int m=((l+r)>>1);
        //arr[m]>=both neighbor, it is max
        if(a[m]>a[m+1]&&a[m]>a[m-1]) return a[m];
        //arr[m]>next and <pre, update right
        if(a[m]>a[m+1]&&a[m]<a[m-1]) return recur(a, l, m-1);
        //arr[m]<next and >pre, update left
        else return recur(a, m+1, r);
}
int run(vector<int> a){
        return recur(a, 0, a.size()-1);
}
```

Maximum value in an array after m range increment operations/range query/range increment:

Efficient update:

1. **add the needed value to lower bound of the range**
2. **reduce upper bound+1 index by the value**

After all updates, add all values, return the maximum sum

Time: O(m+n)

```
//a: start indexes; b: end indexes; c: added value
int run(vector<int> a, vector<int> b, vector<int> c, int n){
        vector<int> t(n, 0);
        for(int i=0;i<a.size();++i){
                int l=a[i], r=b[i];
                t[l]+=c[i];
                t[r+1]-=c[i];
```

```
        }
        int sum, rst=-INF;
        for(int i=0;i<n;++i){
                sum+=t[i];
                rst=max(rst, sum);
        }
        return rst;
}
```

Search in rotated sorted array:

Reservoir Sampling: