Data Structure and Algorithm

yan zhao

Contents

Chapter 1

Data structure

Data structure is known as ADT (Abstract Date Type), which is independent of implementation. For example, stack can be implemented by array or linked list inside. Such as `std::stack` in STL library, It's implemented by `std::deque` default.

Linear and hierarchical structures. Array, linked lists, stacks and queue are linear structures. And tree, graph, heaps are hierarchical.

STL is a good reference when you learn data structure, especially (API) design and interface of STL container. such as `push`, `pop` and `top` for `stack` and `priority_queue`. For `queue`, besides `push, pop`, there are also `front()` and `back()`. Just remember five words: **push, pop, front, back and top**.

1.1 Array

1.1.1 Basic operation of array

All languages support array and it is a build-in data type in most of language.

```
//C/C++ language
int a[3]; or int a[4][5].
vector<int> vi(4,0); //STL
array<int, 4> a1 = {1, 2, 3 , 4}; //new array in STL

//C# language
object[] a1 = {"string", 123, true};
```

Array support three basic operations: random access (index), insert, erase.

Dynamic two dimension array (ragged array) can be implemented below: `char ** p;`. Pay attention to `p[1][2]` and `pa[1][2]` in the below code. They are all correct forms.

All ragged array should include terminate flag, so you can know the length of each row. For C-style string ragged array, the 'NULL' in the end of string can be used as terminate flag directly, so ragged array is idea container for C-style string array.

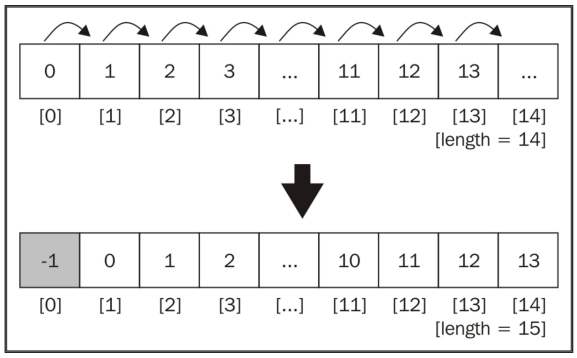
```
char **p;
char* a[]={"zhao","yan","hello"};
p = a;     // use pointer of pointer to catch array of pointer.
printf("%c\n",*(*(p+1)+2)); //p[1][2]

int (*pa)[3];  //pointer to array.
```

```
7  int aa[2][3] = {{1,2,3},{4,5,6}};
8  pa = aa;
9  int bb[][3]; //bb must have 3 as last dimension. c++ syntax.
10 bb = aa;
11 printf("%d\n",pa[1][2]);//*(*(pa+1)+2);
```

line 7-10 this is array of **array**. so you have to use **a pointer to array** to catch it. Pay attention to the syntax of a pointer to array in C++ language: `int (*pointer_to_array)[3];`

If you have an array of big object, and you need to sort it. You can build a array of pointer to point to each object in the array. When you sort the an array of big object, you just move pointer instead, It is called **"indirect addressing"**, It can help you to avoid moving big chunk of data.

If you want to insert an element into an array, you **need to copy backward from end.** if erase an element from an array, **copy forward from erasing point**. Most of time, you can just call insert(pos) or erase(pos) in STL container. **It inserts the element before the pos and erase the element at th pos**, and pos is an iterator.



Reverse array can be implemented by **swap + double pointers**, But reversing list need different algorithm, so in STL , `std::list` has its own reverse member function. Reverse list is a typical interview question, and you should know it very well.

Boundary about array structure.

- Remember: if `n` is size of array, `n/2` is the middle element (odd) or the first element in the second half (even). This pattern is very helpful for you to write reverse function.
- Remember: if `end` is end of index, `end/2` is the middle element (odd) or the last element in the first half(even). Be careful with this, most of time, D&C like to use start and end index, not size of array.
- `std::distance(v.begin(), v.end());` is size of array/vector.

```
1  void reverseArray(int a [], int n){
2    int i,j,temp;
3    for(i=0,j=n-1;i<n/2;++i,--j){
4      swap(a[i], a[j]);
5    }
6  }
7
8  //another is based on iterator
9  auto bi = v.begin();
10 auto ei = v.end();
11 auto mi = next(bi, distance(bi, ei)/2 );
12
13 for(auto i = bi, j = prev(ei); i<mi; ++i, --j){
```

```
14    iter_swap(i,j);
15  }
16
17
18  for(last--; first<last; ++first, --last){
19    iter_swap(first,last);
20  }
21
22  //pay attention next and distance usage, put being in front of end.
23  //distance(v.end(), v.begin()) is just size of array/vector.
```

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.

- Single rotate method

```
1  while(d-->0){ //d-->0 is good trick
2    int temp = Array[0];
3    for (int i = 0; i < n-1; i++){ //left open, right close, when i<n-1, it means
         that we need to access array[i+1]
4      Array[i] = Array[i + 1];
5    }
6    Array[n-1] = temp;
7  }
```

- Reverse method. That is a very efficient implementation.

```
1  rotate(arr[], d, n)
2    reverse(arr[], 1, d) ;
3    reverse(arr[], d + 1, n);
4    reverse(arr[], l, n);
```

Idea: middle part reverse twice, so keep origin value

- For `std::list`, you can use `std::rotate`, but the time complexity is O(n), you can use splice function to reach O(1).

```
1  std::list l{1, 2, 3, 4, 5};
2  auto i2 = std::next(i1, 2);
3
4  l.splice(l.end(), l, l.begin(), i2 ); //output 3, 4, 5, 1, 2
5  //this is a good pattern, need to remember, l.splace(l.end(), ....)
```

- You can use `std::rotate` in insert sort method. It also can be used in slide. Detail can be found in "Top 5 Beautiful C++ std Algorithms Examples"

```
1  for (auto i = start; i != end; ++i)
2    std::rotate(std::upper_bound(start, i, *i), i, std::next(i));
3    //all stl algorithms deal with left close, right open space.
4    //i and std::next(i) is right open point.
5
6  template <typename It>
7  auto slide(It f, It l, randIter p) -> std::pair<It, It>{
8    //  p < [f...l] < p, two possibles
9    if (p < f) return { p, std::rotate(p, f, l) };
10   if (l < p) return { std::rotate(f, l, p), p };
11   return { f, l };
12 }
13
14 ForwardIt rotate( ForwardIt first, ForwardIt n_first, ForwardIt last );
15 // rotate returns Iterator to the new location of the element pointed by first
```

line 2: rotate(a, b, c) is [a,b) and [b,c) scope. they are all left close, right open scope. In this way, you can understand why we need `std::next(i)` here.

- You need to remember below code snippet. **reverse–>rotate–> insertion order and slide**.

We can also use partition to implement gather. **Partition return: Iterator to the first element of the second group.**

Gather all eligible element around iterator p

```
template <typename BiIt, typename UnPred>
auto gather(BiIt f, BiIt l, BiIt p, UnPred s) -> std::pair <BiIt, BiIt>{
  return { stable_partition(f, p, not1(s)),
    stable_partition(p, l, s) };
}
```

1.1.2 Interview questions

- Two sum.
  1. Answer 1: First Sort O(n*log(n)), then use two pointers. **for the two pointers, from two ends to the middle, use while(l<r) is a good strategy.** Two typical applications are reverse and the two sum.

```
bool hasArrayTwoCandidates(int A[], int arr_size, int sum) {
    int l, r;
    sort(A, A + arr_size);

    /* Now look for the two candidates in  the sorted array*/
    l = 0;
    r = arr_size - 1;
    while (l < r) {
        if (A[l] + A[r] == sum)
            return 1;
        else if (A[l] + A[r] < sum)
            l++;
        else // A[i] + A[j] > sum
            r--;
    }
    return 0;
}
```

     Idea: Don't need keep index information, so you can sort it.
  2. Answer 2: Hash O(n)
- Count triplets with sum smaller than a given value.

```
1) Sort the input array in increasing order.
2) Initialize result as 0.
3) Run a loop from i = 0 to n-2.  An iteration of this loop finds all
   triplets with arr[i] as first element.
     a) Initialize other two elements as corner elements of subarray
        arr[i+1..n-1], i.e., j = i+1 and k = n-1
     b) Move j and k toward each other until they meet, i.e., while (j < k)
            (i) if (arr[i] + arr[j] + arr[k] >= sum), then do k--

            // Else for current i and j, there can (k-j) possible third elements
                that satisfy the constraint.
            (ii) Else Do ans += (k - j) followed by j++
```

Idea: Same Idea as previous question, Don't need keep index information, so you can sort it. brutal force, only sort can help here.

- Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time and constant space. Example: I/P = [1, 2, 3, 2, 3, 1, 3] and O/P = 3. The Best Solution is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x.

```
result = result ^ a[i];
```

Idea: Perform some calculation on all elements.

- You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

```
1) Get the sum of numbers
      total = n*(n+1)/2
2  Subtract all the numbers from total and you will get the missing number.
```

Idea: Perform some calculation on all elements.

- Max product of the three numbers for a given array of size N. **Find the three largest numbers in the array (n1, n2, n3) and the two smallest numbers (m1, m2). The answer is either n1 x n2 x n3 or n1 x m1 x m2. The tricky thing here is there are two potential answers.**

```
nth_element(...)
```

Idea: Not sort, but nth max element based on heap

- Given an unsorted array of non-negative integers, find a **continuous** subarray which adds to a given number. Examples: Input: arr[] = 1, 4, 20, 3, 10, 5, sum = 33; Ouptut: Sum found between indexes 2 and 4.

```
int arr[] = {1, 4, 20, 3, 10, 5};
int left = 0; int right = 0;
int sum = 0;
while(right< 6){
  sum+=arr[right];
  ++right;

  while(sum>33){
    sum-=arr[left];
    ++left;
    if (sum == 33){
      cout<<left<<" "<<right<<" "<<endl;
    }
  }
}
```

Idea: 1) You must keep index information, so you can't sort. That is a typical sliding window problems. Keep left and right two index. Sliding window can be found in algorithm chapter.

- Given an array `arr[]` of integers, find out the difference between any two elements such that larger element appears after the smaller number in arr[]. Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then returned value should be 2 (Diff between 7 and 9). That is also a one time stock buy sell problem.

```
In this method, instead of taking difference of the picked element with every
    other element, we take the difference with the minimum element found so far.
    So we need to keep track of 2 things:
```

```
1) Maximum difference found so far (max\_diff).
2) Minimum number visited so far (min\_element).
```

Keep position, but this problem can be divided by sub-problem. Such as, [7, 9, 5, 6, 3, 2] can be seen as [7,9] [5, 6] [3] [2] Once current element is less than min_element, begin a new sub-problem.

- Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum. Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j. Example: The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

```
1  int main(){
2    vector<int> vi = {2, 4, 1, 3, 5};
3    set<int> si;
4    int sum = 0;
5    for(auto i :vi){
6      si.insert(i);
7      cout<<i<<endl;
8      auto it1 = si.upper_bound(i);
9      if (it1 != si.end())
10     cout<<"***"<<*it1<<endl;
11     sum += distance(it1, si.end());
12   }
13
14   cout<<sum<<endl;
15   return 0;
16 }
```

Line 10: set also support iterator, it will output sorted element from beginning to end

Line 12 distance function, you have to input smaller iterator first. for set, if you input end, it will fall into dead loop.

For this questions, there are three points: 1) You need keep index(position) information. (No sort, no calculation). 2) you can't divid it to subj-problem. 3) You have to use auxiliary DS outside. such as set. which store how many element is greeter than current element.

- Convert array into Zig-Zag fashion ? The converted array should be in form a < b > c < d > e < f. Just like bubble sort, use swap to make it satisfy order. Pay attention here, no global order requirement, so just **divide task to atomic operation.**

```
1  void zigZag(int arr[], int n) {
2    // Flag true indicates relation "<" is expected,
3    // else ">" is expected. The first expected relation is "<"
4    bool flag = true;
5    for (int i = 0; i <= n - 2; i++) {
6      if (flag){ /* "<" relation expected */
7        /* If we have a situation like A > B > C,
8        we get A > C < B by swapping B and C */
9        if (arr[i] > arr[i + 1])
10       swap(arr[i], arr[i + 1]);
11     }
12     else{ /* ">" relation expected */
13       /* If we have a situation like A < B < C,
14       we get A < C > B by swapping B and C */
15       if (arr[i] < arr[i + 1])
16       swap(arr[i], arr[i + 1]);
17     }
18     flag = !flag; /* flip flag */
19   }
```

```
}
// use i%2 == 0 to instead flag is also OK here.
```

1.1.2.1 Mono stack

- Write a program to print all the LEADERS in the array. An element is leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. For example int the array 16, 17, 4, 3, 5, 2, leaders are 17, 5 and 2.

```
Scan all the elements from right to left in array and keep track of maximum till
    now. When maximum changes it's value, print it.
```

```
can also use mono stack, from right to left, when stack is empty, it's LEASDERS
    number.
```

- Next Greater Element. Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1. For the input array [4, 5, 2, 25], the next greater elements for each element are as follow.

```
4      -->   5  |   5      -->   25 | 2      -->   25 | 25     -->   -1
```

```
void printNGE(int arr[], int n){
  stack<int> s;
  /* push the first element to stack */
  s.push(arr[0]);

  // iterate for rest of the elements
  for (int i = 1; i < n; i++) {

    if (s.empty()) {
      s.push(arr[i]);
      continue;
    }
    /* if stack is not empty, then pop an element from stack.
    If the popped element is smaller than next, then
    a) print the pair
    b) keep popping while elements are
    smaller and stack is not empty */
    while (s.empty() == false && s.top() < arr[i]) {
      cout << s.top()
      << " ---> " << arr[i] << endl;
      s.pop();
    }
    /* push next to stack so that we can find
    next greater for it */
    s.push(arr[i]);
  }
  /* After iterating over the loop, the remaining
  elements in stack do not have the next greater
  element, so print -1 for them */
  while (s.empty() == false) {
    cout << s.top() << " ---> " << -1 << endl;
    s.pop();
  }
}
```

- One of the main ideas of a monotonic stack is to discard elements that do not meet certain conditions and avoid pushing them onto the stack, as they are irrelevant to the solution.
- Given a string s, remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

  Our algorithm should pops elements from the stack when stk.peek() > c. However, there are two cases to consider:

  1. Case 1: If the character at stk.peek() appears later in the string, it can be safely popped. Since it will reappear later, it can be pushed back onto the stack, satisfying lexicographical order requirements.
  2. Case 2: If the character at stk.peek() does not appear again later, it cannot be popped. As mentioned earlier, the stack will not contain duplicate elements. Popping it means losing this character permanently.

  Returning to the example s = "bcac", when inserting the character 'a', we see that the lexicographical order of 'c' is larger than 'a', and 'c' also appears later in the string. Hence, 'c' is popped from the stack.

  The while loop continues, and we find that the lexicographical order of 'b' is still larger than 'a'. However, since 'b' does not appear again after 'a', it should not be popped. The key question is: how can the algorithm determine how many 'b's and 'c's exist after the current 'a'?

```java
String removeDuplicateLetters(String s) {
  Stack<Character> stk = new Stack<>();

  // Maintain a counter to record the frequency of each character in the string
  // Since the input consists of ASCII characters, size 256 is sufficient
  int[] count = new int[256];
  for (int i = 0; i < s.length(); i++) {
    count[s.charAt(i)]++;
  }

  boolean[] inStack = new boolean[256];
  for (char c : s.toCharArray()) {
    // Decrease the count for each character as it is traversed
    count[c]--;

    // Skip if the character is already in the stack
    if (inStack[c]) continue;

    while (!stk.isEmpty() && stk.peek() > c) {
      // Stop popping if the stack top character will not appear again
      if (count[stk.peek()] == 0) {
        break;
      }
      // If it will appear again, it can be safely popped
      inStack[stk.pop()] = false;
    }
    stk.push(c);
    inStack[c] = true;
  }

  // Build the result string from the stack
  StringBuilder sb = new StringBuilder();
  while (!stk.isEmpty()) {
    sb.append(stk.pop());
  }
  return sb.reverse().toString();
}
```

1.1.2.2 Mono queue

- Given an array and an integer K, find the maximum for each and every contiguous subarray of size k. It looks like two sum problem. there are three different levels time complexity.
  1. The first brute force, O(n*k).
  2. The second, use balance Search Tree(std::set), O(n*log(k)).
  3. the third, use queue O(n)

```
std::deque<int> Qi(k);
/* Process first k (or first window) elements of array */
int i;
for (i = 0; i < k; ++i) {
  // For every element, the previous smaller elements are useless so
  // remove them from Qi
  while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])

  Qi.pop_back();     // Remove from rear
  Qi.push_back(i);     // Add new element at rear of queue
}
// Process rest of the elements, i.e., from arr[k] to arr[n-1]
for (; i < n; ++i) {
  // The element at the front of the queue is the largest element of
  // previous window, so print it
  cout << arr[Qi.front()] << " ";
  // Remove the elements which are out of this window
  while ((!Qi.empty()) && Qi.front() <= i - k)
  Qi.pop_front();   // Remove from front of queue
  // Remove all elements smaller than the currently
  // being added element (remove useless elements)
  while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
  Qi.pop_back();
  // Add current element at the rear of Qi
  Qi.push_back(i);
}
// Print the maximum element of last window
cout << arr[Qi.front()];
```

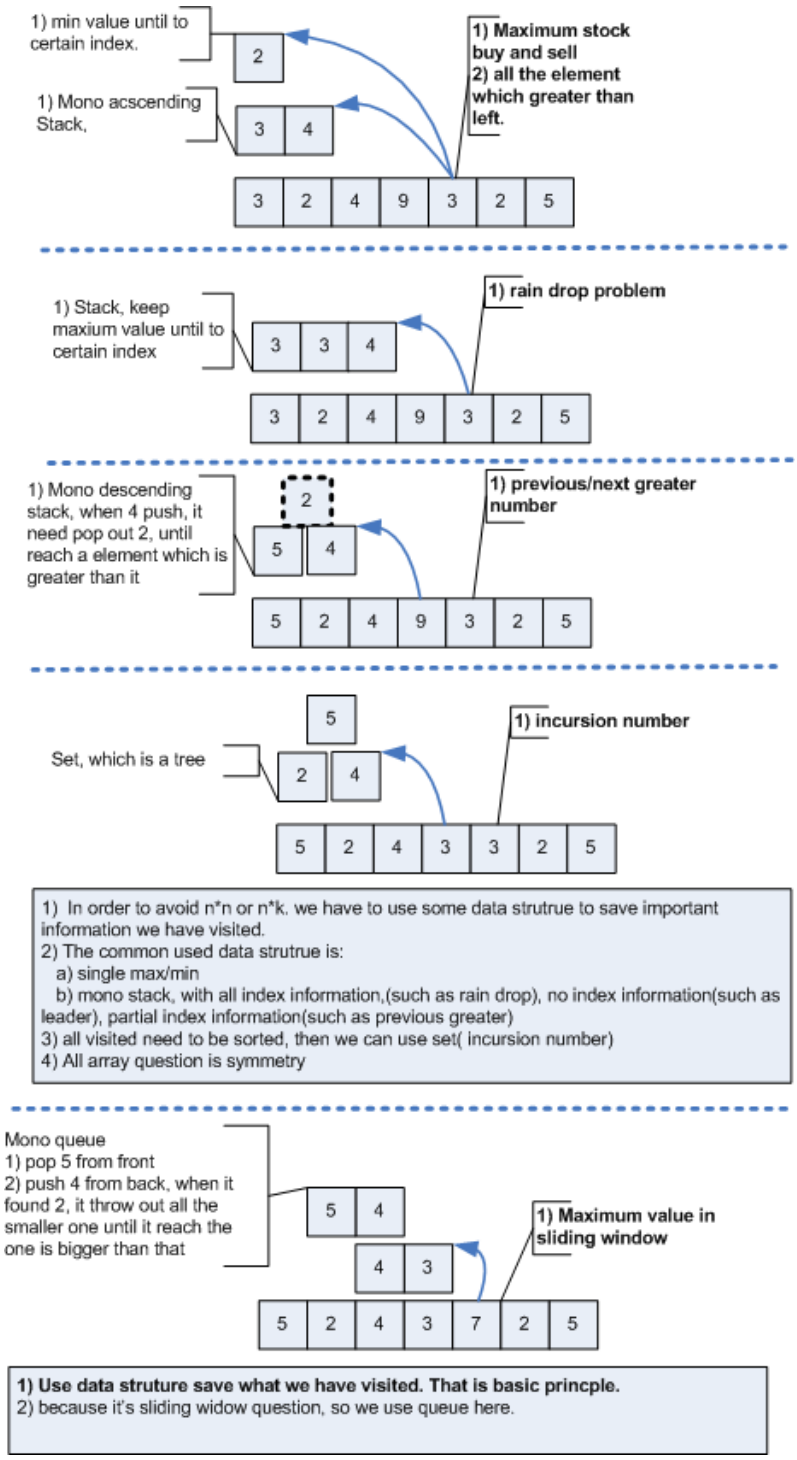
1.1.2.3 Hints for array questions.

There are a few common technology when you work on array related interview questions:

- **Basic rule:** prefer to use STL algorithms and views, such as `views::split`, `views::chunk_by`, `views::slide`. Given any question, First try these existing algorithms and views.
- Sort, when you don't need index information, you can use sort + double pointers to avoid brute force. A common examples are two sum and three sum less than value.
- Get max and min value, based on heap, init a heap need O(n), when you get the second largest or third largest one, you only need O(log(n)). It's better than sort O(n*log(n)).
- Two pointers algorithms
  1. Two-Pointer Technique (from both ends towards the center): This involves starting two pointers from the two ends of a sequence and moving them towards the center. This technique is commonly used in palindrome checking or finding pairs that sum to a specific value. It's also used to implement reverse algorithems.
  2. Fast and Slow Pointer Technique: This uses two pointers moving at different speeds, typically with the fast pointer moving twice as fast as the slow one. partition and remove can be implemented by this method.

3. slide windows: is similar with Fast and slow pointer, but it has some differencens. The sliding window technique is often used for problems related to substring manipulation. Some leetcode example is lettcode 3, 76, 438 and 567.

- Calculation, example is missing one element, single non-duplicate element(XOR).
- index, Make use of index information, an example is leetcode 448: Given an array nums of n integers where nums[i] is in the range [1, n], return an array of all the integers in the range [1, n] that do not appear in nums. Example 1: Input: nums = [4,3,2,7,8,2,3,1]Output: [5,6]. Another famous example to use index information is https://www.geeksforgeeks.org/counting-sort/
- partrial sum: The prefix sum technique, previously detailed, is a widely-used algorithmic approach. It is particularly suitable in scenarios where the original array remains unchanged, allowing for efficient repeated queries of cumulative sums over specific intervals. Another example is leetcode 528.
- adjacent difference: The primary use case for the difference array technique is when you need to frequently increment or decrement elements within a specific range of the original array.
- Mono stack: Leetcode 496, 503 and 739. Next Greater Element.
- Mono queue: Leetcode 239, Sliding window maximum
- Use other auxiliary DS, such as, map, set or hash_map. They can be used as count and sort. how many unorder? 0316.Remove Duplicate Letters, leetcode 1122



1.2 Linked list

1.2.1 Basic

List in STL is bidirectional list(double-linked), in C++11, you can use `std::forward_list` as single-linked list.

```
struct node{    //C implementation
int a;
 struct node* link
} ;

Template <class T> //C++ implementation
Class ChainNode{
   friend Chain<T>;
   T data
  ChainNode<T> *link;
}

template<class T>
class Chain{
  ......
  ChainNode<T> *head;
}
```

1.2.2 Operations

Insert after p node, **get an index to previous pointer**

```
//Consider if it's the first element.
Insert(int k, const T &x)
chainNode<T> *i = new chainNode<T> (x);

//leftside is pointer pointed direction
//rightside is node, but in list, all nodes only can be accessed by a pointer.
//insert after p node
i->link = p->link; //p->link is at right
p->link = i;       //p->link is at left now.
// i is inserted element, p is position.
//You can remember i=p p=i  and fore three links.
```

Insert before p node, what should I do? **Just swap two node, then deal with next node. Don't touch next node in one atomic action. That is rule.**

```
1)create temp auto node, new  i node.

while(p->link)
2) copy value from p to temp node
3) copy value i to p.
4) copy temp value to i value
p = p->link.
```

Delete the node after a given node.

```
Delete(int k, const T&x)
d = p->link;
p->link = d->link ;
delete d;
//d=p p=d and last three links
```

Delete the given node. That is a typical question. you need to remember it.

```
void deleteNode(Node* node_ptr){
  // If the node to be deleted is the last node of linked list
  if (node_ptr->next == NULL){
```

```
    free(node_ptr); // this will simply make the node_ptr NULL.
    return;
  }

  // if node to be deleted is the first or any node in between the linked list.
  Node* temp = node_ptr->next;
  node_ptr->data = temp->data;
  node_ptr->next = temp->next;
  free(temp);
}
```

Source code Psychically, still delete the next node, logically, we use next value to overwrite current node value, so we simulate "delete" in this way. remember, **We don't need while loop to reach the last node in the list**.

Three different operations:

- Normal single link list: insert and delete given a node, always insert after the given node, or erase the next node of the given node.
- Normal single link list: If you want to insert before or delete the given node, You still need to do it physically, but copy value to simulate the "insert" and "delete".
- Double link list, such as `std::list`, insert value before the given node, and delete the given node.

How to rotate a List? This question in interesting, you need to understand splice usage. splice perform 1) move like question, 2) before node of next(i,3) points to null, 3) pointer of end(li) points to i. All these things perform automatically.

```
list<int> li = {1, 2, 3, 4, 5};
auto i = begin(li);
//rotate(i, std::next(i, 3), end(li) ); //must use std::next complexity is O(n).
li.splice(i, li, next(i, 3), end(li) ); //better, use splice, O(1)
```

1.2.3 Interview questions

- Use dumb head for easy deleting node.

```
while (head != NULL && head->val == val) { // pay attention while here,
  ListNode* tmp = head;      //5-->5-->5--> for this condition.
  head = head->next;
  delete tmp;
}

// delete non head
ListNode* cur = head;
while (cur != NULL && cur->next!= NULL) {
  if (cur->next->val == val) {
    ListNode* tmp = cur->next;
    cur->next = cur->next->next;
    delete tmp;
  } else {
    cur = cur->next;
  }
}
return head;
}
```

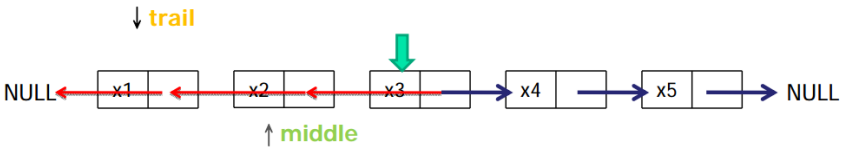
If we build dumb node, code is much easier now

```
ListNode* dummyHead = new ListNode(0); // dumb node
dummyHead->next = head;
// Set the virtual (dummy) head node to point to the head, which makes it easier
    to perform deletion operations later.
ListNode* cur = dummyHead;

while (cur->next != NULL) {
  if(cur->next->val == val) {
    ListNode* tmp = cur->next;
    cur->next = cur->next->next;
    delete tmp;
  } else {
    cur = cur->next;
  }
}
return dummyHead->next;
```

- Print the middle element of link list. When exit while, even node, fast_ptr will be NULL, odd node, fast_ptr->next will be null. Just like n/2, odd number will be in the middle, even number will be the first element in the second half.

```
void printMiddle(struct Node *head) {
  struct Node *slow_ptr = head;
  struct Node *fast_ptr = head;

  if (head!=NULL) {
    while (fast_ptr != NULL && fast_ptr->next != NULL) {  //this statement is very
        important.
      fast_ptr = fast_ptr->next->next;
      slow_ptr = slow_ptr->next;
    }
    printf("The middle element is [%d]\n\n", slow_ptr->data);
  }
}
```

- How to reverse a list **There are two points: 1)while(p), 2) inside loop, do p = p->next, and necessary action(print out value and change pointer direction...)**



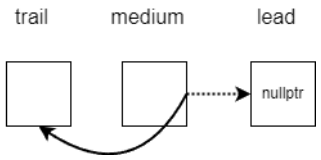

```
middle = NULL;
while (lead) {
  trail = middle;
  middle = lead;
  lead = lead->link;
  middle->link = trail;
}
return middle;
```

First, initial the middle = NULL
And continue loop until lead == NULL

```
node* lead, middle, *tail;
middle = nullptr;
tail = head;
while(tail){
  lead = middle;
  middle = tail;
```

```
7     tail = tail->next;
8     middle->next = lead;
9   }
10  head = middle;
```

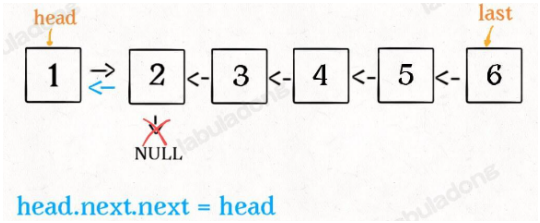
- Three steps to resolve this problem. this pattern can be applied on all link list problem.
  1. The end status is like below figure.
  2. so normal status is also like this(previous atomic operation end statu and next atomic operation start status are all like below), write while main logic.
  3. when you finish while main logic, according to this logic customize your beginning status. in this question, only tail = head and middle = nullptr;



- Reverse linklist

```
1   // Definition: Given the head node of a singly linked list, reverse the list and
        return the new head node.
2   ListNode reverse(ListNode head) {
3     if (head == null || head.next == null) {
4       return head;
5     }
6     ListNode last = reverse(head.next);
7     head.next.next = head;
8     head.next = null;
9     return last;
10  }
11  //1) in reverse, we don't use last node
12  //2) in each reverse, we just take action, don't need return something.
13  //3) just like reverse output a linklist.
14
15
16  //below is my implementation, Pay attention to the ListNode* &, we need to modify
17  //head->next, so we have to return reference. just return pointer doesn't work
        here.
18  ListNode* ghead;
19  ListNode * &  dfs(ListNode* head){
20    if(head == nullptr){
21      ghead = head;
22      return ghead;
23    }
24    if(head->next == nullptr){
25      ghead = head;
26      return ghead->next;
27    }
28    ListNode* & p = dfs(head->next);
29    cout<<head->val<<endl;
30    p=head;
31    head->next = nullptr;
32    return head->next;
33  }
34
35  ListNode* reverseList(ListNode* head) {
36    dfs(head);
37    return ghead;
```

```
}
```



- Reverse the first k element

```
ListNode successor = null; // Successor node
// Reverse the first n nodes starting from head and return the new head node
ListNode reverseN(ListNode head, int n) {
  if (n == 1) { // Record the (n + 1)th node
    successor = head.next;
    return head;
  }
  // Starting from head.next, reverse the first n - 1 nodes
  ListNode last = reverseN(head.next, n - 1);

  head.next.next = head;
  // Connect the reversed head node to the successor node
  head.next = successor;
  return last;
}
```

- Reverse m to n

```
ListNode reverseBetween(ListNode head, int m, int n) {
  // base case
  if (m == 1) {
    return reverseN(head, n);
  }

  // advance to revert beginning point to trigger base case.
  head.next = reverseBetween(head.next, m - 1, n - 1);
  return head;
}
```

- Per k group to reverse

```
/** Reverse elements in the interval [a, b), note that it's left-closed and right-
    open */
ListNode reverse(ListNode a, ListNode b) {
  ListNode pre = null, cur = a, nxt = a;
  // Just change the while loop's termination condition
  while (cur != b) {
    nxt = cur.next;
    cur.next = pre;
    pre = cur;
    cur = nxt;
  }
  // Return the new head node after reversal
  return pre;
}

ListNode reverseKGroup(ListNode head, int k) {
  if (head == null) return null;
```

```
  // Interval [a, b) contains k elements to be reversed
  ListNode a = head, b = head;
  for (int i = 0; i < k; i++) {
    // If fewer than k nodes, no need to reverse; base case
    if (b == null) return head;
    b = b.next;
  }
  // Reverse the first k elements
  ListNode newHead = reverse(a, b);
  // Recursively reverse the remaining list and connect them
  a.next = reverseKGroup(b, k);
  return newHead;
}
```

- Given a singly linked list, determine if its a palindrome. This method takes O(n) time and O(1) extra space.
  1. Get the middle of the linked list.
  2. Reverse the second half of the linked list.
  3. Check if the first half and second half are identical.
  4. **Idea: 1) find the middle 2)reverse**. If it's odd number, you need to move the middle pointer to next position.
- Swap nodes in a linked list without swapping data. The idea it to first search x and y in given linked list. If any of them is not present, then return. While searching for x and y, keep track of current and previous pointers.

```
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
     If the length is odd, the extra node should go in the front list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL){
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else{
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL){
      fast = fast->next;
      if (fast != NULL){
        slow = slow->next;
        fast = fast->next;
      }
    }

  /*'slow' is before the midpoint in the list, so split it in two at that point.*/
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}
```

- Prints the given Linked List in reverse manner. (resurisve)

```
void fun1(struct node* head){
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

- Write a program function to detect loop in a linked list

```
This is the fastest method. Traverse linked list using two pointers. Move one
   pointer by one and other pointer by two. If these pointers meet at some node
   then there is a loop. If pointers do not meet then linked list doesn't have
   loop.
```

a circle has at last three nodes, so fast pointer each time step 2, So it will always go around inside the circle

Summary of link list:

- **Just like tree, most link-list problems can be resolved by recursive**.
- Most of time, In single link list, you need to keep previous node and current node.

	array	list
insert erase	O(n)	O(1)
sort	quicksort	mergesort
reverse	swap element	just manipulate pointer
rotate	easy pointer manipulate	swap element
merge		
other		

`std::list` has its own sort member function, but `std::vector` doesn't have. you can use std::sort algorithme. `std::list` has its own reverse, std::vector doesn't have, just use std::reverse algorithm. std::list and vector don't have their own rotate function. `std::list` has its own merge, vector use std::merge algorithm.

1.3 Matrix

1.3.1 Sparse matrices

Sparse matrix can implemented by on dimension array or linked list.

```
template<typename T>
class term{
  int row, int col;
  T value;
}

term<T> *sparseMatrix;  // using one linked list
```

Define sparse matrix by using std::list.

```
using Row = list< pair<int, float>> ;
using Matrix = list< int, Row>;

list< pair<int, float>> row0;
```

```
row0.push_back({0, 1.0});
row0.push_back({3, 3.0});

Matrix matrix;
matrix.push_back({0, row0});
```

You can build a matrix class. use m(2,3) to access a element. And support +, - * . You also can define index begin from 1. It will more reasonable for you when you use a matrix.

```
template<typename T>
class Matrix{
  T& operator()(int row, int col){
    ..
  }
}
```

Some special square matrices

- Diagonal $i \neq j, M(i,j) = 0$ Can use one dimension array to describe it.
- Low triangular $i < j, M(i,j) = 0$ We can use one dimension array to implement lower triangular because it will save many spaces. Index can be calculated by $i * (i-1)/2 + j - 1$, and save it into a one dimension array.

```
[a11  0   0   0 ]
[a21 a22  0   0 ]
[a31 a32 a33  0 ]
[a41 a42 a43 a44]

a11: (1 - 1) * (1) / 2 + 1 - 1 = 0
a21: (2 - 1) * (2) / 2 + 1 - 1 = 1
a22: (2 - 1) * (2) / 2 + 2 - 1 = 2
a31: (3 - 1) * (3) / 2 + 1 - 1 = 3
a32: (3 - 1) * (3) / 2 + 2 - 1 = 4
.....
```

- Symmetric. Just like a low triangular matrix, can be described by one dimension array.

1.3.2 Matrices traversal methods and operation

- Row and column traversal, just use two for statement. Pay attention to the cache missing problem when you perform column traversal.
- For square matrix (n*n), you can perform in place transpose. There are two points:
  1. Swap `i` and `j`.
  2. If botton-left to top-right, then use `n-1` to substract.

```
 for (int i = 0; i < n; i++) {
  for (int j = i; j < n; j++) {
     swap(matrix[i][j], matrix[j][i]);
  }
}

// Mirror-symmetric 2D matrix along the bottom-left to top-right diagonal.
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n - i; j++) {
     swap(matrix[i][j], matrix[n-1-j][n-1-i]);
  }
}
```

- Rotate clockwise or counter clockwise matrix, first transpose, then reverse each row, just remember this trick.
- Spiral print a matrix. Four boundary, then shrink step by step.

```
int m = matrix.length, n = matrix[0].length;
int upper_bound = 0, lower_bound = m - 1;
int left_bound = 0, right_bound = n - 1;
List<Integer> res = new LinkedList<>();
// When res.size() == m * n, the entire array has been traversed
while (res.size() < m * n) {
  if (upper_bound <= lower_bound) {
    // Traverse from left to right on the top
    for (int j = left_bound; j <= right_bound; j++) {
      res.add(matrix[upper_bound][j]);
    }
    upper_bound++; // Move the upper boundary down
  }

  if (left_bound <= right_bound) {
    // Traverse from top to bottom on the right side
    for (int i = upper_bound; i <= lower_bound; i++) {
      res.add(matrix[i][right_bound]);
    }
    right_bound--; // Move the right boundary left
  }

  if (upper_bound <= lower_bound) {
    // Traverse from right to left on the bottom
    for (int j = right_bound; j >= left_bound; j--) {
      res.add(matrix[lower_bound][j]);
    }
    lower_bound--; // Move the lower boundary up
  }

  if (left_bound <= right_bound) {
    // Traverse from bottom to top on the left side
    for (int i = lower_bound; i >= upper_bound; i--) {
      res.add(matrix[i][left_bound]);
    }
    left_bound++;  // Move the left boundary right
  }
}
```

- DFS, recursive and iterator implementation.

```
//leetcode 200, Main function to count the number of islands
int numIslands(char[][] grid) {
  int res = 0;
  int m = grid.length, n = grid[0].length;
  for (int i = 0; i < m; i++) { // Traverse the whole grid
    for (int j = 0; j < n; j++) {
      if (grid[i][j] == '1') {
        // Increase the island count when a new island is found.
        res++;
        dfs(grid, i, j); // Then use DFS to sink the island.
      }
    }
  }
  return res;
}

```

```
// Starting from (i, j), turn all adjacent land into water
void dfs(char[][] grid, int i, int j) {
  int m = grid.length, n = grid[0].length;
  if (i < 0 || j < 0 || i >= m || j >= n) {
    return; // Out of index bounds
  }
  if (grid[i][j] == '0') {
    return; // Already water
  }

  grid[i][j] = '0'; // Turn (i, j) into water
  // Sink the land above, below, left, and right
  dfs(grid, i + 1, j);
  dfs(grid, i, j + 1);
  dfs(grid, i - 1, j);
  dfs(grid, i, j - 1);
}
```

There are a few points you need to pay attention to when you implements dfs by iteration:

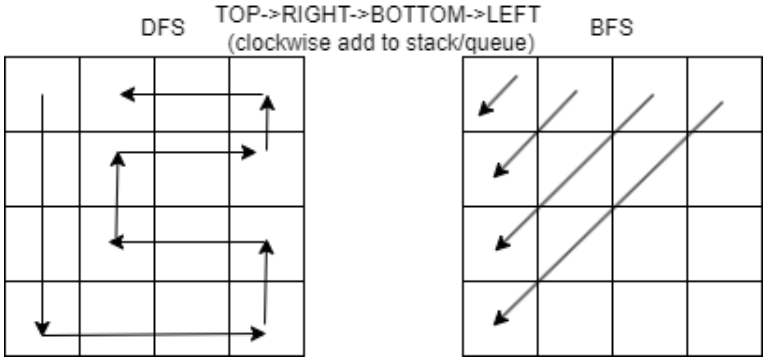
1. top and pop should stay together;
2. use pair to remember position,
3. use s!=empty();
4. check four boundary,
5. when pop, check if we visited, when push, check if we visited. (two positions checks)

```
void dfs(vector<vector<char>>& grid, int i, int j){
  int row = grid.size();
  int col = grid[0].size();
  if(grid[i][j] == '0'){
    return ;
  }

  stack<pair<int, int> > s;
  s.push({i,j});
  while(!s.empty()){
    auto [i,j] = s.top(); //top and pop should be stay together
    s.pop();
    if(grid[i][j] == '1'){
      grid[i][j] = '0';
      if(i>0 && grid[i-1][j] == '1'){
        s.push({i-1, j});
      }
      if(j<col-1 && grid[i][j+1] == '1'){
        s.push({i, j+1});
      }
      if(i<row-1 && grid[i+1][j] == '1'){
        s.push({i+1, j});
      }
      if(j>0 && grid[i][j-1] == '1'){
        s.push({i, j-1});
      }
    }
  }
}
```

- BFS. A few difference with DFS:
  1. No need to check when pop, because when we push, we have visited, all the node in queue are visited.
  2. Change stack to queue, change top to front.

3. When add the first element, mark it as visited.

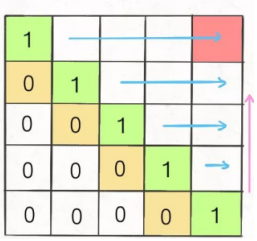
```
void BFS(vector<vector<char>>& grid, int i, int j){
  int row = grid.size();
  int col = grid[0].size();
  if(grid[i][j] == '0'){
    return ;
  }

  queue<pair<int, int> > q;
  q.push({i,j});
  grid[i][i] = '0'; //the first element also need to marked as visited.
  while(!q.empty()){
    auto [i,j] = q.front();
    q.pop();
    cout<<i<<" "<<j<<endl;
    if(i>0 && grid[i-1][j] == '1'){
      grid[i-1][j] = '0';
      q.push({i-1, j});
    }
    if(j<col-1 && grid[i][j+1] == '1'){
      grid[i][j+1] = '0';
      q.push({i, j+1});
    }
    if(i<row-1 && grid[i+1][j] == '1'){
      grid[i+1][j] = '0';
      q.push({i+1, j});
    }
    if(j>0 && grid[i][j-1] == '1'){
      grid[i][j-1] = '0';
      q.push({i, j-1});
    }
  }
}
```



About the traversing direction

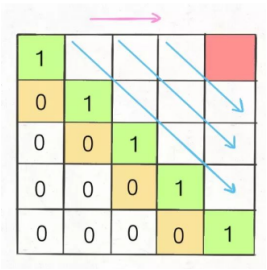
- Reverse traverse, from bottom to up, from left to right.

```
for (int i = n - 1; i >= 0; i--) {
  for (int j = i + 1; j < n; j++) {
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);  // current after left and right
        cell
```



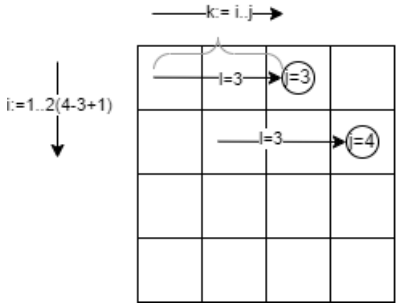
- Diagonal traverse

```
for (int l = 2; l <= n; l++) { //l is length.
  for (int i = 0; i <= n - l; i++) {
    int j = l + i - 1;
    // dp[i][j]
  }
}
       i:= 0 . . . . . n
l:=2   (0,1) (1,2) (2,3)
l:=3   (0,2) (1,3) (2,4)
l:=4   . . . . . .
```



- The most implicated traversal, dp to calculate matrix multiplication. How to a kind of diagonal traversal matrix? this method is useful for matrix multiplication DP problem.

```
for(L = 2;L<n;L++)
  for(i = 1;i<n-L+1;++i)   //n-L+1  remember n-L+1 and i+L-1.
    j = i+L-1              //i+L-1
    for (k = i; k <= j - 1; k++){
      ...
```



Fill and color difference. if you want to color fill, need to check if origin color and new color are the same, otherwise, it will lead dead loop.

matrix and spiral print all indicate a good hint:

- identify a sub task(function).

- then design this function API (what information this function need to finish it's task.)
- Once we have correct API, then loop call this function with correct parameter, this step will be easy now.

1.3.3 Interview questions

- leetcode 48. You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise). You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.
- Two matrix multiply. `sum += v1[row][i]*v2[i][col];` is the key points to understand the whole problem.

```
int inner_product(int row, int col, int num, const vector<vector<int>> &v1,
const vector<vector<int>> &v2){
  int sum = 0;
  for (int i =0 ;i<num ;i++){
    sum += v1[row][i]*v2[i][col];
  }
  return sum;
}
vector<vector<int> > m1 = {{1,2,3}, {4, 5, 6}}; //list initialization
vector<vector<int> >m2 = { {1,2}, {3,4}, {5,6}};

int m1_row = m1.size(); int m1_col = m1[0].size();
int m2_row = m2.size(); int m2_col = m2[0].size();
vector<vector<int> > result = {{0,0}, {0,0}};
for(int i = 0;i<m1_row;i++){
  for(int j= 0;j<m2_col;j++){
    int element = inner_product(i, j,3, m1, m2); //call function here.
    result[i][j] = element;
  }
}
```

Source code In this questions, inner_product is key point to resolve this problem. First, get atomic operation and change it into a function. Then, design this function parameter and in the end, design outside environment to call this automic function. This is very an important strategy.

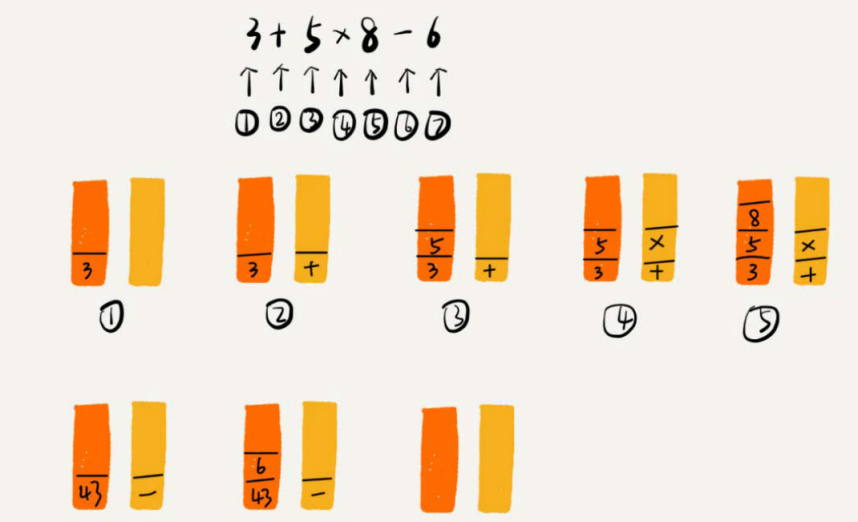
1.4 Stack

In C++ STL , `std::stack` is container adaptor, specifically designed to operate in a LIFO context. Stack is last-in first-out, and queue is frist-in and first-out

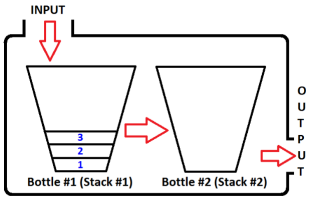
1.4.1 Application of Stack

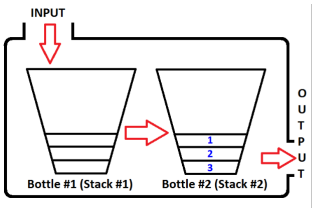
The most famous application about stack is **Recursive**. **The most famous applications based on stack is: Maze (depth first search). Hanoi tower (Recursive). brace Match.**

Some typical two stacks examples. One is expressions evaluation. we can build two stacks, one is for operands and the other for operators. When the operators precedence is higher than the stack top element, push the operators into the stack, otherwise, pop the operator and get two operatands from operands stacks, get the result and push back to the operands stack.



Two stacks trick can also be used in browsers back and forward function. Another typical two stack usage are use two stacks to implement queue.





1.4.1.1 Offline Equivalence class

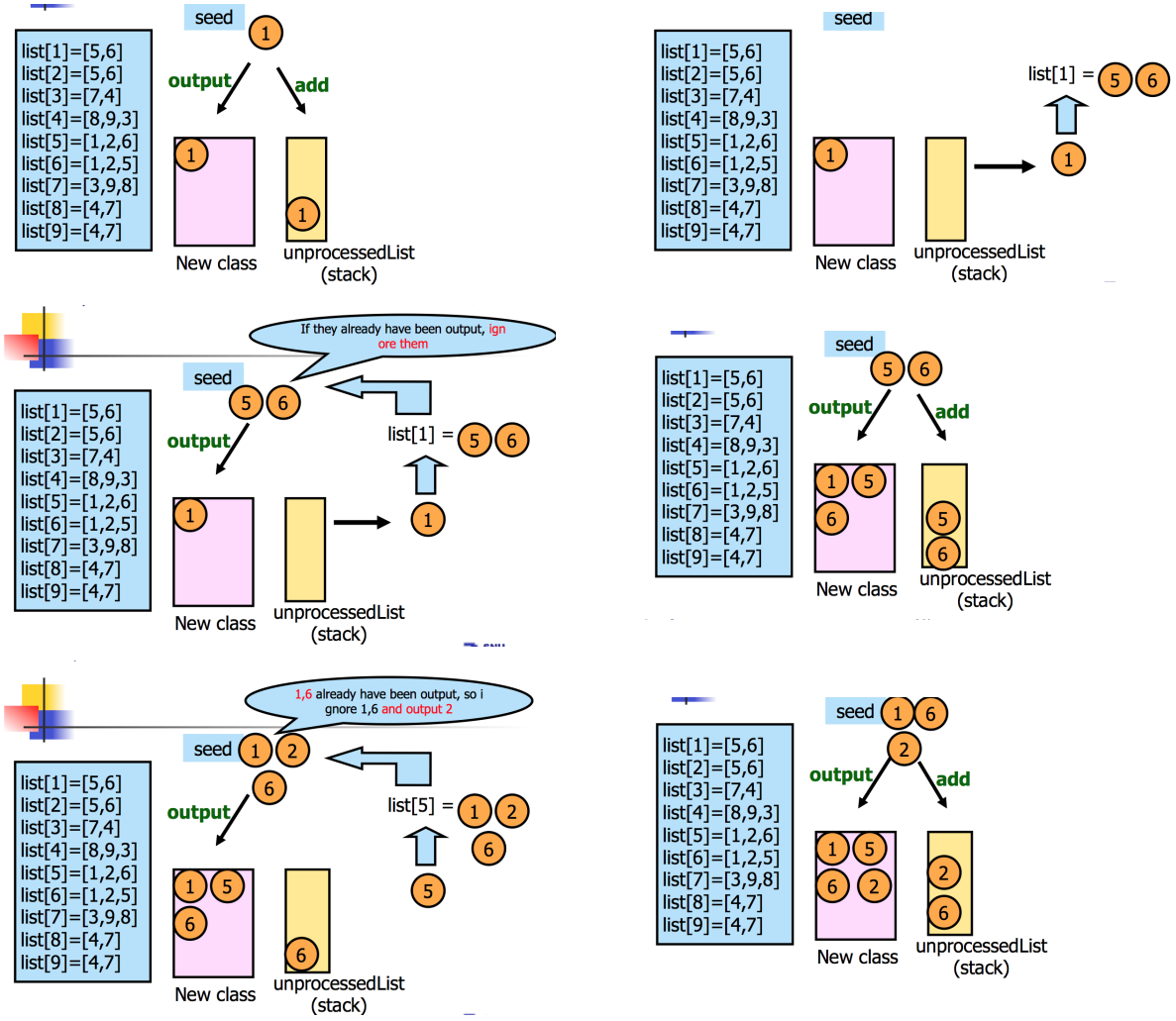
Offline equivalence class, known n and R, get all Equivalence class.

n = 9, r = 11 and relation pairs are: (1,5), (1,6),(3,7),(1,5), (4,8),(5,2) (6,5), (4,9),(9,7),(7,8), (3,4),(6,2)

One word, **from relation pairs build a graphs(adjacency list), Then use stack to perform DFS search, use equivalence class record if we have visited.**

Phase 1: Equivalence pairs (i, j) are read in and adjacency (linked) list of each object is built. Phase 2: Trace (output) the equivalence class containing object i with stack (depth-first search). Next find another object not yet output, and repeat.

```
for (int i = 1; i <= n; i++)  // output equivalence classes
if  (!out[i]) { // start of a new class
  out[i] = true;
  unprocessedList.push(new Integer(i)) ;
  while (!unprocessedList.empty()) {
    // get rest of class from unprocessedList
    int j = ((Integer) unprocessedList.pop()).intValue();
    while  (!list[j].empty()) {
      // elements on list[j] are in the same class
      int q = ((Integer) list[j].pop()).intValue();
      if (!out[q]) { // q not yet output
        System.out.print(q + " ");
        out[q] = true;
        unprocessedList.push(new Integer(q));  }
    } //end of while
  }  //end of while
} //end of if
```



1.4.2 Interview questions

- MinStack, all operation are constant time. Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack. All these operations of SpecialStack must have a time and space complexity of O(1). Note: To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, lists, etc

```
class MinStack {
public:
  void push(int val) {
    if (stack.empty()) {
      stack.push_back({val, val});
    } else {
      stack.push_back({val, min(stack.back().second, val)});
    }
  }
  void pop() {
    stack.pop_back();
  }
  int top() {
    return stack.back().first;
  }
  int getMin() {
    return stack.back().second;
```

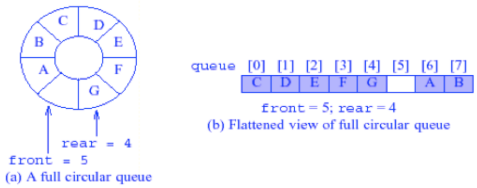
```
18    }
19  private:
20    // first = val, second = min as of that insert
21    vector<pair<int, int>> stack;
22  };
```

1.5 Queue

1.5.1 Circule Queue

You can use three different way to implement queue, array, linked list and circular array.

Circular array representation of a queue.



- Initial condition : front = rear = 0
- front is for pop, rear is for push (STL use back), so you can remember it easily.
- use rear+1%length and front+1%length to simulate rear++ and front++
- rear+1%length == front means full; front==rear is empty.
- **front point to an empty cell. So at most it can save length-1 elements.** Otherwise, we can't distinguish empty and full.
- how to understand (rear+1)%length. an array with 10 element, length is 10, the last index is 9. (9+1)%10 = 0, it help use change 9 to 0(from end array to begin array)

```
1  void push(T t){
2    if ((rear+1)%length == front){
3      throw "full" exception
4    }
5    rear = (rear+1)%length; //use (rear+1)%length simulate ++rear
6    array[rear] = t;
7  }
```

- Pop only increase front, so if front == rear, means front catch up rear, it means empty.

```
1  T pop(){
2    if (front == rear){
3      throw "empty" exption
4    }
5    front = (front+1)%length;
6    return array[front];
7  }
```

1.5.2 Application of queue

For BFS, path is not save in stack. So you need a new kind of data type.

```
struct position {
        int x;
        int y;
        struct position* parent;
}
```

Just like DFS, before you add a position to queue, mark it first, and also marked it as visited.

1.6 Heap

1.6.1 Basic

A heap only supports three operations: initialization, insertion, and deletion of the maximum (or minimum) value. For insertion, the new element is placed at the last position and then repeatedly swapped with its parent to satisfy the heap condition. For deletion, the maximum (or minimum) value is removed, and the value from the last position is moved to the root. Then, it is compared with the left and right child nodes to decide which child to swap with in order to satisfy the heap condition. An important point during programming is the relationship between parent and child nodes, which is as follows:

```
Root is at index 0 in array. Relationship between child and parent are:
Left child of i-th node is at (2*i + 1)th index.
Right child of i-th node is at (2*i + 2)th index.
Parent of i-th node is at (i-1)/2 index.
```

Max(min) tree is each node is greater or equal to its children node. Heap just guarantees that elements on higher levels are greater (for max-heap) or smaller (for min-heap) than elements on lower levels, whereas BST guarantees order (from "left" to "right"). If you want sorted elements, go with BST.

Heap is **1)complete 2) binary 3)Max(min) tree.** Because Heap is complete binary tree. We can use an array to store it.

Insert is easier than you think, When you insert, put it on the end position, then exchange with parent.

```
insert(x){
  int c = ++currentSize;
  while(c!=1 && x>heap[c/2]){ //get its parent node and compare
    heap[c] = heap[c/2];  //move down parent
    // move parent  down because it smaller
    c/=2;  //move up a level
  }
  heap[c] = x;  //the right position of insert.
  //at this time heap[c] has been moved away by previous
  //statement heap[c] = heap[c/2]
}
 //c is used to control level,  c/=2 move up level
 //heap[c/2] get its parent
```

Heap just support delete Max(Min) value. When delete, delete the first element, then put the end position element to the first position, then exchange it with either left child or right child.

```
T y = heap[CurrentSize--];
int i = 1, ci = 2;
//because ci will increase to get bigger one from two children.
//so I have to use another i to keep current level
```

```
while(ci<=CurrentSize){
  if(heap[ci]<heap[ci+1]) ci++; // get the bigger child

  if(y>=heap[ci]{
    heap[i] = y;   // i always empty for a new value.
    break;
  }
   //i is current node
   //ci  is i biggest child node
  heap[i] = heap[ci]; //move up child, insert is "move down parent"
  i = ci;  //move to next level
  ci *=2;
}
```

Initialize a Heap: **For an array, from the middle point, loop backward until reach to beginning(0 index)**. the time complexity is O(n).

```
void heapify(int arr[], int n, int i){
  int largest = i; // Initialize largest as root
  int l = 2 * i + 1; // left = 2*i + 1
  int r = 2 * i + 2; // right = 2*i + 2

  if (l < n && arr[l] > arr[largest]) // If left child is larger than root
  largest = l;

  // If right child is larger than largest so far
  if (r < n && arr[r] > arr[largest])
  largest = r;

  // If largest is not root
  if (largest != i) {
    swap(arr[i], arr[largest]);
    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
  }
}

// Function to build a Max-Heap from the given array
void buildHeap(int arr[], int n){
  int startIdx = (n / 2) - 1; // Index of last non-leaf node

  // Perform reverse level order traversal from last non-leaf node and heapify each
      node
  for (int i = startIdx; i >= 0; i--) {
    heapify(arr, n, i);
  }
}
```

In STL, heap is called priority_queue, Default is maximum heap, if you want to build minimum heap, use std::greater<int> below, it's a little against intuitive.

```
//default compare function is less<T>
std::priority_queue<int, std::vector<int>, std::greater<int> > q2;
for(int n : {1,8,5,6,3,4,0,9,7,2})
q2.push(n);
// 0, 1, 2 ,3 ,4, 5...
```

STL support a basic heap operation. just remember three functions: std::make_heap, std::push_heap and std::pop_heap.

```
vector<int> vi = {2, 4, 1, 3, 5,6,7};
cout<<"ddd"<<distance(end(vi),begin(vi))<<endl;
make_heap(begin(vi),end(vi)); // inpace modify

//after you pop_heap, you need to use v.pop_back()
//to make begin(vi), end(vi) is still heap.
v.pop_heap(begin(vi),end(vi));
v.pop_back();

//if you want to use push_heap, you have to
//call push_back first.
v.push_back(10);
v.push_heap(v.begin(vi) ,end(vi));

//still inpace modify
sort_heap(v.begin(vi) ,end(vi));
```

source code push_back and push_heap must appear together. At the same time, **pop_heap and pop_back must appear together.**

Why heap sort has nlog(n)? https://stackoverflow.com/questions/54078858/why-is-the-time-complexity-of-heap-sort-onlogn

1.6.2 Application

Huffman tree use external nodes to represent a character, It assure no prefix is repeated. You can use minHeap to build huffman tree. In the minHeap, each element is child tree, and then pop twice, combine two child trees. then insert it back to minHeap. Until there is only one element in the Heap.

```
struct MinHeapNode {  // A Huffman tree node
  char data;  // One of the input characters
  unsigned freq;  // Frequency of the character
  struct MinHeapNode *left, *right; // Left and right child of this node
};

// Prints huffman codes from the root of Huffman Tree.
// It uses arr[] to store codes
void printCodes(struct MinHeapNode* root, int arr[],
int top) {
  if (root->left) {  // Assign 0 to left edge and recur
    arr[top] = 0;
    printCodes(root->left, arr, top + 1);
  }

  if (root->right) {  // Assign 1 to right edge and recur
    arr[top] = 1;
    printCodes(root->right, arr, top + 1);
  }

  // If this is a leaf node, then it contains one of the input
  // characters, print the character and its code from arr[]
  if (isLeaf(root)) {
    printf("%c: ", root->data);
    printArr(arr, top);
  }
}
```

Maximum k item in data steam. **use K minimum heap**. If you want Minimum k items (less than), use k maximum heap. A little anti intuitive, just remember it.

median number in data steam. **use one minimum heap and one maximum heap, and keep these two heaps has at least 1 number difference size.** What's difference with previous Maximum k item in data stream? median number or 10% top, answer depends on the whole size, so we need to keep all the data in the stream, Maximum k element does **NOT** depends on the size of data stream.

Another usage of heap is to use k-merge, you need to remember it.

1.7 Tree

1.7.1 Basic knowledge

1.7.1.1 Basic conception

List has a first node, every tree has a root node. Root node is a key component when you deal with most of tree problems. Pre-order, In-order and Post-order are based on middle node. **and they are all dfs algorithm.**

complete, full, perfect tree are different. Full Binary can be skewed shape.

- Full Binary Tree: A Binary Tree is full if every node has 0 or 2 children.
- Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible. (complete tree is heap, so complete tree is very important conception.)
- Perfect Binary Tree: A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

```
       18  //full binary tree
      /    \
    15      20
   /  \
  40   50
 /  \
30  50
```

```
        18 //complete (heap is complete tree)
       /        \
      15        30
     /  \      /  \
    40    50  100   40
   /  \   /
  8   7  9
```

```
      18  //perfect binary tree.

    /      \
  15        30
 /  \      /  \
40  50   100  40
```

Heap is CBT(complete binary tree), not perfect binary tree.

Balance search tree includes AVL and RB tree, in order to keep it balance, you need to rotate child tree.

Tournament Tree: Tournament tree is a complete binary tree with some properties. Remove and Replay of Tournament tree gives you "sorting".

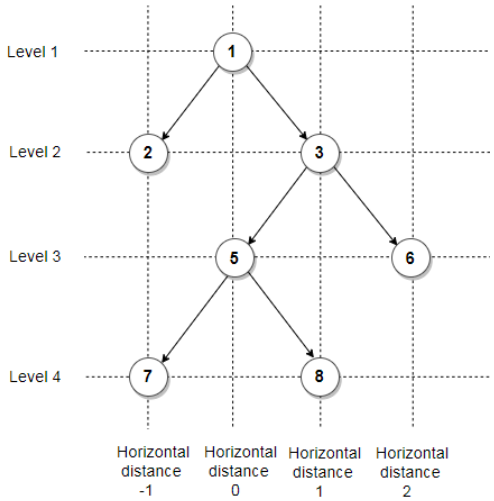
Binary Search Tree: BST is a binary tree with some properties (not necessarily CBT). In-order traversal of BST gives you "sorting" Can be skewed and unbalanced, so we need Balanced Tree.

1.7.1.2 Other property of tree

Horizontal, and diagonal traversal. horizontal distance is distance between root node. When you want to have bottom view or top view, you can use it. detail can be found here:
`https://www.techiedelight.com/print-bottom-view-of-binary-tree/`

```
map<int, pair<int, int>> map;
printBottom(root, 0, 0, map);
//------------------------------------------
void printBottom(Node* node, int dist, int level, auto &map){
  if (node == nullptr) { // base case: empty tree
    return;
  }

  //if the current level is more than or equal to the maximum level seen so far
  //for the same horizontal distance or horizontal distance is seen for the first time,
       update the map

  if (level >= map[dist].second){
    // update value and level for the current distance
    map[dist] = { node->key, level };
  }

// recur for the left subtree by decreasing horizontal distance and increasing level by
     1
  printBottom(node->left, dist - 1, level + 1, map);

//recur for the right subtree by increasing both level and horizontal distance by 1
  printBottom(node->right, dist + 1, level + 1, map);
}
```

Source code Distance and level should be stack type value. Map is reference can shared by all recursive functions. And this is very important data structure in this question.

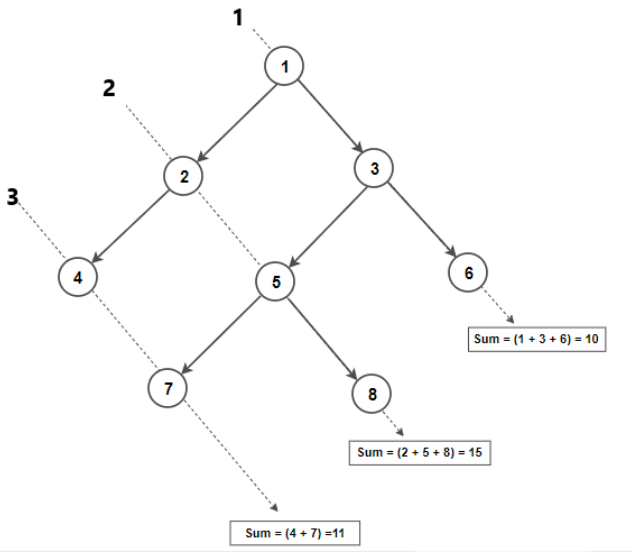


Diagonal, Given a binary tree, calculate the sum of all nodes for each diagonal having negative slope Àssume that the left and right child of a node makes a 45–degree angle with the parent.
`https://www.techiedelight.com/find-diagonal-sum-given-binary-tree/`

```
// Recursive function to perform preorder traversal on the tree and
// fill the map with the diagonal sum of elements
```

```
3  unordered_map<int, int> map;
4  void diagonalSum(Node* root, int diagonal, auto &map)
5  {
6    // base case: empty tree
7    if (root == nullptr) {
8      return;
9    }
10
11   // update the current diagonal with the node's value
12   map[diagonal] += root->data;
13
14   // recur for the left subtree by increasing diagonal by 1
15   diagonalSum(root->left, diagonal + 1, map);
16
17   // recur for the right subtree with the same diagonal
18   diagonalSum(root->right, diagonal, map);
19 }
```

line 15 diagonal is just add 1 when you go to the left tree. Don't change when you go right tree.
line 4 when you use sum of the whole tree, you must pass a reference type `auto& map` here.



1.7.2 Search Tree

Search tree is the key in each node must be **greater than all keys stored in the left sub-tree, and smaller than all keys in the right sub-tree.**

Difference between BST and hash

- So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables? what are advantages? Following are some important points in favor of BSTs.
- We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
- Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.
- BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

- With BSTs, all operations are guaranteed to work in O(Log(n)) time. But with Hashing, O(1) is average time and some particular operations may be costly, especially when table resizing happens.

ascending or descending order of BST. Pay attention here, they are all in order traversal.

```
void ascending(BST* root){
    if(root == NULL) return;
    ascending(root->left);
    std::cout<<root->data<<" ";
    ascending(root->right);
}

void descending(BST* root){
    if(root == NULL) return;
    descending(root->right);
    std::cout<<root->data<<" ";
    descending(root->left);
}
```

1.7.3 Tree and recursion

1.7.3.1 Base case

Nearly all the tree problems can be resolved by recursive. **Only focus on the current root logic.** For example, kth element problem, or lowest common ancester problem. The basic patter is below:

```
TreeFun(root*...){
  BaseCase;
  TreeFun(root->Left...);
  TreeFun(root->right...);
  Join logic with current node according to different problem.
}
```

Global variables, reference variables, local variables, and return variables: differences in their usage within tree recursion programs

95% Base Case is very simple.**Forgetting to check if the root is null. It's an important base case. In nearly all interview questions, the first statement inside recursive should be check root, then return nullptr, return 0 or return nothing.**

```
if(root == nullptr){
  return ...
}
```

The easier part is recursive all with left child and right child. At this time, you should design the recursive interface according to specific question. For example, If it is required from you to write a function that returns a value (e.g. the number of nodes in a binary tree), you have to make sure that the function actually "returns". One of the common mistakes is just writing the recursive call without writing the word "return" before it.

The most difficult part is **Join**, You should write some logic here according to specific problem. For example, for the height of tree, we need to get max value of the left and right tree and increment it.

```
int height(Node t){
  if(t == null)  {
```

```
3       return -1;
4     }
5     else
6     {
7       int left = height(t->left);
8       int right = height(t->right);
9       //below are joint logic combine with current node
10      int height = 1 + max( left , right );
11      //put left and right recursive into assignment
12      return height;
13    }
14  }
```

There are two problems involving returning nodes: one is finding the lowest common ancestor, and the other is deleting a node with a certain value from a BST.

The inorder position is primarily used in BST (Binary Search Tree) scenarios, and you can consider the inorder traversal of a BST as traversing a sorted array.

The preorder position doesn't inherently have any special properties. The reason you find that many problems seem to involve writing code in the preorder position is simply because we tend to place code that isn't sensitive to preorder, inorder, or postorder in the preorder position.

You'll notice that code in the preorder position executes from top to bottom, whereas code in the postorder position executes from bottom to top. This is significant because code in the preorder position can only access data passed from the parent node via function parameters, whereas code in the postorder position can access both parameter data and data returned from the subtrees via the function's return value.

Can we obtain the solution by traversing the binary tree just once? If so, implement it using a traverse function along with external variables. This approach is called the "traversal" thinking pattern. Can we define a recursive function to derive the solution to the original problem using the answers from subproblems (subtrees)? If so, write the definition of this recursive function and fully utilize its return value. This is known as the "divide and conquer" thinking pattern. No matter which thinking pattern you use, you must consider:

- If you isolate a single node of the binary tree, what does it need to do? When should it do it (preorder/inorder/postorder)? You don't need to worry about other nodes—the recursive function will execute the same operation on all nodes.
- Differences between global variables, reference variables, local variables, and return variables when used in tree recursion programs.
- Referring back to the previous discussion: when encountering subtree problems, the first thing to consider is setting a return value for the function and focusing on postorder traversal.
- Conversely, if you've written a recursive-in-recursive solution like in the beginning, it's likely worth considering whether it can be optimized using postorder traversal.

1.7.3.2 Traversal Order

Next big question is pre-order, in-order or post-order. **90% we use post-order.** But it also depends on practical requirement.

In order examples:

- inorder: two applications Find k'th smallest and k'th largest element in a BST:

```
1 int kthSmallest(Node* root, int *i, int k){
```

```
  if (root == nullptr) { // base case
    return INT_MAX;
  }

  int left = kthSmallest(root->left, i, k); // Left

  if (left != INT_MAX) { // Join
    return left;
  }

  // if the current element is k'th smallest, return its value
  if (++*i == k) {
    return root->data;
  }

  return kthSmallest(root->right, i, k); // right
}
```

source code Only In order tranversal give ascending order, we have to use here. for example, kth element, you only think that the element in left tree, then return it, current root, then return it. if it's in right tree, then recursive call with root->right. return INT_MAX can be thought as a flag. if you want to look for a integer, you can use INT_MAX to denote a flag. I should be shared by all recursive call, so we use pointer or reference here.

- Print an expression tree

```
      *
    /   \
  +      c
 / \
a   b

void inorder(Node* root)
{
  if (root == nullptr) {
    return;
  }

  // if the current token is an operator, print open parenthesis
  if (isOperator(root->data)) {
    cout << "(";
  }

  inorder(root->left);
  cout << root->data;
  inorder(root->right);

  // if the current token is an operator, print close parenthesis
  if (isOperator(root->data)) {
    cout << ")";
  }
}
```

Pre order examples:

- Print left view of a binary tree

```
// Recursive function to print the left view of a given binary tree
void leftView(Node* root, int level, int &last_level)
{
  // base case: empty tree
```

```
  if (root == nullptr) {
    return;
  }

  // if the current node is the first node of the current level
  if (last_level < level)
  {
    // print the node's data
    cout << root->key << " ";

    // update the last level to the current level
    last_level = level;
  }

  // recur for the left and right subtree by increasing the level by 1
  leftView(root->left, level + 1, last_level);
  leftView(root->right, level + 1, last_level);
}
```

Post order examples:

- Find maximum sum root to leaf path in a binary tree

```
// Function to calculate the maximum root-to-leaf sum in a binary tree
int getRootToLeafSum(Node* root){
  // base case: tree is empty
  if (root == nullptr) {
    return 0;
  }

  // calculate the maximum node-to-leaf sum for the left child
  int left = getRootToLeafSum(root->left);

  // calculate the maximum node-to-leaf sum for the right child
  int right = getRootToLeafSum(root->right);

  // consider the maximum sum child
  return (left > right? left : right) + root->data;
}
```

- combined: find LCA

```
bool findLCA(Node* root, Node* &lca, Node* x, Node* y)
{
  // base case 1: return false if the tree is empty
  if (root == nullptr) {
    return false;
  }

  // base case 2: return true if either `x` or `y` is found
  if (root == x || root == y){  //
    // set lca to the current node
    lca = root;
    return true;
  }

  // recursively check if `x` or `y` exists in the left subtree
  bool left = findLCA(root->left, lca, x, y);

  // recursively check if `x` or `y` exists in the right subtree
  bool right = findLCA(root->right, lca, x, y);
```

```
20
21    // if 'x' is found in one subtree and 'y' is found in the other subtree,
22    // update lca to the current node
23    if (left && right) {
24      lca = root;
25    }
26
27    // return true if 'x' or 'y' is found in either left or right subtree
28    return left || right;
29  }
```

Only focus on the current root logic. If left anf right , then return root. otherwise just return left or right. so simple!!

Level order examples:

```
1  // Function to print spiral order traversal of a given binary tree
2  void spiralOrderTraversal(Node* root){
3    if (root == nullptr) {
4      return;
5    }
6
7    // create an empty double-ended queue and enqueue the root node
8    list<Node*> deque;          // or use deque
9    deque.push_front(root);
10
11   // 'flag' is used to differentiate between odd or even level
12   bool flag = false;
13
14   while (!deque.empty()) {  // loop till deque is empty
15     // calculate the total number of nodes at the current level
16     int nodeCount = deque.size();
17
18     if (flag){ // print left to right
19       // process each node of the current level and enqueue their
20       // non-empty left and right child to deque
21       while (nodeCount) {
22         Node* curr = deque.front(); // pop from the front if 'flag' is true
23         deque.pop_front();
24
25         cout << curr->key << " ";
26
27         // it is important to push the left child into the back,
28         // followed by the right child
29         if (curr->left != nullptr) {
30           deque.push_back(curr->left);
31         }
32         if (curr->right != nullptr) {
33           deque.push_back(curr->right);
34         }
35         nodeCount--;
36       }
37     }
38     else {  // print right to left
39       // process each node of the current level and enqueue their
40       // non-empty right and left child
41       while (nodeCount){
42         Node* curr = deque.back(); // it is important to pop from the back
43         deque.pop_back();
44
45         cout << curr->key << " ";     // print front node
```

```
46
47          // it is important to push the right child at the front,
48          // followed by the left child
49          if (curr->right != nullptr) {
50            deque.push_front(curr->right);
51          }
52          if (curr->left != nullptr) {
53            deque.push_front(curr->left);
54          }
55          nodeCount--;
56        }
57      }
58      flag = !flag; // flip the flag for the next level
59      cout << endl;
60    }
61  }
```

Source code use BFS, so use queue, but we also need spiral, so have to use deque which support pop_front and pop_back. In order to print each level, use below source code pattern.

```
1  while (!deque.empty()){
2    // calculate the total number of nodes at the current level
3    int nodeCount = deque.size();
4    while(nodeCound){
5      nodeCount--;
```

Traversal order summary

- Most of time, traversal order can be decided by the question itself very quickly.
- **Any time when you find Maximum or ancestor, this is a strong indication which we can use postorder.**
- You can combined use preorder visit and postorder logic combine together, such as LCA.

1.7.3.3 Interface

Next questions is interface design. (all auxiliary parameter) Find k'th smallest and k'th largest element in a BST: `int *i` or `int& i` is shared variable among all recursive. which record how many nodes we have visited so far. Return value is just your question answer. that is very easy to understand.

If you change inside function, sometimes, you need to change it back when you return.

```
1  // Recursive function to find paths from the root node to every leaf node
2  void printRootToleafPaths(Node* node, vector<int> &path){
3    if (node == nullptr) { // base case
4      return;
5    }
6
7    // include the current node to the path
8    path.push_back(node->data);
9
10   if (isLeaf(node)){ // if a leaf node is found, print the path
11     for (int data: path) {
12       cout << data << " ";
13     }
14     cout << endl;
15   }
16
17   // recur for the left and right subtree
```

```
18    printRootToleafPaths(node->left , path);
19    printRootToleafPaths(node->right , path);
20
21    // backtrack: remove the current node after the left , and right subtree are done
22    path.pop_back();
23  }
```

Another simple is use local stack variable. If you don't use reference, you don't need to pop_back. That is not memory friendly solution, but you can avoid pop_back function call.

```
1  // Recursive function to find paths from the root node to every leaf node
2  void printRootToleafPaths(Node* node, vector<int> path){
3    if (node == nullptr) {
4      return;
5    }
6
7    path.push_back(node->data);
8
9    // if a leaf node is found, print the path
10   if (isLeaf(node))   {
11     for (int data: path) {
12       cout << data << " ";
13     }
14     cout << endl;
15   }
16
17   // recur for the left and right subtree
18   printRootToleafPaths(node->left , path);
19   printRootToleafPaths(node->right , path);
20
21 }
```

Information flow:

- return value from child to parent, (only return one value, so need some calculation inside from both left and right children)
- pass from parent to child, parameter to remember auto value(level) ,
- shared between children and parent. and pass a pointer parameter(max_level) to remember static, which all recursive

1.7.3.4 Span child tree

Determine whether the given binary tree nodes are cousins of each other. Two nodes of a binary tree are cousins if they have the same depth, but have different parents. **traversal all, use global variable to store the good result when you traversal all.**

```
1  // Perform inorder traversal on a given binary tree and update `x` and `y`
2  void inorder(Node* root, Node* parent, int level, NodeInfo &x, NodeInfo &y)
3  {
4    // base case: tree is empty
5    if (root == nullptr) {
6      return;
7    }
8
9    // traverse left subtree
10   inorder(root->left , root, level + 1, x, y);
11
12   // if the first element is found, save its level and parent node
```

```
13    if (root->key == x.key){
14      x.level = level;
15      x.parent = parent;
16    }
17
18    // if the second element is found, save its level and parent node
19    if (root->key == y.key){
20      y.level = level;
21      y.parent = parent;
22    }
23
24    // traverse right subtree
25    inorder(root->right, root, level + 1, x, y);
26  }
```

Source code Use two global valure as reference, then search the whole tree.

1.7.3.5 Two trees

Recursive can be used to deal with two trees at the same time.

```
1   int identicalTrees(struct node* a, struct node* b){
2     /*1. both empty */
3     if (a==NULL && b==NULL)
4     return true;
5
6     /* 2. both non-empty -> compare them */
7     if (a!=NULL && b!=NULL)     {
8       return(
9       a->data == b->data &&
10      identicalTrees(a->left, b->left) &&
11      identicalTrees(a->right, b->right)
12      );
13    }
14
15    /* 3. one empty, one not -> false */
16    return false;
17  }
```

Some problems can be changed to two tree problems. such as if mirror.

```
1   // Function to check if subtree rooted at 'X' and 'Y' mirror each other
2   bool isSymmetric(Node* X, Node* Y)
3   {
4     // base case: if both trees are empty
5     if (X == nullptr && Y == nullptr) {
6       return true;
7     }
8
9     // return true if
10    // 1. Both trees are non-empty, and
11    // 2. The left subtree is the mirror of the right subtree, and
12    // 3. The right subtree is the mirror of the left subtree
13    return (X != nullptr && Y != nullptr) &&
14    isSymmetric(X->left, Y->right) &&
15    isSymmetric(X->right, Y->left);
16  }
```

Fill any nodes next pointer:

```
Node connect(Node root) {
  if (root == null) return null;
  connectTwoNode(root.left, root.right);
  return root;
}

void connectTwoNode(Node node1, Node node2) {
  if (node1 == null || node2 == null) {
    return;
  }

  node1.next = node2;
  //sibling
  connectTwoNode(node1.left, node1.right);
  connectTwoNode(node2.left, node2.right);
  //cousin
  connectTwoNode(node1.right, node2.left);
}
```

Source code Any time you need to access cousin, you should use two nodes. Three branch recursive, That is very different with others.

1.7.3.6 Traps

You can check if one node is leaf node. The basic idea is just like to find if a value exist in the tree. But you can't ignore base case.

```
      1     //must have base case to deal with this tree
    /   \
  null   2


      1     //if (root->left == null && root->right == null) return;
    /   \   //can deal with this context. but can't deal with previous tree.
 null   null

int countLeaves(Node* t){ //an example of isLeaf
  if(t == null)
    return 0;

  if(isLeaf(t))  {
    return 1;
  }
  else
    return countLeaves(t.left) +countLeaves( t.right);
}
```

- **Accessing the children values. You SHOULD NOT do that unless if want to judge if it's leaf node**. The child passed to the recursive call is treated as the new root.

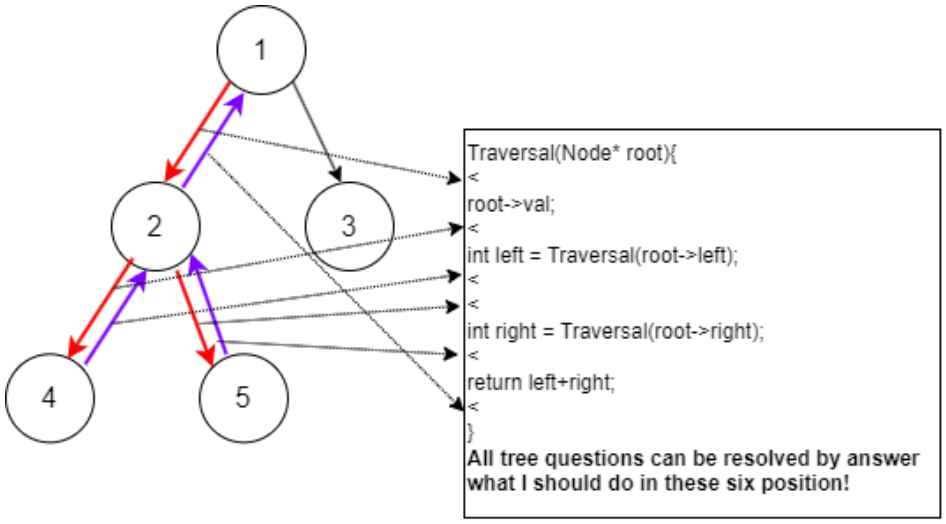
1.7.3.7 Conclusion

According to question, if it will return a value, such as leaf count, height. It has to have a function return value. What value should we return? Usually, this is related to question. For example, kth element should return integer, LCA should return a pointer. At the same time, integer and pointer also can act as flag value.

The first statement always should be base case.

```
if(root == nullptr)
        return [value]
```

- decide pre-order, in-order, and post-order.
- For some easy problems, you have finished the problems. Then for some difficult problems, you need to some logic comparison or calculation. such as Maximum path sum in a binary tree. or max height tree.
- Logic inside the function, an good example is LCA: there are four different possible. 1) The root node is x or y, 2) the x or y only in left tree, 3) the x or y only in the right tree, 4) the x and y in the two separate tree. That is all the possibles, then write all logic to deal previous 4 possibles.
- That is ALL!, never try to recursive call in your head.



1.7.4 Interview questions

1.7.4.1 Print part of tree

Usually use pre-order, then use some condition to suppress cout. At the same time, also need to pass some condition as parameter, (left view, level) return some flag(path include a node)

Print left view **When you return, the level will be local stack value, each value has stack scope.**

```
// Recursive function to print the left view of a given binary tree
void leftView(Node* root, int level, int &last_level)
{
  if (root == nullptr) { // base case: empty tree
    return;
  }

  // if the current node is the first node of the current level
  if (last_level < level){
    cout << root->key << " "; // print the node's data

    // update the last level to the current level
    last_level = level;
```

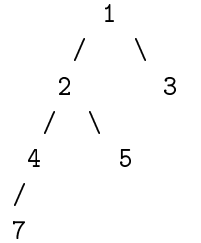
```
  }

  // recur for the left and right subtree by increasing the level by 1
  leftView(root->left, level + 1, last_level);
  leftView(root->right, level + 1, last_level);
}
```

Print the certain level

```
// Function to print all nodes of a given level from left to right
bool printLevelLeftToRight(Node* root, int level)
{
  if (root == nullptr) {
    return false;
  }

  if (level == 1)
  {
    cout << root->key << " ";
    return true;  //return, don't go deeper to save time complexity.
  }

  // process left child before the right child
  bool left = printLevelLeftToRight(root->left, level - 1);
  bool right = printLevelLeftToRight(root->right, level - 1);

  return left || right;  //that is why we need to return bool.
}

// Function to print level order traversal of a given binary tree
void levelOrderTraversal(Node* root)
{
  // start from level 1  till the height of the tree
  int level = 1;

  // run till printLevel() returns false
  while (printLevel(root, level)) {
    level++;
  }
}
```

Print path include a node. If target is present in tree, then prints all the ancestors and returns true, otherwise returns false.

- The trick in this question is that it doesn't tell us the function should return bool, you should guess it out.
- preorder and postorder should be combined. when search key, use preorder, when print all ancester, use post order.
- this combined order model also used in LCA.



key is 7, then your function should print 4, 2 and 1.

```
bool printAncestors(struct node *root, int target){
  /* base cases */
  if (root == NULL)
    return false;
  if (root->data == target)
    return true;

  /* If target is present in either left or right subtree of this node, then print this
      node */
  if ( printAncestors(root->left, target) ||
  printAncestors(root->right, target) ){
    cout << root->data << " ";
    return true;
  }
  /* Else return false */
  return false;
}
```

Source cde: Idea: just like max height example, return bool instead value, logic || instead max. the basic idea are quite same.

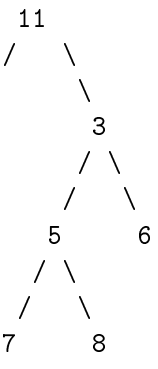
Source code: use return bool value to suppress cout

Source code: focuse local root principal also can be used here. for local root, there are only three possibles. current == target, target in left tree, target in right tree.

1.7.4.2 Build tree

The problem of constructing a binary tree usually follows the "divide and conquer" approach: constructing the entire tree = root node + constructing the left subtree + constructing the right subtree.

serialize and deserialize a tree




```
//11,2,4,.,.,.,3,5,7,.,.,8,.,.,6,.,.
```

```
string serialize(Node* root) {  //serialize the root
  if (root == NULL) {
    return ".";   //a '.' character represents a null node
  }

  string serialized = "";

  serialized += to_string(root->key);
  //add the current node to the serialized string then traverser its left and right
      children respectively
```

```
10
11    serialized += "," + serialize(root->left);      //add the seperator after each node
12    serialized += "," + serialize(root->right);
13
14    return serialized;
15  }
16
17  Node* deserializeUtil(vector<string>& serialized, int& i) {
18
19    if (i == serialized.size()) {    //if the whole array is exhausted
20      return NULL;
21    }
22
23    string val = serialized[i++];
24
25    if (val == ".") {    //if we encounter the '.' character => a null pointer
26      return NULL;
27    }
28    //build the tree using the serialized array
29    Node* node = new Node{ stoi(val) };
30
31    node->left = deserializeUtil(serialized, i);
32    node->right = deserializeUtil(serialized, i);
33
34    return node;
35
36  }
```

Source code: with nullptr(.) in string, we can use in-order unique define a tree.

Construct a binary tree from inorder and preorder traversal.

```
1  Node* construct(int start, int end, vector<int> const& preorder,
2  int& pIndex, unordered_map<int, int>& map, int level){
3
4    if (start > end) { // base case
5      return nullptr;
6    }
7
8    // The next element in `preorder[]` will be the root node of subtree
9    // formed by sequence represented by `inorder[start, end]`
10   Node* root = newNode(preorder[pIndex++]);
11
12   // get the root node index in sequence `inorder[]` to determine the
13   // left and right subtree boundary
14   int index = map[root->key];
15
16   // recursively construct the left subtree
17   for (int i = 0; i < level; ++i)
18   cout << "␣";
19   cout << root->key << "␣" << start << "␣" << index << "␣" << end << endl;
20
21   root->left = construct(start, index - 1, preorder, pIndex, map, level+1);
22
23   // recursively construct the right subtree
24   root->right = construct(index + 1, end, preorder, pIndex, map, level+1);
25
26   // return current node
27   return root;
28 }
29
```

```
// Construct a binary tree from inorder and preorder traversals.
// This function assumes that the input is valid
// i.e., given inorder and preorder sequence forms a binary tree
Node* construct(vector<int> const& inorder, vector<int> const& preorder){
  // get the total number of nodes in the tree
  int n = inorder.size();

  // create a map to efficiently find the index of any element in a given inorder
      sequence
  unordered_map<int, int> map;
  for (int i = 0; i < n; i++) {
    map[inorder[i]] = i;
  }

  // `pIndex` stores the index of the next unprocessed node in preorder;
  // start with the root node (present at 0th index)
  int pIndex = 0;

  return construct(0, n - 1, preorder, pIndex, map, 0);
}
//output

          1
        /   \
       /     \
      2       3
     /       / \
    /       /   \
   4       5     6
          / \
         /   \
        7     8

1 0 2 7
 2 0 1 1
  4 0 0 0
 3 3 6 7
  5 3 4 5
   7 3 3 3
   8 5 5 5
  6 7 7 7
```

Source code: Iterate one time each time in pre-order. Build map to store in-order index, divide it into left tree and right tree. **just like quick_sort**, pick an element from pre-order, then divide left and right according to index value in map

How many BST?

```
/* Main function, how many different BST tree given a number */
int numTrees(int n) {
  // Calculate the number of BSTs that can be formed from the closed interval [1, n]
  return count(1, n);
}

/* Calculate the number of BSTs that can be formed from the closed interval [lo, hi] */
int count(int lo, int hi) {
  if (lo > hi) return 1; // base case

  int res = 0;
  for (int i = lo; i <= hi; i++) {
    // Use the value of i as the root node
```

```
    int left = count(lo, i - 1);
    int right = count(i + 1, hi);
    // The total number of BSTs is the product of combinations of the left and right
        subtrees
    res += left * right;
  }
  return res;
}
```

Build BST

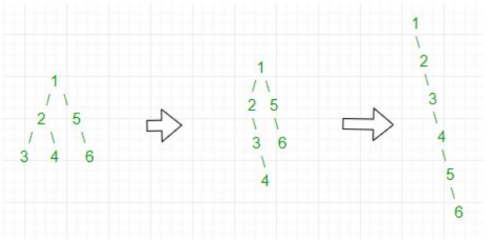
```
/* Main function */
public List<TreeNode> generateTrees(int n) {
  if (n == 0) return new LinkedList<>();
  // Construct BSTs that can be formed from the closed interval [1, n]
  return build(1, n);
}

/* Construct BSTs that can be formed from the closed interval [lo, hi] */
List<TreeNode> build(int lo, int hi) {
  List<TreeNode> res = new LinkedList<>();
  // base case
  if (lo > hi) {
    res.add(null);
    return res;
  }

  // 1. Enumerate all possible root nodes.
  for (int i = lo; i <= hi; i++) {
    // 2. Recursively construct all valid BSTs for the left and right subtrees.
    List<TreeNode> leftTree = build(lo, i - 1);
    List<TreeNode> rightTree = build(i + 1, hi);
    // 3. Enumerate all combinations of left and right subtrees for the root node.
    for (TreeNode left : leftTree) {
      for (TreeNode right : rightTree) {
        // Use i as the value of the root node
        TreeNode root = new TreeNode(i);
        root.left = left;
        root.right = right;
        res.add(root);
      }
    }
  }

  return res;
}
```

1.7.4.3 Change tree

Flatten tree. In order and post order are both OK, but preorder is not good.

```
void flatten(struct Node* root){
  if (root == NULL || root->left == NULL &&
  root->right == NULL) {
    return;
  }

  // if root->left exists then we have to make it root->right
  if (root->left != NULL) {
```

```
9       flatten(root->left); // move left recursively
10
11      // store the node root->right
12      struct Node* tmpRight = root->right;
13      root->right = root->left;
14      root->left = NULL;
15
16      // find the position to insert
17      // the stored value
18      struct Node* t = root->right;
19      while (t->right != NULL) {
20        t = t->right;
21      }
22
23      t->right = tmpRight; // insert the stored value
24    }
25
26    // now call the same function for root->right
27    flatten(root->right);
28  }
```



Truncate a binary tree to remove nodes that lie on a path having a sum less than k

- Only node on a path, should use postorder.
- Only print(or not print) or keep (or delete), should return bool
- calculate sum, should input a parameter sum
- **When delete leaf, parent will become leaf, That is why we have to use postorder**
- You have to use reference to pointer.

```
1  void trunc(Node* &curr, int k, int sum){
2    if (curr == nullptr) { // base case: empty tree
3      return;
4    }
5
6    // update sum of nodes in the path from the root node to the current node
7    sum = sum + (curr->data);
8
9    // Recursively truncate left and right subtrees
10   trunc(curr->left, k, sum);
11   trunc(curr->right, k, sum);
12
13   // Since we are doing postorder traversal, the subtree rooted at the current
14   // node may be already truncated, and the current node is a leaf
15   // if the current node is a leaf node and its path from the root node has a sum
16   // less than the required sum, remove it
17   if (sum < k && isLeaf(curr)){
18
19     delete(curr);  // free the memory allocated to the current node
20
```

```
21      curr = nullptr; // set current node to null (node is passed by reference)
22    }
23  };
```

These two methods have common points: we both modify tree from bottom to top. When we modify current noe, we assume that all the child tree has finished. One use in-order, other use post-order. (Why? think about later.)

1.7.4.4 Other

Given the root of a binary tree, return all root-to-leaf paths in any order. (you need root == nullptr, also need to judge if it's leaf node). That is an interesting example.

```
1  void pathTree(TreeNode* root, string path, vector<string>& result){
2    if(root == nullptr){
3      return;
4    }
5    if(path ==""){
6      path = to_string(root->val);
7    }
8    else{
9      path+="->";
10     path+=to_string(root->val);
11   }
12   if(root->left == nullptr && root->right == nullptr){
13     result.push_back(path);
14     return;
15   }
16
17   pathTree(root->left, path, result);
18   pathTree(root->right, path, result);
19 }
20
21 vector<string> binaryTreePaths(TreeNode* root) {
22   string s = "";
23   vector<string> result;
24   pathTree(root, s, result);
25   return result;
26 }
```

In order without recursive,

```
1  // An iterative process to print preorder traversal of Binary tree
2  void iterativePreorder(node *root){
3      if (root == NULL) // Base Case
4          return;
5
6      // Create an empty stack and push root to it
7      stack<node *> nodeStack;
8      nodeStack.push(root);
9
10     /* Pop all items one by one. Do following for every popped item
11        a) print it
12        b) push its right child
13        c) push its left child
14     Note that right child is pushed first so that left is processed first */
15     while (nodeStack.empty() == false)     {
16         // Pop the top item from stack and print it
17         struct node *node = nodeStack.top();
```

```
18          printf ("%d ", node->data);
19          nodeStack.pop();
20
21          // Push right and left children of the popped node to stack
22          if (node->right)
23              nodeStack.push(node->right);
24          if (node->left)
25              nodeStack.push(node->left);
26      }
27  }
```

Idea: A common implementation of DFS, without check two unvisited condition. Because tree is a 1) minimally connected graph and having only one path between any two vertices. 2) no loop.

Determine whether a binary tree is a subtree of another binary tree. both inorder and preorder together identify a tree quniquely.

Can be changed into a binary tree problem!

- print all subsequence, select current or don't select
- stair case, walk one or two
- house robbery
- All possible combine in array.

```
1  Input:  digits[] = { 1, 2, 2, 1 }
2  {1, 2, 2, 1}
3  {1, 2, 21}
4  {1, 22, 1}
5  {12, 2, 1}
6  {12, 21}
```

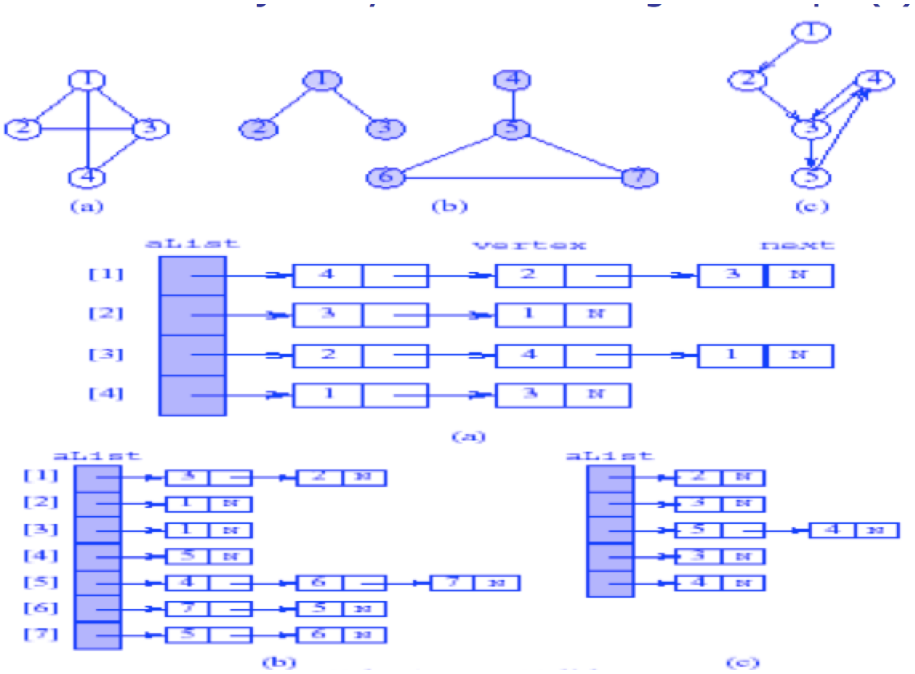
1.8 Graph

1.8.1 Basic

A Tree is just a restricted form of a Graph. They fit with in the category of Directed Acyclic Graphs (or a DAG). So Trees are DAGs with the restriction that a child can only have one parent.

Graphs are generally search breath first or depth first. The same applies to Tree.

Basic represent way: **adjacency list** and **adjacency matrix**:



You want to mirror the problem using a tree-like structure:For this we have boost graph library. The BGL currently provides two graph classes and an edge list adaptor: adjacency_list, adjacency_matrix and edge_list. The adjacency_list class is the general purpose "swiss army knife" of graph classes.

1.8.2 DFS and BFS

1.8.2.1 DFS

DFS has three versions 1) recursive 2) recursive to loop version. 3) uniform version.

- Recursive version:

```
DFS(current){
   visit current;
   for all(next to current){
      if(unvisited(next)){
           DFS(next);
       }
    }
}
```

- Recursive to loop

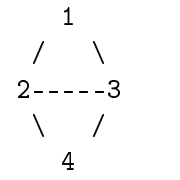
```
DFS(start){
 S.push(start)
   while(!S.empty()){
     current = top();
     visit current;
     If (current has one unvisited next){
          stack.push(next);
     }
     else{
         stack.pop()
     }
}
}
```

- Unified version:

```
DFS(start){
 S.push(start)
  while(!S.empty()){
    current = S.pop
    if(unvisited(current){
          visit current.
          for all(next to current and unvisited)
             S.push(next);
    }
  }
 }
```

Why we need two "unvisited" words in DFS unified version.

- When we pop node from S, then we visit it, then we add all adjacent nodes to S, In this way, When we pop 1 from S, we add 2, 3 to S. at this time, 3 is unvisited, then when we pop 2 from S, we will add 4, 3 to S. so there are two 3 in the S.
- Based on, we need to first unvisit to avoid visit 3 twice
- When we reach 3; 1,2,4 have been visited, so when we want to add all 3 adjacent, we need second unvisited to avoid 1,2,4 to S.



DFS is an algorithm to search a hierarchical structure. But, pre-order traversal seems to be something similar also. So, what is the difference between the two?? DFS says:

1. If element found at root, return success.
2. If root has no descendants, return failure
3. Recursive DFS on left subtree: success if element found
4. If not, Recursive DFS on right subtree: success if element found

Pre-order Traversal says:

1. Visit the root
2. Recursive pre-order on left subtree
3. Recursive pre-order on right subtree

1.8.2.2 BFS

There are two version of BFS:

- Form DFS unify version, We can change S(stack) to Q(queue) and get unified version. **That is why we call it unified version**

```
BFS(start){
 Q.push(start)
  while(!Q.empty()){
    current = Q.pop
    if(unvisited(current){
```

```
            visit current.
            for all(next to current and unvisited)
               Q.push(next);
      }
    }
  }
```

- Common version: Unified version will have multi-copy vertex in the queue. We can improve it as below: 1) when you add it to queue, visit it. In this way, avoid add multi-copy vertex in queue 2) Because All the vertex in queue is visited, when you pop it, you don't need to judge if it's visisted.

```
DFS(start){
 D.push(start)
  while(!D.empty()){
    current = D.pop  //2) no if here
    for all(next to current and unvisited){
          visit current.  //1) push and visit at the same time.
          D.push(next);
     }
  }
 }
```

- conclusion:
  1. **Conclusion, just remember unified version. 1) pop current, push neighbor, 2) check two unvisited**
  2. unified version will push two nodes into stack or queue, so for BFS, you can visit and push at the same time, in this way, you don't need to check current if it is unvisited.
  3. For DFS, you can't use this way, because you can only visit only one neighbor, or it will become BFS. so each time, you just push one node into stack. It leads to two other versions. recursive version, and loop version.

Path is saved in the stack. when you quit while loop. just pop up each position from stack, then you get a path. But for BFS, you need extra information to save parent information.

1.8.3 Spanning tree

BFS and DFS will produce spanning tree.

The minimum spanning tree can be obtained in polynomial time: Kruskal's algorithm, Prim's algorithm and Sollin's algorithm.

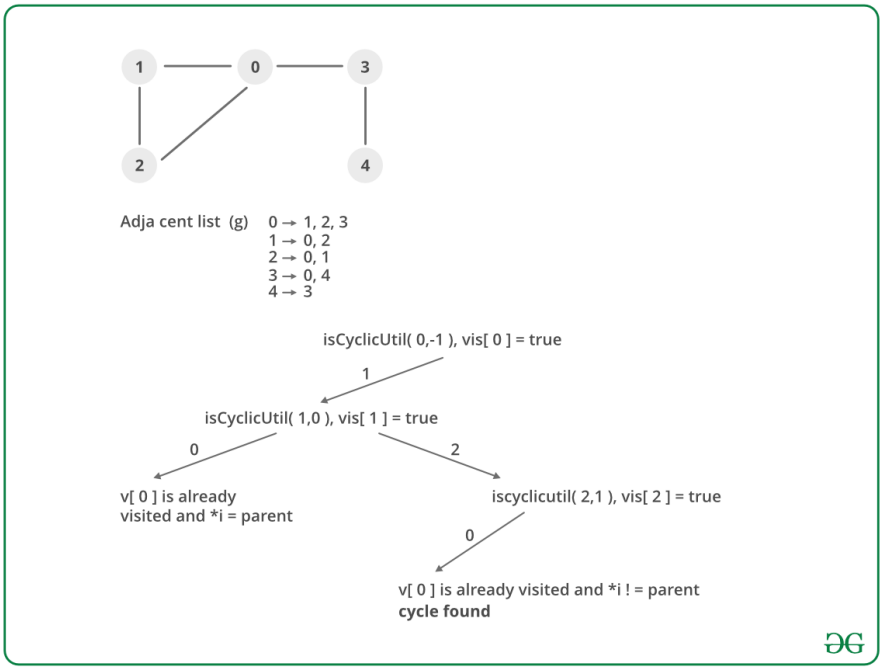
Kruskal algorithm:

```
KRUSKAL(G):
 A = empty
 foreach v in G.V:
    MAKE-SET(v)
 foreach (u, v) ordered by weight(u, v), increasing:
    if FIND-SET(u) != FIND-SET(v):
       A = A  {(u, v)}
       UNION(u, v)
 return A
```

Source code: union find algorithm.

1.8.4 Application

detect a cycle in the graph. **Undirected graph, not directed graph**. The key point is just find visited[i], but if visited[i] is parent, then it's not a cycle. The whole logic is not complex.




```
// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent){
    visited[v] = true; // Mark the current node as visited

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i){
        // If an adjacent is not visited, then recur for that adjacent
        if (!visited[*i])        {
           if (isCyclicUtil(*i, visited, v))
              return true;
        }
//If an adjacent is visited and not parent of current vertex, then there is a cycle.
        else if (*i != parent)
           return true;
    }
    return false;
}
```

Direct graph detect cycle. **need to add a onPath to detect cycle, different with undirected graph.** When we want to push node to stack, don't judge if we have visited them or not? (undirected graph use visited[], directed graph use onPath[])

```
boolean[] onPath;
boolean[] visited;
boolean hasCycle = false;
void traverse(List<Integer>[] graph, int s) {
  if (onPath[s]) {
```

```
6       // find cycle
7       hasCycle = true;
8     }
9     if (visited[s] || hasCycle) {
10      return;
11    }
12    visited[s] = true;
13
14    onPath[s] = true;
15    for (int t : graph[s]) {
16      traverse(graph, t);
17    }
18    onPath[s] = false;
19  }
```

1.9 Other customized data structure

1.9.1 union find

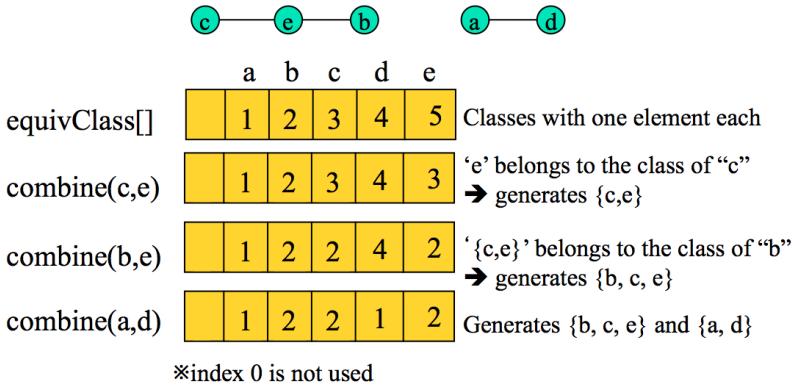
It is also called as "Online Equivalence class"

definition of Equivalence relationship

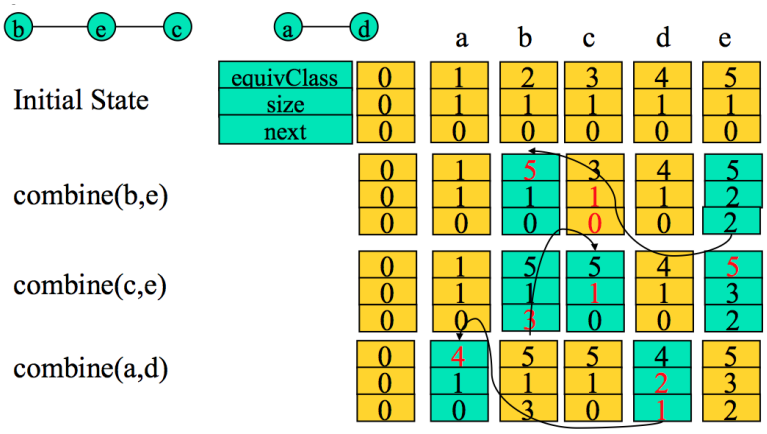
- a a. (Reflexivity)
- a b if and only if b a. (Symmetry)
- if a b and b c then a c. (Transitivity)

online equivalence class, also known union-find question.

- method 1: use array



- method 2: stimulate pointer




```java
class UF {
  // Number of connected components
  private int count;
  // Stores the parent node of each node
  private int[] parent;

  // n is the number of nodes in the graph
  public UF(int n) {
    this.count = n;
    parent = new int[n];
    for (int i = 0; i < n; i++) {
      parent[i] = i;
    }
  }

  // Connects node p and node q
  public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);

    if (rootP == rootQ)
      return;

    parent[rootQ] = rootP;
    count--;  // Merges two connected components into one
  }

  // Checks if node p and node q are connected
  public boolean connected(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    return rootP == rootQ;
  }

  public int find(int x) {
    if (parent[x] != x) {
      parent[x] = find(parent[x]);
    }
    return parent[x];
  }

  // Returns the number of connected components in the graph
  public int count() {
    return count;
  }
```

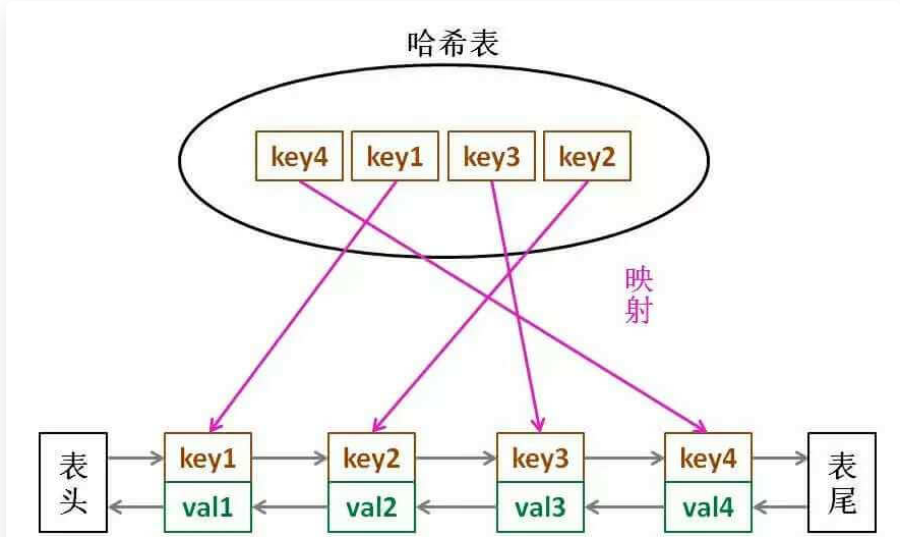
```
46 }
```

1.9.2 LRU and LFU

LRU is hash + list. (listedhashmap)

LFU is hash + heap. hashMap<key, pairvalue, frequent>. we still need key data structure to manage the frequency hashMap<frequent, hashLinkSet> freqToKeys. freqToKeys is key to this question.

In LRU and LFU, The key data structure is LinkedHashMap, in LFU, it use LinkedSetMap, in fact, LinkedSetMap still use the LinkedHashMap behind the scene.



1.9.3 O(1) insert, delete and getRandom

Insert, delete get random O(1) hash +vector(no need to keep vector sort). When you want to delete, swap it with the last element in the vector, then delete it. **Most of complex data structure need hash + list/vector combine, it's a good clue**

- C++ does not offer a collection template with the behavior that would mimic Java's LinkedHashMap<K,V>, so you would need to maintain the order separately from the mapping.
  This can be achieved by keeping the data in a std::list<std::pair<K,V», and keeping a separate std::unordered_map<k,std::list<std::pair<K,V»::iterator> map for quick look-up of the item by key:
  1. On adding an item, add the corresponding key/value pair to the end of the list, and map the key to the iterator std::prev(list.end()).
  2. On removing an item by key, look up its iterator, remove it from the list, and then remove the mapping.
  3. On replacing an item, look up list iterator from the unordered map first, and then replace its content with a new key-value pair.
  4. On iterating the values, simply iterate std::list<std::pair<K,V».

Chapter 2

Algorithm

Big O notation. Remember below English words: constant, logarithmic (log n), linear (n) linearithmic (n*log(n)) , quadratic 2^n (2 to the power of n), polynomial(quadratic cubic), exponential c^n. factorial $n!$. O(10^N): trying to break a password by testing every possible combination (assuming numerical password of length N). traveling salesman problem(TSP)is factorial $n!$. Both factorial and exponential are non-polynomial(NP).

verb|https://webusers.imj-prg.fr/ jan.nekovar/co/en/en.pdf| list some common mathematical English.

space complexity is much simple, usually, we only use O(1), O(n) and O(n^2), we don't use other forms very often.

2.1 Math and geometry basic

2.1.1 Geometry

If three points make a triangle?

```
// Calculation the area of triangle. We have skipped
// multiplication with 0.5 to avoid floating point computations
// if a is 0, don't make a triangle.
int a = x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2);
```

Judge three points are in the same line?

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{y_3 - y_1}{x_3 - x_1}$$

Get one points in the triangle.

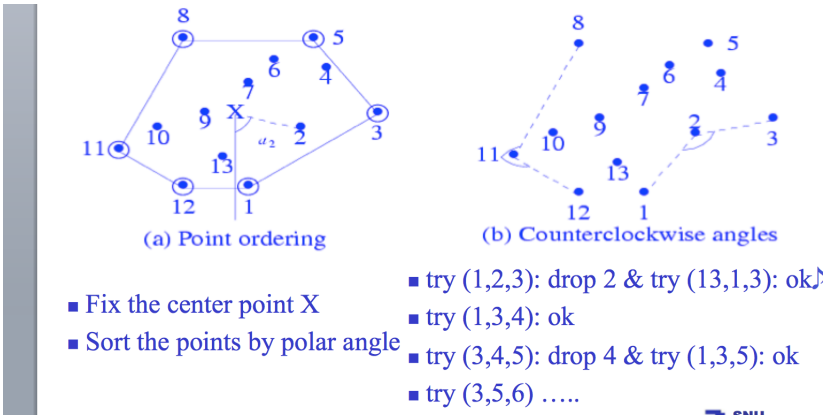
$$x = \frac{\frac{x_1 + x_2}{2} + x_3}{2}$$

If two rectangles overlap? l1: Top Left coordinate of first rectangle. r1: Bottom Right coordinate of first rectangle. l2: Top Left coordinate of second rectangle. r2: Bottom Right coordinate of second rectangle. 1) One rectangle is above top edge of other rectangle. 2) One rectangle is on left side of left edge of other rectangle.

```
// Returns true if two rectangles (l1, r1) and (l2, r2) overlap
bool doOverlap(Point l1, Point r1, Point l2, Point r2){
    // If one rectangle is on left side of other
    if (l1.x > r2.x || l2.x > r1.x)
        return false;
    // If one rectangle is above other
    if (l1.y < r2.y || l2.y < r1.y)
        return false;
    return true;
}
```

2.1.1.1 Convex hull

1. Find a point C that is inside the convex hull of S ((sum of x coordinates)/ n, (sum of y coordinates)/n)
2. Sort S by polar angle and within polar angle by distance from C. Use atan2(y-Cy , x-Cx) c++ function.
3. Create a doubly linked circular list of points using above order Let right link to the next point in the order and left link to the previous point



4. Let p be the point that the smallest y-coordinate (break a tie, if any, by selecting the one with largest x-coordinate)
5. use Law of cosines to calculate angle formed by x, rx and rrx

$$c^2 = a^2 + b^2 - 2ab\cos\lambda$$

```
for (x = p; rx = point to the right of x; x != rx) {
      rrx = point to the right of rx;
      if (angle formed by x, rx, and rrx is <=180 degrees) {
          delete rx from the list;
          rx = x;
          x = point on left of rx;
      }
      else {
      x = rx; rx = rrx;}
  }
```

6. For convex hull problem, you can use double link list to store all the extreme points. Because it need to use three consequence points to measure the degree to judge if it's less than 180 degrees. So double link list is the best data structure to use.

2.1.2 Permutation and combination

repeated / unrepeated (Permutation/Combination)

repeated permutation: Three number, how many permutation 10^3. Another Three position, For each position, you can select 10 options and fill it. If three position, For each position, you can selet 2 options and fill it. You get powerset problem 2^3 {a,b, c} all the subset.

unrepeated permutation: $n! = factorial function$

unrepeated combination:

$$C(n,r) = {}^nC_r = {}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

repeated combination:

$$\binom{r+n-1}{r} = \frac{(r+n-1)!}{r!(n-1)!}$$

where ***n*** is the number of things to choose from, and we choose ***r*** of them (Repetition allowed, order doesn't matter)

Detail can been seen here: https://www.mathsisfun.com/combinatorics/combinations-permutations.html

2.1.3 Bit

Basic operations are: and, or, xor, not, right shift, left shit.

```
& | ^ ! << >> //six operators related with bits.

number |= 1UL << n; // set
number &= ~(1UL << n); //clear
number ^= 1UL << n;  //toggle, ^= can be used for toggle.
```

Many C compilers choose which right shift to perform depending on what type of integer is being shifted; often signed integers are shifted using the arithmetic shift, and unsigned integers are shifted using the logical shift.

```
a|b|c|d //after we right shift, we change it to below
x|a|b|c
1) with a logical shift put 0 in x
2) with a arithmetic shift put a in x
```

XOR is exclusive or, You can think that it's subset of OR, for OR, if both bits are 1, the result is 1. but In XOR, we exclusive this possible, You can think that it's "pure" or. Based on previous definition. There are two characteristics:

```
x^x = 0
x^0 = x
```

from previous two examples, we can use it to find any number occur even number in the array.

change the last 1 to zeor. The basic idea is very interesting.

```
n&(n-1);
//110100-> 110000

//when we n-1: 110100, the right most 1 will become 0, all zeor after
//right most 1 will become 1.
110100
110011
*--
```

Based on previous trip, we can test if a number is power of 2. There is only one 1 in the binary formation.

```
bool isPowerOfTwo(int n) {
  if (n <= 0) return false;
  return (n & (n - 1)) == 0;
}
```

Hamming weight, Hamming weight of an integer is defined as the number of set bits in its binary representation.

```
int hammingWeight(uint32_t n) {
  int res = 0;
  while (n != 0) {
    n = n & (n - 1); //change n, you need to assign new value back to n.
    res++;
  }
  return res;
}
```

As noted in this answer, n&(n-1) unsets the last set bit. So, if we unset the last set bit and xor it with the number; by the nature of the xor operation, the last set bit will become 1 and the rest of the bits will return 0

```
int set_bit = n ^ (n&(n-1));
```

2.1.4 Usage of modulus %

Some applicatoin you can use with modulus (reminder).

- circular queue
- hash
- to know the last three digit.

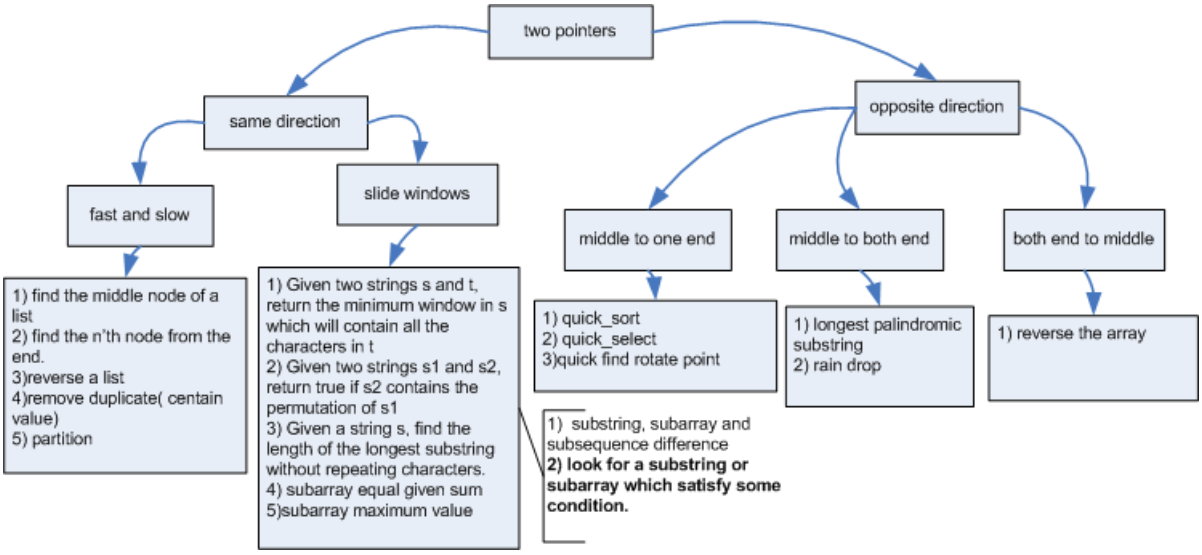
We usually use the

```
int[] arr = {1,2,3,4,5};
int n = arr.length, index = 0;
while (true) {
  // Looping through a circular array
  print(arr[index % n]);
  index++;
}

int[] nextGreaterElements(int[] nums) {
  int n = nums.length;
  int[] res = new int[n];
  Stack<Integer> s = new Stack<>();
```

```
13    // Doubling the array length to simulate a circular array
14    for (int i = 2 * n - 1; i >= 0; i--) {
15      // Modulo operation for index i, the rest follows the template
16      while (!s.isEmpty() && s.peek() <= nums[i % n]) {
17        s.pop();
18      }
19      res[i % n] = s.isEmpty() ? -1 : s.peek();
20      s.push(nums[i % n]);
21    }
22    return res;
23  }
```

2.2 Two pointers

Two pointers are very popular algorithms in many interview questions.



2.2.1 Middle to both ends

The longest palindrome substring

```
1   string palindrome(string& s, int l, int r) {
2     //from l go to left, from r go to right.
3     //<-----l   r----->
4
5     // check if it is out of boundary
6     while (l >= 0 && r < s.size()
7     && s[l] == s[r]) {
8       // both side move
9       l--; r++;
10    }
11    // return result
12    return s.substr(l + 1, r - l - 1);
13  }
14
15  string longestPalindrome(string s) {
16    string res;
```

```
  for (int i = 0; i < s.size(); i++) {
    // s[i] is middle
    string s1 = palindrome(s, i, i);
    // s[i] and s[i+1]
    string s2 = palindrome(s, i, i + 1);
    // res = longest(res, s1, s2)
    res = res.size() > s1.size() ? res : s1;
    res = res.size() > s2.size() ? res : s2;
  }
  return res;
}
```

Rain drop

```
int trap(vector<int>& height) {
  if (height.empty()) return 0;
  int n = height.size();
  int res = 0;
  // memo
  vector<int> l_max(n), r_max(n);
  // base case
  l_max[0] = height[0];
  r_max[n - 1] = height[n - 1];
  // calculate l_max from left to right
  for (int i = 1; i < n; i++)
  l_max[i] = max(height[i], l_max[i - 1]);
  // calculate r_max from right to left
  for (int i = n - 2; i >= 0; i--)
  r_max[i] = max(height[i], r_max[i + 1]);

  // calculate the last answer.
  for (int i = 1; i < n - 1; i++)
  res += min(l_max[i], r_max[i]) - height[i];
  return res;
}
```

2.2.2 slow and fast

remove duplicate

```
int removeDuplicates(int[] nums) {
  if (nums.length == 0) {
    return 0;
  }
  int slow = 0, fast = 0;
  while (fast < nums.length) {
    if (nums[fast] != nums[slow]) {
      slow++;
      // maintain nums[0..slow] without duplicate
      nums[slow] = nums[fast];
    }
    fast++;
  }
  // length should be  + 1
  return slow + 1;
}
```

partition

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
  first = std::find_if_not(first, last, p);
  if (first == last) return first;

  for (ForwardIt i = std::next(first); i != last; ++i) {
    if (p(*i)) {
      std::iter_swap(i, first);
      ++first;
    }
  }
  return first;
}
```

Note here, in removeDuplication, we ++ first, then assign, but in partition, we swap first, then ++. you need to know that slow pointer is pointing by these two methods.

2.2.3 slide windows

Code template of slide windows. Slide windows is suitable for some sub-string problem.

```
int left = 0, right = 0;

while (right < n) {`
  // increase windows
  window.add(a[right]);
  right++;

  //update windows
  //.......

  while (window needs shrink) {
    // decrease window
    window.remove(s[left]);
    left++;

    //update windows here.
    //......
  }
}
```

An example. minimal substring in source which includes all the letters of in target. leetcode 76. s = "ADOBECODEBANC", t = "ABC, output "BANC"

```
string minWindow(string s, string t) {
  unordered_map<char, int> need, window;
  for (char c : t) need[c]++;

  int left = 0, right = 0;
  int valid = 0;
  // Variables to record the starting index and length of the minimum window substring
  int start = 0, len = INT_MAX;
  while (right < s.size()) {
    // `c` is the character that will enter the window
    char c = s[right];
    right++;  // Expand the window
    if (need.count(c)) { // Update the data inside the window
```

```
14          window[c]++;
15          if (window[c] == need[c])
16          valid++;
17      }
18
19      // Check if the left side of the window needs to shrink
20      while (valid == need.size()) {
21          // Update the minimum window substring here
22          if (right - left < len) {
23              start = left;
24              len = right - left;
25          }
26          // 'd' is the character that will exit the window
27          char d = s[left];
28          // Shrink the window
29          left++;
30          // Update the data inside the window
31          if (need.count(d)) {
32              if (window[d] == need[d])
33              valid--;
34              window[d]--;
35          }
36      }
37    }
38    // Return the minimum window substring
39    return len == INT_MAX ? "" : s.substr(start, len);
40  }
```

Basic rules:

1. If a subarray or substring has constraints, if not, use two for brute force, an example is leetcode 2913
2. if there is constraints. double pointers, leetcode 3, 76, 567 and 3258.
3. Do we need map to store substring information? for 3, 76, and 567. we need it. For 3258, we don't need map to store character information in substring, that is very important clue, for 3, we need to judge if there is repeated character, so we need a map. A similar questions can be found.

2.3 Recursive

2.3.1 Basic

Consider an array/string: `1,2,3,4`

```
Subarray: contiguous sequence in an array i.e.{1,2},{1,2,3}

Subsequence: Need not to be contiguous, but maintains order i.e. {1,2,4}

Subset: Same as subsequence except it has empty set i.e. {1,3},{}

Given an array/sequence of size n, possible
Subarray = n*(n+1)/2
Subseqeunce = (2^n) -1 (non-empty subsequences)
Subset = 2^n
```

recursive is not an algorithm, it's just a method. D&C, DP, backtrack, bound and boundary all use

recursive in their implementation, but different ways.

First, the inline specification on a function is just a hint. The compiler can (and often does) completely ignore the presence or absence of an inline qualifier. With that said, a compiler can inline a recursive function, much as it can unroll an infinite loop. It simply has to place a limit on the level to which it will "unroll" the function.

One recursive(Factorial), Two recursive call(Hanoti) (Tree), Multi Recursive (Permutation)

result is single (Factorial), Result is many steps,(Hanoti) (Maze), Result is a set(permutation). you can see that 1) any recursive function should at least one input parameter, and this parameter should to be pass sub-problem. 2) for return value, it has two different kind, if result is single value, you should return a value, such as Factorial. If result is set(permutation) or many steps(in-order traversal of treeor Hanoi tree), you can declare your function as void. 3) Some functions need input another parameter, such as level information in the tree or position information in the permutation. 4) Base case can be understood differently, most of time base case is 0 or null, but for permutation, base case is i==n. most subproblem, i decrease, but for permutation, for each subproblem i increase.

- single result

```
int Factorial(int n){
if (n == 1)
     return 1;
 return n*Factorial(n-1);
}
```

- result is serial steps (Hanoti) DFS(Maze)

```
FUNCTION MoveTower(disk, source, dest, spare):
IF disk == 0, THEN:
    move disk from source to dest
ELSE:
    MoveTower(disk - 1, source, spare, dest)    // Step 1 above
    move disk from source to dest               // Step 2 above
    MoveTower(disk - 1, spare, dest, source)    // Step 3 above
END IF
```

- Result is a set(permutation)

It includes direct recursive and indirect recursive. direct recursive is R() call R() again in it's funciton body. It has two parts: **1)base and 2) recursive component**

2.3.2 Solution space

Why this basic conception is so important, Because you can use these four to describe solution space. repeated permutation can be use in 0/1 backpack problem. LCS problem.

Nearest point pair's solution space is n*(n-1). It's n^2. A good hint is that it can be reduce $n * log(n)$. So We should consider D&C (divide and conquer)

0/1 packback's solution's space is 2^n, A good hint is that I can be reduce to n^c. (c is constant) by dynamic programming.

If there is not D&C or DP, you have to use branch and bound or backtracking. Such as TSP traveling sales problem

If you need optimal answer, You need to traversal the whole solution space. (such as TSP and 0/1 backpacking). If you don't need optimal answer, you don't need to do that, In these two conditions, you all need to **backtracking.**

Why I can't use DP to TSP, but I can use DP to 0/1?

Recursion is a mathematical induction.

Modeling is mapping your problem to a abstract structure.

- Permutation: arrangement, tour, ordering sequence
- subset: cluster, collection, committee, group, packaging, selection
- tree:hieracrchy, dominance, ancestor/descendant, taxonomy
- graphs: network, circuit, web, relationship
- points
- polygon
- string

Some typical applications, you need to remember them.

- 5 cities, TSP, time complexity is n! (5! = 120)
- 5 number, whole set partition. bell number, for 5, last result is 42. please see set partition number in recursive section.
- 5 object, 0/1 backpack, Subset, time complexity is 2^n (2^5=32)
- 5 icecream, 3 spoon, combination, (c(5, 3) = 10). combination is subset(number of element is 3) of subset, they are different.

2.3.3 Basic recursive pattern

You should understand Big O here. For example:

- O(n) is found max and min
- O(n*n) is bubble sort
- O(n*n*n) is solutions for a multi-variable equation: 3x+4y+5z = 108, then get all x, y z.
- O(n*logn) is quick sort. binary search.
- O(2^n) print all subsequences
- catalan number. print all parentheses.
- O($n!$) print all permutation.

Print all subsequences, why is power(2,n)?

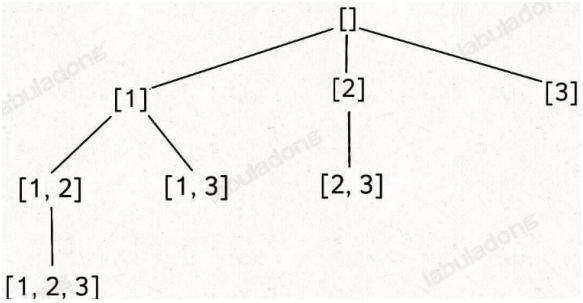
```
void sub(string& s, int i, int j, string res) {
  if (i == j) {
    cout << res << endl;
    return;
  }
  sub(s, i + 1, j, res); //don't select current character

  res.push_back(s[i]); //select current character.
  sub(s, i + 1, j, res);
}
```

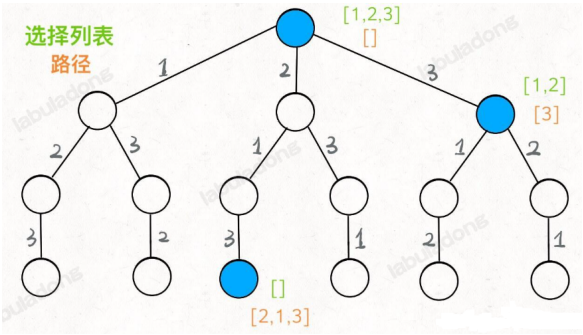
We can use another method to resolve this quesiton. That is bucket perspective, above code is ball perspective.

```
void backtrack(int[] nums, int start) {
  // pre-order, every node is a subset. such as, empty subset[]
  res.add(new LinkedList<>(track));

  for (int i = start; i < nums.length; i++) {
```

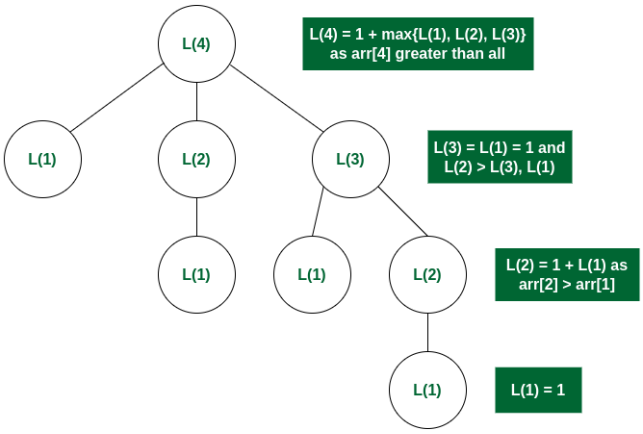
```
6       track.addLast(nums[i]); // make a choice
7
8       // Control the traversal of branches using the start parameter to avoid generating
            duplicate subsets.
9       backtrack(nums, i + 1);
10
11      track.removeLast(); // revoke a choice
12    }
13  }
```

In this tree, each edge is choice, each node is an answer, for n = 3, there are 8 nodes in below graph. We call it back track tree.





A similar idea is to LIS problem. it's also 2^n you can think in this way, previous one is 2*f(n-1), below is f(n-1) f(n-2)...f(1) but f(n-2)...f(1) is just another f(n-1), so equal 2*f(n-1). Detail is we have a lot of overlap problem. Time complexity is power(2,n) The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems as explained in the recursive tree diagram above.

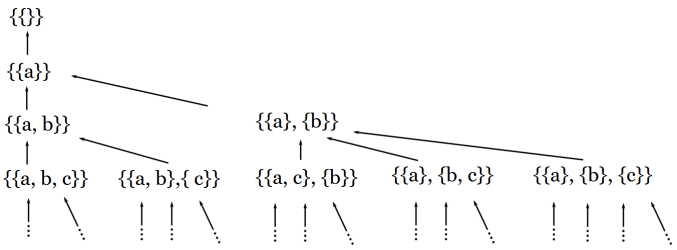



```
1   int _lis( int arr[], int n, int *max_ref){
2     if (n == 1) /* Base case */
3     return 1;
4
5     // 'max_ending_here' is length of LIS ending with arr[n-1]
6     int res, max_ending_here = 1;
7
8     /* Recursively get all LIS ending with arr[0], arr[1] ... arr[n-2]. If arr[i-1] is
          smaller
9     than arr[n-1], and max ending with arr[n-1] needs to be updated, then update it */
10    for (int i = 1; i < n; i++){
11      res = _lis(arr, i, max_ref);
12      if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
```

```
    max_ending_here = res + 1;
  }

  // Compare max_ending_here with the overall max. And update the overall max if needed
  if (*max_ref < max_ending_here)
    *max_ref = max_ending_here;

  // Return length of LIS ending with arr[n-1]
  return max_ending_here;
}
```

If the answer is one, such as LIS, the result should be in the root node, we should use post order, If the answer are a lot , the result shoudl be in leaf node, we should use base case. such as all possible subsequence.

Given a set A = 1, 2, 3, . . ., n . It is called a partition of the set A if the following conditions follow: 1) The union of all the sets is the set A. 2) The intersection of any two sets is an empty set.




```
// Function to print a partition
void printPartition(vector<vector<int>> ans)
{
  for (auto i : ans) {
    cout << "{ ";
      for (auto element : i) {
        cout << element << " ";
      }
      cout << "} ";
  }
  cout << endl;
}

// Function to generate all partitions
void Partition(vector<int> set, int index, vector<vector<int>>& ans){

  // If we have considered all elements in the set print the partition
  if (index == set.size()) {
    printPartition(ans);
    return;
  }

  // For each subset in the partition add the current element to it and recall
  for (int i = 0; i < ans.size(); i++) {
    ans[i].push_back(set[index]);
    Partition(set, index + 1, ans);
    ans[i].pop_back();
  }

  // Add the current element as a singleton subset and recall
  ans.push_back({ set[index] });
```

```
32    Partition(set, index + 1, ans);
33    ans.pop_back();
34  }
35
36  // Function to generate all
37  // partitions for a given set
38  void allPartitions(vector<int> set){
39    vector<vector<int> > v;
40    Partition(set, 0, v);
41  }
42
43
44  int main(){
45    int n = 5;
46    vector<int> set(n);
47    for (int i = 0; i < n; i++) {
48      set[i] = i + 1;
49    }
50
51    // Generate all partitions of the set
52    allPartitions(set);
53    return 0;
54  }
```

```
1  { 1 2 3 4 5 }
2  { 1 2 3 4 } { 5 }
3  ........
4  { 1 2 4 } { 3 5 }
5  { 1 } { 2 } { 3 } { 4 } { 5 }
```

Catalan number, all possible parentheses combination. (It's not good implementation, some child problems have been calcualted many time. You should use dynamic programming to remember the child problems result. but child program result is very big. It's a question for ALL , not for ONE result. So backtrace algorithm is more good for this questions.

Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, Cn counts the number of expressions containing n pairs of parentheses which are correctly matched:

```
1    ((()))      (()())      (())()      ()(())      ()()()
```

Cn is the number of different ways n + 1 factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator, as in the matrix chain multiplication problem). For n = 3, for example, we have the following five different parenthesizations of four factors:

```
1    ((ab)c)d     (a(bc))d     (ab)(cd)      a((bc)d)      a(b(cd))
```

```
1  vector<string> ap(string& s, int i, int j) {
2    vector<string> result;
3    if (i == j-1 && i<s.size()) {
4      result.push_back(s.substr(i,1));
5      return result;
6    }
7    for (int k = i; k < j-1; ++k) {
8      vector<string> v1 = ap(s, i, k+1);
9      vector<string> v2 = ap(s, k+1 , j);
10
11     string rs;
12     for (auto e : v1) {
13       for (auto e1 : v2) {
```

```
        rs = "(";
        rs +=  e;
        rs += e1;
        rs += ")";
        result.push_back(rs);
      }
    }
  }
  return result;
}

string s = "abcd";
auto com = ap(s, 0, 4);

for (auto e : com) {
  cout << e << endl;
}

(a(b(cd)))
(a((bc)d))
((ab)(cd))
((a(bc))d)
(((ab)c)d)
```

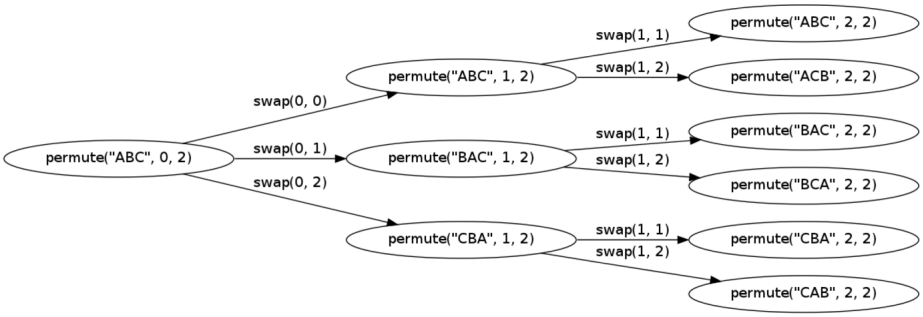
The same idea is to build binary search tree

```
// Recursive function to return a list of tree pointers of all possible
// binary trees having the same inorder sequence as `in[start, end]`
vector<Node*> generateBinaryTrees(vector<int> &in, int start, int end)
{
  // create an empty list to store the root of the constructed binary trees
  vector<Node*> trees;

  // base case
  if (start > end)
  {
    trees.push_back(nullptr);
    return trees;
  }

  // consider each element in the inorder sequence as the root
  for (int i = start; i <= end; i++)
  {
    // recursively find all possible left subtrees for root `i`
    vector<Node*> left_subtrees = generateBinaryTrees(in, start, i - 1);

    // recursively find all possible right subtrees for root `i`
    vector<Node*> right_subtrees = generateBinaryTrees(in, i + 1, end);

    // do for each combination of left and right subtrees
    for (Node* l: left_subtrees)
    {
      for (Node* r: right_subtrees)
      {
        // construct a binary tree with i'th element as the root and whose
        // left and right children point to `l` and `r`, respectively
        Node* tree = new Node(in[i], l, r);

        // add this tree to the output list
        trees.push_back(tree);
      }
```

```
    }
  }

  return trees;
}
```

```
void subset(int arr[], vector<int>& result, int k) {
  if (k == 6) {
    cout << "one answer is (";
    for (int i = 0; i < result.size(); i++) {
      cout << result[i] << " ";
    }
    cout << ")" << endl;
    return;
  }
  result.push_back(arr[k]);
  subset_bucket(arr, result, k + 1);
  result.pop_back();
  subset_bucket(arr, result, k + 1);
  return;
}
```

permutation, It is also worth noting that when you make a recursive call, you advance down an individual branch of the tree, and an additional branch is added with every iteration of a 'for' or 'while' loop. One confusing thing about this problem is the second swap after the recursive call to permute. This can be interpreted as 'unswap,' and is required because the char array is passed by reference, not by value, and every time you swap elements in the array the change is visible down stream.




```
void permute(char a[], int i, int n){
  int j;
  if (i == n)
  cout << a << endl;
  else    {
    for (j = i; j <= n; j++)        {
      swap(a[i], a[j]);
      permute(a, i+1, n);
      swap(a[i], a[j]);
    }
  }
}
```

compared with subset code, You will find that they are all recursive. permutation is n!

```
permute(i)
```

```
2 for(j...)  //calculate j times same size(i+1) sub problem.
3 permute(i+1)  //similar with n*f(n-1)
```

2.4 Sort

2.4.1 Kth element

There is a few options to implement Kth element:

- n*log(n) sort
- n*k partial select
- n*log(k) heap
- quick select, average O(n)

k small number, use maxium heap(default)

```
1  std::priority_queue<int> pq;
2  for(auto e: arr){
3    if(pq.size()<3){
4      pq.push(e);
5    }
6    else if(pq.top()>e){
7      pq.push(e);
8      pq.pop();
9    }
10 }
11
12 while(pq.size()>0){
13   cout<<pq.top()<<endl;
14   pq.pop();
15 }
```

Basic idea is **quick select**, but we use median of median method to get good pivot, to make at the worst 30-70 partition. Pay attention to the k value need to be adjusted when you call a function recursively.

- WE should use left close, and right close method to deal with Divide conquer array problem. It will make your code a little easier.
- k-1, index is based on 0, but k is based on 1, that is why we need k-1 in line 18

```
1  int partition(vector<int>& arr, int l, int r) {
2    int x = arr[r], i = l;
3    for (int j = l; j <= r - 1; j++) {
4      if (arr[j] <= x) {
5        swap(arr[i], arr[j]);
6        i++;
7      }
8    }
9    swap(arr[i], arr[r]);
10   return i;
11 }
12
13 int quickselect(vector<int>& A, int left, int right, int k){
14   //p is position of pivot in the partitioned array
15   int p = partition(A, left, right);
16
17   //k equals pivot got lucky
```

```
  if (p == k-1){
    return A[p];
  }
  //k less than pivot
  else if (k - 1 < p){
    return quickselect(A, left, p - 1, k);
  }
  //k greater than pivot
  else{
    return quickselect(A, p + 1, right, k);
  }
}

vector<int> A = {1,3,8,2,4,9,7};
int k = 5;
cout<<quickselect(A, 0, 6, 5); //output is 7
```

Stable_partition. The whole idea is just like merge sort, Pay attention here, merge sort is also stable sort.

```
#include <algorithm>

template <typename Iterator, typename Predicate>
Iterator my_stable_partition(Iterator first, Iterator last, Predicate predicate) {
  auto n = std::distance(first, last);
  if (n <= 1) {
    if (n == 1 && predicate(*first))
      ++first;
    return first;
  }
  auto middle = first;
  std::advance(middle, n / 2);
  auto a = my_stable_partition(first, middle, predicate);
  auto b = my_stable_partition(middle, last, predicate);
  return std::rotate(a, middle, b);
}
```

median of medians (advance part, skip it if you don't have enough time.)

```
int array[] = { 1,12,3,4,1,      5,2,7,8,88,      5,2,32,1,35,   -1,7,5,38,-11 };

int insertSort(int left, int right) {
  for (int i = left + 1; i <= right; i++) {
    int temp = array[i], j;
    for (j = i; j > left && array[j - 1] > temp; j--) array[j] = array[j - 1];
    array[j] = temp;
  }
  return (left + right) >> 1;
}

int BFPRT(int, int, int);

int getPivotIndex(int left, int right) {
  if (right - left < 5) return insertSort(left, right);
  int back = left - 1;
  for (int i = left; i + 4 < right; i += 5) {
    std::cout << "i" << i <<"␣"<< right<<std::endl;
    int index = insertSort(i, i + 4);
    std::swap(array[++back], array[index]);
  }
  return BFPRT(left, back, ((left + back) >> 1) + 1);
}


```

```
int partition(int left, int right, int pivotIndex) {
  std::swap(array[right], array[pivotIndex]);
  int mid = left;
  for (int i = left; i < right; i++) {
    if (array[i] < array[right])
    std::swap(array[i], array[mid++]);
  }
  std::swap(array[right], array[mid]);
  return mid;
}

int BFPRT(int left, int right, int k) {
  int pivotIndex = getPivotIndex(left, right);
  int mid = partition(left, right, pivotIndex);
  int count = mid - left + 1;
  if (count == k) {
    return mid;
  }
  else if (count > k) {
    return BFPRT(left, mid - 1, k);
  }
  else {
    return BFPRT(mid + 1, right, k - count);
  }
}

int main() {
  int k = 5;
  int length = sizeof(array) / sizeof(array[0]);
  for (int i = 0; i < length; i++) {
    std::cout << array[i] << "␣␣";
  }
  std::cout << std::endl << "␣" << k << "␣";
  std::cout << array[BFPRT(0, length - 1, k)] << std::endl;
  return 0;
}
```

2.4.2 Simple sort

All sort algorithm don't compare equal relationship. So some algorithm is Stable, and other is not stable. Bubble is simple source code stable sorting. All the sorting is from minimum to maximum default.

for all sort comparsion, there is good ref website:
https://www.toptal.com/developers/sorting-algorithms/selection-sort

Selection sort is an in-place comparison sort. O(n2). The algorithm finds the minimum value, swaps it with the value in the first position. And repeat From the comparisons presented here, one might conclude that selection sort should never be used. It does not adapt to the data in any way (notice that the four animations above run in lock step), so its runtime is always quadratic. However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice. Selection sort only need O(n) write, so if write is expensive operation, you need to use it.

```
For(int size=n ;size>1;size--){
  Int j = Max(a,size);
  Swap(a[j],a[size-1]);
}
```

Insertion sort is suitable for small list and mostly sorted list. Although it is one of the elementary sorting algorithms with O(n2) worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low

overhead). For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort. Insertion sort is also stable sort.

```
For(int I = 0;i<n;i++){
Int iv = a[i];
Insert(a,I, iv);
}
Insert(a[],int n,int lv){
For(int I = n-1;i>=0&&lv<a[i];i--)
    A[i+1]=a[i]  //move each element from tail to head one by one
A[i]  = lv;  //insert here.
}
```

Bubble sort is stable. Bubble sort can stops after reaching a sorted array. In the best case (already sorted), every insert requires constant time. So Bubble and insert can reach O(n) time in Best context. Bubble sort has many of the same properties as insertion sort, but has slightly higher overhead. In the case of nearly sorted data, bubble sort takes O(n) time, but requires at least 2 passes through the data (whereas insertion sort requires something more like 1 pass).

```
For (int I = n;i>0;i--){
  For(int j= 0;j<I;j++)
    If(a[j]<a[j+1]) swap(a[j],a[j+1]);
}
```

In insertion sort elements are bubbled into the sorted section, while in bubble sort the maximums are bubbled out of the unsorted section.

```
  //insert sort
  sorted   | unsorted
  1 3 5 8 | 4 6 7 9 2
  1 3 4 5 8 | 6 7 9 2

  bubble sort
  unsorted   | biggest
  3 1 5 4 2 | 6 7 8 9
  1 3 4 2 | 5 6 7 8 9
```

Difference between insertion sort and selection sort? The real question is the speed of comparison vs. copying. The time a selection sort will win is when a comparison is a lot faster than copying. Just for example, let's assume two fields: a single int as a key, and another megabyte of data attached to it. In such a case, comparisons involve only that single int, so it's really fast, but copying involves the entire megabyte, so it's almost certainly quite a bit slower. Since the selection sort does a lot of comparisons, but relatively few copies, this sort of situation will favor it. The insertion sort does a lot more copies, so in a situation like this, the slower copies will slow it down quite a bit.

As far as worst case for a insertion sort, it'll be pretty much the opposite – anything where copying is fast, but comparison is slow. There are a few more cases that favor insertion as well, such as when elements might be slightly scrambled, but each is still within a short distance of its final location when sorted. In another word, insertion sort is more efficient than selection sort when the input array is partially sorted or almost sorted.

If the data doesn't provide a solid indication in either direction, chances are pretty decent that insertion sort will work out better.

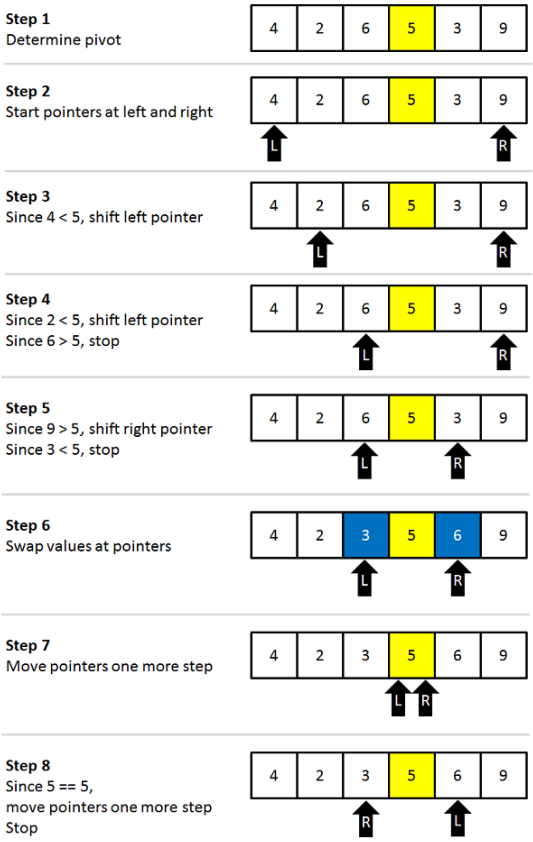
2.4.3 quick sort

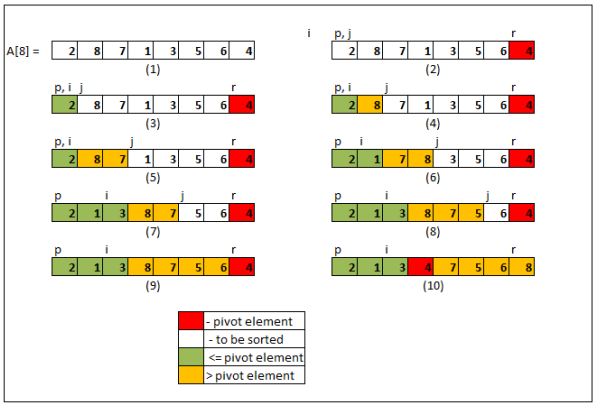
2.4.3.1 Quick sort

Three quick heapSort, MergeSort, quickSort. The time complexity are all (Olog(n)). MergeSort need O(N) extra space, When you merge two lists, this shortcoming can be avoid.

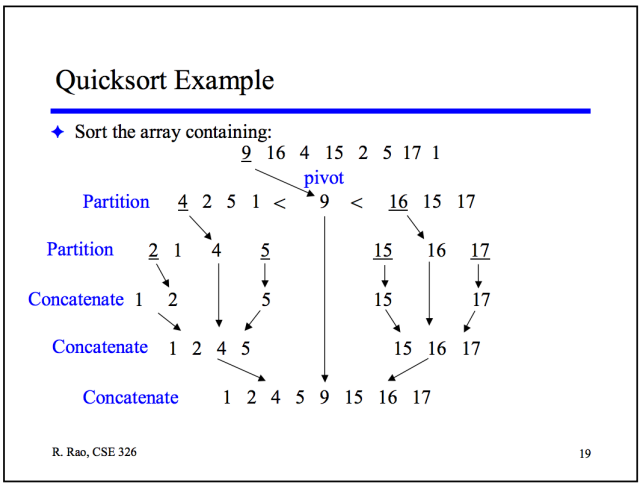
partition

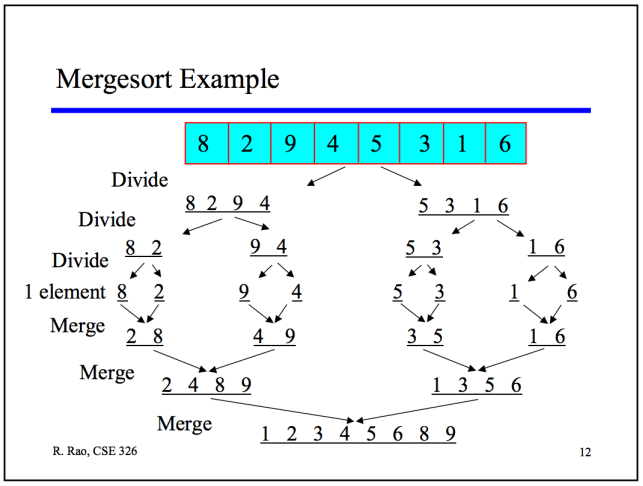
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */

partition (arr[], low, high){
    // pivot (Element to be placed at right position)
    pivot = arr[high];
    i = (low - 1)  // Index of smaller element
    for (j = low; j <= high- 1; j++)    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```





Quick sort need random access ability.





2.4.3.2 Merge sort

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible. MergeSort(headRef)

```
1) If head is NULL or there is only one element in the Linked List
then return.
2) Else divide the linked list into two halves.
FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
MergeSort(a);
MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
and update the head pointer using headRef.
*headRef = SortedMerge(a, b);
```

Idea: 1) No random access 2) recursive

```
/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a, b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL)){
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}
```

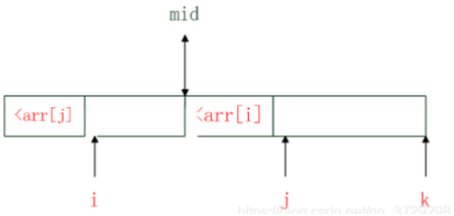
```
/* See http://geeksforgeeks.org/?p=3622 for details of this function */
struct node* SortedMerge(struct node* a, struct node* b){
  struct node* result = NULL;
  /* Base cases */
  if (a == NULL)
  return(b);
  else if (b==NULL)
  return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data){
    result = a;
    result->next = SortedMerge(a->next, b);
  }
  else{
    result = b;
    result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

Source code: Idea: recursive

Inplace merge

```
void swapNums(vector<int>&ivNums, int lo, int hi){
  --hi;
  while (lo<hi){
    swap(ivNums[lo++], ivNums[hi--]);
  }
}

void MergeInPlace(vector<int>&ivNums, int lo, int mid, int hi){
  int i = lo, j = mid;
  while (i<mid && j<hi){ //stop when the first or latter half is empty
    while (i < j&&ivNums[i] <= ivNums[j]){
      ++i;
    }
```

```
14      while (j<hi&&ivNums[i] > ivNums[j]){
15        ++j;
16      }
17      //just std::rotate
18      swapNums(ivNums, i, j);
19      swapNums(ivNums, i, i + j - mid);
20      swapNums(ivNums, i + j - mid, j);
21
22      //reset the boundary and continue.
23      i = i + j - mid;
24      mid = j;
25    }
26  }
```



2.4.3.3 heap sort

The most import thing is heapify, 0then push the maxium element to the 0 index, then swap to the last position, then heapify the (0, len-1), then (0, len-2)....

```
1  // To heapify a subtree rooted with node i which is
2  // an index in arr[]. n is size of heap
3  void heapify(int arr[], int n, int i)
4  {
5    int largest = i; // Initialize largest as root Since we are using 0 based indexing
6    int l = 2 * i + 1; // left = 2*i + 1
7    int r = 2 * i + 2; // right = 2*i + 2
8
9    // If left child is larger than root
10   if (l < n && arr[l] > arr[largest])
11   largest = l;
12
13   // If right child is larger than largest so far
14   if (r < n && arr[r] > arr[largest])
15   largest = r;
16
17   // If largest is not root
18   if (largest != i) {
19     swap(arr[i], arr[largest]);
20
21     // Recursively heapify the affected sub-tree
22     heapify(arr, n, largest);
23   }
24 }
25
26 // main function to do heap sort
27 void heapSort(int arr[], int n)
28 {
29   // Build heap (rearrange array)
30   for (int i = n / 2 - 1; i >= 0; i--)
```

```
31    heapify(arr, n, i);
32
33    // One by one extract an element from heap
34    for (int i = n - 1; i >= 0; i--) {
35      // Move current root to end
36      swap(arr[0], arr[i]);
37
38      // call max heapify on the reduced heap
39      heapify(arr, i, 0);
40    }
41  }
```

2.4.3.4 introsort

This is the method used in STL.

```
1  // A Utility function to perform intro sort
2  void IntrosortUtil(int arr[], int * begin, int * end, int depthLimit){
3    // Count the number of elements
4    int size = end - begin;
5
6    // If partition size is low then do insertion sort
7    if (size < 16){
8      InsertionSort(arr, begin, end);
9      return;
10   }
11
12   // If the depth is zero use heapsort
13   if (depthLimit == 0){
14     make_heap(begin, end+1);
15     sort_heap(begin, end+1);
16     return;
17   }
18
19   // Else use a median-of-three concept to
20   // find a good pivot
21   int * pivot = MedianOfThree(begin, begin+size/2, end);
22
23   // Swap the values pointed by the two pointers
24   swapValue(pivot, end);
25
26   // Perform Quick Sort
27   int * partitionPoint = Partition(arr, begin-arr, end-arr);
28   IntrosortUtil(arr, begin, partitionPoint-1, depthLimit - 1);
29   IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1);
30
31   return;
32 }
```

2.4.4 Bucket and radix sort

Bucket sort is not comparison algorithm. we must know the maximum value in the unsorted array and how many objects in the unsorted array. If maximum value is not big, and objects is a lot. such as all student score (<100) in a country(1000000). At this time, we can use Bucket sort.

Firstly, we must know how to handle duplicates. Secondly, Thirdly, we must have enough memory. If you sort telephone which will not allow duplicated value, you can build an bit array. And set bit 1

if there are number exist. It will save a lot of memories. For duplicate, you can store a link in each bucket. (array item)

Radix sort is cousin of Bucksort. BucketSort is more efficient for 'Dense' arrays, while RadixSort can handle sparse (well, not exactly sparse, but spaced-out) arrays well. Use % operator to get radix each digit.

$1 \sim n^c - 1$, such as $1 \sim 10^3 - 1$ You can use 10 as radix, and sort it 3 times.

2.4.4.1 Bucket sort

Basic step: Here, Link list is best option to implement a bucket. Because Nobody will now how many elements will be put into the each bucket.

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.(Optional)
4. Gather: Visit the buckets in order and put all elements back into the original array.

2.4.4.2 Radix sort

r is radix, $x\%r, (x\%r^2)/r, (x\%r^3)/r^2$ get each digit from least significant digit (LSD) radix sorts to most significant digit.

```
int i ;
int k;
while(i%10){
  k = i%10;
  i = i/10;
}
```

1. according to r, prepare two arrays buckets A, B.
2. Sort according to one position and put them into A array buckets.
3. loop through A array buckets, and get next position digit. and then according to its value put into B array buckets.
4. swap A,B two array, then goto step 2.

Difference between radix sort and quick sort. Quicksort/Introsort is more flexible:Quicksort and Introsort work well with all kinds of data. All you need for sorting is the possibility to compare items. This is trivial with numbers but you can sort other data as well. Radix sort on the other hand just sorts things by their binary representation. It never compares items against each other. Radix sort needs more memory. All radix sort implementations that I've seen use a secondary buffer to store partial sorting results. This increases the memory requirements of the sorting algorithm. That may not be a problem if you only sort a couple of kilobytes, but if you go into the gigabyte range it makes a huge difference. If I remember right a in place radix-sort algorithm exist on paper though.

```
void countSort(int arr[], int n, int exp){

  int output[n];  // Output array
  int i, count[10] = { 0 };

  // Store count of occurrences in count[]
  for (i = 0; i < n; i++)
  count[(arr[i] / exp) % 10]++;

```

```
10    // Change count[i] so that count[i] now contains actual position
11    // of this digit in output[]
12    for (i = 1; i < 10; i++)
13    count[i] += count[i - 1];
14
15    // Build the output array
16    for (i = n - 1; i >= 0; i--) {
17      output[count[(arr[i] / exp) % 10] - 1] = arr[i];
18      count[(arr[i] / exp) % 10]--;
19    }
20
21    // Copy the output array to arr[], so that arr[] now contains sorted
22    // numbers according to current digit
23    for (i = 0; i < n; i++)
24    arr[i] = output[i];
25  }
26
27  // The main function to that sorts arr[] of size n using Radix Sort
28  void radixsort(int arr[], int n){
29
30    // Find the maximum number to know number of digits
31    int m = getMax(arr, n);
32
33    // Do counting sort for every digit. Note that instead of passing digit
34    // number, exp is passed. exp is 10^i where i is current digit number
35    for (int exp = 1; m / exp > 0; exp *= 10)
36      countSort(arr, n, exp);
37  }
```

In order to understand previous code, you need to understand counting sort, Detail can be found here: //

https://www.geeksforgeeks.org/counting-sort/ // it use prefix sum(std::partial_sum). it's also interesting application of std::partial_sum, you need to remember it.

2.4.5 Summary

You can compare sorting algorithms against the following criteria:

- Time Complexity (Big-O notation). You should note that best-case, worst-case and average runtime can have different time complexity. For example best-case for Bubble Sort is only O(n), making it faster than Selection Sort when the original list is mostly in order (not many elements out of place).
- Memory Complexity. How much more memory is required to sort a list as n grows?
- Stability. Does the sort preserve the relative ordering of elements that have equivalent sort values? (For example if you were sorting a list of catalog items by their price, some elements may have equal prices. If the catalog was originally sorted alphabetically by item name, will the chosen sort algortihm preserve the alphabetical ordering within each group of equal-priced items.)
- Best/Worst/Averavge number of comparisons required. Important when compare operations are expensive. (For example: comparing efficiencies of alternative designs where efficiency is calculated via some simulation or otherwise complex calculation).
- Best/Worst/Average number of swap operations required. Important when swap operations are expensive. (For example: sorting shipping containers that must be physically moved on the deck of a ship)

- Code size. Bubble-sort is known for its small code footprint.

2.5 Greedy

Idea of Greedy:

- Solve a problem by making a sequence of decisions.
- Decisions are made one by one in some order.
- Each decision is made using a greedy criterion. At each stage we make a decision that appears to be the best at the time.
- A decision, once made, is (usually) not changed later.

You need a **greedy criterion** to make a local decision. For examples: making change, 0/1 knappack, activity selection (largest subset) sorted according to their finishing time, topological orders. Dijkstra, Kruskal, prim and collin.

Applications: Container Loading, 0/1 knapsack problem, Topological sorting, Bipartite cover, Single-source shortest paths, Minimum-cost spanning trees.

2.5.0.1 Intervals questions

two basic tricks, sort, then do something when it overlap.

Meeting room questions.

1. Minimum empty time. That is 0/1 knapback problem.
2. Two group collision intervals.

Overlap (three questions):

1. Non-overlap number. one meeting room, maximum meeting number. Sort by end time, delete(ignore) if overlap.

```
public int intervalSchedule(int[][] intvs) {
  if (intvs.length == 0) return 0;
  // sort
  Arrays.sort(intvs, new Comparator<int[]>() {
    public int compare(int[] a, int[] b) {
      return a[1] - b[1];
    }
  });

  int count = 1;
  int x_end = intvs[0][1];
  for (int[] interval : intvs) {
    int start = interval[0];
    if (start >= x_end) {
      // find next non-overlap interval,
      //update x_end.
      count++;
      x_end = interval[1];
    }
  }
  return count;
}
```

2. Maximum overlap number. Minimum meeting room numbers. sort by start time. you can use scan line.
   - Sort the intervals by their start times.
   - Use a min-heap to keep track of the end times of meetings currently using a room.
   - Iterate through the sorted intervals and for each interval:
     (a) If the room due to free up the earliest is free, remove it from the heap.
     (b) Add the current meeting's end time to the heap.
     (c) The size of the heap at any point will give the number of rooms required at that time.

```cpp
// Function to find the minimum number of meeting rooms required
int minMeetingRooms(vector<vector<int>>& intervals) {
  if (intervals.empty()) return 0;

  // Sort the intervals by start time
  sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<
      int>& b) {
    return a[0] < b[0];
  });

  // Min-heap to keep track of end times
  priority_queue<int, vector<int>, greater<int>> minHeap;

  // Add the end time of the first meeting
  minHeap.push(intervals[0][1]);

  for (int i = 1; i < intervals.size(); ++i) {
    // If the room due to free up the earliest is free, remove it from the heap
    if (intervals[i][0] >= minHeap.top()) {
      minHeap.pop();
    }
    // Add the current meeting's end time to the heap
    minHeap.push(intervals[i][1]);
  }

  // The size of the heap is the number of rooms required
  return minHeap.size();
}

int main() {
  vector<vector<int>> intervals = {{5, 10}, {0, 30}, {15, 20}};
  cout << "Minimum number of meeting rooms required: " << minMeetingRooms(
      intervals) << endl;
  return 0;
}
```

3. Merge overlap. sort by left. If overlap, then update right point; if non-overlap, find a new current intervals.

```cpp
vector<vector<int>> mergeOverlap(vector<vector<int>>& arr) {

  // Sort intervals based on start values
  sort(arr.begin(), arr.end());

  vector<vector<int>> res;
  res.push_back(arr[0]);

  for (int i = 1; i < arr.size(); i++) {
    vector<int>& last = res.back();
    vector<int>& curr = arr[i];
```

```
12
13      // If current interval overlaps with the last merged
14      // interval, merge them
15      if (curr[0] <= last[1])
16      last[1] = max(last[1], curr[1]);
17      else
18      res.push_back(curr);
19    }
20
21    return res;
22  }
23
24  int main() {
25    vector<vector<int>> arr = {{7, 8}, {1, 5}, {2, 4}, {4, 6}};
26    vector<vector<int>> res = mergeOverlap(arr);
27
28    for (vector<int>& interval: res)
29    cout << interval[0] << " " << interval[1] << endl;
30
31    return 0;
32  }
```

Interval cover. (two questions)

1. Minimum cover

```
1  int videoStitching(int[][] clips, int T) {
2    if (T == 0) return 0;
3    // Sort in ascending order by the start point, and in descending order if the
         start points are the same.
4    Arrays.sort(clips, (a, b) -> {
5      if (a[0] == b[0]) {
6        return b[1] - a[1];
7      }
8      return a[0] - b[0];
9    });
10   // Keep track of the number of selected short videos
11   int res = 0;
12
13   int curEnd = 0, nextEnd = 0;
14   int i = 0, n = clips.length;
15   while (i < n && clips[i][0] <= curEnd) {
16     // Greedily select the next video within the interval of the res-th video.
17     while (i < n && clips[i][0] <= curEnd) {
18       nextEnd = Math.max(nextEnd, clips[i][1]);
19       i++;
20     }
21     // Find the next video and update curEnd.
22     res++;
23     curEnd = nextEnd;
24     if (curEnd >= T) {
25       // The interval [0, T] can already be assembled.
26       return res;
27     }
28   }
29   // Unable to continuously assemble the interval [0, T].
30   return -1;
31 }
```

2. Erase cover. Once find overlap, update right, find non-overlap, update left and right.

```
int removeCoveredIntervals(int[][] intvs) {
  // Sort in ascending order by the start point, and in descending order if the
      start points are the same.
  Arrays.sort(intvs, (a, b) -> {
    if (a[0] == b[0]) {
      return b[1] - a[1];
    }
    return a[0] - b[0];
  });

  // Record the start and end points of the merged intervals.
  int left = intvs[0][0];
  int right = intvs[0][1];

  int res = 0;
  for (int i = 1; i < intvs.length; i++) {
    int[] intv = intvs[i];
    // Case 1: Find the covering intervals.
    if (left <= intv[0] && right >= intv[1]) {
      res++;
    }
    // Case 2: Find the overlapping intervals and merge them.
    if (right >= intv[0] && right <= intv[1]) {
      right = intv[1];
    }
    // Case 3: Completely non-overlapping, update the start and end points.
    if (right < intv[0]) {
      left = intv[0];
      right = intv[1];
    }
  }

  return intvs.length - res;
}
```

2.5.1 Application

Machine Scheduling, **sort all task according to start time, and minheap of machine end time.**

Container Loading and 0/1 knapsack problem are different. For 0/1 knapsack problem, we consider the value and weight together, For greed algorithm, it change time complexity from $2^n$to $n.log(n)$, but 583/600 is withing 10%, so it's very practical algorithm. Don't underestimate it.

Topological sorting:

```
1) Select any one among vertices having no incoming edge
2) Put the node into the solution &  Remove the node and its outgoing edges from the graph
4) Repeat the above steps until no nodes remain

You need to InDegree[n] array to save and update all the vetex InDegree, and stack to store all
```

Bipartite-cover problems are NP-hard

```
A greedy method to develop a fast heuristic
Construct the cover A' in stages
```

```
Select a vertex of A  using the greedy criterion:
Select a vertex of A that covers the largest # of uncovered vertices of B
```

Dijkstra algorithm.

```
1:  function Dijkstra(Graph, source):
2:      for each vertex v in Graph:                 // Initialization
3:          dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
4:          previous[v] := undefined                // Previous node in optimal path from source
5:      dist[source] := 0                           // Distance from source to source
6:      Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
7:      while Q is not empty:                       // main loop
8:          u := node in Q with smallest dist[ ]
9:          remove u from Q
10:         for each neighbor v of u:               // where v has not yet been removed from Q.
11:             alt := dist[u] + dist_between(u, v)
12:             if alt < dist[v]                    // Relax (u,v)
13:                 dist[v] := alt
14:                 previous[v] := u
15:     return previous[ ]
```

- The greedy idea lies in line 8
- In line 8, it must shortest path to this vetex, Because if it's not, There is shorter path, then, according to we select min in line 8, shorter one will be selected first, and this value HAS BEEN updated before.
- $O(n^2)$
- the shortest value is saved, it's also idea of dynamic programming.
- Just like vertabi, you can think that they are the same.

2.6 Divide Conquer

2.6.1 Binary search

A binary search (or divide-and-conquer) idea can be applied to the problem of finding the closest pair of points on a plane. By utilizing a subproblem, you can prune some candidate points. Another application is using merge sort to count the number of smaller elements to the right of each element, which is introduced by labuladong. The fundamental idea is still binary search, but the specifics come from the fact that merge sort maintains order, and this characteristic can be used to count the number of smaller elements on the right. This is a rather unique application.

The key idea here is leveraging the sorting order inherent in merge sort to solve a problem related to counting elements, and it also shows the pruning advantage in divide-and-conquer algorithms.

Binary search.

```
1  int binarySearch(int arr[],int target, int i, int j) {
2    int left = i;
3    int right = j;
4    while ( left<= right) {
5      int mid = (left+right)/2;
6      if (arr[mid] > target)
7        right = mid - 1;
8      else if (arr[mid] < target)
9        left = mid + 1;
10     else if (arr[mid] == target)
```

```
11        return mid;
12    }
13    return -1;
14  }
```

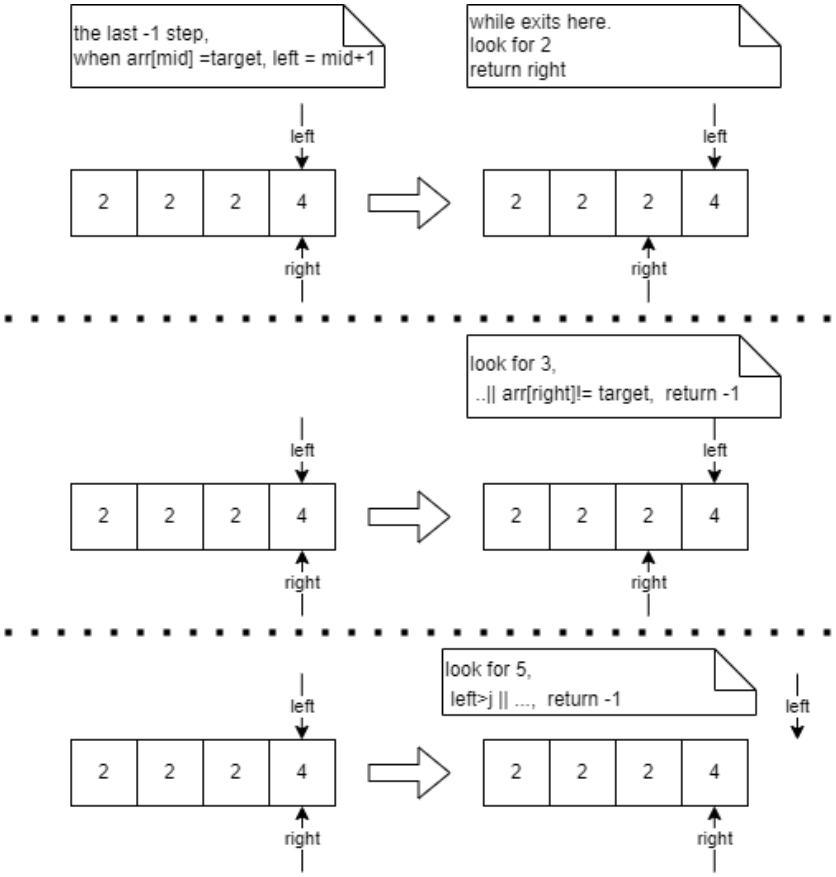
```
1  int left_bound(int arr[],int target, int i, int j) {
2    int left = i;
3    int right = j;
4    while ( left<= right) {
5      int mid = (left+right)/2;
6      if (arr[mid] > target)
7        right = mid - 1;
8      else if (arr[mid] < target)
9        left = mid + 1;
10     else if (arr[mid] == target)
11       right = mid-1;  //1) reduce to right in order to find left bound
12   }
13
14   if (left > j || arr[left] != target)
15     return -1;  //3) in order to return left, left will go to the right boundary,
16   return left; //2) return left.
17 }
```

```
1  int right_bound(int arr[],int target, int i, int j) {
2    int left = i;
3    int right = j;
4    while ( left<= right) {
5      int mid = (left+right)/2;
6      if (arr[mid] > target)
7        right = mid - 1;
8      else if (arr[mid] < target)
9        left = mid + 1;
10     else if (arr[mid] == target)
11       left = mid+1;  //1) step left to find right bound
12   }
13
14   if (right < 0 || arr[right] != target)
15     return -1;  //3) in order to return right, right will be too small
16   return right; //2) return right.
17 }
```

Three methods summary are below:

- close left and right
- left bound reduce right, right bount add left;
- left bound return left, right bound return right.
- left may be too big, right maybe too small.
- if no found,

```
1 //if we want to find 4, then right point to 3, left point to 5
2 //when we quit while loop.
3 //so left_bount return the 5 (smallest bigger than target),
4 //right bount return 3( biggest smaller than target.)
5 right left
6   3    5
```



Right boundary logic can be used to found square root.

```
int mySqrt(int x) {
  int l = 0;
  int r = x/2;
  while(l<=r){
    int mid = l + (r-l)/2;
    int temp = mid*mid;
    if(temp<x){
      l = mid+1;
    }
    else if(temp>x){
      r = mid-1;
    }
    else if(temp == x){
      r = mid;
      break;
    }
  }
  return r; //round down value, for 8, result is 2
  return l; //round up value, for 8, result is 3
}
-------
l
2 3     // (l+r)/2 is till l
  r
-------
  l
2 3    //so ++l,
  r
```

```
29 ------
30    l
31 2  3    //mid>x, so --r; l<r, while end.
32 r       //at this time l points 3, r points 2.
```

2.6.2 Other

An element in a sorted array can be found in O(log n) time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.

The basic idea is DC(divided conquer.)

1. Use DC, The first part is end condition. For this question, the end atomic condition is e-b == 1(only two elements in it)
2. If use array, better use index directly.
3. When you have last atomic condition, you need to decide boundary b, (e-b)/2, e. **When you decide boundary , you have to make half part of contents still includes answer. That is very useful hint for you to decide boundary value**

```
1  size_t find_rot(const vector<int> &vi, int b, int e){
2    if (e-b ==1){
3      return b;
4    }
5    int pivot = b + (e-b)/2;
6    cout<<"pivot"<<pivot<<endl;
7    if (vi[b]<vi[pivot]){
8      return find_rot(vi, pivot, e);
9    }
10   else{
11     return find_rot(vi, b, pivot);
12   }
13 }
```

label If you can divide it according certain condition, then you can use DC

2.7 Backtracking

2.7.1 Basic

Backtracking is particularly useful for solving problems where you need to explore all possible solutions, but you can prune certain paths that are unlikely to lead to a valid solution (i.e., "backtrack" when a partial solution cannot be completed). Problems suitable for backtracking typically involve making a sequence of decisions and can be described as "constraint satisfaction" problems.

Here are common types of questions suitable for backtracking:

- Combinatorial Problems Permutations and Combinations: Finding all permutations or combinations of a set of numbers or characters. Example: Generate all permutations of a given array of numbers. Subsets: Generating all subsets of a set. Example: Finding all subsets of a given set of numbers (power set).
- Constraint Satisfaction Problems (CSPs) Sudoku Solver: Filling in numbers on a Sudoku board while satisfying the row, column, and grid constraints. N-Queens Problem: Placing N queens on

an N×N chessboard so that no two queens attack each other. Crossword Puzzle Filler: Filling a crossword grid while ensuring valid word placements.

- Graph and Tree Search Problems Maze Solver: Finding a path from start to finish in a maze. Hamiltonian Path/Cycle: Finding a path or cycle in a graph that visits every vertex exactly once. Word Search: Searching for words in a grid of characters by moving horizontally, vertically, or diagonally.
- Partitioning Problems Partition to K Equal Sum Subsets: Partitioning an array into K subsets such that the sum of elements in each subset is equal. Palindrome Partitioning: Partitioning a string into substrings, where each substring is a palindrome.
- Combinatorial Optimization Problems Knapsack Problem: Finding combinations of items that maximize value without exceeding the capacity of a knapsack (although dynamic programming is often more efficient for this problem). Job Scheduling: Scheduling jobs on a limited number of resources (machines) without conflicts.
- String Matching Regular Expression Matching: Checking if a string matches a given pattern with wildcard characters.
- Puzzle Solving Crossword or Word Puzzle: Solving word puzzles by filling blanks with valid words from a dictionary. Solving Puzzles like the 8-Puzzle: Rearranging pieces on a grid to reach a specific target configuration.

In all these cases, backtracking explores the possible decisions recursively but abandons a path (backtracks) if it becomes clear that it cannot yield a valid solution, thus reducing unnecessary computation.

The basic backtracking code

```
result = []
def backtrack(track, select_list):
  if end_condition:
    result.add( track)
    return

  for option in select_list:
    select one
    backtrack(track, select_list)
    revert the select
```

If we don't want to return all the result, but one, we can change it to below code.

```
result = []
bool backtrack(track, select_list):
  if end_condition:
    result.add(track)
    return true;

  for option in select_list:
    select one
    if(backtrack(track, select_list))
      return true;
    revert the select
```

Line9-10 This if will stop the recursive and return the true direct to the root.

If we want to remember all the result, we need to List<List>. List inside remember a track, List outside remember all the valid resule.

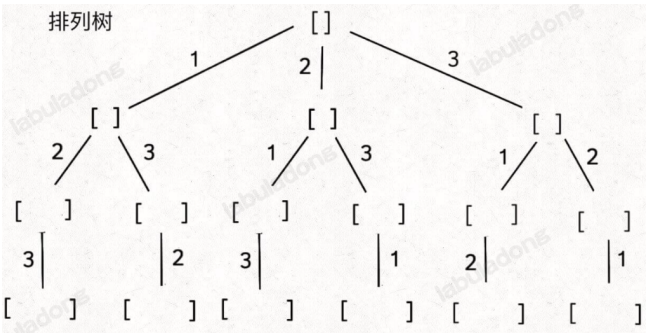
```
List<List<Integer>> res = new LinkedList<>();
LinkedList<Integer> track = new LinkedList<>(); //remember recursive path

```

```
4  public List<List<Integer>> subsets(int[] nums) { // main function
5    backtrack(nums, 0);
6    return res;
7  }
8
9  void backtrack(int[] nums, int start) {
10   res.add(new LinkedList<>(track));
11   for (int i = start; i < nums.length; i++) { // main frame
12     track.addLast(nums[i]); // select
13     backtrack(nums, i + 1); //  use start to trim
14     track.removeLast(); // unselect
15   }
16 }
```

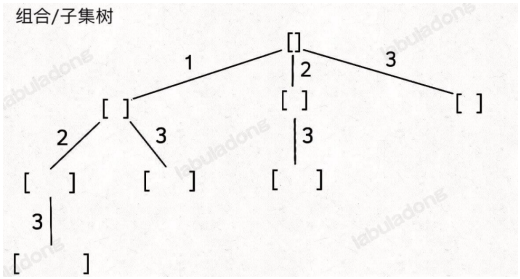
2.7.2 Permutation, combination and subset

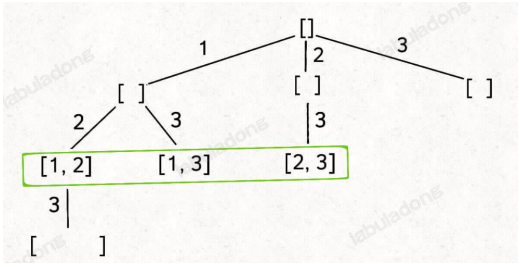
2.7.2.1 No duplicate, single select

Two important tree structures.



Combination can be seen as a child question of subset. permutation n!, subset is 2^n, combination is c(n, k). six elements. $5! = 120$, $2^5 = 32$. $c(5, 2) = 10$.





Subset problem.

1. Each node(from current node to root,including root(empty set)) is an answer

2. Don't need base case here. when i>=len, for will return without any execution.
3. Refer the combination tree figure to help to understand this code

```
void sub(int arr[], int len, int start, vector<int>& result) {
  cout << "one answer is ( ";
  for (int i = 0; i < result.size(); i++) {
    cout << result[i] << " ";
  }
  cout << ")" << endl;

  for (int i = start; i < len; ++i) {
    result.push_back(arr[i]);
    sub(arr, len, i + 1, result);
    result.pop_back();
  }
}

sub(arr, 3, 0, result)
```

Combination problem, the same idea as subset, just add a parameter k. When one answer length is k, print it out.

```
void com(int arr[], int len, int start, int k, vector<int>& result){
  if (result.size() == k) {
    cout << "one answer is ( ";
    for (int i = 0; i < result.size(); i++) {
      cout << result[i] << " ";
    }
    cout << ")" << endl;
  }

  for (int i = start; i < len; ++i) {
    result.push_back(arr[i]);
    com(arr, len, i + 1, k, result);
    result.pop_back();
  }
}
```

Permutation problem.

- Base case is size == len
- You must use used array here to remember which element you have used.

```
void per(int arr[], int len, int used[], vector<int>& result) {
  if (result.size() == len) {
    cout << "one answer is ( ";
    for (int i = 0; i < result.size(); i++) {
      cout << result[i] << " ";
    }
    cout << ")" << endl;
  }
  for (int i = 0; i < len; ++i) {
    if (used[i] == 1)
      continue;
    result.push_back(arr[i]);
    used[i] = 1;
    per(arr, len, used, result);
    result.pop_back();
    used[i] = 0;
  }
```

```
}
```

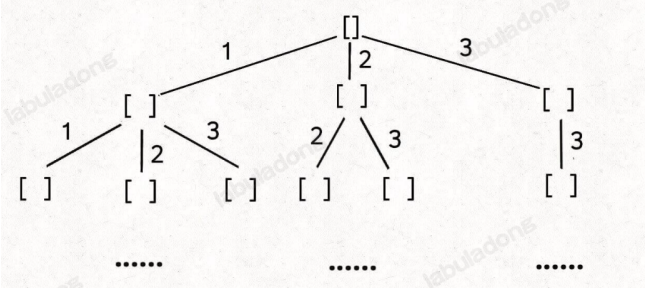
For permutation problem, there is another easy implementation. You don't need extra result vector, but you have to use swap. They use the same basic backtrack idea.

```
//all permutation. here there is a trick, we don't need track, but the basic idea is
    the same.
void permute(char a[], int i, int n){
  int j;
  if (i == n){  //end_condition,
    cout << a << endl;
    return;
  }

  for (j = i; j <= n; j++){  // all options in select_list
    swap(a[i], a[j]);      //select one
    permute(a, i+1, n);
    swap(a[i], a[j]);     //revoke select.
  }
}
```

2.7.2.2 No duplicate, multi select

The basic idea is just like before, for combination, we use i+1 to call recursion implementation single select, if you can multi select, just use i here. In order to avoid dead loop, this questions need a target to end loop, A typical question is coin problem.

```
void sub_multi_select(int arr[], int len, int start, vector<int>& result, int target) {
  auto sum = std::accumulate(result.begin(), result.end(), 0);
  if (sum > target)
  return;
  if (sum == target) {
    cout << "one answer is (";
    for (int i = 0; i < result.size(); i++) {
      cout << result[i] << " ";
    }
    cout << ")" << endl;
  }

  for (int i = start; i < len; ++i) {
    result.push_back(arr[i]);
    sub_multi_select(arr, len, i, result, target);  //Pay attention here, not pass i+1,
        but pass i.
    result.pop_back();
  }
}

int arr[3] = { 1,2,5};
vector<int> result;
sub_multi_select(arr, 3, 0, result, 10);

one answer is ( 1 1 1 1 1 1 1 1 1 1 )  //Ten result
......
one answer is ( 1 1 2 2 2 2 )
one answer is ( 1 2 2 5 )
one answer is ( 2 2 2 2 2 )
one answer is ( 5 5 )
```

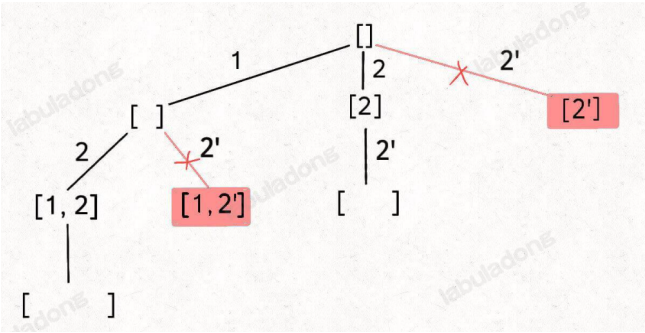


multi select permutation, Code is very easy, don't use used array. The time complexity is power(n, n). That is the only question which has so high time complexity.

2.7.2.3 Duplicate, single select

For combination, need to sort, then trim(prune) a branch which doesn't meet condition.

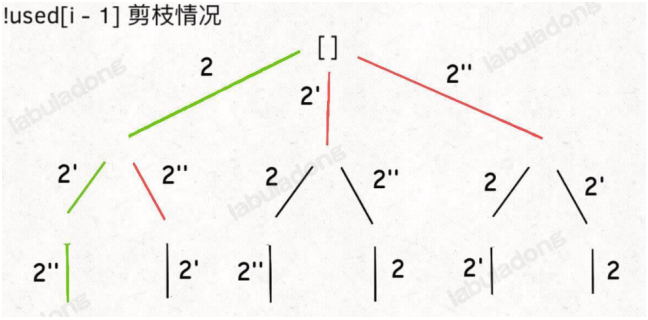
```
....
for (int i = start; i < len; ++i) {
  if(i>start && arr[i] == arr[i-1]) // trimming happens in these two
  continue;                         // statements, so we need to sort first.
  result.push_back(arr[i]);
  sub(arr, len, i + 1, result);
  result.pop_back();
}
....
```



For permutation, need to sort, then trim (prune) a branch which doesn't meet condition.

```
  ....
  for (int i = start; i < len; ++i) {
    if(i>start && arr[i] == arr[i-1] && !used[i-1]) // trimming happens in these two
      continue;                                     // statements, so we need to sort first.
    result.push_back(arr[i]);
    sub(arr, len, i + 1, result);
    result.pop_back();
  }
  ....
```

How to understand `!used[i-1]`?



2.7.2.4 Parenthesis

The trimming idea can be applied in parenthesis problem, the basic problem is also permutation problem, but you need to trimming the branch. anytime, left parenthesis should be greater or equal right parenthesis.

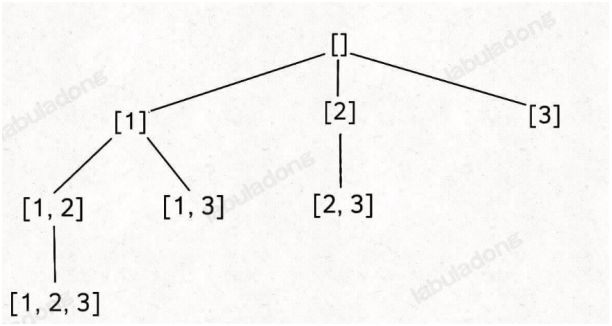
```
vector<string> generateParenthesis(int n) {
  vector<string> res;  //all results
  string track;
  backtrack(n, n, track, res); // left and right parenthesis number are both n
  return res;
}

void backtrack(int left, int right, string& track, vector<string>& res) {

  if (right < left) return; //left is less right, illegal.

  if (left < 0 || right < 0) return;
  // all is 0, get a legal result.
  if (left == 0 && right == 0) {
    res.push_back(track);
    return;
  }

  track.push_back('('); // try to put a left
  backtrack(left - 1, right, track, res);
  track.pop_back();

  track.push_back(')'); // try to put a right.
  backtrack(left, right - 1, track, res);
  track.pop_back();
}
// output is (((())))     (()())     (())()     ()(())     ()()()
```

2.7.2.5 Summary

Four code compare: They are the most common used code template for back trace, you need to understand and

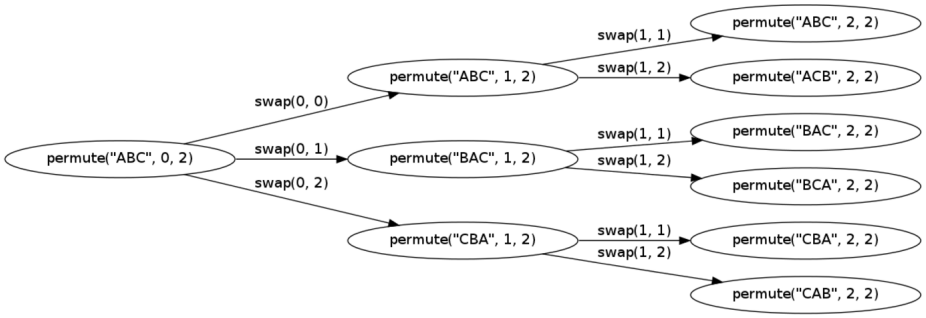
1. code 1: multi tree, call backtrace(i), backtrace(i-1), backtrace(i-2)... This will cause a multi branch tree, and the first level has n branches, The second level has n-1 branches, untill the last level has only 1 branches. O(n*n)

```
  for(i ..n )
  backtrace( i+1)
```



2. code 2: multi tree, on each level, i, i-1, i-2 backtrace O(n!)

```
for(j = i ..n )  // on i level, each backtrace trun (n-i) times.
  backtrace( i+1)
```



3. code 3: binary tree, global result, need reverse
4. code 4: binary tree, stack partial result, don't need reverse. global result.

2.7.3 Interview questions

Eight queen

```
vector<vector<string>> res;
vector<vector<string>> solveNQueens(int n) {

  // '.' is  void, 'Q' is queen
  vector<string> board(n, string(n, '.'));
  backtrack(board, 0);
  return res;
}


void backtrack(vector<string>& board, int row) {
  // base case
  if (row == board.size()) {
    res.push_back(board);
    return;
  }

  int n = board[row].size();
  for (int col = 0; col < n; col++) {
    // exclude
    if (!isValid(board, row, col)) {
      continue;
    }
    // make decision
```

```
25      board[row][col] = 'Q';
26      // go deeper
27      backtrack(board, row + 1);
28      // reverse
29      board[row][col] = '.';
30    }
31  }
```

k equal subset.

```
1  class Solution {
2    public:
3    bool canPartitionKSubsets(vector<int>& nums, int k) {
4      int sum = accumulate(nums.begin(), nums.end(), 0);
5      if (sum % k != 0) return false;
6      sort(nums.begin(), nums.end(), greater<int>());
7      vector<bool> visited(nums.size(), false);
8      return helper(nums, k, sum / k, 0, 0, visited);
9    }
10   bool helper(vector<int>& nums, int k, int target, int start, int curSum, vector<bool
       >& visited) {
11     if (k == 1) return true;
12     if (curSum > target) return false; //return false is used to trim branch
13     if (curSum == target) return helper(nums, k - 1, target, 0, 0, visited);
14     for (int i = start; i < nums.size(); ++i) {
15       if (visited[i]) continue;
16       visited[i] = true;
17       if (helper(nums, k, target, i + 1, curSum + nums[i], visited)) return true;
18       visited[i] = false;
19     }
20     return false;
21   }
22 };
```

2.8 Dynamic programming

2.8.1 Basic Idea

Dynamic Programming is a technique for solving problems with overlapping subproblems. Each subproblem is solved only once and the result of each sub-problem is stored in a table (generally implemented as an array or a hash table) for future references. These sub-solutions may be used to obtain the original solution and the technique of storing the sub-problem solutions is known as memoization. You may think of DP = recursion + re-use

The principle of optimality. No matter what the first decision, the remaining decisions must be optimal with respect to the state that results from this first decision. Dynamic programming may be used only when the principle of optimality holds. Useful when the sub-problems are overlapping. Solve an optimization problem by caching sub-problem solutions rather than recomputing them

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations.
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a traceback step in which the solution itself is constructed.

Three key words in dynamic programming are: state, transition and choice. state is most time defined by question directly, the most difficult question is **How the current states is transitioned through**

choices from previous states, which is the transition equation. If the state is defined in terms of the problem and you find it difficult to write the transition equation, then you must redefine the state; one example is Given an integer array nums, find the subarray with the largest sum, and return its sum. (https://leetcode.com/problems/maximum-subarray/description/)

The top-down approach is generally recursive (but less efficient) and more intuitive to implement as it is often a matter of recognizing the pattern in an algorithm and refactoring it as a dynamic programming solution.The bottom-up approach is generally iterative (and more efficient), but less intuitive and requires us to solve (and know!) the smaller problems first then use the combined values of the smaller problems for the larger solution. We refer to top-down solutions as memorization and bottom-up as tabulation. Memorization is generally more intuitive to implement especially when we don't know the solution to subproblems, whereas tabulation requires us to know the solutions, or bottom, in advance, in order to build our way up.

Difference between DP and BT: The basic question can be asked: can(bool), min or count(one number), all(list or vector). Usually, can and all can be used backtracking algorithm, but max/min is a typical indication of greed or dynamic programming.

Difference between DP and greed: Dynamic Programming often has higher time complexity due to solving all subproblems, typically $O(n^2)$ or $O(n^3)$ for certain problems. Greedy Algorithms Generally faster, with time complexity often $O(n \log n)$ or $O(n)$, depending on the problem.

- Use **DP** when all subproblems need to be solved and combined for the final answer.
- Use **Greedy** when a locally optimal choice at every step leads directly to the global optimum.

Difference between DP and DC: In Divide and Conquer, the sub-problems are independent of each other while in case of Dynamic Programming, the sub-problems are not independent of each other (Solution of one sub-problem may be required to solve another sub-problem). Divide and Conquer works by dividing the problem into sub-problems, conquer each sub-problem recursively and combine these solutions. solves a problem by combining the solutions to sub-problems

- Partition the problem into non-overlapping sub-problems
- Solve the sub-problems recursively
- Combine their solutions to solve the original problem

2.8.2 Three questions

Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). Below is Min jump question

```
int arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9};

int minj(int arr[], int last){
  if (last ==1 && arr[last]>=1){
    return 1;
  }

  int min = last;
  for(int i = last-1; i>=0;i--){
    if (i+arr[i]>=last){
      int step = 1+minj(arr,i);
      if( step<min){
        min = step;
      }
    }
  }
```

```
  return min;
}
```

Below is getting all questions.

```
int arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9};

vector<vector<int> >  minj(int arr[], int last){
  if (last ==1 && arr[last]>=1){
    vector<int> v1 = {1};
    vector<vector<int> > v = {v1};
    return v;
  }
  vector<vector<int> > result;

  int min = last;
  for(int i = last-1; i>=0;i--){
    if (i+arr[i]>=last){
      vector<vector<int> > result1 = minj(arr,i);
      for(auto e: result1){
        e.push_back(arr[i]);
        result.push_back(e);
      }
    }
  }
  return result;
}
```

below is Can questions.

```
int arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9};

bool  minj(int arr[], int last){

  if (last ==1 && arr[last]>=1){
    return true;
  }

  bool result = false;

  for(int i = last-1; i>=0;i--){
    if (i+arr[i]>=last){
      result = true && minj(arr,i);
      if(result)
      break;
    }
  }
  return result;
}
```

Conclusion:

- can and min is little easy, min return one number, can return one bool.
- "all" type question need return a list, because each answer is a list, so here we return vector< vector >.
- "all" type question is suitable Top-bottom method. can and min is suitable for bottom-top method.

2.8.3 one dimension

Longest Increasing Subsequence, top bottom method. There are a lot of overlap problem, so efficient is not good.

```
int _lis( int arr[], int n, int *max_ref){
  if (n == 1) /* Base case */
  return 1;

  // 'max_ending_here' is length of LIS ending with arr[n-1]
  int res, max_ending_here = 1;

  /* Recursively get all LIS ending with arr[0], arr[1] ... arr[n-2]. If arr[i-1] is
      smaller
  than arr[n-1], and max ending with arr[n-1] needs to be updated, then update it */
  for (int i = 1; i < n; i++){
    res = _lis(arr, i, max_ref);
    if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
    max_ending_here = res + 1;
  }

  // Compare max_ending_here with the overall
  // max. And update the overall max if needed
  if (*max_ref < max_ending_here)
    *max_ref = max_ending_here;

  // Return length of LIS ending with arr[n-1]
  return max_ending_here;
}
```

Longest Increasing Subsequence, bottom-top method. There are no overlap problem, so efficiency is better.

```
int lis( int arr[], int n ){
  int lis[n];
  lis[0] = 1;

  /* Compute optimized LIS values in
  bottom up manner */
  for (int i = 1; i < n; i++ ){
    lis[i] = 1;
    for (int j = 0; j < i; j++ )
    if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
    lis[i] = lis[j] + 1;
  }
  // Return maximum value in lis[]
  return *max_element(lis, lis+n);
}
```

House robber

```
for (i ->n)
  for(j->i)
    max(dp[j]...)
```

Can target string be built by input string, top-bottom.

```
vector<string> input = {"ab", "abc", "cd", "def", "abcd", "ef" , "c"};
string target = "abcdef";
```

```
vector<vector<string> > ac(string target, const vector<string> & in ){
  if (target.size() == 0){
    return vector<vector<string>> {{}};
  }

  vector<vector<string>> last_result;
  for(auto i: input){
    if (target.find(i) == 0){
      string sub = target;
      sub.erase(0, i.size());
      vector<vector<string>> re = ac(sub, in);
      for(auto j : re){
        j.push_back(i);
        last_result.push_back(j);
      }
    }
  }
  return last_result;
}
```

How many way can target string be built by input string, top-bottom.

```
vector<string> input = {"purp", "p", "ur", "le", "purpl"};
string target = "purple";

int ac(string target, const vector<string> & in ){
  int arr[7] = {1, 0, 0, 0, 0, 0, 0};
  for(int i = 0;i<7;i++){
    for (auto s: input){
      int length = s.length();
      if(target.substr(i, length) == s)
      {
        arr[i+length] += arr[i];
      }
    }
  }
  return arr[6];
}
```

2.8.3.1 Conclusion

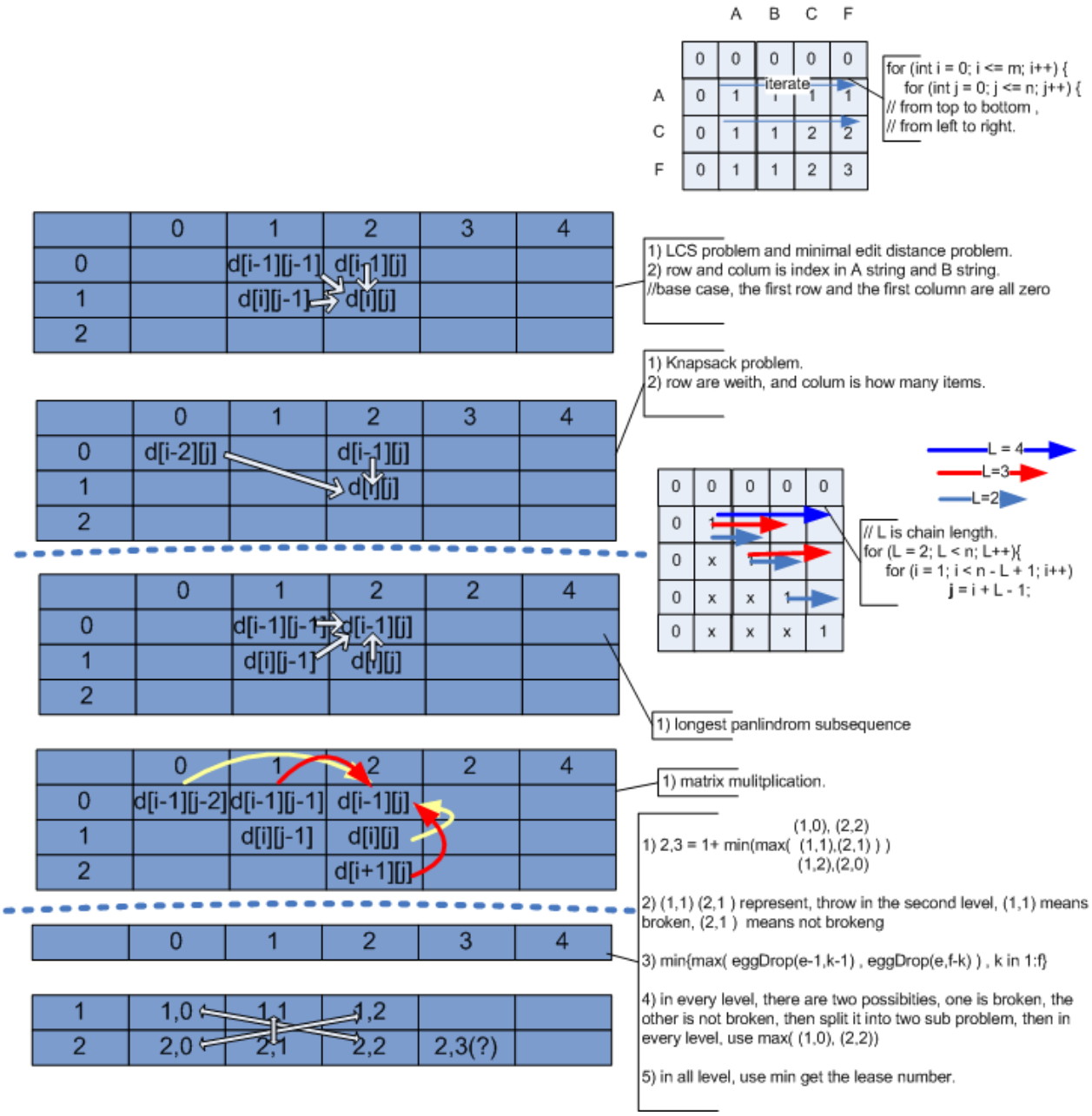
For this problem, first, decide to use one dimension dp (lis array in previous example). Then dp definition(state) should be match with your questions, here, state is the length of LIS.

dp and recursive use the same state. For example, dp and return value of recursive are both length of LIS.

