# Ⅰ.Array

#### Tow pointer method

One pointer move from left to right,another one move from right to left.Based on certain regulation,these two pointer move in different direction to achieve the goal.

For example:

Container with most water,using two pointer method to comparing area is able to reduce complexity(O(N)).

Triplet(three numbers with sum equal to 0).Fixing one numbers,using two pointer to traverse array and check sum based on sum = nums[i] + nums[left] + nums[right].

First of all,the array is sorted by ascend,then exploiting two pointers method to traverse.According the value of sum,we move the different pointers.

If sum > 0,the number pointed by right pointer is too large,so right pointer --;

If sum < 0,the number pointed by left pointer is too small,so left pointer++;

If sum = 0,we get the triplet conforming condition.

for (int i = 0; i < nums.size(); i++)

        {

            if (nums[i] > 0)

                break;

            if (i > 0 && nums[i-1] == nums[i])

                continue;

            left = i+1;

            right = nums.size() - 1;

            while (left < right)

            {

                sum = nums[i] + nums[left] + nums[right];

                if (sum > 0)

                    right--;

                else if (sum < 0)

                    left++;

                else

                {

                    vector<int> one;

                    one.push\_back(nums[i]);

                    one.push\_back(nums[left]);

                    one.push\_back(nums[right]);

                    output.push\_back(one);

                    last\_left = nums[left];

                    last\_right = nums[right];

                    while(left < right && nums[left] == last\_left)

                        left++;

                    while(left < right && nums[right] == last\_right)

                        right--;

                }

            }

        }

3 Sum closest problem(output the sum of three numbers that is closest to target value).It has the same principle as last example.

sort(nums.begin(), nums.end());

for (int i = 0; i < nums.size()-2; i++)

        {

            int left = i + 1, right = nums.size() - 1;

            while(left < right)

            {

                int sum = nums[i] + nums[left] + nums[right];

                if (sum == target)

                    return sum;

                if (abs(sum - target) < diff)

                {

                    diff = abs(sum - target);

                    closest = sum;

                }

                (sum > target) ? right-- : left++;

            }

        }

4 Sum problem(output four numbers with sum equal to target).

sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 1; i++)

        {

            for (int j = i + 1; j < nums.size(); j++)

            {

                left = j + 1;

                right = nums.size() - 1;

                while(left < right)

                {

                    long long sum = (long long)nums[i] + (long long)nums[j] + (long long)nums[left] + (long long)nums[right];

                    if (sum == target)

                    {

                        unique.insert({nums[i], nums[j], nums[left], nums[right]});

                    }

                    (sum > target) ? right-- : left++;

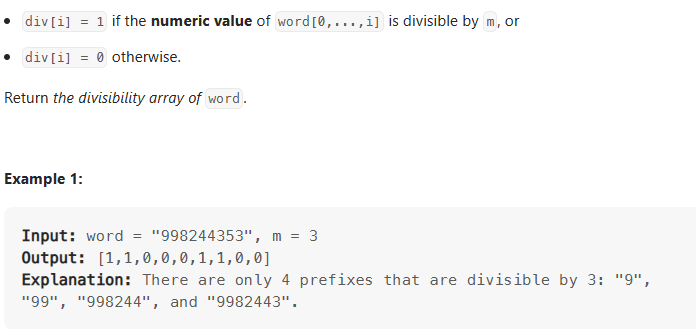
                }

            }

        }

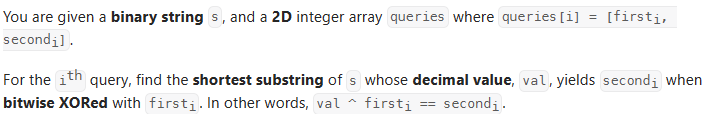
# **Ⅱ.String**

Find the Divisibility Array of a String:



Due to the limitation of size of integer,there is a stepwise method to avoid using oversized integer.To be specified,if the input is “9945” and 3, first number 9 is divisible with remainder equal to 0,so 99 is equivalent to 09,994 is equivalent to 04 that is not divisible with remainder 1,then 9945 is equivalent to 15 that is divisible.

Substring XOR Queries：



Analysis:

Exploiting nested loop finds all combination of binary and stores them into the map in decimal representation of binary - <index\_left,index\_right> style.Then we traverse queries to check the specified the condition such that value ^ first = second equivalent to value = first ^ second.Therefore,we use map.count() to assure whether required value exists in map.

for (int i = 0; i < s.length(); i++)

        {

            x = 0;

            for (int j = 0;i+j < s.length() && j < 32; j++)

            {

                x = (x << 1) + (s[i + j] == '1');

                if (tmp.count(x) == 0)

                {

                    tmp[x].push\_back(i);

                    tmp[x].push\_back(i+j);

                }

                if (s[i] == '0')

                    break;

            }

        }

        for (auto &i : queries)

             {

                 if (tmp.count(i[0]^i[1]))

                 {

                     ans.push\_back(tmp[(i[0]^i[1])]);

                 }

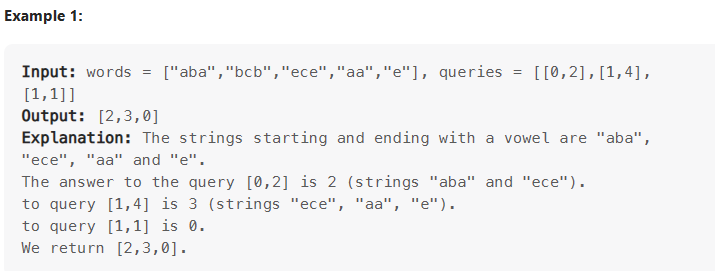
                 else

                     ans.push\_back({-1, -1});

             }

#### Prefix sum method:

Count Vowel Strings in Ranges:



Analysis:If we take nested loop method to solve with O(nxq) time complexity,it will be over time consuming.First of all,pre-traversing words gets the numbers of vowel string.Then we traverse the queries to get number of required strings.

vector<int> preCheck(words.size()+1);

        preCheck[0] = 0;

        int i = 0;

        for (auto str: words)

        {

            preCheck[i + 1] = preCheck[i] + (isVowel(str.at(0)) && isVowel(str.at(str.length()-1)));

            i++;

        }

        for (auto q : queries)

        {

            ans.push\_back(preCheck[q[1]+1] - preCheck[q[0]]);

        }

Notes:To achieve erasing repeating char,we are able to use unordered\_set to erase repeating char:

unordered\_set<char> s(str.begin(), str.end());

            string type(s.begin(), s.end());

            sort(type.begin(), type.end());

#### Vector::emplace\_back:

Parameters are the arguments to forward to the constructor of the element

Return value is a reference to the inserted element.

Application instance:

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<President> elections;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "emplace\_back:\n";

auto& ref = elections.emplace\_back("Nelson Mandela", "South Africa", 1994);

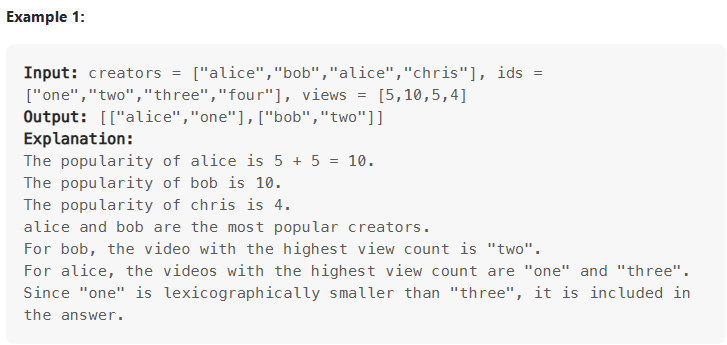
[assert](http://en.cppreference.com/w/cpp/error/assert)(ref.year == 1994 && "uses a reference to the created object (C++17)");

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<President> reElections;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "\npush\_back:\n";

reElections.push\_back(President("Franklin Delano Roosevelt", "the USA", 1936));

Most Popular Video Creator:



vector<vector<string>> ans;

map<string, long> cnt;

        map<string, vector<pair<long,string>>>pmap;

        long m = 0;

        for (int i = 0; i < creators.size(); i++)

        {

            cnt[creators[i]] += views[i];

            m = max(m, cnt[creators[i]]);

            pmap[creators[i]].emplace\_back(-views[i], ids[i]);

        }

        for (auto [c, vec] : pmap)

        {

            if (cnt[c] != m)

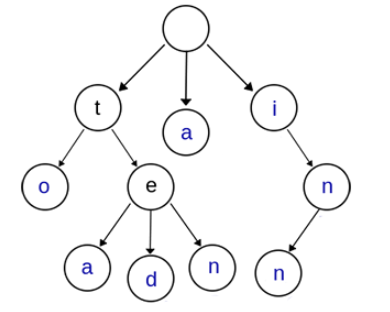
                continue;

            ans.push\_back({c, min\_element(vec.begin(),vec.end())->second});

        }

Show the application of emplace\_back and certain operation of map container.

**Trie(Prefix tree) Method**



{“a”, “to”, “tea”, “ted”, “ten”, “i”, “in”, “inn”}

The figure shows the characteristics of trie :

1. root node has no char
2. All chars on the path from root node to any node construct the string of corresponding node.
3. Every nodes has different number and type of char.
4. Two string have the same prefix,they must have the same pre-path.

Application:

String retrieval:

struct trie\_node

{

bool isKey; // flag whether current node represent a key

trie\_node \*children[26]; // child nodes

};

Frequency counting:

struct trie\_node

{

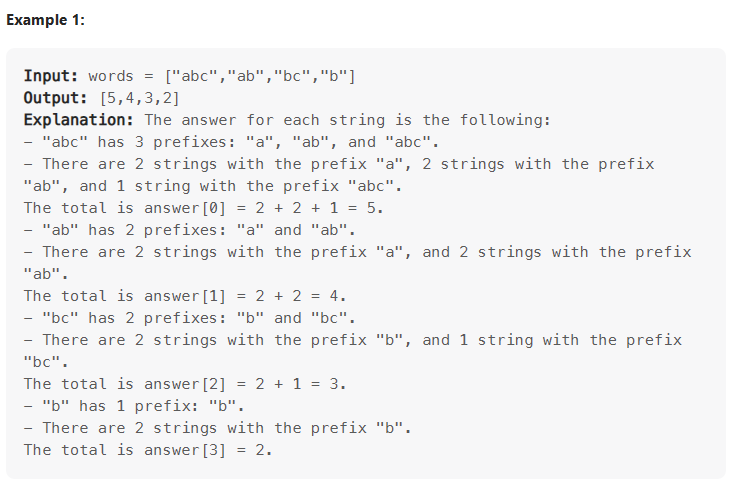
int count; // 记录该节点代表的单词的个数

trie\_node \*children[26]; // 各个子节点

};

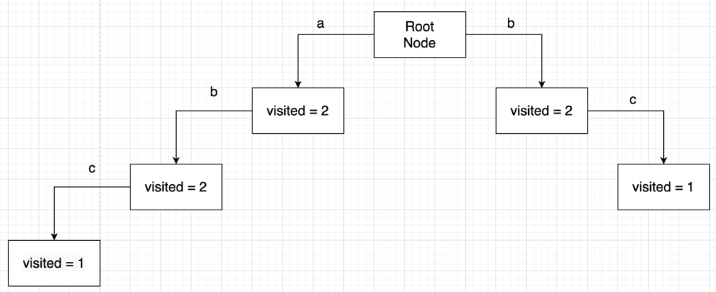
String sorting and prefix matching.

Example:



Analysis:

We construct the trie of the words



struct Trie{

        Trie \*chl[26] = {};

        int visited = 0;

    };

Trie t;

//construct the trie

        for (auto word : words)

        {

            auto tp = &t;

            for (auto ch : word)

            {

                if (!tp->chl[ch - 'a'])

                    tp->chl[ch - 'a'] = new Trie;

                tp = tp->chl[ch - 'a'];

                tp->visited++;

            }

        }

//

        for (auto word : words)

        {

            int num = 0;

            auto tp = &t;

            for (auto ch : word)

            {

                num += tp->chl[ch - 'a']->visited;

                tp = tp->chl[ch - 'a'];

            }

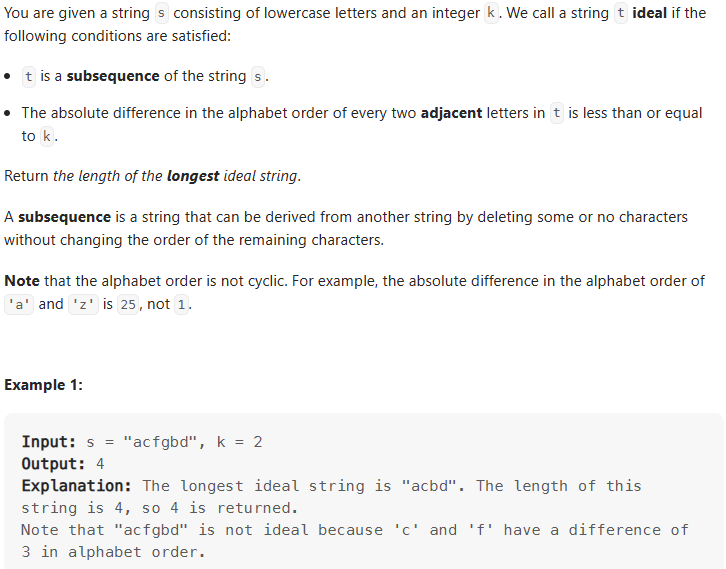
            ans.push\_back(num);

        }

The difference between i++ and ++i:

Both method will increase i by 1,but they return different type.i++ return the temporary variable.++i return reference.if i = 0,i++ return 0 and ++i return 1.

Longest Ideal Subsequence:



int dp[150] = {}, ans = 0;;

        for (auto &ch : s)

        {

            for (auto i = ch-k; i <= ch + k; i++)

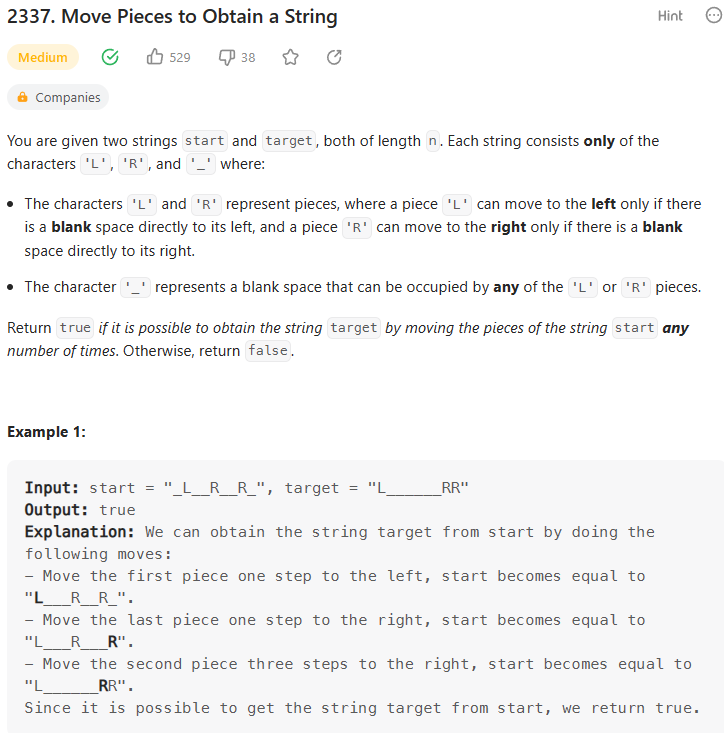
            {

                dp[ch] = max(dp[ch], dp[i]);

            }

            ans = max(ans, ++dp[ch]);

        }



Analysis:

There are several sufficient conditions to support target:

1. the sequence of LR string(remove all ‘\_’) has to be identical
2. The number of “L” or “R” has to be identical
3. if current char is 'L',then this condition must hold i>= j , if in target string it found at index j and , in start string it found at i.Besides,if current char is “R”,then then this condition must hold i<= j.

If it obey any of them above condition,it return false.

int len = start.length();

        int i = 0, j = 0;

        while (i <= len && j <= len)

        {

            while(start[i] == '\_')

                i++;

            while(target[j] == '\_')

                j++;

            if (i == len|| j == len)//condition2 check

                return (i == len && j == len);

            if (start[i] != target[j])//condition1 check

                return false;

            if (start[i] == 'L')//condition 3 check

            {

                if (i < j) return false;

            }

            else

            {

                if (i > j) return false;

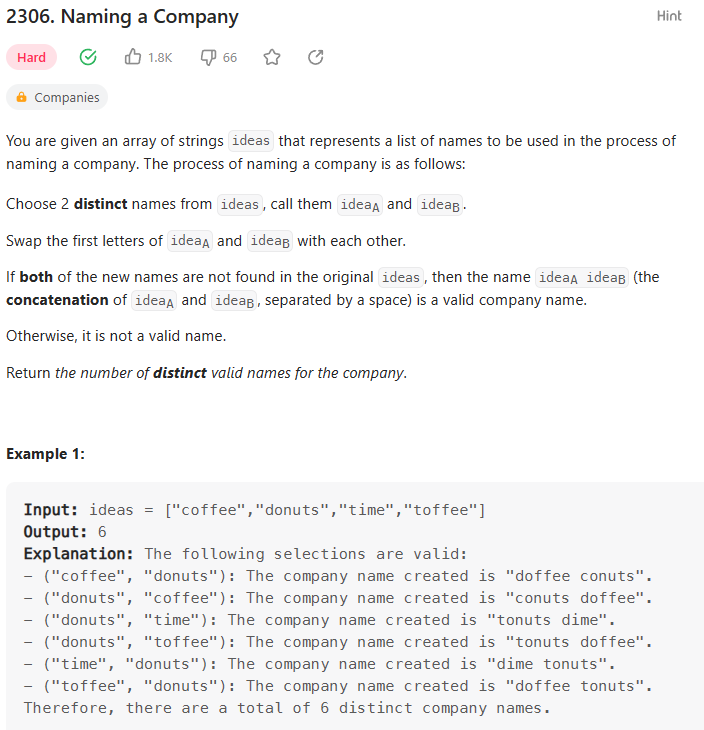
            }

            i++;

            j++;

        }

        return true;

Naming a company: 

Analysis:

We store the ideas in hash\_map in <head, tails> style.It becomes:

No head tails

1 c offee

2 d onuts

3 t ime,offee

To make sure the name being valid,we have to compare the tails and count the occurrence of repeated tail.To be more specified,using nested loop to compare every element in the hash\_map and accumulate the number of valid name.For instance,when we use first element compare with third element,we find 3rd element incorporates repeated tail “offee”,so repeated counter(minus) add 1.Therefore,according to this round comparing,ans += 2 \* (1 - 1) \* (2 - 1).

long long ans = 0;

        unordered\_map<char, set<string>>mp;

        for (auto &str : ideas)

        {

            mp[str[0]].insert(str.substr(1, str.length()-1));

        }

        for (int i = 97; i < 127; i++)

        {

            for (int j = i+1; j < 128; j++)

            {

                int minus = 0;

                for (auto &ss1 : mp[i])

                {

                    if (mp[j].find(ss1) != mp[j].end())

                        minus++;

                }

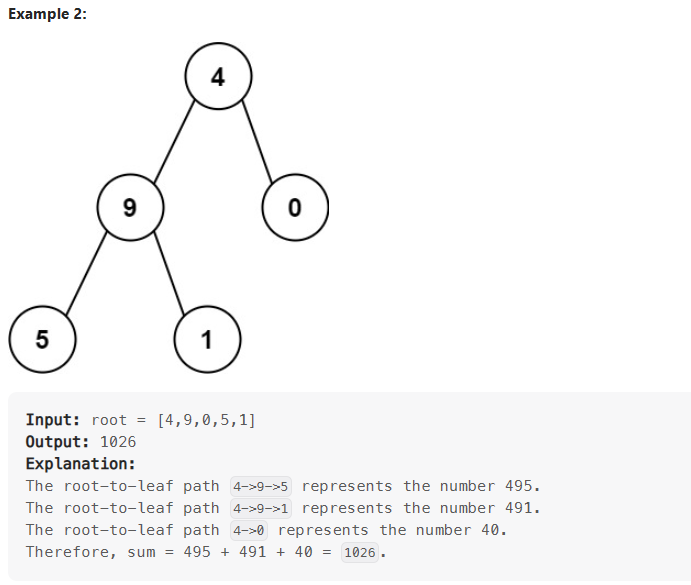
                ans += 2 \* (mp[i].size()-minus) \* (mp[j].size()-minus);

            }

        }

# **Ⅲ.Tree**

Sum Root to Leaf Numbers:



Analysis:

Exploiting pre-order method(root,left,right) traverse the tree.Based on

tmp = tmp\*10 + node->val,

we are able to figure out every number constructed by specified regulation.When it traverses to the new layer,tmp is multiplied by 10 and add current node value.Importantly,the value of tmp will not be modified by calculation of other nodes meaning every node has fixed tmp.However,value ans will be delivered to calculation of other node to make sure accumulating the tmp.

int sumNumbers(TreeNode\* root) {

        long long answer = 0;

        calculator(root, 0, answer);

        return (int)answer;

    }

    void calculator(TreeNode\* node, int tmp, long long &ans){

        if (node->left == nullptr && node->right == nullptr)

        {

            tmp = tmp\*10 + node->val;

            ans += tmp;

            return;

        }

        tmp = tmp\*10 + node->val;//

        if (node->left)

            calculator(node->left, tmp, ans);

        if (node->right)

            calculator(node->right, tmp, ans);

    }

# **Ⅳ.Back Tracking**

When we use brute-force method to traverse all possible results and the layer of nested loop is changeable,exploiting the back tracing control the layers of nested loop.

Application scenario:tree,combination,permutation,subset,segmentation,subsequence,checkerboard

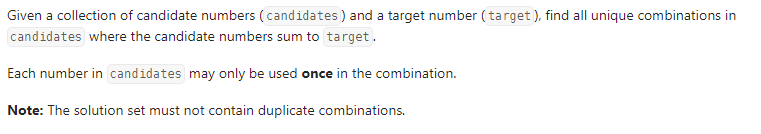
Note:most cases need optimization(pruning)

Analysis method:

1. build the tree data diagram
2. determine the arguments and return value of recursion function
3. Determine the return condition
4. Define recursive logic of single layer
5. Pruning(optimization, Deduplication)

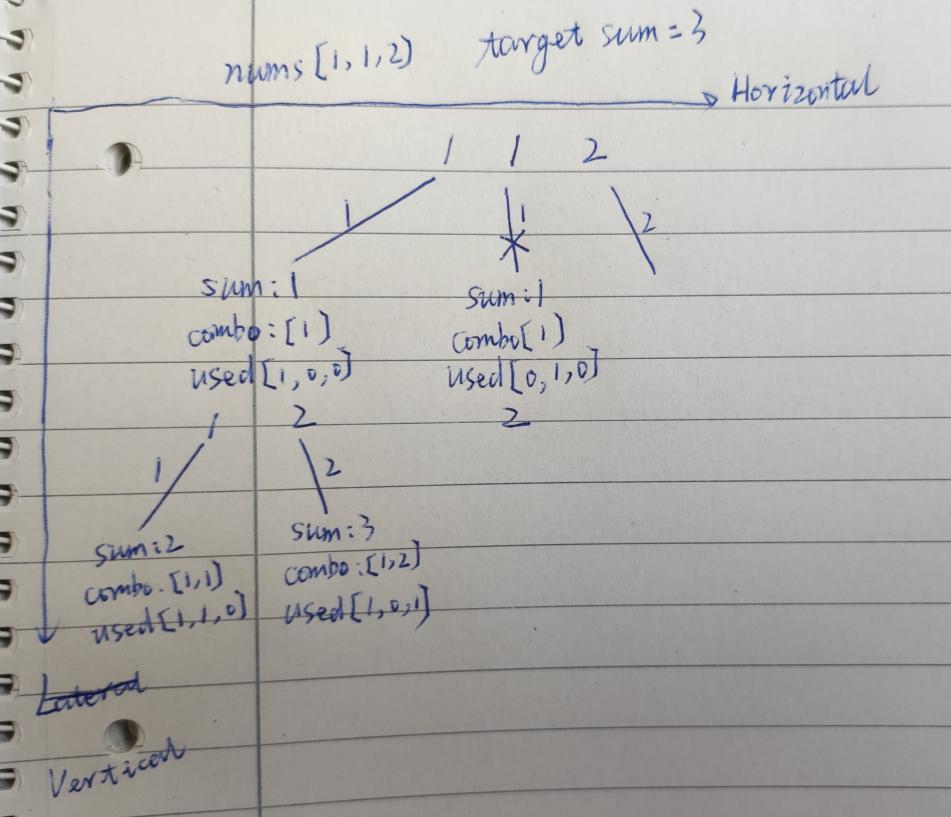
For instance:

Leetcode40:



Analysis:

Tree data diagram:



Cross means that current branch is supposed to prune

vector<int>path;//store all path

    vector<vector<int>>res;//store valid path

    void backTracing(int startIndex, int &sum, int tar, vector<int>& candidates, vector<bool>&used)

    {

        if (sum >= tar)

        {

            if (sum == tar)

                res.push\_back(path);

            return;

        }

        for (int i = startIndex; i < candidates.size(); i++)

        {

//pruning for deduplication

            if (i > 0 && candidates[i] == candidates[i-1] && used[i-1] == 0)

                continue;

            path.push\_back(candidates[i]);

            sum += candidates[i];

            used[i] = 1;

            backTracing(i+1, sum, tar, candidates, used);

            path.pop\_back();

            sum -= candidates[i];

            used[i] = 0;

        }

    };

    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

        vector<bool>used(candidates.size(),0);

        sort(candidates.begin(), candidates.end());

        int sum = 0;

        backTracing(0, sum, target, candidates, used);

        return res;

    }

# **Ⅴ.Greedy Algorithm**

Local solution deduces the global solution.

For instance:

Leetcode376:Wiggle subsequence

Local solution:when the prediff(previous number minus current number) is more than and equal to zero and curdiff(current number minus next number) is less than zero or prediff is less than and equal to zero and curdiff is more than zero,the number of wiggle subsequence increases by 1.

Traversing first n-1 number,we are about to get the global solution.

curdiff = nums[i+1] - nums[i];

            if (prediff <= 0 && curdiff > 0 || prediff >= 0 && curdiff < 0)

            {

                res++;

                prediff = curdiff;

            }

Leetcode 53:Maximum subarray

Local solution:when the current accumulation is less than the current number,we drop the accumulation and restart accumulating from current number on,simultaneously recording the maximum value of accumulation.

sum += nums[i];

            if (sum < nums[i])

                sum = nums[i];

            res = max(res, sum);

# **Ⅴ.Dynamic Programming**

**Some concepts:**

**Recursion with memorization**

**Space replace time**

**Recursive pruning**

**Analysis method:**

**First step:**

**Find the state representation including definition of state set and attribute(max/min/count).**

**Second step:**

**State calculation separate the state set into several subset,defining the state equation of every subset ,handling subsets and combining the outcome of all subset then get the result.**

**Optimization:**

**Space**

For example:

**01 Knapsack problem:**

Given N items and their value w[N] and volume v[N].Find the max value with specified volume constrain m.Every item can only be used once.

Analysis:

First step:find the state representation:

f(i,j) represents the set of all plans that first i items with volume being not exceeding j have.

Attribute is the max value.

Second step:state calculation

Separating the set into two subset based on whether choose i th item.

Don’t Choose: f[i][j] = f[i-1][j]

Choose: f[i][j] = f[i-1][j-v[i]] + w[i] when j >= v[i]

Brute coding:

int maxValue(vector<int> W, vector<int> V, int m){

Const int N = 1010;

int f[N][N];

for (int i = 1; i <= W.size(); i++)

        {

for (int j = 0; j <= m; j++)

        {

f[i][j] = f[i-1][j];

if (j >= v[i]) f[i][j] = max(f[i][j],f[i-1][j-V[i]] + W[i]);

}

}

return f[n][m];

}

Optimization:

Considering the space optimization

int f[N];

for (int i = 1; i <= W.size(); i++)

for (int j = m; i >= V[i]; j--)

f[j] = max(f[j], f[j-V[i]]+W[i]);//right side belongs to i-1 //layer and left side belongs to i layer.To make sure f[j-V[i]] from //last layer,we control j varying from big to small.

return f[m];

**Completed Knapsack problem:**

The item can be used multiple times.

Analysis:

First step:

same as the Knapsack problem

Second step:

Separating the set into many subsets based on how many times the i th item is used.An item can be used 0,1,...,k,...,n times constrained by the targeting volume.

f[i][j]=max(f[i-1][j],f[i-1][j-v[i]]+w[i],f[i-1][j-2\*v[i]]+2\*w[i],...,f[i-1][j-k\*v[i]]+k\*w[i],... ).

Due to:

f[i][j-v[i]]=max(f[i-1][j-v[i]],f[i-1][j-2\*v[i]]+w[i],...,f[i-1][j-k\*v[i]]+(k-1)\*w[i], ... );

So we can get state equation:

f[i][j] = max(f[i-1][j], f[i][j-v[i]]+w[i]);

Brute coding:

int maxValue(vector<int> W, vector<int> V, int m){

Const int N = 1010;

int f[N][N];

for (int i = 1; i <= W.size(); i++)

        {

for (int j = 0; j <= m; j++)

        {

f[i][j] = f[i-1][j];

if (j >= v[i]) f[i][j] = max(f[i][j],f[i][j-V[i]] + W[i]);

}

}

return f[n][m];

}

Optimization:

int f[N];

for (int i = 1; i <= W.size(); i++)

for (int j = V[i]; i <= m; j++)

f[j] = max(f[j], f[j-V[i]] + W[i]);//different from Knapsack problem,here //uses i th f[j-V[i]],so it control j varying from small to big.

Return f[m];

**Multiple Knapsack problem:**

The item i can be used s[i] times.

There will be one more cycle to enumerate the using times(1~s).It will increase the time complexity.So there is a method to optimize this situation:

We make the problem be equivalent to 01 Knapsack problem,which is achieved by reconstructing the items pool,putting s item i into items pool.So,the number of items becomes the sumo f s.Undoubtedly,it still has a huge time complexity,so we have to optimize the process of problem conversion by binary optimization method.

Binary optimization method:every number within s[i] can be represented by a binary rule.For example:

If s[i] = 7,all number less than and equal to 7 can be presented by 1,2,4.

if s[i] = 10,all number less than and equal to 10 can be presented by 1,2,4,x.

x = 10-1-2-4 = 3.

So,10 item i is equivalent to four items with w = w[i]\*(1/2/4/3),v = v[i]\*(1/2/4/3).

Coding:

struct Good

{

int v, w;

};

int main()

{

cin >> n >> m;

/\*normal version

for (int i = 1; i <= n; i++){

Int v,w,s;

cin << v << w << s;

for (int j = m; j >= v; j--)

for (int k = 1; k<=s&&k\*v<=j; k++)

f[j] = max(f[j], f[j-k\*v] + k\*w);

}

\*/

/\*Binary optimization version

vector<Good> goods;

for (int i = 1; i <= n; i++)

{

int v, w, s;

cin >> v >> w >> s;

for (int k = 1; k <= s; k \*= 2)

{

s -= k;

goods.push\_back({v\*k, w\*k});

}

if (s > 0)

goods.push\_back({v\*s, w\*s});

}

for (int i = 0; i < goods.size(); i++)

for (int j = m; j >= goods[i].v; j--)

f[j] = max(f[j], f[j-goods[i].v] + goods[i].w);

\*/

cout<<f[m]<<endl;

return 0;

}

**Interval DP:**

Interval DP has the following characteristics:

Merge : to integrate two or more parts, of course, it can also be reversed;

Features : can decompose the problem into a form that can be combined in pairs;

Solving : Set the optimal value for the entire problem, enumerate the merge points, decompose the problem into left and right parts, and finally merge the optimal values of the two parts to obtain the optimal value of the original problem.

For example: Stone merger

There are N piles of stones S[N],every pile has one or more stones.To merge the piles to one pile,you can only merge adjacent piles every time.The cost is the number of stones in the adjacent piles.Find the smallest cost.

Analysis:

1. Finding the state representation

f[i][j] represents the set of all plans of merging piles in interval[i,j].

Attribute is the minimum of cost.

1. Separating the set into multiple subsets and defining state equations.

Separating principle:

The set can be separated into multiple subsets based on the index of merging point(k).

The problem that find the lowest cost to merge piles in interval[i,j] is equivalent to the sum of the lowest cost in left interval [i,k] and the lowest cost in right interval[k+1,j] and add the prefix sum from i to j i.e:

State equation:

f[i][j] = f[i][k] + f[k+1][j] + prefix\_s[j]-prefix\_s[i-1]

K is the merging point that is supposed to be enumerated.

Code:

int minCost(vector<int> S){

Const int N = 1010;

int f[N][N];

vector<int>prefix\_s(S);

for (int i = 1; i <= S.size(); i++)

        prefix\_s[i] += prefix\_s[i-1];

for (int len = 2; len <= S.size(); len++)//enumerate the interval length

        {

for (int i = 1; i + len - 1 <= S.size(); i++)

        {

int j = len + i - 1;

f[i][j] = 1e8;

for (int k = i; k < j; k++)

f[i][j] = min(f[i][j], f[i][k] + f[k+1][j] + prefix\_s[j]-prefix\_s[i-1]);

}

}

return f[1][S.size()];

}

Longest common subsequence:

There are two strings A and B.Find the max length of the common subsequence.

Analysis:

1. Finding the state representation

f[i][j] represents the set of all common subsequences of A[1~i] and B[1~j]

Attribute is the maximum of common subsequence.

1. Separating the set into multiple subsets and defining state equations.

Separating principle:

Based on that the longest common subsequence of A[1~i] and B[1~j] whether incorporate A[i],B[j],there are four subsets:

Both are not incorporated(00).

f[i][j] = f[i-1][j-1];

Only B[j] is incorporated(01).

f[i][j] = f[i-1][j];

According to the definition of set,f[i-1][j] not only cover the situation of only incorporating B[j] but also cover situation of incorporating none of them.Because we aim to find the max length,so f[i-1][j] works despite it generates overlapped calculation.

Only A[i] is incorporated(10)

f[i][j] = f[i][j-1];

According to the definition of set,f[i][j-1] not only cover the situation of only incorporating A[j] but also cover situation of incorporating none of them.Because we aim to find the max length,so f[i][j-1] works despite it generates overlapped calculation.

Both are incorporated(11)

f[i][j] = f[i-1][j-1]+1;

Meaning A[i] = B[j].

Because f[i-1][j-1] is involved by f[i][j] = f[i-1][j-1] or f[i][j] = f[i][j-1].we get the state equation:

f[i][j] = max(f[i-1][j], f[i][j-1]);

f[i][j] = max(f[i][j], f[i-1][j-1]+1); when A[i] = B[j]

Coding:

int maxLength(vector<int> A, vector<int> B){

Const int N = 1010;

A.push\_front(0);

B.push\_front(0);

int f[N][N];

for (int i = 1; i <= A.size(); i++)

        {

for (int j = 1; j <= B.size(); j++)

        {

f[i][j] = max(f[i-1][j], f[i][j-1]);

if (A[i] == B[j]) f[i][j] = max(f[i][j],f[i-1][j-1] + 1);

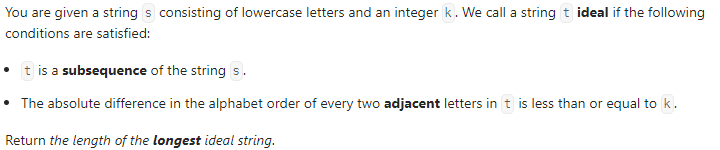
}

}

return f[A.size()][B.size()];

}

Longest Ideal Subsequence:



Analysis:

1. Finding the state representation

dp[i] represents the collection of ideal subsequences ending with character i

Attribute:max length of ideal subsequence

1. Separating the set into multiple subsets and defining state equations.

Separating principle:

There are 2\*k+1 subsets.i.e dp[i-k]+1,...,dp[i]+1,...,dp[i+k]+1.To find the maximum,we get the state equation:

dp[i] = max(dp[i-k]+1,...,dp[i]+1,...,dp[i+k]);

Coding:

int longestIdealString(string s, int k) {

        int dp[150] = {}, ans = 0;

         for (auto &ch : s)

         {

             for (auto i = ch-k; i <= ch + k; i++)

            {

                 dp[ch] = max(dp[ch], dp[i]);

             }

             ans = max(ans, ++dp[ch]);

         }

        return ans;

    }