**Algorithm Overview**

**Algorithms and Data Structures**

**What is an algorithm**

Algorithm, a concept from the field of mathematics, is a formula and idea used to solve a certain type of problem. In the field of computer science, the essence of an algorithm is a series of program instructions used to solve specific calculation and logic problems.

There are simple algorithms and complex ones. There are efficient ones and there are poor ones. In the computer field, there are two important criteria for measuring the quality of an algorithm.

* time complexity
* Space complexity

Algorithms can be applied in many different fields and in a variety of application scenarios, such as calculation, search, sorting, optimal decision-making, etc.

**What is Data Structure**

Data structure is the organization, management and storage format of data. Its purpose is to access and modify data efficiently.

Data structures include linear data structures such as arrays and linked lists, as well as complex data structures such as trees and graphs.

**time complexity**

**What is time complexity**

Time complexity is a measure of the running time of an algorithm, expressed using the Big O method and denoted as T(n) = O(f(n)).

**Number of basic operations performed**

That is, the number of times a certain program code block is executed, recorded as n. This n can be a **constant** , **linear** , **logarithmically** calculated, or **polynomially** calculated.

**Asymptotic time complexity**

We T(n)simplify the relative execution time function of the program to an order of magnitude, which can be n, n2 , n3 , etc.

Several principles for deriving time complexity:

* If the running time is of constant magnitude, it is represented by a constant of 1.
* Keep only the highest order terms in the time function
* If the highest-order term exists, the coefficient in front of the highest-order term is omitted.

In summary, common time complexities are in order from low to high, including O(1), O(logn), O(n), O(nlogn), etc.O(n2)

**Space complexity**

**What is space complexity?**

Space complexity is a measure of the amount of storage space temporarily occupied by an algorithm during its operation, expressed using the Big O method and denoted as S(n) = O(f(n)).

**Calculation of space complexity**

Space complexity is similar to time complexity, with several different growth trends.

* Constant Space

When the storage space size of the algorithm is fixed and has no direct relationship with the input size, the space complexity is recorded as O(1).

* Linear Space

When the space allocated by the algorithm is a linear collection (such as an array), and the size of the collection is proportional to the input size n, the space complexity is recorded as O(n).

* Two-dimensional space

When the space allocated by the algorithm is a two-dimensional array collection, and the length and width of the collection are proportional to the input size n, the space complexity is recorded as .O(n2)

* Recursive Space

Recursion is special. When the computer executes a program, it will allocate a piece of memory to store the "method call stack". It includes two behaviors: **push** and **pop** . The memory space required to perform recursive operations is proportional to the recursive depth. Therefore, the space complexity of pure recursive operations is also linear. If the depth is n, then the space complexity is O(n).

In summary, common space complexities are in order from low to high, including O(1),,, etc. Among them O(n), the space complexity of the recursive algorithm is proportional to the recursive depth.O(n2)

**The trade-off between time and space**

Since the computing speed and space resources of computers are limited, people need to evaluate the time complexity and space complexity of algorithms to achieve the best use of computers. But often we need to sacrifice one aspect to achieve the other. In most cases, time complexity is more important, after all, we pursue faster program execution speed.

**Data structure basics**

**What is an array**

An array is an ordered collection of a finite number of variables of the same type. Each variable in the array is called an element.

Array is the simplest and most commonly used data structure. Its characteristics are as follows:

Arrays are stored sequentially in memory (memory is composed of continuous memory cells, each of which has its own address).

Each element in the array is stored in a small memory unit, and the elements are closely arranged. The storage order of the elements cannot be disrupted, nor can a storage unit be skipped for storage.

**Basic operations on arrays**

**Reading Elements**

Since arrays are stored sequentially in memory, the corresponding array elements can be read through the array subscript.

const arr = new Array(1, 2, 3, 4);

console.log(arr[2]) // 3

**Update Elements**

Also use the subscript to assign the value to the corresponding array element to modify the value.

const arr = new Array(1, 2, 3, 4);

arr[3] = 5

console.log(arr) // [ 1, 2, 3, 5 ]

**Inserting Elements**

Since the actual number of elements in an array may be less than the length of the array, there are three situations for inserting elements into the array.

* Insert at the end of the line

In this case, just place the element to be inserted in the free position at the end of the array.

const arr = new Array(1, 2, 3, 4);

arr[4] = 5

console.log(arr) // [1, 2, 3, 4, 5]

* Intermediate and out-of-range insertion

Since each element of the array has a fixed index, we have to first move the insertion position and the following elements backward to make room, and then put the element to be inserted into the corresponding array position. At the same time, we have to consider the situation of out-of-range insertion, that is, inserting new elements when the array is full.

/\*\*

\* Idea:

\* 1. First generate an array with a given capacity (capacity). The default number of actual elements is 0.

\* 2. Insert an element into the array, pass in the insertion position index, and the inserted element element

\* 3. Determine whether the incoming index is out of bounds. If it is out of bounds, an error will be thrown.

\* 4. Loop from right to left and move the elements one position backward one by one.

\* 5. Place the element to be inserted at the position corresponding to the index subscript of the array

\*

\* You need to pay attention here to the situation of out-of-range insertion. The solution is to create a new array that is twice as long as the original array, and copy all the elements in the original array to the new array to expand the array.

\*/

class MyArray {

constructor(capacity) {

this.array = new Array(capacity);

this.size = 0;

}

resize () {

const arrayNew = new Array(this.array.length \* 2);

for (let i = 0; i < this.array.length; i++) {

arrayNew[i] = this.array[i]

}

this.array = arrayNew

}

insert(index, element) {

// Boundary judgment

if (index < 0 || index > this.size) {

throw new Error(' Exceeds the actual element range of the array！')

}

// Insert out of range

if (this.size >= this.array.length) {

this.resize()

}

for (let i = this.size - 1; i >= index; i--) {

this.array[i + 1] = this.array[i]

}

this.array[index] = element;

this.size++;

}

output() {

for (let i = 0; i < this.size; i++) {

console.log(this.array[i])

}

}

}

const arr = new MyArray(4);

// arr.insert(-1, 3)

// arr.insert(5, 9)

arr.insert(0, 3)

arr.insert(1, 7)

arr.insert(2, 9)

arr.insert(3, 5)

arr.insert(1, 6)

arr.output() // 3 6 7 9 5

console.log(arr) // MyArray { array: [ 3, 6, 7, 9, 5, <3 empty items> ], size: 5 }

**Deleting an element**

The deletion and insertion processes of an array are opposite. If the deleted element is in the middle of the array, the elements behind it will be moved forward one position.

class MyArray {

constructor(capacity) {

this.array = new Array(capacity);

this.size = 0;

}

// ...

output() {

for (let i = 0; i < this.size; i++) {

console.log(this.array[i])

}

}

delete(index) {

if (index < 0 || index >= this.size) {

throw new Error(' Exceeds the actual element range of the array！')

}

const deletedElement = this.array[index];

// Loop from left to right and move the elements forward one position one by one.

for (let i = index; i < this.size - 1; i++) {

this.array[i] = this.array[i + 1]

}

this.size--;

return deletedElement

}

}

const arr = new MyArray(4);

arr.insert(0, 3)

arr.insert(1, 7)

arr.insert(2, 9)

arr.insert(3, 5)

arr.insert(1, 6)

arr.delete(3)

arr.output() // 3 6 7 5

**What is a linked list**

A linked list is a physically non-continuous and non-sequential data structure consisting of several nodes.

* Singly linked list

Each node of a singly linked list consists of two parts, one part is the variable data that stores the data, and the other part is the pointer next pointing to the next node.

The first node of the linked list is called the head node, the last node is called the tail node, and the next pointer of the tail node points to null.

* Doubly linked list

Compared with a unidirectional linked list, each node of a doubly linked list has a prev pointer pointing to the previous node in addition to data and next pointers.

The storage method of linked lists in memory is **random storage** . Each node of the linked list is distributed in different locations in the memory and is associated with each other by next. This can flexibly and effectively utilize scattered fragmented space.

**Basic operations on linked lists**

class Node {

constructor(data) {

this.data = data

this.next = null

}

}

class LinkedList {

constructor() {

this.head = new Node("head");

this.tail = new Node("tail");

this.size = 0;

}

// ...

}

**Find Node**

When searching for an element, the linked list needs to start from the head node and search backwards one node at a time through the next pointer.

// ...

/\*\*

\* @param { Find the position of an element} index

\* @memberof LinkedList

\*/

find(index) {

if (index < 0 || index > this.size) {

throw new Error(' Beyond the linked list node range！')

}

// Temporary variables, used for node replacement

let currentNode = this.head

for (let i = 0; i < index; i++) {

currentNode = currentNode.next

}

return currentNode;

}

// ...

**Update Node**

The update process of the linked list is also very simple, just replace the old data with the new data (replace data).

this.currentNode.data = 'newData';

**Insert Node**

When inserting a node into a linked list, there are three cases.

* Tail Insert

Just point the next pointer of the last node to the newly inserted node.

* Head Insertion

First, point the next pointer of the new node to the original head node, and then make the new node the head node of the linked list.

* Intermediate Insertion

First, the next pointer of the new node points to the node at the insertion position, and then the next pointer of the node before the insertion position points to the new node.

Elements can be inserted into the linked list infinitely without capacity limit.

// ...

/\*\*

\*

\* @param { data to insert} data

\* @param { where to insert} index

\* @memberof LinkedList

\*/

insert(data, index) {

if (index < 0 || index > this.size) {

throw new Error(' Beyond the linked list node range！')

}

// node to insert

let insertedNode = new Node(data);

//

if (this.size === 0) { // Empty linked lists

this.head = insertedNode

this.tail = insertedNode

} else if (index === 0) { // Head insertion

insertedNode.next = this.head

this.head = insertedNode

} else if (this.size === index) { // Tail insertion

this.tail.next = insertedNode

this.tail = insertedNode

} else { // Insert in the middle

let prevNode = this.find(index - 1); // Get the previous node

insertedNode.next = prevNode.next;

prevNode.next = insertedNode

}

this.size++;

}

// ...

let llist = new LinkedList();

llist.insert(3, 0)

llist.insert(7, 1)

llist.insert(9, 2)

llist.insert(5, 3)

llist.insert(6, 1)

llist.output() // 3 6 7 9 5

**Deleting a Node**

The deletion operation of the linked list is also divided into three cases.