```
for (i ->n) // A common template
for(j->i)
max(dp[j]...)
```

2.8.4 Two dimension

The basic idea is below:



LCS problem

```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
  int L[m + 1][n + 1];
  int i, j;

  /* Following steps build L[m+1][n+1] in
  bottom up fashion. Note that L[i][j]
  contains length of LCS of X[0..i-1]
  and Y[0..j-1] */

  for (i = 0; i <= m; i++) {
    for (j = 0; j <= n; j++) {
      if (i == 0 || j == 0)
```

```
      L[i][j] = 0;
      //base case, the first row and the first column are all zero

      else if (X[i - 1] == Y[j - 1])
      L[i][j] = L[i - 1][j - 1] + 1;

      else
      L[i][j] = max(L[i - 1][j], L[i][j - 1]);
    }
  }

  /* L[m][n] contains length of LCS
  for X[0..n-1] and Y[0..m-1] */

  return L[m][n];
}
```

minimal edit distance problem

```
int min(int x, int y, int z) { return min(min(x, y), z); }
int editDistDP(string str1, string str2, int m, int n)
{
  // Create a table to store results of subproblems
  int dp[m + 1][n + 1];

  // Fill d[][] in bottom up manner
  for (int i = 0; i <= m; i++) {
    for (int j = 0; j <= n; j++) {
      // If first string is empty, only option is to
      // insert all characters of second string
      if (i == 0)
      dp[i][j] = j; // Min. operations = j

      // If second string is empty, only option is to
      // remove all characters of second string
      else if (j == 0)
      dp[i][j] = i; // Min. operations = i

      // If last characters are same, ignore last char
      // and recur for remaining string
      else if (str1[i - 1] == str2[j - 1])
      dp[i][j] = dp[i - 1][j - 1];

      // If the last character is different, consider
      // all possibilities and find the minimum
      else
      dp[i][j] = 1 + min(dp[i][j - 1], // Insert
      dp[i - 1][j], // Remove
      dp[i - 1][j - 1]); // Replace
    }
  }

  return dp[m][n];
}
```

longest panlindrom subsequence (LPS)

```
// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
  int n = strlen(str);
```

```
  int i, j, cl;
  int L[n][n];  // Create a table to store results of subproblems

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

  // Build the table. Note that the lower diagonal values of table are
  // useless and not filled in the process. The values are filled in a
  // manner similar to Matrix Chain Multiplication DP solution (See
  // https://www.geeksforgeeks.org/matrix-chain-multiplication-dp-8/). cl is length of
  // substring
  for (cl=2; cl<=n; cl++){
    for (i=0; i<n-cl+1; i++){
      j = i+cl-1;
      if (str[i] == str[j] && cl == 2)
      L[i][j] = 2;
      else if (str[i] == str[j])
      L[i][j] = L[i+1][j-1] + 2;
      else
      L[i][j] = max(L[i][j-1], L[i+1][j]);
    }
  }
  return L[0][n-1];
}
```

Matrix multiply

```
int MatrixChainOrder(int p[], int n){
  /* For simplicity of the program, one
  extra row and one extra column are
  allocated in m[][]. 0th row and 0th
  column of m[][] are not used */
  int m[n][n];

  int i, j, k, L, q;

  /* m[i, j] = Minimum number of scalar
  multiplications needed to compute the
  matrix A[i]A[i+1]...A[j] = A[i..j] where
  dimension of A[i] is p[i-1] x p[i] */

  // cost is zero when multiplying
  // one matrix.
  for (i = 1; i < n; i++)
  m[i][i] = 0;

  // L is chain length.
  for (L = 2; L < n; L++){
    for (i = 1; i < n - L + 1; i++){
      j = i + L - 1;
      m[i][j] = INT_MAX;
      for (k = i; k <= j - 1; k++){

        // q = cost/scalar multiplications
        q = m[i][k] + m[k + 1][j]
        + p[i - 1] * p[k] * p[j];
        if (q < m[i][j])
        m[i][j] = q;
      }
    }
```

```
34    }
35
36    return m[1][n - 1];
37  }
```

- $M_{ij} \leftarrow M_i \times M_{i+1} \times \ldots \times M_j,\ i \leq j$
- c(i,j) ← cost of an optimal way to compute M_{ij}
- kay(i,j) = k be such that the optimal computation of M_{ij} computes $M_{ik} \times M_{k+1,j}$

$$c(i,i) = 0, \quad 1 \leq i \leq q \quad (M_{ii} = M_i)$$
$$c(i,i+1) = r_i r_{i+1} r_{i+2}, \quad 1 \leq i < q \quad (M_{ii+1} = M_i \times M_{i+1})$$
$$c(i,i+s) = \min_{i \leq k < i+s} \{ c(i,k) + c(k+1,i+s) + r_i r_{k+1} r_{i+s+1} \}$$

kay(i,i+s) is the value of k that yields the above min.♪

knapsack capacity

```
int knapSack(int W, int wt[], int val[], int n)
{
  int i, w;
  int K[n + 1][W + 1];

  // Build table K[][] in bottom up manner
  for(i = 0; i <= n; i++)
  {
    for(w = 0; w <= W; w++)
    {
      if (i == 0 || w == 0)
      K[i][w] = 0;
      else if (wt[i - 1] <= w)
      K[i][w] = max(val[i - 1] +
      K[i - 1][w - wt[i - 1]],
      K[i - 1][w]);
      else
      K[i][w] = K[i - 1][w];
    }
  }
  return K[n][W];
}
```

egg drop

```
int eggDrop(int n, int k){
  /* A 2D table where entery
  eggFloor[i][j] will represent
  minimum number of trials needed for
  i eggs and j floors. */
  int eggFloor[n + 1][k + 1];
  int res;
  int i, j, x;

  // We need one trial for one floor and 0
  // trials for 0 floors
  for (i = 1; i <= n; i++) {
```

```
    eggFloor[i][1] = 1;
    eggFloor[i][0] = 0;
  }

  // We always need j trials for one egg
  // and j floors.
  for (j = 1; j <= k; j++)
  eggFloor[1][j] = j;

  // Fill rest of the entries in table using
  // optimal substructure property
  for (i = 2; i <= n; i++) {
    for (j = 2; j <= k; j++) {
      eggFloor[i][j] = INT_MAX;
      for (x = 1; x <= j; x++) {
        res = 1 + max(
        eggFloor[i - 1][x - 1],
        eggFloor[i][j - x]);
        if (res < eggFloor[i][j])
        eggFloor[i][j] = res;
      }
    }
  }

  // eggFloor[n][k] holds the result
  return eggFloor[n][k];
}
```

below is dp implementation of knapsack.

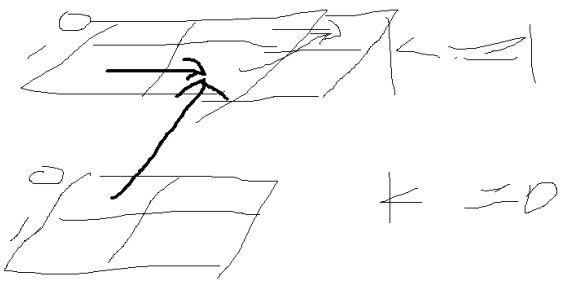
```
int knapsack_dp() {
  int W = 10;
  int N = 5;
  int val[] = {40, 50, 100, 95, 30};
  int wt[] = { 2, 4, 2, 5, 3 };
  int dp[6][11];
  for (int i = 0; i <= N; i++) {
    for (int w = 0; w <= W; w++) {
      if (i == 0)
      dp[i][w] = 0;
      if (w == 0)
      dp[i][w] = 0;
    }
  }
  for (int i = 1; i <= N; i++) {
    for (int w = 1; w <= W; w++) {
      if (w - wt[i - 1] < 0) {

        dp[i][w] = dp[i - 1][w];
      }
      else {

        dp[i][w] = max(
        dp[i - 1][w - wt[i - 1]] + val[i - 1],
        dp[i - 1][w]
        );
      }
    }
  }

  for (int i = 1; i <= N; i++) {
```

```
32      for (int w = 1; w <= W; w++) {
33        cout << dp[i][w] << " ";
34      }
35      cout << endl;
36    }
37
38    return dp[N][W];
39  }
40  // 0 40 40 40 40 40 40 40 40 40
41  // 0 40 40 50 50 90 90 90 90 90
42  // 0 100 100 140 140 150 150 190 190 190
43  // 0 100 100 140 140 150 195 195 235 235
44  // 0 100 100 140 140 150 195 195 235 235
```

2.8.5 Three dimension

A typical question is stock buy and sell.

```
1  for (int i = 0; i < n; i++)
2  for (int k = max_k; k >= 1; k--) {
3    if (i - 1 == -1) {
4      // deal i = -1  base case
5      dp[i][k][0] = 0;
6      dp[i][k][1] = -prices[i];
7      continue;
8    }
9    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
10   dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]);    // k-1 here.
       three dimension array, go to the next level, please refer the below picture.
11 }
12 return dp[n - 1][max_k][0];
```



2.8.6 Summary

The general form of a dynamic programming problem is to find the optimal value. Since the goal is to find the optimal value, the core issue in solving dynamic programming problems is enumeration. The steps are: define the base case, clarify the "**states**", clarify the "**choices**", and define the meaning of the dp array/function. I summarize this theory as "one model with three characteristics."

First, let's look at what "one model" refers to. It describes the model of problems that dynamic programming is suited for solving. I define this model as the "optimal solution model with multi-stage

decision-making." Each decision stage corresponds to a set of states. We then search for a sequence of decisions that, when followed, results in the optimal value we seek.

Now, let's explore what the "three characteristics" are. They are **optimal substructure**, **no after-effect**, and **overlapping subproblems**. These concepts are somewhat abstract, so I'll explain them one by one:

- Optimal substructure means that the optimal solution to the problem contains the optimal solutions to its subproblems. In other words, we can derive the optimal solution of the problem from the optimal solutions of its subproblems. When we map optimal substructure to the dynamic programming problem model we defined earlier, it means that the states in the later stages can be derived from the states in the earlier stages.
- No after effect (or Memorylessness). This has two meanings. The first is that when deriving the state of a later stage, we only care about the state values of the earlier stages, not how those states were derived step by step. The second meaning is that once the state of a certain stage is determined, it is not affected by decisions made in later stages. No after effect is a relatively "relaxed" requirement. As long as the dynamic programming problem model mentioned earlier is satisfied, the no after effect property will generally hold.
- Overlapping Subproblems This concept is easier to understand. As mentioned several times in the previous section, in summary, it means that different decision sequences may reach the same stage, potentially resulting in repeated states.

Dynamic programming involves the following steps:

1. Transform the current problem into subproblems: Once you break it down into subproblems, define three things:
   (a) states
   (b) choice.
   (c) the state transition equation, which explains how the solution to the current problem is derived from the combination (or transformation) of the subproblems.
2. Decide whether to use top-down DP (subproblem, subproblem state 1, subproblem state 2, etc.) or bottom-up DP: For example, dp[subproblem], dp[subproblem state 1], dp[subproblem state 2], etc.
3. Determine the traversal direction and base case based on the meaning of the dp function.
4. Apply the template code to implement the solution.

Below are some common examples: the egg-drop problem, subsequence problems and finding the minimum. The 0/1 knapsack problem and stock trading problem.

Why does the n-floor egg-drop problem require considering all subproblems from 0 to i-1 for each i, while the stock trading problem only needs to consider the previous state? One thing to note is that in the stock trading problem, k represents the maximum number of allowed transactions, not the number of transactions already completed. It's a way of describing the state.

(Based on the problem, define the dp, then take dp[i-1] and try to see if you can combine dp[i-1] to form dp. Generally, there are two possibilities: either it involves all previous states (as in the egg-drop problem), or it involves just the previous two states (as in the stock trading problem). The current state can transition from buying or not buying.)"

Four questions: matrix multiply, shoot ballon, gambling(piles stone, take from left and right) and maxium panlinmon subsequecne. The answer is located in the top-right corner of a 2D array, and you need to use diagonal traversal or reverse traversal. The transition equations are almost identical.

```
1 for (int i = n; i >= 0; i--) {
```

```
  // j from left to right
  for (int j = i + 1; j < n + 2; j++) {
    // "Which balloon is the last to be burst?"

    for (int k = i + 1; k < j; k++) {
      // choose the best option
      dp[i][j] = Math.max(
      dp[i][j],
      dp[i][k] + dp[k][j] + points[i]*points[j]*points[k]
      );
    }
  }
}
```

Two questions: Regular expressions and edit distance: The result is in the bottom-right corner, and a forward traversal is needed. The subfunction represents an edit or match operation.

```
bool dp(string& s, int i, string& p, int j) {
  if (s[i] == p[j] || p[j] == '.') {
    // match
    if (j < p.size() - 1 && p[j + 1] == '*') {
      // 1.1 regex match 0 or more
      return dp(s, i, p, j + 2)
      || dp(s, i + 1, p, j);
    } else {
      // 1.2 normal match 1
      return dp(s, i + 1, p, j + 1);
    }
  } else {
    // not match
    if (j < p.size() - 1 && p[j + 1] == '*') {
      // 2.1 regex match 0 times
      return dp(s, i, p, j + 2);
    } else {
      // 2.2 can't continue match
      return false;
    }
  }
}

// return s1[0..i] and s2[0..j] minimum edition distance
int dp(String s1, int i, String s2, int j) {
  // base case
  if (i == -1) return j + 1;
  if (j == -1) return i + 1;

  if (s1.charAt(i) == s2.charAt(j)) {
    return dp(s1, i - 1, s2, j - 1); // do nothing
  }
  return min(
  dp(s1, i, s2, j - 1) + 1,     // insert
  dp(s1, i - 1, s2, j) + 1,     // delete
  dp(s1, i - 1, s2, j - 1) + 1 // replace
  );
}
```

```
for (int i = 1; i <= n; ++i) {
  for (int j = 1; j <= min(i, m); ++j) {
    if (j == 1) {
      dp[i][j] = sum[i];
```

```
    } else {
      for (int k = 1; k <= i - 1; ++k) {
        dp[i][j] = min(dp[i][j], max(dp[k][j - 1], sum[i] - sum[k]));
      }
    }
  }
}
return dp[n][m]
```

2.9 Branch and bound

Backtracking mainly use BFS, not DFS.

Backtracking is used to find all possible solutions available to the problem. It traverse state tree by DFS (Depth First Search) until it found a solution.. It realizes that it has made a bad choice & undoes the last choice by backing up. It involves feasibility function.

Branch-and-Bound, it realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution. It involves bounding function. Usually searches a tree in a breadth-first / least-cost manner (unlike backtracking)

Difference between branch-bound and backtracking is about size

- backtracking use stack and only add one child each time, so it use less memory compared with branch-bound
- branch-bound store all possible child in queue, so if answer is near root, it will find answer quickly.
- win in size, but lose in time. backtracking will find answer slower if answer is near the root.

Below I present three different methods to resolve 0-1 back pack problem. dp only can deal with integral weight. but branch-bound can deal with float value. This code can run and print out all the value, it can help you to understand the idea behind these two important algorithms.

below is knapSack recursive implementation, it's also backtracking, different with using stack. The idea is same, you can see the code is simpler.

```
int knapSack(int W, int wt[], int val[], int n){
  if (n == 0 || W == 0) // Base Case
  return 0;

  // If weight of the nth item is more
  // than Knapsack capacity W, then
  // this item cannot be included
  // in the optimal solution
  if (wt[n - 1] > W)
  return knapSack(W, wt, val, n - 1);

  //There are two options, so there's no need to write a for-loop. Based on the options
      , change the state, pass the state into the recursive call, and take the maximum
      of the returned results.

  // Return the maximum of two cases:
  // (1) nth item included
  // (2) not included
  else

  return max(
  val[n - 1]
  + knapSack(W - wt[n - 1],
```

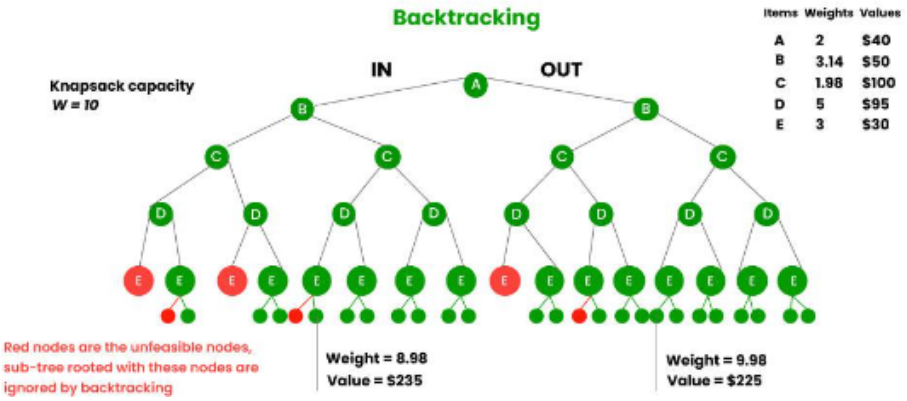
```
    wt, val, n - 1),
    knapSack(W, wt, val, n - 1));
}
```

```
// Structure for Item which store weight and corresponding value of Item
struct Item{
    float weight;
    int value;
};
// Node structure to store information of decision tree
struct Node{
    // level  --> Level of node in decision tree (or index in arr[]
    // profit --> Profit of nodes on path from root to this node (including this node)
    // bound ---> Upper bound of maximum profit in subtree of this node/
    int level, profit, bound;
    float weight;
};
// Comparison function to sort Item according to val/weight ratio
bool cmp(Item a, Item b){
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// below is knapsack backtracking implementation.
int knapsack_back(int W, Item arr[], int n){
    sort(arr, arr + n, cmp); // sorting Item on basis of value per unit weight.

    stack<Node> S; // make a queue for traversing the node
    Node u, v;

    u.level = -1; // dummy node at starting
    u.profit = u.weight = 0;
    S.push(u);

    // One by one extract an item from decision tree compute profit of all
    // children of extracted item and keep saving maxProfit int maxProfit = 0;
    while (!S.empty()){
        u = S.top();
        S.pop();
        // If it is starting node, assign level 0
        if (u.level == -1)
            v.level = 0;

        // If there is nothing on next level
        if (u.level == n - 1)
            continue;

        // Else if not last node, then increment level,
        // and compute profit of children nodes.
        v.level = u.level + 1;

        // Do the same thing,  but Without taking the item in knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        S.push(v);

        // Taking current level's item add current level's
        // weight and value to node u's weight and value
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;
```

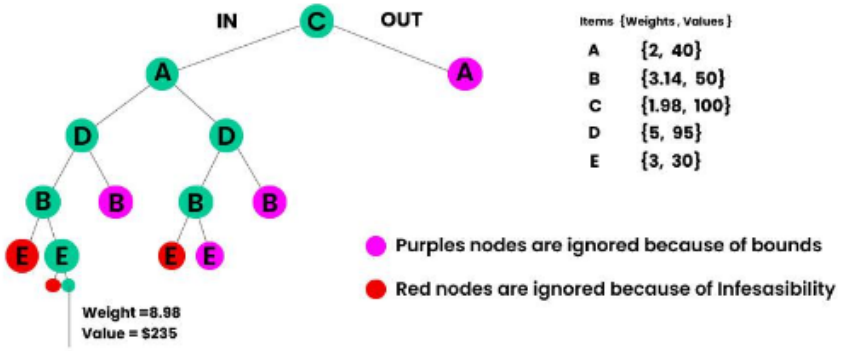
```
58      if(v.weight>W)
59        continue;
60
61      // If cumulated weight is less than W and profit
62      // is greater than previous profit, update maxprofit
63      if (v.weight <= W && v.profit > maxProfit)
64        maxProfit = v.profit;
65
66      // If bound value is greater than profit,
67      // then only push into queue for further consideration
68      S.push(v);
69    }
70    return maxProfit;
71  }
72
73  //-----------------------------------------------------------------
74  // Returns bound of profit in subtree rooted with u.
75  // This function mainly uses Greedy solution to find
76  // an upper bound on maximum profit.
77  int bound(Node u, int n, int W, Item arr[]){
78    // if weight overcomes the knapsack capacity, return 0 as expected bound
79    if (u.weight >= W)
80      return 0;
81
82    // initialize bound on profit by current profit
83    int profit_bound = u.profit;
84
85    // start including items from index 1 more to current item index
86    int j = u.level + 1;
87    int totweight = u.weight;
88
89    // checking index condition and knapsack capacity condition
90    while ((j < n) && (totweight + arr[j].weight <= W)){
91      totweight += arr[j].weight;
92      profit_bound += arr[j].value;
93      j++;
94    }
95    // If k is not n, include last item partially for  upper bound on profit
96    if (j < n)
97      profit_bound += (W - totweight) * arr[j].value / arr[j].weight;
98    return profit_bound;
99  }
100
101 // Returns maximum profit we can get with capacity W
102 int knapsack_bb(int W, Item arr[], int n){
103   // sorting Item on basis of value per unit weight.
104   sort(arr, arr + n, cmp);
105   queue<Node> Q;  // make a queue for traversing the node
106   Node u, v;
107
108   u.level = -1;  // dummy node at starting
109   u.profit = u.weight = 0;
110   Q.push(u);
111
112   // One by one extract an item from decision tree compute
113   // profit of all children of extracted item and keep saving maxProfit
114   int maxProfit = 0;
115   while (!Q.empty()) {
116     u = Q.front();
117     Q.pop();
118     if (u.level == -1)
```

```
119       v.level = 0;
120
121     // If there is nothing on next level
122     if (u.level == n - 1)
123       continue;
124
125     // Else if not last node, then increment level,
126     // and compute profit of children nodes.
127     v.level = u.level + 1;
128
129     // Taking current level's item add current level's
130     //weight and value to node u's weight and value
131     v.weight = u.weight + arr[v.level].weight;
132     v.profit = u.profit + arr[v.level].value;
133
134     // If cumulated weight is less than W and profit
135     // is greater than previous profit, update maxprofit
136     if (v.weight <= W && v.profit > maxProfit)
137       maxProfit = v.profit;
138
139     // Get the upper bound on profit to decide whether to add v to Q or not. If bound
            value
140     //is greater than profit, then only push into queue for further consideration
141     v.bound = bound(v, n, W, arr);
142     if (v.bound > maxProfit)
143       Q.push(v);
144
145     // Do the same thing,  but Without taking the item in knapsack
146     v.weight = u.weight;
147     v.profit = u.profit;
148     v.bound = bound(v, n, W, arr);
149     if (v.bound > maxProfit)
150       Q.push(v);
151   }
152   return maxProfit;
153 }
154 int main(){
155   int W = 10;   // Weight of knapsack
156   Item arr[] = { {2, 40}, {3.14, 50}, {1.98, 100}, {5, 95}, {3, 30} };
157   int n = sizeof(arr) / sizeof(arr[0]); //n object.
158
159   knapsack_back(W, arr, n); //use backtracking
160   knapsack_bb(W, arr, n); //use branch and bound
161 }
```

Below figure illustrates the backtracking



Below figure illustrates the branch and bound



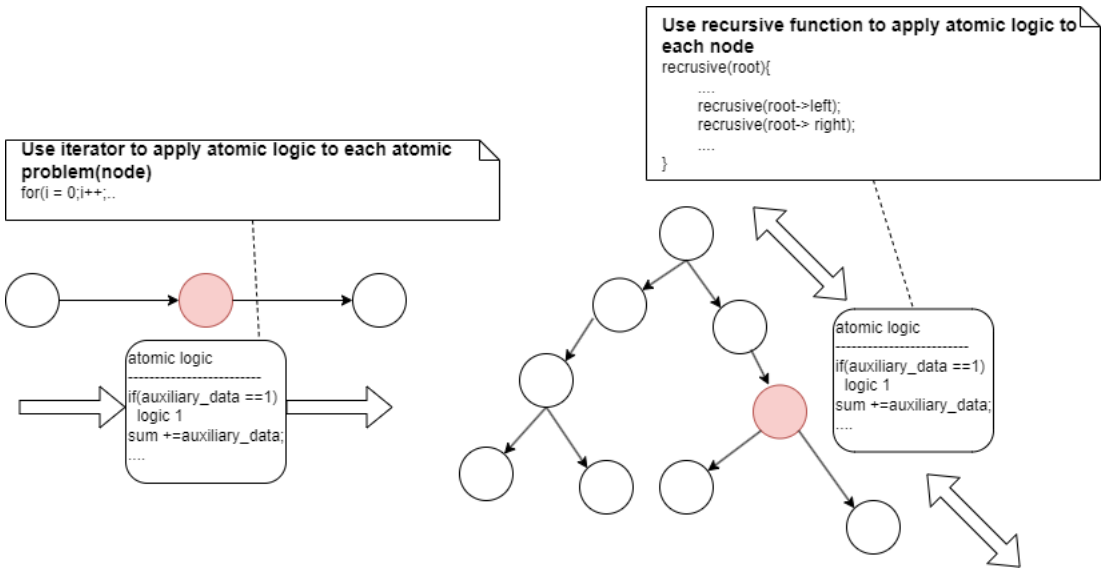
Recursive 0-1 pack pack. This is back trace, but why don't we have reverse decision? because in eight queen, we use reference parameter pass it to function(all function share). Here, we use stack, in each level, we have saved the local variable(you can think that it has been reversed by poping stack.) That is the difference. Another important thing is that for 0-1 pack and eight queen, we need a logically recursive tree. So we need to simulate a "level" conception. **0-1 pack stack solution, we save level information to the each node. 0-1 recursive solution, we use int i (array index)as level information. in eight queen, we use row information as level information** . level is the level information in the recursive solution tree.

```
1  public int maxW = Integer.MIN_VALUE;
2  // cw current weight, i: current item call with f(0, 0, a, 10, 100)
3  public void f(int i, int cw, int[] items, int n, int w) {
4    if (cw == w || i == n) { // cw==w is full or select all items
5      if (cw > maxW) maxW = cw;
6      return;
7    }
8    f(i+1, cw, items, n, w);
9    if (cw + items[i] <= w) {// bigger than w, give up
10     f(i+1,cw + items[i], items, n, w);
11   }
12 }
```

2.10 Summary of algorithms

The most import rule: Break the big problem down into atomic problems. If the atomic problem requires some global conditions, then include the necessary data and data structures. Write the logic to handle this atomic problem (you may need to use a if..else to handle different conditions inside this atomic problem). Then apply this atomic problem handling logic into **recursion** or **iteration**. Difference between recursive and iterate.

- Recursive: Simple, intuitive code unless care is taken to avoid recomputing previously computed values, the recursive program will have prohibitive complexity
- Iterative: Not require additional space for the recursion stack. To minimize run time overheads, and hence to reduce actual run time, dynamic programming recurrences are almost always solved iteratively. (no recursion).



Three levels:

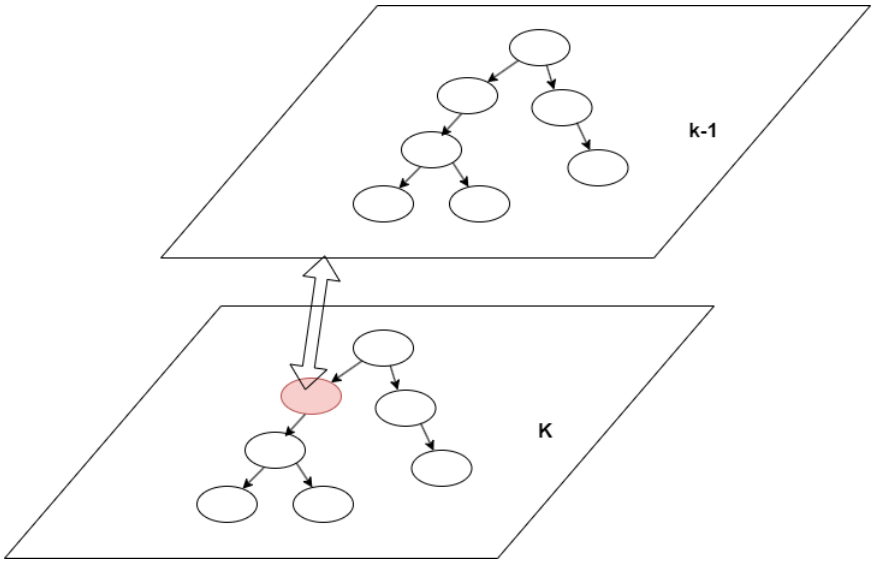
1. The first level consists of two basics: recursion and iteration.
2. The second level builds on these two. In the domain of trees, recursion manifests as three types of traversals; in the domain of graphs, recursion is represented by DFS, while iteration is represented by BFS.
3. The third level involves practical applications, where recursion and iteration are expressed through five major algorithms: divide-and-conquer (DC), dynamic programming (DP), backtracking (BT), branch-and-bound (BB), and greedy algorithms.

Difference between backtracking and DP:

- Both backtracking and dynamic programming involve a series of decisions. The difference is that in backtracking, certain constraints must be satisfied during the decision-making process (such as in the N-Queens problem). If the constraints are not met, you need to undo that decision. Dynamic programming, on the other hand, deals with the global optimal solution. So for each decision, you typically compare and record the optimal solution for the subproblem in order to eventually derive the global optimal solution.
- Another difference is that backtracking often uses recursion, while dynamic programming uses iteration (usually over a 2D array). However, sometimes the two approaches can be combined, such as in the partition problem, where backtracking is primarily used, but you can also record

solutions to subproblems (using dynamic programming techniques) to reduce the number of recursive calls.

- typical question for DP is stock buy and sell k time. typical question for backtrack is k equal subset. They all involve extra k parameter. With k parameter, you can think that it's k level, on each level, there is tree or array structure. That is the most complex problem you can meet in the interview questions. With k, you can think it's multi layer, for back track, each layer has a tree (k equal subset). For dp, each layer has two dimension array.



Chapter 3

Interview preparation

3.1 C++ code points

3.1.1 container

- 4753, 47743. container methods summary.
- know common container initialization:

```
//1) use copy constructor
vector<vector<int>> vv{{1,2,3},{4, 5, 6}};
vector<vector<int>> ww{vv};

//2) all zero
vector<vector<int>> ww(2, vector<int>(10, 0));

//3) build from different container.
set<int> s {1, 2, 3};
vector<int> v2{s.begin(), s.end()};

vector<int> v2{1, 2, 3, 3, 3,4};
set<int> s {v2.begin(), v2.end()};

map<int, int> m {{1, 2}, {2, 4}};
vector<pair<int, int>> v1{m.begin(), m.end()};

vector<pair<int, int>> v1{{1, 2}, {2, 4}};
map<int, int> m {v1.begin(), v1.end()};
```

- for map, insert, emplace, emplace_hint, try_emplace difference. [] and insert_or_assign difference. Use more emplace if it is possible.
- Use list.splice in list to finish rotate.
- for std::array, the main interface are access, iterator and fill. std::span also has access, iterator and first, last and subspan.
- erase+remove for vector, remove for list and erase for associatate.
- range assign, construct and insert and erase
- min priority queue

```
priority_queue<int, vector<int>, greater<int>> min_pq;
```

- some auto usages:

```
for(auto &[f, s] :map1) //use & here, no need const,
```

```
2
3 auto [it, ins] = map1.insert_or_assign(1, 3);
4
5
6 conat auto [f, s]  = ranges::remove() //no & here, use const for better security.
7
8 auto [fi, si] = multimap.equal_range(1); // struct binding, fi is first iterator
9 for(auto p = fi; p!= si; ++p){
10   auto &[key, value] = *p;  //struct bind, key is first element
11 }
```

- std::erase and std::erease_if for vector and string list and deque

3.1.2 string

- Own functions:
  1. string.find(ch/str, pos)
  2. string.substr(pos, count)
  3. replace(it1, it2..), replace(pos, count ...),
  4. compare
  5. starts_with, ends_with, contains
- common functions: ::toupper, ::tolower, isupper, islower, isdigit, isalnum, isalpha, isspace,
- to_string and stol, stof, stoul
- trim(use find_if_not and ::ispace) and split (getline + stringstream).
- find + while pattern

3.1.3 algorithm and pattern

3.1.3.1 useful pattern

- Three useful pattern.

```
1 for(int i ; auto e: container){ //loop without know the size.
2   ...
3   ++i;
4 }
5
6 flag == 2 ? ++sum1 : ++sum2; //ternary
7
8 maxs = max(e, maxs); //find max
```

- know the different below.

```
1 if()
2 if()
3
4 if()
5 else
```

- variable name: cnt, result, st, ans, max1, max2
- nest if can be written &&

```
1 if(s==1){
2   if(t==2){
3   }
4 }
```

```
5
6 if(s==1 && t == 2) //a better way to express.
```

3.1.3.2 algorithm

- ACCMFS, RRRRSSSSPPCFMT, HMSSS
- Find(_if)(_not) and find_first_of and adjacent_find is the basic. range add find_last_if_not. std::string add find_last_not_of.
- Ranges doesn't support four math algorithms.
- Customization with callable: 1) change operation(accumulate change + to *), 2)change projection. (compare pair::first) 3) action 4) map 5) generator 6) predicate 7) compare
- emplace use () inside, not .
- ranges:: return value
  1. find use iterator to mark position, ranges:: version don't change it.
  2. most out iterator, add one or two in
  3. Some ranges:: return subrange. 1) search 2) remove/unique 3) rotate, shift and partition.
- How to use projection on stl function? can't use directly, must use your own lambda. A good example of leetcode 1637,

```
1 sort(c, {}, &vector<int>::front);
2 //That is not correct, must use your own lambda to wrap it.
3
4 ranges::sort(points, {}, [](auto &e ){return e.front();});
```

3.1.4 views

- GPTSSSRZS
- views::itoa(1) and ranges::to<string>() works in leetcode.

```
1 string s = "12345678";
2
3 auto vw = s | views::chunk(3);
4 auto vw1 = vw | views::join_with(',')|ranges::to<string>();
5 cout<<vw1<<endl;
6 -std=c++23 -O2 -Wall -Wextra -Wpedantic
```

- filter is not SIZE, take_while is not COMMON. Can't use `auto &` for views::transform. element_view iterator has no -> operator

3.1.5 C++ utilities

- math(6), compare(6), logic(3), bit(4) (3466)
- stoi and to_string
- ::toupper, ::tolower, ::isalnum ::isalpha, ::isdigit, ::ispunct, ::isspace
- sqrt, cbrt, pow, abs
- numeric_limits<int>::max()

3.1.6 bit

- n&(n-1)

- big and small endian
- set, clear, flip
- find right and left bit.
- ~(-1 « n), -1 is all set bit, use as left shift with 1, not 0.

3.2 Coding tips and traps

Pay more attention to below points when you write C++ code:

- Write slowly, try you best to write zero bug code.
- Use more std container. **Most of complex data structure need hash and list combine**. Use more std ranges algorithms and views.
- For vector and array, pay attention to size-1 is the last index. Also stick to left close and right open when using ranges algorithm. Pay left edge (0) and right edge (last element). Understand index and length.
- Check variable name, especially long variable name.
- Don't forget return in lambda and recursive. Don't forget ; in the end.
- Add () if expression is complicated.
- When while exist, understand index position and what it represent?

