**COMP90038 Algorithms and Complexity**

**期 末 考 试 机 经**

注：

1. 标了星星的地方是课外或者tute的骚操作，其余没标星星的地方是基本操作
2. 课件上有图的地方一定一定一定要亲手画几遍，伪代码来得及就背来不及至少排序的要会写

**PART I: Big-O Algorithm Complexity Cheat Sheet**

**赌5毛考试必考**

1. **Common Data Structure Operations**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Search** | **Insertion** | **Deletion** |
| **Array** | O(*n*) | O(*n*) | O(*n*) |
| **Linked List** | O(*n*) | O(1) | O(1) |
| **Stack** | O(*n*) | O(1) | O(1) |
| **Queue** | O(*n*) | O(1) | O(1) |
| **Binary Search Tree** | Average: Θ(log*n*)  Worst: O(*n*) | Average: Θ(log*n*)  Worst: O(*n*) | Average: Θ(log*n*)  Worst: O(*n*) |
| **AVL Tree** | O(log*n*) | O(log*n*) | O(log*n*) |
| **Hash Table** | Average: Θ(1)  Worst: O(*n*) | Average: Θ(1)  Worst: O(*n*) | Average: Θ(1)  Worst: O(*n*) |

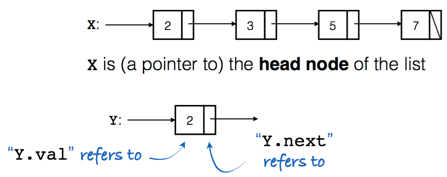
1. **Sorting and Searching Algorithms** (distribution counting sort那个我乱写的)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Best** | **Average** | **Worst** | **In-place** | **Stable** | **Input-insensitive** |
| **Selection Sort** | Ω(*n*2) | Θ(*n*2) | O(*n*2) | √ | × | √ |
| **Insertion Sort** | Ω(*n*) | Θ(*n*2) | O(*n*2) | √ | √ | × |
| **Shellsort** | \*Ω(*n*log*n*) | \*Θ(*n*log2*n*) | \*O(*n*log2*n*) | √ | × | × |
| **Mergesort** | Ω(*n*log*n*) | Θ(*n*log*n*) | O(*n*log*n*) | × | √ | √ |
| **Quicksort** | Ω(*n*log*n*) | Θ(*n*log*n*) | O(*n*2) | √ | × | × |
| **Heapsort** | Ω(*n*log*n*) | Θ(*n*log*n*) | O(*n*log*n*) | √ | × | √ |
| **Distribution Counting Sort** | \*Ω(*n + u – l*) | \*Θ(*n + u – l*) | \*O(*n + u – l*) | \*× | \*× | \*√ |
| **Linear Search** | Ω(1) | Θ(*n*) | O(*n*) | – | – | – |
| **Binary Search** | Ω(1) | Θ(log*n*) | O(log*n*) | – | – | – |
| **Interpolation Search** | Ω(1) | Θ(loglog*n*) | O(*n*) | – | – | – |
| **Quick Select** | Ω(*n*) | Θ(*n*) | O(*n*2) | – | – | – |
| **BST Search** | Ω(1) | Θ(log*n*) | O(*n*) | – | – | – |
| **AVL Tree Search** | Ω(1) | Θ(log*n*) | O(log*n*) | – | – | – |
| **Hash Table Search** | Ω(1) | Θ(1) | O(*n*) | – | – | – |
| **Depth-First Search** | matrix: Θ(|*V*|2) list: Θ(|*V*|+ |*E*|) | | | – | – | – |
| **Breadth-First Search** | matrix: Θ(|*V*|2) list: Θ(|*V*|+ |*E*|) | | | – | – | – |
| **Warshall’s Algorithm** | Θ(*n*3) | | | – | – | – |
| **Floyd’s Algorithm** | Θ(*n* 3) | | | – | – | – |
| **Prim’s Algorithm** | matrix: Θ(|*V*|2) list: Θ(|*E*|log|*V*|) | | | – | – | – |
| **Dijkstra’s Algorithm** | matrix: Θ(|*V*|2) list: ??????????? | | | – | – | – |
| **Brute Force String Search** | Ω(*m*) | – | O(*mn*) | – | – | – |
| **Horspool’s String Search** | Ω(*m*) | – | O(*mn*) | – | – | – |

这里有一个记忆的小技巧：除了distribution counting sort这个骚东西，所有排序算法里只有insertion sort和mergesort是stable的，只有mergesort是not in-place的，input-insensitive会背best和worst自然就会了，是不是很腻害(〃’▽’〃)

**PART II: Data Structures**

1. **Linked List**

****

1. **Stack**

Last-In-First-Out (LIFO)

* CreateStack
* Push
* Pop
* Top
* EmptyStack?

1. **Queue**

First-In-First-Out (FIFO)

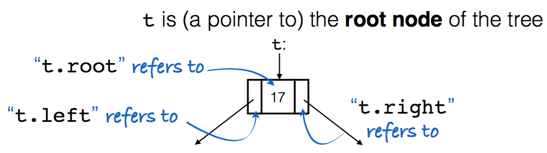
* CreateQueue
* Enqueue
* Dequeue
* Head
* EmptyQueue?

1. **Binary Tree**

* Definition:

A binary tree is a [tree](https://en.wikipedia.org/wiki/Tree_structure) [data structure](https://en.wikipedia.org/wiki/Data_structure) in which each node has at most two [children](https://en.wikipedia.org/wiki/Child_node), which are referred to as the left child and the right child.

* Linked-list representation:



* Level:

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).

* Height:

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.

* Full binary tree:

Each node has 0 or 2 (non-empty) children.

* Complete binary tree:

Each level filled left to right.

* Calculating the height:

**function** HEIGHT(*T*)

**if** *T* = *null* **then**

**return** -1

**else**

**return** *max*(HEIGHT(*T.left*), HEIGHT(*T.right*)) + 1

Comparisons: 2*n* + 1

1. **Binary Search Tree (BST)**

* Definition:

A binary search tree, or BST, is a binary tree that stores elements in all internal nodes, with each sub-tree satisfying the BST property: each element in the left sub-tree is smaller than the root and each element in the right sub-tree is larger than the root.

* Search:
* *Step 1*: To search *k* in a BST, compare against its root *r*;
* *Step 2*: If *r* = *k*, we are done;
* *Step 3*: Otherwise, search in the left or right sub-tree, according as *k* < *r* or *k* > *r*.
* Insertion:
* *Step 1*: To insert a new element *k* into a BST, we pretend to search for *k*;
* *Step 2*: When the search has taken us to the fringe of the BST (we find an empty sub-tree), we insert *k* where we would expect to find it.
* Binary tree traversal:
* Preorder traversal:

**function** PREORDERTRAVERSE(*T*)

**if** *T* ≠ *null* **then**

visit *T.root*

PREORDERTRAVERSE(*T.left*)

PREORDERTRAVERSE(*T.right*)

**function** PREORDERTRAVERSE(*T*)

*init*(*stack*)

*push*(*stack*, *T*)

**while** the stack is non-empty **do**

*T* ← *pop*(*stack*)

visit *T*.*root*

**if** *T.right* is non-empty **then**

*push*(*stack*, *T.right*)

**if** *T.left* is non-empty **then**

*push*(*stack*, *T.left*)

* Inorder traversal:

**function** INORDERTRAVERSE(*T*)

**if** *T* ≠ *null* **then**

INORDERTRAVERSE(*T.left*)

visit *T.root*

INORDERTRAVERSE(*T.right*)

* Postorder traversal:

**function** POSTORDERTRAVERSE(*T*)

**if** *T* ≠ *null* **then**

POSTORDERTRAVERSE(*T.left*)

POSTORDERTRAVERSE(*T.right*)

visit *T.root*

* Level-order traversal:

**function** LEVELORDERTRAVERSE(*T*)

*init*(*queue*)

*inject*(*queue*, *T*)

**while** the queue is non-empty **do**

*T* ← *eject*(*queue*)

visit *T*.*root*

**if** *T.left* is non-empty **then**

*inject*(*queue*, *T.left*)

**if** *T.right* is non-empty **then**

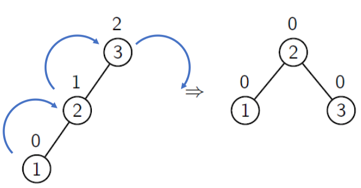
*inject*(*queue*, *T.right*)

1. **AVL Tree**(我猜这个和23树里面会选考一个)

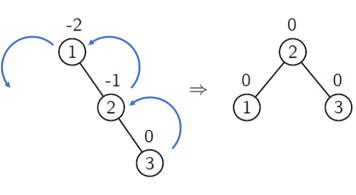
* Definition:

For a binary (sub-) tree, let the balance factor be the difference between the height of its left sub-tree and that of its right sub-tree. An AVL tree is a BST in which the balance factor is -1, 0, or 1, for every sub-tree.