* Tail Delete

Just point the next pointer of the second-to-last node to null.

* Header Delete

Just set the head node of the linked list to the next pointer of the original head node.

* Middle Delete

Just point the next pointer of the predecessor node of the node to be deleted to the next node of the node to be deleted.

// ...

/\*\*

\* Linked list deletion

\* @param {Find the location of the element} index

\* @memberof LinkedList

\*/

remove(index) {

if (index < 0 || index > this.size) {

throw new Error('Out of the range of linked list nodes！')

}

//

let removedNode = new Node(null);

if (index === 0) { // Delete the head node

removedNode = this.head;

this.head = this.head.next;

} else if (index === this.size - 1) { // Delete the tail node

let prevNode = this.find(index - 1);

removedNode = prevNode.next;

prevNode.next = null;

this.tail = prevNode;

} else { // Delete the intermediate node

let prevNode = this.find(index - 1);

let nextNode = this.find(index + 1); // prevNode.next.next;

removedNode = prevNode.next;

prevNode.next = nextNode

}

this.size--;

return removedNode;

}

// ...

let llist = new LinkedList();

llist.insert(3, 0)

llist.insert(7, 1)

llist.insert(9, 2)

llist.insert(5, 3)

llist.insert(6, 1)

llist.remove(0)

llist.output() // 6 7 9 5

Through learning, we know that arrays and linked lists are both linear data structures. So, let's compare their performance.

| **--** | **Find** | **renew** | **insert** | **delete** |
| --- | --- | --- | --- | --- |
| Arrays | O(1) | O(1) | O(n) | O(n) |
| Linked List | O(n) | O(1) | O(1) | O(1) |

It can be seen that arrays can quickly locate elements, and are more advantageous in scenarios with more read operations and fewer write operations. Linked lists are more flexible in inserting and deleting elements, and are more advantageous in scenarios where elements need to be frequently inserted and deleted at the end.

**Stacks and Queues**

**What is a stack**

A stack is a linear data structure. The elements in a stack can only **be entered in a first-in-last-out (** FILO) order. The location where the earliest element is stored is called **the bottom of the stack** , and the location where the last element is stored is called **the top of the stack** .

A stack can be implemented using an array or a linked list.

**Basic operations of stack**

**Push**

The push operation is to put elements into the stack. Elements are only allowed to be placed from the top of the stack. The position of the new element will become the new top of the stack.

**Pop**

The pop operation (pop) is to pop an element from the stack. Only the element at the top of the stack is allowed to be popped, and the element before the popped element will become the new top of the stack.

class Stack {

constructor() {

this.data = [];

// Position at the top of the stack

this.top = 0;

}

// into the stack

push(element) {

this.data[this.top++] = element

}

// Out of the stack

pop() {

return this.data[--this.top]

}

// Return to the top element of the stack

peek() {

return this.data[this.top - 1]

}

// Returns the length of the stack

size() {

return this.top

}

clear() {

this.top = 0

this.data = []

}

output() {

for (let i = 0; i < this.data.length; i++) {

console.log(this.data[i])

}

}

}

let stack = new Stack();

stack.push(1)

stack.push(2)

stack.push(3)

stack.push(4)

// console.log(stack.peek()) // 4

// stack.output() // 1 2 3 4

console.log(stack.pop()) // 4

console.log(stack.size()) // 3

stack.clear()

console.log(stack.size()) // 0

**What is a queue**

A queue is a linear data structure. Elements in a queue can only **be entered in** a first-in-first-out (FIFO) order. The exit of a queue is called the front, and the entry is called the rear.

Queues can be implemented using arrays or linked lists.

**Basic operations of queues**

**Join the team**

Enqueue is to put new elements into the queue. Elements are only allowed to be placed at the end of the queue. The next position of the new element will become the new end of the queue.

**Team Out**

Dequeue is to remove elements from the queue. Elements are only allowed to be removed from the head of the queue. The last element of the dequeued element will become the new head of the queue.

class Queue {

constructor() {

this.data = [];

}

front() {

return this.data[0]

}

rear() {

return this.data[this.data.length - 1];

}

enqueue(element) {

this.data.push(element);

}

dequeue() {

return this.data.shift();

}

empty() {

return this.data.length == 0 ? true : false;

}

output() {

for (let i = 0; i < this.data.length; i++) {

console.log(this.data[i])

}

}

}

let queue = new Queue();

queue.enqueue(1)

queue.enqueue(2)

queue.enqueue(3)

queue.enqueue(4)

// queue.output() // 1 2 3 4

queue.dequeue()

console.log(queue.front()) // 2

console.log(queue.rear()) // 4

queue.dequeue()

queue.dequeue()

queue.dequeue()

console.log(queue.empty()) // true

**Hash Table**

Hash table is also called hash table. This data structure provides a mapping relationship between key and value. As long as a key is given, the matching value can be found efficiently.

Hash function  
In some way, the key and the array index are converted. The simplest way to achieve this conversion is to perform a modulo operation according to the length of the array.  
index = HashCode(key) % Arrary.length

**Hash table read and write operations**

**Write operation (put)**

A write operation is to insert a new key-value pair into the hash table.

First, the key is converted into an array index through a hash function. Then, if there is no element at the position corresponding to the array index, the new key-value pair is filled into this position.

However, since different keys may have the same subscript obtained through the hash function, **a hash conflict** occurs . There are two main methods to resolve hash conflicts: **open addressing** and **linked list** .

* Open addressing

When a key is found to have an occupied array index through the hash function, we can look for the next empty position.

* Linked List Method

The key-value pair object points to the next node through the next pointer. When the new object is mapped to the array position that conflicts with it, it only needs to be inserted into the corresponding linked list.

**Read operation (get)**

The read operation is to find the corresponding value in the hash table through the given key.

First, the hash function is used to convert the element into a simple index. Then, if the element's key already exists in the array, it has been found, otherwise it continues to search.

**Resize**

When the hash table reaches a certain saturation after multiple element insertions, the probability of key mapping position conflicts will gradually increase. To resolve the conflicts, it is necessary to expand the length, that is, expand the capacity.

There are two factors that affect capacity expansion.

* Capacity, which is the current length of the hash table
* LoadFactor, that is, the load factor of the hash table, the default is 0.75f

The conditions for expansion are as follows:

HashMap.Size >= Capacity x LoadFactor

This expansion process includes two steps:

1. Expand and create a new empty array whose length is twice that of the original array

2. Re-hash: Traverse the original array and re-hash all objects into the new array

class HashTable {

constructor() {

this.table = new Array(137);

}

simpleHash(data) {

let total = 0;

for (let i = 0; i < data.length; ++i) {

total += data.charCodeAt(i);

}

console.log("Hash value: " + data + " -> " + total);

return total % this.table.length;

}

put(data) {

let pos = this.simpleHash(data);

this.table[pos] = data;

}

output() {

for (let i = 0; i < this.table.length; ++i) {

if (this.table[i] != undefined) {

console.log(i + ": " + this.table[i]);

}

}

}

}

let hashtable = new HashTable();

let someNames = ["David", "Jennifer", "Donnie", "Raymond",

"Cynthia", "Mike", "Clayton", "Danny", "Jonathan"];

for (let i = 0; i < someNames.length; ++i) {

hashtable.put(someNames[i]);

}

hashtable.output()

/\*\*

Hash value: David -> 488

Hash value: Jennifer -> 817

Hash value: Donnie -> 605

Hash value: Raymond -> 730

Hash value: Cynthia -> 720

Hash value: Mike -> 390

Hash value: Clayton -> 730

Hash value: Danny -> 506

Hash value: Jonathan -> 819

35: Cynthia

45: Clayton

57: Donnie

77: David

95: Danny

116: Mike

132: Jennifer

134: Jonathan

\*/

**Tree**

A tree is a finite set of n (n>=0) nodes. When n=0, it is called an empty tree. In any non-empty tree, there are the following characteristics.

1. There is only one specific node called the root.

2. When n>1, the remaining nodes can be divided into m (m>0) non-intersecting finite sets, each of which is itself a tree and is called a subtree of the root.

The maximum number of levels in a tree is called the height or depth of the tree.

**What is a Binary Tree**

A binary tree is a special form of tree. Each node of this tree **has at most two child nodes** . The two child nodes of a binary tree node are called **the left child** and **the right child** . The order of the two child nodes is fixed and cannot be reversed.

* Full Binary Tree

A binary tree is a full binary tree if all non-leaf nodes have left and right children, and all leaf nodes are on the same level. Every branch of a full binary tree is full.

* Complete Binary Tree

For a binary tree with n nodes, all nodes are numbered in hierarchical order, and the nodes are numbered from 1 to n. If all the nodes of this tree are in the same position as the nodes numbered from 1 to n in a full binary tree of the same depth, then this binary tree is a complete binary tree.

What physical storage structures can be used to represent a binary tree?

1. Chain storage structure

Each node of a binary tree consists of 3 parts.

* Data variables to store data
* left pointer pointing to the left child
* right pointer pointing to the right child

2. Array

When using array storage, the nodes of the binary tree are placed in the corresponding positions in the array in hierarchical order. If the left child or right child of a node is vacant, the corresponding position in the array is also vacant.

**Applications of Binary Trees**

**Find**

Binary search tree is used for search operations.

* If the left subtree is not empty, then the values ​​of all nodes in the left subtree are smaller than the root node.
* If the right subtree is not empty, then the values ​​of all nodes in the right subtree are greater than the root node
* The left and right subtrees are also binary search trees

For a binary search tree with a **relatively balanced node distribution , if the total number of nodes is n, then the time complexity of searching the nodes is O(logn), which is the same as the depth of the tree.**

**Maintain relative order**

A binary search tree requires that the left subtree is smaller than the parent node and the right subtree is larger than the parent node. This ensures the order of the binary tree, so it is also called a binary sort tree.

**Traversal of a binary tree**

From the perspective of the positional relationship between nodes, there are four types of binary tree traversal.

1. Pre-order traversal

The output order of pre-order traversal of a binary tree is root node, left subtree, and right subtree.

2. In-order traversal

The in-order traversal of a binary tree outputs in the order of left subtree, root node, and right subtree.

3. Post-order traversal

For a post-order traversal of a binary tree, the output order is left subtree, right subtree, and root node.

/\*\*

\* Binary lookup tree (also known as binary sort tree).

\* This is done in the form of a linked list

\*/

class TreeNode {

constructor(data) {

this.data = data

this.left = null

this.right = null

}

}

class BinarySearchTree {

constructor() {

this.root = null

}

// Inserts a node into the tree, and determines whether the size is inserted to the left or right

insert(data) {

let newNode = new TreeNode(data);

let insertNode = function (root, newNode) {

if (newNode.data < root.data) {

if (root.left === null) {

root.left = newNode

} else {

insertNode(root.left, newNode)

}

} else {

if (root.right === null) {

root.right = newNode

} else {

insertNode(root.right, newNode)

}

}

}

if (this.root === null) {

this.root = newNode;

} else {

insertNode(this.root, newNode)

}

}

// Find a node in the tree

find(data) {

let findNode = function (node, key) {

if (node === null) return null;

if (key < node.data) {

return findNode(node.left, key)

} else if (key > node.data) {

return findNode(node.right, key)

} else {

return node

}

}

return findNode(this.root, data)

}

// Smallest node

min(node = this.root) {

let minNode = function (node) {

if (node === null) return null;

while (node && node.left !== null) {

node = node.left

}

return node

}

return minNode(node)

}

// Largest node

max(node = this.root) {

let maxNode = function (node) {

if (node === null) return null;

while (node && node.right !== null) {

node = node.right

}

return node

}

return maxNode(node)

}

// Pre-order traversal

preOrderTraveral(callback) {

let preOrderTraveralNode = function(node, callback) {

if (node !== null) {

callback(node.data)

preOrderTraveralNode(node.left, callback)

preOrderTraveralNode(node.right, callback)

}

}

preOrderTraveralNode(this.root, callback)

}

// Middle-order traversal

inOrderTraveral(callback) {

let inOrderTraveralNode = function(node, callback) {

if (node !== null) {

inOrderTraveralNode(node.left, callback)

callback(node.data)

inOrderTraveralNode(node.right, callback)

}

}

inOrderTraveralNode(this.root, callback)

}

// Post-order traversal

postOrderTraveral(callback) {

let postOrderTraveralNode = function(node, callback) {

if (node !== null) {

postOrderTraveralNode(node.left, callback)

postOrderTraveralNode(node.right, callback)

callback(node.data)

}

}

postOrderTraveralNode(this.root, callback)

}

// Removes a node from the tree

remove(data) {

let removeNode = function(node, key) {

if (node === null) return null;

if (key < node.data) {

node.left = removeNode(node.left, key)

return node

} else if (key > node.data) {

node.right = removeNode(node.right, key)

return node

} else {

// Scenario 1: There are no child nodes

if (node.left === null && node.right === null) {

node = null

return node

}

// Scenario 2: There is only one child node

if (node.left === null) {

node = node.right

return node

} else if (node.right === null) {

node = node.left

return node

}

// Situation 3: Both the left child and the right child are present

let temp = this.min(node.left)

node.data = temp.data

node.right = removeNode(node.right, temp.data)

return node

}

}

this.root = removeNode(this.root, data)

}

}

let output = function(data) {

console.log(data)

}

let btree = new BinarySearchTree();

btree.insert(3)

btree.insert(4)

btree.insert(2)

btree.insert(7)

btree.insert(9)

btree.remove(7)

console.log('Pre-order traversal：')

btree.preOrderTraveral(output) // 3 2 4 7 9

console.log('Middle-order traversal：')

btree.inOrderTraveral(output) // 2 3 4 7 9

console.log('Post-order traversal：')

btree.postOrderTraveral(output) // 2 9 7 4 3

4. Level-order traversal

From a more macro perspective, binary tree traversals can be divided into two categories.

1. Depth-first traversal (pre-order traversal, in-order traversal, post-order traversal)

2. Breadth-first traversal (level-order traversal)

As the name implies, level-order traversal is to traverse each node of a binary tree layer by layer according to the hierarchical relationship from the root node to the leaf node. There is no direct relationship between nodes at the same level of a binary tree. This can be achieved with the help of a data structure such as **a queue .**

**Binary Heap**

A binary heap is essentially a complete binary tree, which comes in two types.

1. **Max Heap**

The value of any parent node of a maximum heap is greater than or equal to the value of its left and right child nodes.

1. **Minimum Heap**

The value of any parent node of the minimum heap is less than or equal to the value of its left and right child nodes.

The root node of a binary heap is called **the top of the heap** . The top of a max heap is **the largest element** in the entire heap , and the top of a min heap is **the smallest element** in the entire heap .

**Self-adjustment of binary heap**

For binary heaps, there are several operations, all based on the self-adjustment of the heap, that is, adjusting a complete binary tree that does not conform to the properties of a heap into a heap.

1. Insert Node

Insert the element from the last position of the binary tree, and then adjust the position according to the characteristics of the heap ("float up" after comparing the sizes), and finally achieve a result that conforms to the heap.

1. Deleting a Node

The deletion process is the opposite of inserting a node. The deleted node is the top element of the heap. Then, in order to complete the structure of the binary tree, the last node of the heap is temporarily replaced at the original top position of the heap, and then the position of each node is adjusted according to the heap's value.

1. Constructing a binary heap

Constructing a binary heap means adjusting a completely unordered complete binary tree into a binary heap, and the essence is to let all non-leaf nodes "sink" in turn.

Binary heap is the basis for implementing heap sort and priority queue.

/\*\*

\* Binary pile

\* This is done in an array

\*

\* Assuming that the parent node's subscript is parent, then its left child's subscript is 2xparent+1, and its right child's subscript is 2xparent+2.

\*/

/\*\*

\* "Float" adjustment

\* @param {\*} array Heaps to be adjusted

\*/

function upAdjust(array) {

let childIndex = array.length - 1;

let parentIndex = (childIndex - 1) / 2;

// tempSaves the inserted leaf node values for final assignment

let temp = array[childIndex];

while (childIndex > 0 && temp < array[parentIndex]) {

// It is not a true exchange, and a one-way assignment is sufficient

array[childIndex] = array[parentIndex]

childIndex = parentIndex

parentIndex = (parentIndex - 1) / 2;

}

array[childIndex] = temp

}

/\*\*

\* "Sinking" adjustment

\* @param {\*} array Heaps to be adjusted

\* @param {\*} parentIndex The parent node to be "sunk".

\* @param {\*} length The effective size of the heap

\*/

function downAdjust(array, parentIndex, length) {

// temp Saves the parent node value for the final assignment

let temp = array[parentIndex];

let childIndex = parentIndex \* 2 + 1;

while (childIndex < length) {

// If there is a right child, and the right child is less than the value of the left child, the right child is located

if (childIndex + 1 < length && array[childIndex + 1] < array[childIndex]) {

childIndex++;

}

// If the parent node is less than the value of any child, it will jump out

if (temp < array[childIndex]) {

break;

}

// It is not a true exchange, and a one-way assignment is sufficient

array[parentIndex] = array[childIndex]

parentIndex = childIndex

childIndex = childIndex \* 2 + 1

}

array[parentIndex] = temp

}

/\*\*

\* Build the heap

\* @param {\*} array Heaps to be adjusted

\*/

function buildHeap(array) {

// Starting with the last non-leaf node, make the "Sink " adjustment in turn

for (let i = (array.length - 2)/2; i >= 0; i--) {

downAdjust(array, i, array.length)

}

}

function output(array) {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

let array = [1, 3, 2, 6, 5, 7, 8, 9, 10, 0];

upAdjust(array)

// output(array) // 1 3 2 6 0 7 8 9 10 5

buildHeap(array)

// output(array) // 0 1 2 6 3 7 8 9 10 5

**Priority Queue**

The priority queue no longer follows the first-in-first-out rule, but is divided into two cases.

* The maximum priority queue, regardless of the order of entry, is the current largest element first out of the queue
* The minimum priority queue, regardless of the order of entry, is the current smallest element first out of the queue

**Implementation of priority queue**

Because binary heap has such characteristics.

1. The top of a max heap is **the largest element in the entire heap**
2. The top of a minimum heap is **the smallest element in the entire heap.**

Therefore, the maximum priority queue can be implemented with a maximum heap, so that each queue entry operation is an insertion operation of the heap, and each queue exit operation is to delete the top node of the heap. Similarly, the minimum priority queue can be implemented with a minimum heap.

/\*\*

\* Priority queues

\* This is done in a binary heap

\*/

class PriorityQueue {

constructor() {

this.array = new Array()

this.size = 0

}

enqueue(key) {

this.array[this.size++] = key

this.upAdjust()

}

dequeue() {

if (this.size <= 0) {

throw new Error('The queue is empty！')

}

Get the top element of the heap

let head = this.array[0]

Move the last element to the top of the pile

this.array[0] = this.array[--this.size]

this.downAdjust()

return head

}

upAdjust() {

let childIndex = this.size - 1;

let parentIndex = (childIndex - 1) / 2;

// temp holds the inserted leaf node value for the final assignment

let temp = this.array[childIndex];

while (childIndex > 0 && temp > this.array[parentIndex]) {

// It is not a true exchange, and a one-way assignment is sufficient

this.array[childIndex] = this.array[parentIndex]

childIndex = parentIndex

parentIndex = parentIndex / 2;

}

this.array[childIndex] = temp

}

downAdjust() {

let parentIndex = 0

let temp = this.array[parentIndex];

let childIndex = 1;

while (childIndex < this.size) {

// If there is a right child, and the right child is less than the value of the left child, the right child is located

if (childIndex + 1 < this.size && this.array[childIndex + 1] < this.array[childIndex]) {

childIndex++;

}

// If the parent node is greater than the value of any one child, it will jump out directly

if (temp >= this.array[childIndex]) {

break;

}

// It is not a true exchange, and a one-way assignment is sufficient

this.array[parentIndex] = this.array[childIndex]

parentIndex = childIndex

childIndex = childIndex \* 2 + 1

}

this.array[parentIndex] = temp

}

}

let pqueue = new PriorityQueue();

pqueue.enqueue(3)

pqueue.enqueue(5)

pqueue.enqueue(10)

pqueue.enqueue(2)

pqueue.enqueue(7)

console.log('Out of the queue element：', pqueue.dequeue()) // 5

console.log('Out of the queue element：', pqueue.dequeue()) // 7

**Sorting Algorithm**

According to the different time complexities, mainstream sorting algorithms can be divided into three categories.

1. A sorting algorithm with a time complexity of O(n 2 ).

* Bubble Sort
* Selection Sort
* Insertion Sort
* Shell sort (its performance is better than O(n 2 ), but not as good as O(nlogn).

2. A sorting algorithm with a time complexity of O(nlogn).

* Quick Sort
* Merge Sort
* Heap Sort

3. Sorting with linear time complexity.

* Counting Sort
* Bucket sort
* Radix sort

**In addition, it can also be divided into stable sorting** and **unstable sorting** according to the stability of its algorithm . That is, if the elements with the same value still maintain the order before sorting after sorting, it is stable; if the elements with the same value disrupt the order before sorting after sorting, it is unstable.

| **Sorting Algorithm** | **Average time complexity** | **Worst time complexity** | **Space complexity** | **Is the sort stable?** |
| --- | --- | --- | --- | --- |
| Bubble Sort | O(n 2 ) | O(n 2 ) | O(1) | Stablize |
| Selection Sort | O(n 2 ) | O(n 2 ) | O(1) | Unstable |
| Insertion Sort | O(n 2 ) | O(n 2 ) | O(1) | Stablize |
| Shell sort | O(nlogn) ~ O(n 2 ) | O(n 2 ) | O(1) | Unstable |
| Quick Sort | O(nlogn) | O(n 2 ) | O(nlogn) | Unstable |
| Merge Sort | O(nlogn) | O(nlogn) | O(n) | Stablize |
| Heap Sort | O(nlogn) | O(nlogn) | O(1) | Unstable |
| Counting Sort | O(n + m) | O(n + m) | O(m) | Stablize |
| Bucket sort | O(n + m) | O(n 2 ) | O(n + m) | Stablize |
| Radix sort | O(nm) | O(nm) | O(n + m) | Stablize |

**Bubble Sort**

Bubble sort is a basic exchange sort.

Idea: Compare adjacent elements in pairs. When one element is greater than the adjacent element on the right, swap their positions; when one element is less than or equal to the adjacent element on the right, the position remains unchanged.

Bubble sort is a stable sort. Since the sorting algorithm traverses all elements in each round, a total of (number of elements - 1) rounds are traversed, so the average time complexity is O(n 2 ).

Code:

function sort(array) {

for (let i = 0; i < array.length - 1; i++) {

for (let j = 0; j < array.length - 1 - i; j++) {

let tmp = 0;

if (array[j] > array[j+1]) {

tmp = array[j]

array[j] = array[j+1]

array[j+1] = tmp

}

}

}

}

function output() {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

const array = [5, 8, 6, 3, 9, 2, 1, 7];

sort(array)

output() // 1 2 3 5 6 7 8 9

**Quick Sort**

Quick sort is also an exchange sort. Unlike bubble sort, it selects a base element in each round and moves other elements larger than it to one side of the sequence and elements smaller than it to the other side of the sequence, thus splitting the sequence into two parts. This idea is called **divide and conquer** .

Under the idea of ​​divide and conquer, the original list is split into two parts in each round, and each part is split into two parts in the next round until it can no longer be divided. Each round of comparison and exchange requires traversing all elements, so the time complexity of quick sort is O(nlogn).

**Selection of datum elements**

The pivot element is the center of the divide-and-conquer process, and other elements are moved to its left and right sides.

The simplest way to determine the base element is to select the first element of the sequence. However, in special cases, there will be problems. The solution is to randomly select an element as the base element and swap the base element with the first element of the sequence.

**Exchange of elements**

After selecting the base element, swap all elements smaller than it to one side of it, and all elements larger than it to the other side. There are two ways to implement this.

1. Bilateral Round Robin

2. One-sided loop method

Code:

/\*\*

\* Sorting algorithm

\* Quick sort

\*/

function quickSort(arr, startIndex, endIndex) {

// The condition under which the recursion ends: startIndex >= endIndex

if (startIndex >= endIndex) {

return;

}

// Get the position of the datum element

// let pivotIndex = partition(arr, startIndex, endIndex);

let pivotIndex = partition2(arr, startIndex, endIndex);

// According to the datum element, it is divided into two parts for recursive sorting

quickSort(arr, startIndex, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, endIndex);

}

/\*\*

\* Divide and conquer (bilateral circulation）

\* arr Arrays to be swapped

\* startIndex Start subscript

\* endIndex End subscript

\*/

function partition(arr, startIndex, endIndex) {

// Take the element with the first position (you can also randomly select the //position) as the base element

let pivot = arr[startIndex];

let left = startIndex;

let right = endIndex;

while (left !== right) {

// Control the right pointer to compare and move left

while (left < right && arr[right] > pivot) {

right--;

}

// Controls the left pointer comparison and moves it to the right

while (left < right && arr[left] <= pivot) {

left++;

}

// Swap the element to which the left and right pointers point

if (left < right) {

let tmp = arr[left];

arr[left] = arr[right];

arr[right] = tmp;

}

}

// pivot and pointer coincidence point swapping

arr[startIndex] = arr[left];

arr[left] = pivot;

return left;

}

/\*\*

\* Divide and conquer (unilateral circulation).

\* arr array to be swapped

\* startIndex Start subscript

\* endIndex End subscript

\*/

function partition2(arr, startIndex, endIndex) {

// The element with the first position (or a random location) is taken as the base // element

let pivot = arr[startIndex];

let mark = startIndex;

for (let i = startIndex + 1; i <= endIndex; i++) {

if (arr[i] < pivot) {

mark++;

let tmp = arr[mark];

arr[mark] = arr[i];

arr[i] = tmp;

}

}

arr[startIndex] = arr[mark];

arr[mark] = pivot;

return mark;

}

function output(array) {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

const array = [4, 4, 6, 5, 3, 2, 8, 1];

quickSort(array, 0, array.length - 1);

output(array) // 1 2 3 4 4 5 6 8

**Heap Sort**

According to the characteristics of the binary heap, the top of the maximum heap is the largest element in the entire heap, and the top of the minimum heap is the smallest element in the entire heap. In this way, each time the old top of the heap is deleted, the size of the new top of the heap is adjusted to be the node second only to the old top of the heap. So as long as the top of the heap is deleted repeatedly and the binary heap is adjusted repeatedly, the resulting set will be an ordered set.

Steps of Heap Sort Algorithm:

1. Build the unordered array into a binary heap. If it needs to be sorted from small to large, it will form a maximum heap; if it needs to be sorted from large to small, it will form a minimum heap.

2. Loop through the top elements of the heap and replace them at the end of the binary heap, adjusting the heap to create a new top.

/\*\*

\* Sorting algorithm

\* Heap sorting

\*/

/\*\*

\* "Sinking" adjustment

\* array Heaps to be adjusted

\* parentIndex The parent node to be "sunk".

\* length The effective size of the heap

\*/

function downAjust(array, parentIndex, length) {

// temp Saves the parent node value for the final assignment

let temp = array[parentIndex];

let childIndex = 2 \* parentIndex + 1;

while (childIndex < length) {

// If there is a right child, and the right child is greater than the value of the left child, the right child is located

if (childIndex + 1 < length && array[childIndex + 1] > array[childIndex]) {

childIndex++;

}

// If the parent node is greater than the value of any one of the children, the loop is jumped

if (temp >= array[childIndex]) {

break;

}

// One-way assignment

array[parentIndex] = array[childIndex]

parentIndex = childIndex

childIndex = 2 \* childIndex + 1

}

array[parentIndex] = temp

}

function output(array) {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

/\*\*

\* Heap sorting

\* @param {\*} array Heaps to be adjusted

\*/

function heapSort(array) {

// 1.Build an unordered array into a maximum heap

for (let i = (array.length - 2)/2; i >= 0; i--) {

downAjust(array, i, array.length)

}

output(array)

// 2. Loop through the deletion of the top element of the heap, move it to the tail // of the collection, and adjust the heap to create a new heap

for (let i = array.length; i > 0; i--) {

// The last element is swapped with the first element

let temp = array[i];

array[i] = array[0];

array[0] = temp;

// Sink adjusts the maximum heap

downAjust(array, 0, i)

}

}

const array = [1, 3, 2, 6, 5, 7, 8, 9, 10, 0];

heapSort(array);

output(array) // 0 1 2 3 5 6 7 8 9 10

**Counting Sort**

Counting sort works as follows,

1. Get the maximum value of the sequence

2. Determine the length of the statistical array based on the maximum value of the series

3. Traverse the array to fill the statistical array

4. Traverse the statistical array and output the results

function countSort(array) {

// 1.Get the maximum value of the series

let max = array[0];

for (let i = 1; i < array.length; i++) {

if (array[i] > max) {

max = array[i]

}

}

// 2.Determine the length of the statistical array based on the maximum value of the series

let countArray = new Array(max + 1);

countArray.fill(0);

// 3.Traversing arrays populate statistical arrays

for (let i = 0; i < array.length; i++) {

countArray[array[i]]++;

}

// 4.Iterate through the statistical array and output the result

let index = 0;

let sortedArray = new Array(array.length);

for (let i = 0; i < countArray.length; i++) {

for (let j = 0; j < countArray[i]; j++) {

sortedArray[index++] = i

}

}

return sortedArray;

}

function output(array) {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

const array = [4, 4, 6, 5, 3, 2, 8, 1, 7, 5, 6, 0, 10];

const sortArray = countSort(array);

output(sortArray)

limitation:

1. When the difference between the maximum and minimum values ​​of a sequence is too large, counting sort is not suitable. This will cause a waste of space and increase the time complexity.

2. It is also not suitable when the elements of the sequence are not integers. Decimals cannot create corresponding statistical arrays.

**Bucket sort**

Bucket sort is also a linear time sorting algorithm. It is similar to the counting group created by counting sort, and it needs to create several buckets to assist in sorting. Each bucket represents an interval range, which can hold one or more elements.

The working principle is as follows,

1. Create buckets and determine the range of each bucket

The number of created buckets is equal to the number of elements in the original sequence. Except for the last bucket which only contains the maximum value of the sequence, the intervals of the previous buckets are determined according to the proportion.

Interval span = (maximum value - minimum value) / (number of buckets - 1)

2. Traverse the original sequence and put the elements into each bucket according to their numbers.

3. Sort the elements in each bucket separately.

4. Traverse all buckets and output all elements.

function bucketSort(array) {

// The maximum and minimum values of the series are obtained, and the difference d // is calculated

let max = array[0];

let min = array[0];

for (let i = 1; i < array.length; i++) {

if (array[i] > max) {

max = array[i]

}

if (array[i] < min) {

min = array[i]

}

}

let d = max - min; // Difference

// 2.Initialize the bucket

let bucketNum = array.length; // The number of buckets

let range = d / (bucketNum - 1); // interval

let listArray = []; // Stored by number

// 3.Iterate through the original sequence, placing each element in the bucket

for (let i = 0; i < array.length; i++) {

// Calculate the number of the bucket

let index = Math.floor((array[i] - min) / range);

// 4.Sort the inside of each bucket to determine whether there is already a value in the bucket, and if there is a value, sort it

if (listArray[index]) {

// Gets the subscript of the last value of the bucket

let last = listArray[index].length - 1;

// The last value of the bucket is greater than the inserted value, so you need // to sort the value by inserting this value in front of the bucket

while (last >= 0 && listArray[index][last] > array[i]) {

// The number in front of the bucket is put in the back

listArray[index][last + 1] = listArray[index][last]

last--;

}

// The unsorted ones are added directly to the back of the bucket

listArray[index][last + 1] = array[i];

} else {

listArray[index] = [];

listArray[index][0] = array[i]

}

}

// 5.Outputs all elements

let num = 0;

let sortedArray = [];

while (num < bucketNum) {

if (listArray[num]) {

sortedArray = sortedArray.concat(listArray[num])

}

num++;

}

return sortedArray;

}

function output(array) {

for (let i = 0; i < array.length; i++) {

console.log(array[i])

}

}

const array = [4.12, 6.421, 0.0023, 3.0, 2.123, 8.122, 4.12, 10.09];

const sortArray = bucketSort(array);

output(sortArray) // 0.0023 2.123 3.0 4.12 4.12 6.421 8.122 10.09

**Algorithms in Interviews**

**1. How to determine if a linked list has a loop**

method one:

Starting from the head node, traverse each node in the single linked list in turn. Each time a new node is traversed, check all the nodes before the new node from the beginning, and compare the new node with all the previous nodes in turn. If it is found that the node is the same as a previous node, it means that the node has been traversed and the linked list has a cycle, otherwise there is no cycle.

The time complexity is O(n 2 ), and the space complexity is O(1).

Method Two:

Create a HashSet with the node ID as the key to store the traversed nodes. Start from the head node and traverse each node in turn. Each time a new node is traversed, it is compared with the node stored in the HashSet. If there is an identical node, it proves that the linked list has a loop. Otherwise, store the node in the HashSet and continue with the next node and repeat the previous operation.

Time complexity is O(n), space complexity is O(n).

Method 3:

Create two pointers p1 and p2, and let them point to the head node of the linked list at the same time. Then start a big loop. In the loop body, p1 moves back one node each time, and p2 moves back two nodes each time, and then compares whether the two nodes are the same. If they are the same, it means that the linked list has a cycle, otherwise it enters the next loop.

Time complexity O(n), space complexity O(1). (optimal)

class Node {

constructor(data) {

this.data = data

this.next = null

}

}

function isCircle(head) {

let p1 = head;

let p2 = head;

while(p2 !== null && p2.next !== null) {

p1 = p1.next;

p2 = p2.next.next;

if (p1 === p1) {

return true

}

}

return false;

}

let node1 = new Node(5);

let node2 = new Node(3);

let node3 = new Node(7);

let node4 = new Node(2);

let node5 = new Node(6);

node1.next = node2;

node2.next = node3;

node3.next = node4;

node4.next = node5;

node5.next = node2;

console.log(isCircle(node1)) // true

Question extension:

* If the linked list has a loop, how do you find the length of the loop?

When the pointers meet for the first time, indicating that there is a loop in the linked list, let the two pointers continue to cycle forward from the meeting point and count the number of cycles forward until the two pointers meet again. The number of cycles forward at this time is the length of the loop.

Ring length = first speed difference x number of advances = number of advances

* If the linked list has a cycle, how to find the entry and exit points?

The distance from the head node of the linked list to the loop entry point is equal to the distance from the first encounter point to the loop entry point after n - 1 circles.

D = (n - 1) x (S1 + S2) + S2

Note: S1 represents the distance from the entry point to the first meeting point of the two pointers; S2 represents the distance from the first meeting point back to the entry point.

**2. Implementation of the Minimum Stack**

Question: Implement a stack with three methods: pop, push, and getMin.

Requirements: To ensure that the time complexity of the three methods is O(1)

The solution steps are as follows:

1. Create a main stack A and a backup stack B to assist A.

2. When the first element enters A, let the new element also enter B. This unique element is the current minimum value of stack A.

3. After that, every time a new element enters stack A, compare the element with the current minimum value of stack A. If it is less than the current minimum value of stack A, let the new element enter stack B. At this time, the top element of stack B is the current minimum value of stack A.

4. Whenever an element is popped from stack A, if the popped element is the current minimum value of stack A, the top element of stack B is also popped.

5. When the getMin method is called, just return the top element of stack B.

For this solution, the time complexity of pushing, popping, and taking the minimum value is O(1), and the worst space complexity is O(n).

// ... Stack Classes have been implemented before, skip them

class MinStack {

constructor() {

this.mainStack = new Stack();

this.minStack = new Stack();

}

// into the stack

push(element) {

this.mainStack.push(element);

// If the worker stack is empty, or if the new element is less than or equal to // the top of the worker stack, the new element is pressed into the worker stack

if (this.minStack.empty() || element < this.minStack.peek()) {

this.minStack.push(element);

}

}

// Out of the stack

pop() {

// If the out-stack element and the top-level element of the auxiliary stack are // equal, the auxiliary stack is out of the stack

if (this.minStack.peek() === this.mainStack.peek()) {

this.minStack.pop();

}

return this.mainStack.pop();

}

// Gets the minimum element of the stack

getMin() {

if (this.mainStack.empty()) {

throw new Error('stack is empty!')

}

return this.minStack.peek();

}

}

let stack = new MinStack();

stack.push(4)

stack.push(9)

stack.push(7)

stack.push(3)

stack.push(8)

stack.push(5)

console.log(stack.getMin()) // 3

stack.pop()

stack.pop()

stack.pop()

console.log(stack.getMin()) // 4

**3. How to find the greatest common divisor**

* Euclidean algorithm

The Euclidean algorithm, also known as the Euclidean algorithm, is an algorithm that aims to find the greatest common divisor of two positive integers.

The algorithm is based on a theorem: **the greatest common divisor of two positive integers a and b (a>b) is equal to the greatest common divisor between the remainder c of a divided by b and b** .

Therefore, we can use a recursive method to solve this problem, gradually simplifying the operations between two larger integers into operations between two smaller integers until the two numbers are divisible or one of the numbers is reduced to 1.

function getGreatestCommonDivisor(a, b) {

let big = a > b ? a : b;

let small = a < b ? a : b;

if (big % small === 0) {

return small;

}

return getGreatestCommonDivisor(big%small, small);

}

console.log(getGreatestCommonDivisor(25, 5)) // 5

console.log(getGreatestCommonDivisor(100, 80)) // 20

console.log(getGreatestCommonDivisor(27, 14)) // 1

The time complexity is O(log(max(a,b)))

* The method of reducing loss

Principle: **The greatest common divisor of two positive integers a and b (a>b) is equal to the greatest common divisor of the difference c of ab and the smaller number b** .

Similarly, we can also use recursion to solve this problem, gradually simplifying the operations between two larger integers into operations between two smaller integers until the two numbers can be equal. The greatest common divisor is the two numbers that are finally equal.

function getGreatestCommonDivisor(a, b) {

if (a === b) {

return a;

}

let big = a > b ? a : b;

let small = a < b ? a : b;

return getGreatestCommonDivisor(big-small, small)

}

console.log(getGreatestCommonDivisor(25, 5)) // 5

console.log(getGreatestCommonDivisor(100, 80)) // 20

console.log(getGreatestCommonDivisor(27, 14)) // 1

The time complexity is O(max(a,b))

* Combine the Euclidean algorithm with the phase subtraction method, and use shift operations based on the phase subtraction method

Ideas:

gcd is the abbreviation of getGreatestCommonDivisor method.

1. When a and b are both even, gcd(a, b) = 2 x gcd(a/2, b/2) = 2 x gcd(a>>1, b>>1)

2. When a is even and b is odd, gcd(a, b) = gcd(a/2, b) = gcd(a>>1, b)

3. When a is odd and b is even, gcd(a, b) = gcd(a, b/2) = gcd(a, b>>1)

4. When a and b are both odd numbers, first use the subtraction operation once, gcd(a, b) = gcd(b, a - b), then ab must be an even number, and then continue with the shift operation.

function getGreatestCommonDivisor(a, b) {

if (a === b) {

return a;

}

if ((a & 1) ===0 && (b & 1) === 0) { // a、b All are even

return getGreatestCommonDivisor(a>>1, b>>1)<<1;

} else if ((a & 1)===0 && (b & 1) !== 0) { // A even B odd

return getGreatestCommonDivisor(a>>1, b)

} else if ((a & 1) !==0 && (b & 1) === 0) { // A odd b even

return getGreatestCommonDivisor(a, b>>1)

} else { // a, b均为odd

let big = a > b ? a : b;

let small = a < b ? a : b;

return getGreatestCommonDivisor(big-small, small)

}

}

console.log(getGreatestCommonDivisor(25, 5)) // 5

console.log(getGreatestCommonDivisor(100, 80)) // 20

console.log(getGreatestCommonDivisor(27, 14)) // 1

The time complexity is O(log(max(a,b)))

**4. How to determine whether a number is an integer power of 2**

Ideas:

Using the characteristics of binary, if an integer is an integer power of 2, then when it is converted to binary, only the highest bit is 1, and the other bits are 0. Then, once the integer power of 2 is reduced by 1, all its binary digits become 1. At this time, if you use the original value and the result of it reduced by 1 to perform a bitwise AND operation to see if the result is 0.

function isPowerOf2\_V2(num) {

return (num & (num - 1)) === 0;

}

console.log(isPowerOf2\_V2(32)) // true

console.log(isPowerOf2\_V2(19)) // false

The time complexity is O(1)

**5. Maximum adjacent difference after sorting an unordered array**

* Counting sorting

Ideas:

Among the sorting algorithms we know, the algorithms with linear time complexity include counting sort, bucket sort, and shell sort. Here we can use the idea of ​​counting sort and consider solving the problem through the subscript of the array.

1. First find the interval length k (k=max-min+1) of the maximum value max and the minimum value min in the original array, as well as the offset d=min.

2. Create a new array array of length k.

3. Traverse the original array, and each time you traverse an element, increase the value of the corresponding index of the new array by 1. After the traversal is completed, some of the elements in the array have become 1 or higher, while some of the elements remain 0.

4. Traverse the new array array and count the maximum number of consecutive 0 values ​​in array + 1, which is the maximum difference between adjacent elements.

* Bucket sorting

Ideas:

If there are few array elements and one of them is particularly large, the length of the created array will be too long, which will waste space. Therefore, we can use bucket sort to solve this problem.

1. Create n buckets based on the length n of the original array, each bucket represents a range. The first bucket starts from the minimum value min of the original array.

Interval span = (max - min) / (n - 1)

2. Traverse the original array, insert each element of the original array into the corresponding bucket, and record the maximum and minimum values ​​of each bucket.

3. Traverse all buckets and count the maximum value of each bucket and the difference between it and the minimum value of the non-empty bucket to the right of this bucket. The largest difference is the adjacent maximum difference after the original array is sorted.

class Bucket {

constructor() {

this.min = null;

this.max = null;

}

}

function getMaxSortedDistance(array) {

// 1. Get the maximum and minimum values of the sequence

let max = array[0];

let min = array[0];

for (let i = 1; i < array.length; i++) {

if (array[i] > max) {

max = array[i];

}

if (array[i] < min) {

min = array[i];

}

}

let d = max - min;

// If max and min are equal, all elements of the array are equal, and 0 is returned

if (d === 0) {

return 0;

}

// 2.Initialize the bucket

let bucketNum = array.length; // Number of buckets

let range = d / (bucketNum - 1); // interval

let buckets = new Array(bucketNum); // Bucket array

for (let i = 0; i < bucketNum; i++) {

buckets[i] = new Bucket();

}

// 3. Iterate through the original array to determine the maximum and minimum //values for each bucket

for (let i = 0; i < array.length; i++) {

// Determine the subscript min of the bucket to which the array elements belong

let index = Math.floor((array[i] - min) / range);

if (buckets[index].min === null || buckets[index].min > array[i]) {

buckets[index].min = array[i]

}

if (buckets[index].max === null || buckets[index].max < array[i]) {

buckets[index].max = array[i]

}

}

// 4.Traverse the bucket to find the maximum difference

let leftMax = buckets[0].max;

let maxDistance = 0;

for (let i = 1; i < buckets.length; i++) {

if (buckets[i].min === null) {

continue;

}

if (buckets[i].min - leftMax > maxDistance) {

maxDistance = buckets[i].min - leftMax

}

leftMax = buckets[i].max

}

return maxDistance;

}

const array = [2, 6, 3, 4, 5, 10, 9];

console.log(getMaxSortedDistance(array)); // 3

The time complexity is O(n)

**6. How to implement queues using stacks**

Ideas:

The characteristic of a stack is "first in, last out", and the characteristic of a queue is "first in, first out". This can be achieved with the help of two stacks, one of which is used as the entrance of the queue to insert new elements, and the other stack is used as the exit of the queue to remove old elements.

// ... Stack Classes have been implemented before, skip them

class StackQueue {

constructor() {

this.stackA = new Stack(); // Enqueue elements

this.stackB = new Stack(); // Out of the queue element

}

// Team

enqueue(element) {

this.stackA.push(element)

}

// Out of the team

dequeue() {

if (this.stackB.empty()) {

if (this.stackA.empty()) {

return null

}

this.transfer()

}

return this.stackB.pop();

}

// Stack element transfer

transfer() {

while (!this.stackA.empty()) {

this.stackB.push(this.stackA.pop());

}

}

}

const stackQueue = new StackQueue();

stackQueue.enqueue(1)

stackQueue.enqueue(2)

stackQueue.enqueue(3)

console.log(stackQueue.dequeue()) // 1

console.log(stackQueue.dequeue()) // 2

stackQueue.enqueue(4)

console.log(stackQueue.dequeue()) // 3

console.log(stackQueue.dequeue()) // 4

The time complexity of the enqueue operation is O(1), and the amortized time complexity of the dequeue operation is O(1)

**7. Find the next number in the full permutation**

In layman's terms, it is to find a new integer that is greater than and only greater than the original number among all the combinations of numbers contained in an integer. We know that the integer composed of a fixed number of numbers is the largest when arranged in reverse order, and the smallest when arranged in order. In order to be close to the original number, we need to keep the high digits unchanged as much as possible and change the order of the low digits within the minimum range.

1. Check the reversed area from back to front and find the previous digit in the reversed area, which is the boundary of the digital permutation.

2. Swap the previous digit in the reverse order area with the smallest digit in the reverse order area that is greater than it.

3. Convert the original reverse order area to a sequential state

function findNearestNumber(numbers) {

// 1.Look at the reverse order region from back to front to find the previous bit of the reverse order region, which is the boundary of the number substitution

let index = findTransferPoint(numbers);

// If the number substitution boundary is 0, it means that the whole array has been // reversed, and it is impossible to get a larger integer composed of the same number, // and null is returned

if (index === 0) {

return null;

}

// 2.Swap the position of the number in the inverse region that is just one digit // in the reverse order and the number in the inverse region that is just greater // than it

// Copy and merge parameters to avoid directly modifying the input parameters

let numbersCopy = numbers;

exchangeHead(numbersCopy, index);

// 3.Convert the original reversed region to order

reverse(numbersCopy, index)

return numbersCopy;

}

function findTransferPoint(numbers) {

for (let i = numbers.length - 1; i > 0; i--) {

if (numbers[i] > numbers[i - 1]) {

return i

}

}

return 0;

}

function exchangeHead(numbers, index) {

let head = numbers[index - 1];

for (let i = numbers.length - 1; i > 0; i++) {

if (head < numbers[i]) {

numbers[index - 1] = numbers[i];

numbers[i] = head;

break;

}

}

return numbers;

}

function reverse(numbers, index) {

for (let i = index, j = numbers.length - 1; i < j; i++, j--) {

let temp = numbers[i];

numbers[i] = numbers[j];

numbers[j] = temp;

}

return numbers;

}

let numbers = [1,2,3,4,5];

for (let i = 0; i < 10; i++) {

numbers = findNearestNumber(numbers);

console.log(numbers.join('')) // 12354 12435 12453

}

This algorithm is also called **dictionary algorithm** . The time complexity is O(n)

**8. The minimum value after deleting k numbers**

Ideas:

Given an integer, remove k digits from it, and require the new integer formed by the remaining digits to be as small as possible. Then we want to reduce the high-order digits. The specific method is to compare all the digits of the original integer from left to right. If a digit is found to be greater than the digit to its right, then after deleting the digit, the value of the digit will inevitably be reduced. Each step requires the minimum value after deleting a digit. After k times, it is equivalent to finding the minimum value after deleting k digits.

function removeKDigits(num, k) {

let numbers = String(num);

// The final length of the new integer = the length of the original array - k

let newLength = numbers.length - k;

// Create a stack that receives all the numbers

let stack = new Array(newLength);

let top = 0; // 栈顶

for (let i = 0; i < numbers.length; ++i) {

// Go through the current numbers

let c = numbers.charAt(i);

// When the top digit of the stack is greater than the current digit traversed to, the top digit of the stack is out of the stack

while (top > 0 && stack[top - 1] > c && k > 0) {

top--;

k--;

}

// Traversed to the current number into the stack

stack[top++] = c;

}

// Find the first non-0 position in the stack to construct a new integer string

let offset = 0;

while (offset < newLength && stack[offset] == '0') {

offset++;

}

return offset == newLength ? '0' : getString(stack, offset, newLength - offset)

}

function getString(array, offset, end) {

return array.reduce((acc, curr, index) => {

if (index >= offset && index < end) {

acc += curr

}

return acc

}, '')

}

console.log(removeKDigits(1593212, 3)) // 1212

console.log(removeKDigits(30200, 1)) // 200

console.log(removeKDigits(10, 2)) // 0

console.log(removeKDigits(541270936, 3)) // 120936

This idea of ​​finding **local optimal solutions** in turn and finally getting **the global optimal solution is called a greedy algorithm** . The time complexity is O(n)

**9. How to add large integers**

Ideas:

To calculate the sum of two large integers, the large operation can be broken down into several small operations.

1. Create two arrays with a length equal to the number of digits in the larger integer + 1. Store each integer in the array in reverse order.

2. Create a result array whose length is the number of digits in the larger integer + 1.

3. Traverse the two arrays, add the elements in pairs from left to right according to the corresponding subscripts, and fill the results into the corresponding subscript positions of the result array. When the number is full, add one and move the carry to the next position.

function bigNumber(bigNumA, bigNumB) {

let bigNumberA = String(bigNumA);

let bigNumberB = String(bigNumB);

// 1.Create two arrays with the length of the array being the number of bits of the //larger integer + 1. Store each integer in reverse order to the array.

let maxLength = bigNumberA.length > bigNumberB.length ? bigNumberA.length : bigNumberB.length;

let arrayA = new Array(maxLength+1).fill(0);

for (let i = 0; i < bigNumberA.length; i++) {

arrayA[i] = bigNumberA.charAt(bigNumberA.length - 1 - i) - '0';

}

let arrayB = new Array(maxLength+1).fill(0);

for (let i = 0; i < bigNumberB.length; i++) {

arrayB[i] = bigNumberB.charAt(bigNumberB.length - 1 - i) - '0';

}

// 2.Construct a result array with the number of bits of the integer with a larger //length+1

let result = new Array(maxLength+1).fill(0);

// 3.Traverse the array, adding bits

for (let i = 0; i < result.length; i++) {

let temp = result[i];

temp += arrayA[i];

temp += arrayB[i];

// Determine whether to carry or not

if (temp >= 10) {

temp = temp - 10;

result[i+1] = 1;

}

result[i] = temp;

}

// 4.Reverse the result array again and turn it into a string

// Whether or not the most significant bit of a large integer was found

let str = ''

let findFisrt = false;

for (let i = result.length - 1; i >= 0; i--) {

if (!findFisrt) {

if (result[i] == '0') {

continue;

}

findFisrt = true;

}

str += result[i]

}

return str

}

console.log(bigNumber(426709752318, 95481253129)) // 522191005447

The time complexity is O(n)

**10. How to solve the gold mine problem**

Ideas:

This is a typical "dynamic programming" problem, similar to the famous "knapsack problem". The so-called dynamic programming is to simplify a complex problem into smaller sub-problems, and then recursively work from the simple sub-problems **from the bottom up** step by step, and finally get the optimal solution to the complex problem.

The moving state equation is as follows:

* F(n, w) = F(n-1, w) (n>1, w<p[n-1])
* F(n, w) = max(F(n-1, w), F(n-1, w-p[n-1])+g[n-1]) (n>1, w>=p[n-1])

Among them, n represents the number of optional gold mines, w represents the number of available workers, p array stores the number of available workers, and g array stores the gold content of the gold mine.

Equation 1 shows that if the number of people does not meet the requirement when mining the nth mine, the optimal choice is to use w people to mine the first n-1 mines.

Equation 2 indicates that if we choose to dig the nth mine, and there are enough workers, we can dig this mine or not. If we don't dig it, the harvest is the same as that of w people digging the first n-1 mines. If we dig it, the gold mine we get is the gold content of using wp[n-1] people to dig the first n-1 gold mines plus the gold content of the nth gold mine. Obviously, based on digging and not digging, we should solve the one with the highest gold content in the two cases.

/\*\*

\* Get the best yield from a gold mine 2D array

\* @param {\*} w Number of workers

\* @param {\*} p The number of workers required for gold mining array

\* @param {\*} g Gold Mine Gold Array

\*/

function getBestColdMining(w, p, g) {

// Create a two-dimensional array

let resultTable = new Array(g.length+1);

for (let i = 0; i < g.length+1; i++) {

resultTable[i] = new Array(w+1).fill(0);

}

// 填充

for (let i = 1; i <= g.length; i++) {

for (let j = 1; j <= w; j++) {

if (j < p[i-1]) {

resultTable[i][j] = resultTable[i-1][j]

} else {

resultTable[i][j] = Math.max(resultTable[i-1][j], resultTable[i-1][j-p[i-1]]+g[i-1])

}

}

}

// Returns the value of the last cell

return resultTable[g.length][w];

}

let w = 10;

let p = [3, 4, 3, 5, 5];

let g = [200, 300, 350, 400, 500];

console.log(getBestColdMining(w, p, g)) // 900

The time complexity is O(nw)

**11. Finding the missing integer**

Ideas:

Since there are no duplicates in the 99 numbers in the array, we can first calculate the sum of 1+2+3+...+100, then subtract the elements in the array one by one, and the difference is the missing integer. This solution has a time complexity of O(n) and a space complexity of O(1).

But if the problem is slightly expanded and deformed, the above solution will not work, for example,

Extension 1: There are several positive integers in an unordered array, ranging from 1 to 100, 99 of which appear an even number of times, and only one integer appears an odd number of times. How to find the integer that appears an odd number of times?

Ideas:

XOR operation: When performing bitwise operations, identical bits get 0 and different bits get 1.

Using XOR operation, the numbers that appear even number of times cancel each other out, and only the numbers that appear odd number of times will be left. Time complexity O(n), space complexity O(1)

Extension 2: Suppose there are several positive integers in an unordered array, ranging from 1 to 100, of which 98 integers appear an even number of times and only two integers appear an odd number of times. How to find these two integers that appear an odd number of times?

Ideas:

Using the divide-and-conquer algorithm, we first perform XOR operations on the array elements one by one, and the result is the XOR operation result of two integers that appear an odd number of times. Among these results, at least one result must be binary 1. Based on this conclusion, the original array can be divided into two parts according to the difference in the nth last bit of the binary number, and each part can be used to find the integer that appears an odd number of times according to the previous XOR operation. Time complexity O(n), space complexity O(1)

function findLostNum(array) {

// Used to store 2 integers that occur odd times

let result = [];

// The first time the overall XOR operation is performed

let xorResult = 0;

for (let i = 0; i < array.length; i++) {

xorResult ^= array[i]

}

// If the XOR operation results in 0, the input array does not meet the //requirements

if (xorResult === 0) {

return null

}

// Determine the different bits of two integers to group them

let separator = 1;

while (0 === (xorResult&separator)) {

separator<<=1

}

// The second group is used to perform XOR operations

for (let i = 0; i < array.length; i++) {

if (0 === (array[i]&separator)) {

result[0] ^= array[i]

} else {

result[1] ^= array[i]

}

}

return result

}

const array = [4, 1, 2, 2, 5, 1, 4, 3];

console.log(findLostNum(array)) // [5, 3]

**Practical Application of Algorithms**

**Clever use of Bitmap**

A requirement for user tags. We need to develop a user portrait system to implement user information tagging. User tags include user's social attributes, living habits, consumption behavior and other information. Through user tags, we can make statistics on various user groups, such as the male-female ratio of users, the number of users who like to travel, etc.

**The Bitmap algorithm** is used here , also known as **the bitmap algorithm** . The bitmap here refers to a data structure composed of continuous binary bits in memory. This algorithm is mainly used to perform deduplication and query operations on a large number of integers.

class MyBitmap {

constructor(size) {

// Bitmap The size of the number of digits

this.size = size;

// Each of these words is a long-type element, corresponding to a 64-bit binary //data

this.words = new Array(this.getWordIndex(size - 1) + 1);

}

// Locate a bit of Bitmap word

getWordIndex(bitIndex) {

// Shifting 6 bits to the right is equivalent to dividing by 64

return bitIndex >> 6

}

// judgement Bitmap The state of a certain bit bitIndex Represents the section of //the bitmap bitIndex bit

getBit(bitIndex) {

if (bitIndex<0 || bitIndex > this.size - 1) {

throw new Error('went beyond Bitmap effective range！')

}

let wordIndex = this.getWordIndex(bitIndex);

return (this.words[wordIndex] & (1 << bitIndex)) !== 0;

}

// handful Bitmap A bit is set to true

setBit(bitIndex) {

if (bitIndex<0 || bitIndex > this.size - 1) {

throw new Error('went beyond Bitmap effective range！')

}

let wordIndex = this.getWordIndex(bitIndex);

this.words[wordIndex] |= (1 << bitIndex)

}

}

let bitmap = new MyBitmap(128);

bitmap.setBit(126);

bitmap.setBit(75);

console.log(bitmap.getBit(126)); // true

console.log(bitmap.getBit(78)); // false

If you want to set a bit in a Bitmap to 1, you need to go through two steps:

1. Locate the corresponding element in words

2. Modify the value of an element by using the AND (&) operator

If you want to check whether a bit in the Bitmap is 1, you also need to go through two steps:

1. Locate the corresponding element in words

2. Determine whether the binary bit corresponding to the element is 1

**Application of LRU Algorithm**

A demand for user information. As the company's business becomes more and more complex, it is necessary to extract a user system to provide basic user information to various business systems. Since the business side frequently queries user information, special attention should be paid to performance issues.

Use the LRU algorithm to solve this problem. LRU stands for Least Recently Used, which means the least recently used. It is a memory management algorithm. The algorithm is based on an assumption: data that has not been used for a long time is unlikely to be used in the future. Therefore, when the memory occupied by the data reaches a certain threshold, we need to remove the data that has been used least recently. A data structure called a **hash linked list** is used in the LRU algorithm . This structure makes the originally unordered key-value have its predecessor and successor key-value, just like a bidirectional linked list. According to this feature, the key-value can be sorted according to the time of last use, so the leftmost end of the linked list is the least recently used.

class Node {

constructor(key, value) {

this.key = key;

this.value = value;

this.next = null;

this.prev = null;

}

}

class LRUCache {

constructor(limit) {

this.head = new Node();

this.end = new Node();

// Cache storage limit

this.limit = limit;

this.hashMap = new Map();

}

get(key) {

let node = this.hashMap.get(key);

if (node == null) {

return null;

}

this.refreshNode(node); // Get a value, why update the access location？

return node.value;

}

put(key, value) {

let node = this.hashMap.get(key);

if (node == null) {

// If the key doesn't exist, insert it key-value

if (this.hashMap.size >= this.limit) {

let oldKey = this.removeNode(this.head);

this.hashMap.delete(oldKey);

}

node = new Node(key, value);

this.addNode(node);

this.hashMap.set(key, node);

} else {

// If the key exists, it is updated key-value

node.value = value

this.refreshNode(node);

}

}

remove(key) {

let node = this.hashMap.get(key);

this.removeNode(node);

this.hashMap.delete(key);

}

// Refreshes the node location that was accessed

refreshNode(node) {

// If you are accessing a tail node, you do not need to move the node

if (node == this.end) {

return;

}

// Remove the node

this.removeNode(node);

// Reinsert the node

this.addNode(node);

}

// Delete the node

removeNode(node) {

if (node == this.head && node == this.end) {

// Remove the unique node

this.head = null;

this.end = null;

} else if (node == this.end) {

// Remove the tail node

this.end = this.end.prev;

this.end.next = null;

} else if (node == this.head) {

// Remove the head node

this.head = this.head.next;

this.head.prev = null;

} else {

// Remove the intermediate node

node.prev.next = node.next;

node.next.prev = node.prev;

}

return node.key;

}

// Tail insert node

addNode(node) {

if (this.end.next == null) {

this.end.next = node;

node.prev = this.end;

node.next = null;

}

this.end = node;

if (this.head.prev == null) {

this.head = node;

}

}

}

let lruCache = new LRUCache(5);

lruCache.put('001', 'User 1 Information');

lruCache.put('002', ''User 2 Information'');

lruCache.put('003', ''User 3 Information'');

lruCache.put('004', ''User 4 Information'');

lruCache.put('005', ''User 5 Information'');

// lruCache.get('002');

lruCache.put('004', 'User 4 information updated');

lruCache.put('006', 'User 6 Information');

console.log(lruCache.get('001')); // null

console.log(lruCache.get('006')); // User 6 Information

**What is the A-star pathfinding algorithm?**

A need for maze pathfinding. In this "maze pathfinding" game, some monsters will attack the protagonist. Now we need to add smart AI to the monsters so that they can automatically bypass obstacles in the maze and find the protagonist.

Here we use the A\* Search algorithm to solve the problem. **The A\* Search algorithm** is an algorithm for finding valid paths. Introduce two sets and a formula.

Two collections:

* OpenList: accessible grid
* CLoseList: The grids that have been reached

A formula:

* F = G + H

Each grid has three attributes: F, G, and H. Among them, G represents the cost of walking from the starting point to the current grid, that is, how many steps have been taken. H represents the distance from the current grid to the target grid without considering obstacles, that is, how far away from the target is. F represents the comprehensive evaluation of G and H, that is, the total number of steps from the starting point to the current grid, and then from the current grid to the target grid. This method of determining the search priority based on the valuation is called **heuristic search** .

// Labyrinth map

const MAZE = [

[0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 1, 0, 0, 0],

[0, 0, 0, 1, 0, 0, 0],

[0, 0, 0, 1, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0]

];

class Grid {

constructor(x, y) {

this.x = x;

this.y = y;

this.f = 0;

this.g = 0;

this.h = 0;

this.parent = null;

}

initGrid(parent, end) {

this.parent = parent;

if (parent != null) {

this.g = parent.g + 1;

} else {

this.g = 1;

}

this.h = Math.abs(this.x - end.x) + Math.abs(this.y - end.y);

this.f = this.g + this.h;

}

}

// A\*Pathfinding master logic

function aStartSearch(start, end) {

let openList = [];

let closeList = [];

// Add the starting point to openList

openList.push(start);

// The main loop checks 1 current grid node per round

while (openList.length > 0) {

// Find the node with the smallest F-value in openList and use it as the current grid node

let currentGrid = findMinGrid(openList);

// Removes the current grid node from the openList

openList.remove(currentGrid);

// The current grid node enters closeList

closeList.push(currentGrid);

// Locate all nearby nodes

let neighbors = findNeighbors(currentGrid, openList, closeList);

for (let grid of neighbors) {

if (!openList.includes(grid)) {

// The neighboring node is not there in openList, mark the parent node、G、H、F，and put it in openList

grid.initGrid(currentGrid, end);

openList.push(grid)

}

}

// If the end point is in openList, the end grid is returned

for (let grid of openList) {

if ((grid.x == end.x) && (grid.y == end.y)) {

return grid

}

}

}

// If the endpoint cannot be found after the openList is exhausted, the endpoint cannot be reached and is empty

return null;

}

// Find the grid with the smallest F-number

function findMinGrid(openList) {

let tempGrid = openList[0];

for (let grid of openList) {

if (grid.f < tempGrid.f) {

tempGrid = grid

}

}

return tempGrid

}

function findNeighbors(grid, openList, closeList) {

let gridList = [];

// The next grid above the current grid

if (isValidGrid(grid.x, grid.y-1, openList, closeList)) {

gridList.push(new Grid(grid.x, grid.y-1))

}

// The next cell below the current grid

if (isValidGrid(grid.x, grid.y+1, openList, closeList)) {

gridList.push(new Grid(grid.x, grid.y+1))

}

// A square to the left of the current grid

if (isValidGrid(grid.x-1, grid.y, openList, closeList)) {

gridList.push(new Grid(grid.x-1, grid.y))

}

// A square to the right of the current grid

if (isValidGrid(grid.x+1, grid.y, openList, closeList)) {

gridList.push(new Grid(grid.x+1, grid.y))

}

return gridList;

}

function isValidGrid(x, y, openList, closeList) {

// Whether it is out of bounds

if (x < 0 || x >= MAZE.length || y < 0 || y >= MAZE[0].length) {

return false;

}

// Whether there are obstacles

if (MAZE[x][y] == 1) {

return false;

}

// Whether it is already in openList

if (containGrid(openList, x, y)) {

return false;

}

// Whether it is already in the closeList

if (containGrid(closeList, x, y)) {

return false;

}

return true;

}

// Whether or not nodes are included

function containGrid(grids, x, y) {

for (let grid of grids) {

if ((grid.x == x) && (grid.y == y)) {

return true

}

}

return false;

}

// Set the start and end points

let startGrid = new Grid(2, 1);

let endGrid = new Grid(2, 5);

// Search for maze endpoints

let resultGrid = aStartSearch(startGrid, endGrid);

// Retrace the path of the maze

let path = [];

while(resultGrid != null) {

path.push(new Grid(resultGrid.x, resultGrid.y));

resultGrid = resultGrid.parent;

}

// Outputs mazes and paths, which are denoted by \*

let pathMap = '';

for (let i = 0; i < MAZE.length; i++) {

for (let j = 0; j < MAZE[0].length; j++) {

if (containGrid(path, i, j)) {

pathMap += '\*, ';

} else {

pathMap += `${MAZE[i][j]}, `;

}

}

pathMap += '\n';

}

console.log(pathMap)

/\*\* Final Path

0, 0, \*, \*, \*, \*, 0

0, 0, \*, 1, 0, \*, 0

0, \*, \*, 1, 0, \*, 0

0, 0, 0, 1, 0, 0, 0

0, 0, 0, 0, 0, 0, 0

\*/

**How to implement the red envelope algorithm**

A demand for money. A red envelope function needs to be launched, similar to WeChat's red envelope function. For example, one person sends a 100 yuan red envelope in a group, and 10 people in the group grab it, and the amount each person grabs is randomly distributed.

Specific rules that the red envelope function needs to meet:

1. The sum of the amounts grabbed by all people must be equal to the amount of the red envelope, no more and no less

2. Everyone gets at least 1 cent

3. Ensure that the amount of red envelopes is distributed as evenly as possible, and avoid severe polarization

Taking into account the problem of high concurrency, the amount of the red envelope received by each person cannot be calculated when receiving it. The amount of each red envelope must be calculated first and placed in a queue. Users who receive red envelopes must find their own share in the queue.

solution:

* Double mean method

Assuming the remaining red envelope amount is m yuan and the remaining number of people is n, then the following formula is used:

**The amount of money snatched each time = random interval [0.01, m/nx 2 - 0.01] yuan**

This formula ensures that the average value of each random amount is equal, and there will be no unfairness caused by the order of grabbing.

/\*\*

\* Split the red packet

\* @param {\*} totalAmount The total amount of the red envelope（In fractions）

\* @param {\*} totalPeopleNum Total number of people

\*/

function divideRedPackage(totalAmount, totalPeopleNum) {

let amountList = []; // Store the amount of the split red envelope

let restAmount = totalAmount;

let restPeopleNum = totalPeopleNum;

for (let i = 0; i < totalPeopleNum - 1; i++) {

// Random range: [1, twice the amount remaining per capita - 1] points

let amount = Math.random() \* (restAmount/restPeopleNum \* 2 - 1) + 1;

restAmount -= amount;

restPeopleNum--;

amountList.push(amount);

}

amountList.push(restAmount);

return amountList;

}

let amountList = divideRedPackage(1000, 10);

for (let amount of amountList) {

console.log('Grab the amount of red envelopes：'+ amount.toFixed(2))

}

* Line segment cutting method

It is determined by the "cutting point". When n people compete for a red envelope with a total amount of m, n-1 cutting points need to be determined. The random range is [1, m-1].

But there are two points to note:

1. How to deal with duplicate random cutting points

2. How to reduce time complexity and space complexity as much as possible