```
while(index<length){
}
//what index is now
```

- Use auto as possible. Must know if you need add reference & or const when you use auto and structure bind.
- If you want to loop from length to 0, index must be int. can't be size_t, otherwise, –index will never small than 0. (Must remember)

3.3 Code template

You need to remember some famous algorithms implementations:

- Reverse link-list (recursive and iterative methods).
- Partition and partition_if, use slow and fast pointer and swap.
- Remove and unique, use slow and fast pointer.
- Binary search, and left/right boundary, use binary search, left close and right close.
- Find inserting point and sqrt(x) use mid=l+(r-l)/2
- Reverse, two poitners and swap
- Rotate, three reverse.
- Insertion sort, std::upper_bound and std::rotate.
- nth_element, quick select.
- Sort, quick sort, merge sort and heap sort.
- Inplace merge.

split, use init,

```
"abc-def-gh" --> abc  def  gh
init = 0;
while( (pos = date.find(' ', init)) !=string::npos ){
```

```
    string temp = date.substr(init, pos-init);
    cout<<temp<<endl;
    ++pos;
    init = pos;
}
temp = date.substr(init);
cout<<temp<<endl;
```

```
"abc-def-gh"

int init = date.size()-1;
while( (pos = date.rfind('-', init)) !=string::npos ){
    string temp = date.substr(pos+1, init-pos);
    cout<<temp<<endl;
    --pos;
    init = pos;
}
string temp = date.substr(0, init+1);  //use init+1 here.
cout<<temp<<endl;
```

Sliding windows template

```
int left = 0, right = 0;

while (right < s.size()) {
    window.add(s[right]); // increase windows
    right++;
    ***  //put you logic here.

    //anytime you increase window, check condition.
    while (window needs shrink) {
        // reduce windows.
        window.remove(s[left]);
        left++;
        *** //put your logic here.
    }
}

//1) only use right <s.size as main loop
//2) "windows nees shrink" condition need to be customized.
//3) modify "window.add" and *** to your own logic.
//4) both left and right increments ++
```

When writing the backtrack function, you need to maintain the "path" that has been traversed and the current "list of available choices." When the "termination condition" is triggered, add the "path" to the result set.

```
result = []
def backtrack(path , "list_of_avaiable_choices"):
if (meet condition)
result.add(path)
return

for choic in "list_of_avaiable_choices":{
    choose choice;
    backtrack(path , "maybe_new_list_of_avaiable_choices");
    revoke choice;
}
```

DFS and BFS code. For BFS, when D.pop we don't judge if it's visited. For all next current, visit them all and push them into the D. That is the difference with DFS.

```
DFS(start){
  S.push(start)
  while(!S.empty()){
    current = S.pop
    if(unvisited(current){
      visit current.
      for all(next to current and unvisited)
      S.push(next);
    }
  }
}
```

```
BFS(start){
  D.push(start)
  while(!D.empty()){
    current = D.pop  //2) no if here
    for all(next to current and unvisited){
      visit current.  //1) push and visit at the same time.
      D.push(next);
    }
  }
}
```

Best sell problem,

```
std::array<int, 9> arr = {2, 3, 8, 100, 20, 1, 1, 2, 1 };
std::unordered_map<int, int> dic;
for(auto e: arr){
  dic[e]++;  //don't need find
}

std::vector< std::pair<int, int> > data( dic.begin(), dic.end() );  //build vector directly.

std::partial_sort( data.begin(), data.begin()+2, data.end(), [](auto &e1, auto &e2){return e1.second>e2.second;});
//use lambda to descend order.

for(auto &e: data){
  cout<<e.first<<"——>"<<e.second<<endl;
}
```

3.4 Typical interviwe questions

3.4.1 data structure

3.4.1.1 array

1) maximum difference from right element(one time stock buy and sell). 2) circular queue, three points: 1) empty, 2) full and 3) front+1/length = rear. 3) stock one buy ans sell. 4) next larger (leader problem) (mono stack). 5) maximum sliding window(mono queue). 6) partial sum used for random select with weight.

3.4.1.2 list

1) Reverse link list. 2) judge if there is loop, where is the entry of loop?

3.4.1.3 tree

1) LCA, (How to resolve if only one node is here?) 2) Judge if tree is valid BST? (how to understand min and max). 3) Number of BST? 4) Build tree from serialize. 5) delete node if path sum $<$ k.

3.4.1.4 graph

Offline union-find aglorithm and use this to find spanning tree.

3.4.2 algorithms

3.4.2.1 Two pointers-slow and fast

duplicate and partition

3.4.2.2 Two pointers-sliding windows

leetcode 76. Minimum Window Substring

3.4.2.3 D&C

nth element, find rotate, and left boundary. These three methods has different left and right boundary. In these method, we usually input first index and last index (not size of array), and also left close, right close interval, this makes programming a little easier.

A variant is index is in the middle. these questions includes all parenthesis, child tree and matrix mulitplication.

```
fun( int l, int r)

for(int k = l ;k<r; k++){
  fun( l,k);
  fun( k, r)
}
```

3.4.2.4 binary search

1) monkey eat banana

3.4.2.5 Backtracking

Some common examples: 1) permutation, subset and combination. Knapsack problem. 2) Legal parentheses combination. 3) The most complicated example is k equal subset problem.

3.4.3 Famous questions

- leetcode 1539. binary search.
- 2970, and 1539 are not very directly, but you need to change the question to a logic on a single element. just like a rain drop problem, need a little "building model" problem.

3.4.3.1 String

- 2273, use array to judge if it's anagram. don't use map here, you can use char cnt [26] directly, it's easier than map or unordered_map. Don't build two cnt[26] either, you can break the loop earlier, that is the common style to juge if it's a anagram.
- 1668 and 1566 a little similar. 1566 just check pattern, I don't know what pattern content is. For 1668, we know what the pattern is. so I can use string.starts_with
  1. 1566 Given an array of positive integers arr, find a pattern of length m that is repeated k or more times.

     A pattern is a subarray (consecutive sub-sequence) that consists of one or more values, repeated multiple times consecutively without overlapping. A pattern is defined by its length and the number of repetitions.

     Return true if there exists a pattern of length m that is repeated k or more times, otherwise return false.
  2. 1668: For a string sequence, a string word is k-repeating if word concatenated k times is a substring of sequence. The word's maximum k-repeating value is the highest value k where word is k-repeating in sequence. If word is not a substring of sequence, word's maximum k-repeating value is 0.

     Given strings sequence and word, return the maximum k-repeating value of word in sequence.

```
    int cntRepeating(string_view sv, string word){
  int cnt{};
  int ws = word.size();
  while(sv.starts_with(word)){
    ++cnt;
    sv = sv.substr(ws);
  }
  return cnt;
}

int maxRepeating(string sequence, string word) {
  /* doesn't work,
  auto pos = sequence.find(word, 0);
  int ws = word.size();
  if(pos == string::npos) {
    return 0;
  }
  int ans{1};
  int result{};
  auto init = pos+ws;
  while((pos = sequence.find(word, init)) != string::npos){
    if(pos - init == 0){
      ++ans;
      init = pos+ws;
    }
    else{
      result = max(result, ans);
      ans = 1;
```

```
        init = pos+ws;
      }
    }
    result = max(result, ans);
    return result;*/


    int n = sequence.size();
    int i = 0;
    int result{};
    while(i<n){
      auto vv = sequence | views::drop(i);
      string_view sv(vv);
      result = max(result, cntRepeating(sv,word) );
      ++i;
    }
    return result;
}
```

- leetcode 1576. look for a different characters with left and right, you can loop through "abc" and it's enough, don't need to loop through 26 character, it's a little trick.
- 1078, Given two strings first and second, consider occurrences in some text of the form "first second third", where second comes immediately after first, and third comes immediately after second. Return an array of all the words third for each occurrence of "first second third".
  Answer: learn how to use views::split and views::slide. **The result is subrange of subrange.**

```
auto vw = text| views::split('␣') | views::slide(3);
vector<string> result;

for(auto e : vw){
  auto it = e.begin();
  auto e1 = string_view{*it};
  ++it;
  auto e2 = string_view{*it};
  if(e1 == first && e2 == second){
    ++it;

    result.emplace_back( (*it).begin(), (*it).end() );
  }

}
return result;


//a better version is ajacent<3>
vector<string> findOcurrences(string text, string first, string second) {
  auto vv = text | views::split('␣') | views::adjacent<3>;
  vector<string> ans;
  for( auto const &[f, s, t] : vv){
    if(string_view{f} == first && string_view{s} == second){
      ans.push_back(string(t.begin(), t.end()) );
    }
  }
  return ans;
}
```

- 1071, For two strings s and t, we say "t divides s" if and only if s = t + t + t + ... + t + t (i.e., t is concatenated with itself one or more times). Given two strings str1 and str2, return the largest string x such that x divides both str1 and str2.

```
string gcdOfStrings(string str1, string str2) {
  if (str1 + str2 != str2 + str1) return "";
  int n = __gcd(str1.size(), str2.size());
  return str1.substr(0, n);
}
```

3.4.3.2 Double pointer

- 3318, map and set combine, roof of usage of different containers. A very good example, need to remember it.
- 2903, double pointers.
- 3095, subarry, have to use double pointer(sliding windows.), count all set 1, revoke or.
- 3090, a typical double pointer, but you need to get maximum, so this maximum lie in when you expand right pointer, if you want to know the minimum, the value is in the shrink of left pointer.
- 680, Given a string s, return true if the s can be palindrome after deleting at most one character from it. We use two pointers to point to the left and right ends of the string, respectively. Each time, we check whether the characters pointed to by the two pointers are the same. If they are not the same, we check whether the string is a palindrome after deleting the character corresponding to the left pointer, or we check whether the string is a palindrome after deleting the character corresponding to the right pointer. If the characters pointed to by the two pointers are the same, we move both pointers towards the middle by one position, until the two pointers meet.
  If we have not encountered a situation where the characters pointed to by the pointers are different by the end of the traversal, then the string itself is a palindrome, and we return true.
  Any palindrome questions can be resolved by two pointers.

3.4.3.3 STL container

- 2558, priority_queue usage. A good example.
- 2956, O(1) use unordered_map, or log(n) use sort and binary search. set operation is not suitable for this question.
- 2006, two_sum variant.

3.4.3.4 STL Algorithm

- 3174, how to use std::erase directly.
- 1913, max or min two, max three(nth_element or priority_queue?).

```
1) numeric\_limits<int>::max()
2) numeric\_limits<int>::min()

3) compare with max1 and max2, if bigger then max1, then max2 = max1. That is a
    common pattern.
```

- 1636, use sort, that idea is very interesting. How to second increasing, the first decreasing. **sort on nums directly**

```
vector<int> frequencySort(vector<int>& nums) {
  vector<int> cnt(201);
  for (int v : nums) {
    ++cnt[v + 100];
  }
```

```
  sort(nums.begin(), nums.end(), [&](const int a, const int b) {
    if (cnt[a + 100] == cnt[b + 100]) return a > b;
    return cnt[a + 100] < cnt[b + 100];
  });
  return nums;
}
```

```
map<int, int> m;
for(auto e: nums){
  ++m[e];
}

vector<pair<int, int>> v(m.begin() , m.end());
sort(v.begin(), v.end(), [](const auto& e1, const auto & e2){
  if(e1.second != e2.second){  //The second increasing
    return e1.second<e2.second;
  }
  return e1.first>e2.first; //The first decreasing.
});

vector<int> result;
result.reserve(nums.size());
for(auto [f, s] : v){
  for(int i = 0;i<s;++i){
    result.push_back(f);
  }
}
return result;
```

3.4.3.5 fancy code

- find all space, don't need while(find), use for() directly. find one max and second max, use for loop is ok, find three max, use min-heap, find three min, use max-heap.
- 1800, a typical boundary question, add one element to break the condition will reduce the programming complexity a lot. that is trick I should follow.

```
   int maxAscendingSum(vector<int>& nums) {
    int i = 0;

    nums.push_back(nums.back()-1);

    int n = nums.size();
    int sum{};
    int result{};
    while(i<n-1){
      if(nums[i]<nums[i+1]){
        sum +=nums[i];
      }
      else{
        sum += nums[i];
        if(sum>result){
          result = sum;
        }
        sum = 0;
      }
      ++i;
    }

```

```
23      return result;
24    }
```

- 2138, dumb node idea. add n%k fill to the end. For any complex boundary condition, add dumb node to make programming easier. that is a good idea.
- 2733, how to get middle of three?
- 2108, use more lambda

```
1 auto ll = [](const auto & e){
2   int n = e.size();
3   return equal(e.begin(), e.begin()+n/2, e.rbegin());
4 };
5
6 auto it = ranges::find_if(words, ll);
7 return it == words.end()? "" : *it;
```

- 1816, don't need split and stringstream. A classical answer is the bottom line, after you know the bottom line, then you need to think that if there is better method to do that.

```
1 string truncateSentence(string s, int k) {
2   for (int i = 0; i < s.size(); ++i) {
3     if (s[i] == ' ' && (--k) == 0) {
4       return s.substr(0, i);
5     }
6   }
7   return s;
8 }
```

3.4.3.6 views

- 2190, how to get best sell book name. Pay attention to the decltype and value_type usage.

```
1 auto vw = nums | views::slide(2);
2 unordered_map<int, int> um;
3 for(auto e: vw){
4   int f = *(e.begin());
5   int s = *(e.begin()+1);
6   if(f == key){
7     ++um[s];
8   }
9 }
10
11 1) use ranges.
12 2) use decltype.
13 3) use value_type.
14
15 auto it = ranges::max_element(um , {}, &decltype(um)::value_type::second);
16 return it->first;
```

- 2465, a typical question can be resolves by the STL algorithms very efficiently. Use std::set to calculate the different items count.

```
1  int distinctAverages(vector<int>& nums) {
2   set<int> s;
3   ranges::sort(nums);
4
5   transform(nums.begin(), nums.begin()+nums.size()/2, nums.rbegin(), inserter(s, s
        .begin() ), [](auto e1, auto e2){return e1+e2;});
6
```

```
  return s.size();

}
```

- leetcode 1556. use reverse, chunk and join_with common.

```
string thousandSeparator(int n) {
  string str = to_string(n);
  auto vw = str| views::reverse | views::chunk(3) | views::join_with('.') | views
      ::common;

  string result(vw.begin(), vw.end());

  ranges::reverse(result);
  return result;

}
```

- 561, Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2), ..., (an, bn) such that the sum of min(ai, bi) for all i is maximized. Return the maximized sum. In order to make the sum as large as possible, the b we choose should be as close to a as possible, so as to retain a larger number. Therefore, we can sort the array then divide every two adjacent numbers into a group, and add the first number of each group.
  You need to sort it first, then use views function to deal with it.

```
    ranges::sort(nums);
    auto vw = nums | views::chunk(2);
    int sum{};
    for(auto e: vw){
      int f = e.front();
      sum+=f;
    }
    return sum;
```

- 1694, pay attention to boundary check all the times.
- 3162, Use cartesian_product to simulate two for loop. cartesian_product returns the tuples.

```
 int numberOfPairs(vector<int>& nums1, vector<int>& nums2, int k) {
  int sum{};
  for(auto [f, s]: views::cartesian_product(nums1, nums2)){
    if(f % (s*k) == 0)
    ++sum;
  }
  return sum;
}
```

- 1662, Learn how to use views::join. Another useful function is views::join_with can be used to add , to number.

```
auto vw1 = word1 | views::join;
auto vw2 = word2 | views::join;

return ranges::equal(vw1, vw2);
```

3.4.3.7 subset and subarray

- 2062, how to get all substring? double pointer is O(n) algorithem, all substring must be O(n*n), so we can't use double pointer here.

- 2264, use adjacent_find to chunk equal. pay attention to the last statement after while.

```
 std::vector<int> v1{0, 1, 2, 3, 40, 40, 41, 41, 5, 5, 5};

auto it = v1.begin();
auto pos = v1.begin();
while( (pos = std::adjacent_find(it, v1.end(), not_equal_to{}) ) != v1.end()){
  for(auto i = it; i<next(pos); ++i){
    cout<<*i<<" ";
  }
  cout<<endl;
  it = pos+1;
}
if(it != v1.end()){
  cout<<"Yan";
  for(auto i = it; i<v1.end(); ++i){
    cout<<*i<<" ";
  }
  cout<<endl;
}
```

1. chunk the equal,
2. chunk the digit and alphabet. leetcode 1805, use chunk_by to seperate abc and number.

```
string str = "123abc456efg";

auto v = str | views::chunk_by([](auto e1, auto e2){
  if(::isdigit(e1)){
    if(::isdigit(e2))
    return true;
    return false;
  }
  else{
    if(::isalpha(e2))
    return true;
    return false;
  }
});

for(auto e: v){
  string temp{e.begin(), e.end()};
  cout<<temp<<endl;
}
```

3. only chunk the vowel. 2062, you can use cnt>=1, once meet consonant, cnt = 0; that will be easier. it's different with 2264, 2264 need to check two element to see if they are not equal, 2062 just check one element to see if it's consonant, so can implement a little easier.

- 1863, how to get subset, how to calculate sum? calculating the only one sum, don't need calculate the subsum of each subset, you can add each item in the subset in the last sum. The second question is how to get all subset? tow for doesn't work, because it's suitable for subarray. not subset. subset is pow(2, n), not n*n. Three comman ways 1) get n, build pow(2,n) bit, then use bit to select 2) recursive bucket point of view, 3) recursive ball point of view.

```
 int n = nums.size();
int ans = 0;
function<void (int, int)> dfs = [&](int i, int s) {
  if (i >= n) {
    ans += s;
    return;
  }
```

```
    dfs(i + 1, s);
    dfs(i + 1, s ^ nums[i]);
};
dfs(0, 0);
return ans;

//1) dfs should be function, auto doesn't work here.
//2) nums can use directly inside the lambda, because you have &
//3) s is subsum each subset, don't use reference here.
//4) Base case is i ==n, then accumulate s to the last result ans.
```

- 1588, dp, also subarray question, but have easy solution.
- 2913, how to get all subarray? use two for loop can get all subarray, use set to count all different nums.

3.4.3.8 tree

- 108 use recursive to build a tree;

```
TreeNode* buildTree(vector<int>& nums, int i, int j){
  if( i == j )
  return nullptr;


  int m = i+ (j-i)/2;
  int v = nums[m];
  TreeNode *root = new TreeNode(v);
  root->left = buildTree(nums, i, m);
  root->right = buildTree(nums, m+1, j);
  return root;
}
TreeNode* sortedArrayToBST(vector<int>& nums) {
  int n = nums.size();
  return buildTree(nums, 0, n);

}
```

- 101, use recursive to judge mirror, just like to judge if it's same. the same idea. recursive is power weapon.

```
 bool isSymmetric(TreeNode* root) {
  auto dfs = [&](this auto&& dfs, TreeNode* root1, TreeNode* root2) -> bool {
    if (root1 == root2) {
      return true;
    }
    if (!root1 || !root2 || root1->val != root2->val) {
      return false;
    }
    return dfs(root1->left, root2->right) && dfs(root1->right, root2->left);
  };
  return dfs(root->left, root->right);
}
```

- 700. Search in a Binary Search Tree

```
      TreeNode* searchBST(TreeNode* root, int val) {
        if(root == nullptr){
          return nullptr;
        }
        TreeNode* l = searchBST(root->left, val);
```

```
6         if(l)
7         return l;
8         if(root->val == val)
9         return root;
10        return searchBST(root->right, val);
11      }
```

- 572. Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

```
1       class Solution {
2         public:
3         bool isSubtree(TreeNode* root, TreeNode* subRoot) {
4           if (!root) {
5             return false;
6           }
7           return same(root, subRoot) || isSubtree(root->left, subRoot) ||
                isSubtree(root->right, subRoot);
8         }
9
10        bool same(TreeNode* p, TreeNode* q) {
11          if (!p || !q) {
12            return p == q;
13          }
14          return p->val == q->val && same(p->left, q->left) && same(p->right, q->
                right);
15        }
16      };
```

- 530 Minimum Absolute Difference in BST. This code demonstrates how to keep previous node in BST.

```
1     class Solution {
2       public:
3       TreeNode* prev;
4       int min = numeric_limits<int>::max();
5
6       void getM_re(TreeNode* root){
7         if(root == nullptr)
8         return;
9
10        getM_re(root->left);
11        if(prev != nullptr){
12          int temp = root->val-prev->val;
13          if(temp<min){
14            min = temp;
15          }
16        }
17        prev = root;
18        getM_re(root->right);
19      }
20      int getMinimumDifference(TreeNode* root) {
21        getM_re(root);
22        return min;
23      }
24    };
```

- 2331, no check leaf node, is it right? commonly, it's not right. only check if it's leaf code can't handle below tree. but for questions 2331, it states that it's full binary tree, so you can only check leaf code without handle base case.

```
    1
   / \
 null  2
```

3.4.3.9 bit

- trip

```
n>>1 // n/2
n&1  // n%2
n^1  // flip n

n&0xf //n %16.
```

- 405, how to modulus negative? common bit operation.

```
 string toHex(int num) {
  if (num == 0) return "0";
  string s = "";
  for (int i = 7; i >= 0; --i) {
    int x = (num >> (4 * i)) & 0xf;
    if (s.size() > 0 || x != 0) {
      char c = x < 10 ? (char) (x + '0') : (char) (x - 10 + 'a');
      s += c;
    }
  }
  return s;
}
```

- 2595, sizeof(n) must mulitpy 8

```
  // judge if a bit is 1? use if directly.
  if(n & (1<<index) ){


vector<int> evenOddBit(int n) {
  vector<int> ans(2);
  for (int i = 0; n > 0; n >>= 1, i ^= 1) {
    ans[i] += n & 1;
  }
  return ans;
}
```

- 2103, use bit to represent RGB color, should learn this trick.

```
vector<10, bitset<3> > result;
....
```

- 2220, how to understand $\hat{?}$ and how to know how many set bits in the binary represent.
- 3340, when to use stol? when not to use stol? stol to avoid overflow, how to flip flag.

```
bool isBalanced(string num) {
  int f[2]{};
  for (int i = 0; i < num.size(); ++i) {
    f[i & 1] += num[i] - '0';  //pay attention to i & i.
  }
  return f[0] == f[1];
}
```

- 2315, how to flip the bit? You are given a string s, where every two consecutive vertical bars '|' are grouped into a pair. In other words, the 1st and 2nd '|' make a pair, the 3rd and 4th '|' make a pair, and so forth.
  Return the number of '*' in s, excluding the '*' between each pair of '|'.

```
int countAsterisks(string s) {
  int ans = 0, ok = 1;
  for (char& c : s) {
    if (c == '*') {
      ans += ok;
    } else if (c == '|') {
      ok ^= 1;
    }
  }
  return ans;
}
```

- 2032, map<int, bitset<3> > a new data structure. Because the bitset support count() function, it's very useful.

3.4.3.10 Good modeling

- 2231, for 0-9 digit, you can use array to count the frequency, don't need std::map. similar with counting sort.
- 1854, so many detail. 1) priority_queue(smaller) 2) how to pop up priority_queue.
- 3238, use array, don't use map.
- 1175, how to get prime? The start point is 2. permutation number, use cnt*n-cnt.
- 1275, only last one decide who will win.
- 1886, how to rotate a array? map i,j –>j, i, i,j –> n-j, n-i.

0	i,j	i,j
90	i,j	j, n-i
180	i,j	n-i, n-j
270	i,j	n-j, i

  second –> first, first–> n-first, that is all.
- 2873 We can use two variables mx and mxdiff to maintain the maximum prefix value and maximum difference, respectively. When traversing the array, we update these two variables, and the answer is the maximum value of all
- leetcode 1566, 2765, different with 2760. 2760 can use chunk, but 2765 can't use chunk. has to use iterate method. They all want to get subarray wich satisfy certain contdition. single iterate, but maintain previous condition, if condition ok, update the last result, if condition not ok, initial to the initial condition. These three questions you need to remember, 1) how to check pattern? 2) odd and even can be used splitted so use chunk 3) increase and decrease can't be split, so use double iterate. for example 2 3 4 3 (2, 3) is answer, but 3 4 3 is also answer, it overlop on 3.
- leetcode 1184 A bus has n stops numbered from 0 to n - 1 that form a circle. We know the distance between all pairs of neighboring stops where distance[i] is the distance between the stops number i and (i + 1) % n. The bus goes along both directions i.e. clockwise and counterclockwise. Return the shortest distance between the given start and destination stops.

```
    int distanceBetweenBusStops(vector<int>& distance, int start, int destination)
        {
      int s = accumulate(distance.begin(), distance.end(), 0);
      int t = 0, n = distance.size();
      while (start != destination) {
```

```
        t += distance[start];
        start = (start + 1) % n;
      }
      return min(t, s - t);
    }
```

- 2515, how to deal with circular array. same with previous.
- 942, A permutation perm of n + 1 integers of all the integers in the range [0, n] can be represented as a string s of length n where: Input: s = "IDID" Output: [0,4,1,3,2]
  We can use two pointers low and high to represent the current minimum and maximum values, respectively. Then, we traverse the string s. If the current character is I, we add low to the result array, and increment low by 1; if the current character is D, we add high to the result array, and decrement high by 1. Finally, we add low to the result array and return the result array.
- 1037, how to decide if three points are in line, change divide to multiplication.
- 821, Given a string s and a character c that occurs in s, return an array of integers answer where answer.length == s.length and answer[i] is the distance from index i to the closest occurrence of character c in s. The distance between two indices i and j is abs(i - j), where abs is the absolute value function.
  Next, we traverse the string s from left to right, recording the most recent position of the character c as pre. For position i, the answer is $i - pre$, i.e., $ans[i] = i - pre$. Then, we traverse the string s from right to left, recording the most recent position of the character c as suf. For position i, the answer is $suf - i$, i.e., $ans[i] = \min(ans[i], suf - i)$. Finally, we return the answer array ans. The time complexity is $O(n)$, where n is the length of the string s. Ignoring the space consumed by the answer array, the space complexity is $O(1)$.
- 696, Given a binary string s, return the number of non-empty substrings that have the same number of 0's and 1's, and all the 0's and all the 1's in these substrings are grouped consecutively. Substrings that occur multiple times are counted the number of times they occur.

```
int countBinarySubstrings(string s) {
  int i = 0, n = s.size();
  vector<int> t;
  while (i < n) {
    int cnt = 1;
    while (i + 1 < n && s[i + 1] == s[i]) {
      ++cnt;
      ++i;
    }
    t.push_back(cnt);
    ++i;
  }


  int ans = 0;
  for (i = 1; i < t.size(); ++i)
  ans += min(t[i - 1], t[i]);
  return ans;
  //return 0;
}
```

- leetcode 263, Ugly Number. An ugly number is a positive integer which does not have a prime factor other than 2, 3, and 5. Given an integer n, return true if n is an ugly number.
- leetcode 292, Nim Game. One use or, the other use and, it will satisfy the requirement.

```
  bool NimRecursive(int n, bool flag){
    if( flag == true&& n<=3){
      return true;
```

```
    }
    if(flag == false && n<=3){
      return false;
    }
    if(flag){
      flag = flag ? false: true;
      if(NimRecursive(n-1, flag ) || NimRecursive(n-2, flag ) || NimRecursive(n-3,
          flag ) ){
        return true;
      }
      return false;
    }
    else{
      flag = flag ? false: true;
      if(NimRecursive(n-1, flag ) && NimRecursive(n-2, flag ) && NimRecursive(n-3,
          flag ) ){
        return true;
      }
      return false;
    }

  }
  bool canWinNim(int n) {
    bool flag = true;
    return NimRecursive(n, flag);
  }
```

- 463. Island Perimeter, for each 1 block, only consider right and down, then -=2, it's easier than considering four sides.

```
    int islandPerimeter(vector<vector<int>>& grid) {
      int m = grid.size(), n = grid[0].size();
      int ans = 0;
      for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
          if (grid[i][j] == 1) {
            ans += 4;
            if (i < m - 1 && grid[i + 1][j] == 1) ans -= 2;
            if (j < n - 1 && grid[i][j + 1] == 1) ans -= 2;
          }
        }
      }
      return ans;
    }
```

- 2908, keep left and right minimum, just like rain drop problem.
- 3010, once find min, replace with max, then do next find. flood island. Another method is to find two minimal elements.
- 2073, judge first, then ++ index. that is the basic style.
- 2506. Count Pairs Of Similar Strings

```
int similarPairs(vector<string>& words) {
  int ans = 0;
  unordered_map<int, int> cnt;
  for (const auto& s : words) {
    int x = 0;
    for (auto& c : s) {
      x |= 1 << (c - 'a');
    }
    ans += cnt[x]++;
```

```
    }
    return ans;
}
```

- 2848, update tail correct.

```
int numberOfPoints(vector<vector<int>>& nums) {
    ranges::sort(nums);
    int result{};
    vector<int> origin = nums[0];
    int index = 1;
    int n = nums.size();
    int tail = origin[1];

    for(; index <n; ++index){
        if(nums[index][0]<=tail){
            if(tail< nums[index][1]){
                tail = nums[index][1];
            }
        }
        else{
            result += tail - origin[0] +1;
            origin = nums[index];
            tail = origin[1];
        }
    }

    result += tail - origin[0]+1;

    return result;
}
```

- 2696, A typical usage of stack.
- 1700 usage of stack and queue, pay attention to stack push need reverse order. use stack, that is very good idea, need to remember it.

```
 int countStudents(vector<int>& students, vector<int>& sandwiches) {
    int cnt[2] = {0};
    for (int& v : students) ++cnt[v];
    for (int& v : sandwiches) {
        if (cnt[v]-- == 0) {
            return cnt[v ^ 1];
        }
    }
    return 0;
}
```

- 2932. Maximum Strong Pair XOR I. Learn how to build trie tree.

```
    https://www.geeksforgeeks.org/maximum-xor-of-two-nodes-in-a-tree/
```

- 2652, You need to know the inclusion-exclusion principle.
- 2928, Combination Mathematics + Principle of Inclusion-Exclusion. see the reference

```
    https://leetcode.cn/problems/distribute-candies-among-children-i/solutions
        /2522970/o1-rong-chi-yuan-li-pythonjavacgo-by-end-smj5/
```

3.5 Project

- drop of knowledge C++, sixth edition. includes all modern C++ 17/20 features. Such as move semantic, smart pointer, concurrency, brace initialization and generic programming.
- A compiler project. Intermediate representation, Whirl to LLVM.
- A Trade Repository or Swap Data Repository is an entity that centrally collects and maintains the records of over-the-counter (OTC) derivatives. CFTC. Commodity Futures Trading Commission. Athena is a cross-asset platform transforming technology at JP Morgan. It delivers innovative and efficient applications to a wide range of the firm's businesses, including sales, trading, operations, risk and research. Athena combined the best of open source technologies
- There are two main types of RFID tags: battery-operated and passive.
- Tell me about a time you worked hard at something but the end result was unsuccessful. Why? What did you learn?

  UH64 is a C++ compiler with its own intermediate representation (IR) called WHIRL. Clang, a highly successful commercial C++ compiler, uses LLVM as its IR. My role in the project was to convert WHIRL to LLVM. The goal was to extend LLVM's support for more CPUs while leveraging WHIRL's optimizations in certain areas, combining the strengths of both.

  The main challenge was the large codebase, which consists of 4 million lines of code. To address this, I focused primarily on the IR domain. I also used a powerful IDE called Understand, which helps analyze source code quickly and efficiently.
- What was the most demanding or time-consuming project or initiative in your recent work experience?

  We need to submit reports to the CFTC within 15 minutes, or we face a fine of $15 per transaction. Initially, we used a single server for this task, but due to the volatility of the FX market, which is influenced by politics, we decided to deploy an additional server.

  The reporting system is developed in Python, but since Python doesn't handle concurrency well, we implemented multiprocessing. To manage synchronization, we also used a database. Additionally, we developed algorithms to make the system scalable, similar to a simple Kubernetes setup.

  The results were positive. Each year, the CFTC provides a statistical report on all major banks, and JPMorgan ranked number one, with 98% of transactions being reported within 15 minutes.
- Can you tell me how you discovered a hidden requirement or non-obvious outcome in a recent project?

  move the bar to open the lid. But the bar
- Share an example of when you changed a design approach because of someone else's input/better idea?
- Can you describe a time when you were stuck on a project or solution?

  Double-free errors are very challenging to debug, especially in embedded applications. I attempted to use tools like Valgrind, but it wasn't supported in this environment. I also tried implementing a custom new operator, but it wouldn't compile successfully.

  After reviewing the source code, I discovered that both processes were using the same thread pool. I resolved the issue by switching to a shared_ptr, which fixed the memory problem. I was stuck on this for almost two weeks, but now I feel very confident in handling any memory-related issues, even in highly constrained environments.
- In the most recent case where a team member suggested a change that you disagreed with, what steps did you take to reach a resolution?

  I will develop a test case to approve my stand point view. Stop talking, Show me the source code.

- Vizio was acquired by Walmart and is currently undergoing a transformation, outsourcing technical work to offshore teams while focusing on the advertising business. I've been moved to developing GUI apps and interfaces, which doesn't allow me to fully use my skills in low-level algorithms and C++. That's why I'm looking for a new position.
- My wife is a resident doctor undergoing training at a hospital in Dallas. This is a rare opportunity and a family decision.
- I have expert-level knowledge of C/C++ and have published a book on C++ and you can purchase it from amazon website. I hold a Ph.D. and have a patent in algorithms, which enables me to handle complex algorithmic and machine learning challenges. I have experience working on a C++ compiler, giving me a strong understanding of optimization and machine code. I have many years of experience, including extensive teamwork and CI/CD expertise. I'm familiar with the entire software development process and tools, essentially making me a full-stack developer.