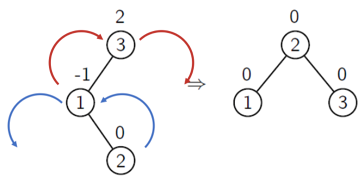
* Rotations:
* R-rotation:



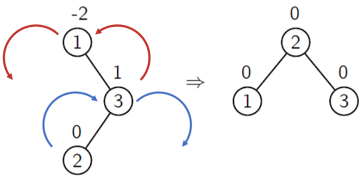
* L-rotation:



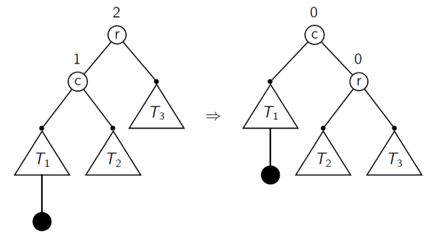
* LR-rotation:



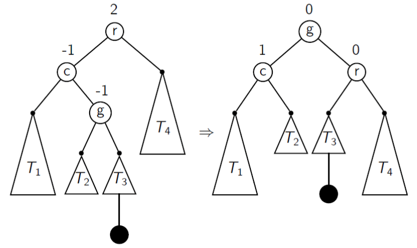
* RL-rotation:



* General rotation: (他AVL出的难绝壁会考这个，你们去搜一个油土鳖视频AVL Tree 7 complete example of adding data to an AVL tree, 这个视频里面有所有可能的骚气考法)
* R-rotation & L-rotation:



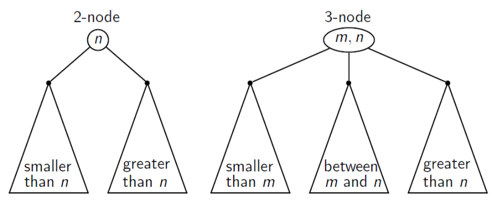
* LR-rotation & RL-rotation:



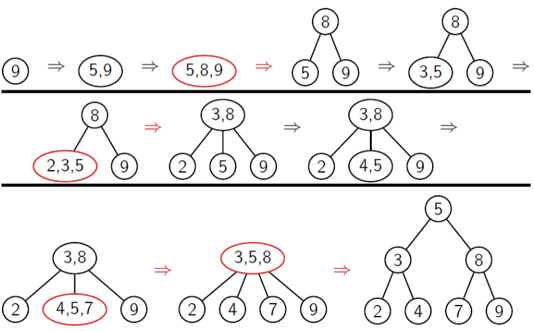
1. **2-3 Tree**

* Definition:

In 2-3 tree, a node that holds a single item has two children (or none, if it is a leaf). A node that holds two items (a so-called 3-node) has three children (or none, if it is a leaf). The height is log3(*n* + 1) – 1 ≤ *h* ≤ log2(*n* + 1) – 1



* Insertion:
* *Step 1*: To insert a key *k*, pretend that we are searching for *k*;
* *Step 2*: This will take us to a leaf node in the tree, where *k* should now be inserted; if the node we find there is a 2-node, *k* can be inserted without further ado;
* *Step 3*: Otherwise we had a 3-node, and the two inhabitants, together with *k*, momentarily from a node with three elements; in sorted order, call them *k*1, *k*2, and *k*3.
* *Step 4*: We now split the node, so that *k*1 and *k*3 form their own individual 2-nodes. The middle key, *k*2 is promoted to the parent one.
* *Step 5*: The promotion may cause the parent node to overflow, in which case it gets split the same way. The only time the tree’s height changes is when the root overflows.



1. **Hashing**

* Definition:

Hashing is a standard way of implementing the abstract data type “dictionary”, a collection of <key, value> pairs. A key identifies each record. It can be anything: integers, alphabetical character, even strings. We will store our records in a hash table of size *m*. The idea is to have a hash function *h* that takes the key *k*, and determines an index in the hash table. The address *h*(*k*) is the hash address.

* The hash table operations:
* Find (这个功能是啥子我也不知道QAQ)
* Insert
* Lookup (search and insert if not found)
* Initialize
* Delete
* Rehash
* The hash functions:
* Integers: *h*(*k*) = *n* mod *m*
* Strings: *h*(*s*) = mod *m*

字符操作不要看屁屁踢上那堆扯淡的p话。用人话讲，比如MYKEY，先把每个字母的序号（从0到25）写出来就是M: 12, Y: 24, K: 10, E: 4, Y: 24, 然后像算二进制那样计算总和12 × 324 + 24 × 323 + 10 × 322 + 4 × 321 + 24 × 320 = 13379736，最后再mod m，so easy！！！

* Handling collisions: (考试绝壁会考！！)
* Separate chaining:

Each element k of the hash table is a linked list, which makes collision handling very easy.

The load factor *α* = *n*/*m*, where *n* is the number of items stored.

Number of probes in successful search ≈ (1 + *α*)/2

Number of probes in unsuccessful search ≈ *α*.

Pros and cons:

Compared with sequential search, reduces the number of comparisons by the size of the table (a factor of m).

Good in dynamic environment, when (number of) keys are hard to predict.

The chains can be ordered, or records may be “pulled up front” when accessed.

Deletion is easy.

However, separate chaining uses extra storage for links.

（我要是老师一定会出这个概念题难死那帮不背概念的学生）

* Linear probing:

In case of collision, try the next cell, then the next, and so on. If we get to the end of the table, we wrap around(回到0).

The load factor α = *n*/*m*, where *n* is the number of items stored.

Number of probes in successful search ≈

Number of probes in unsuccessful search ≈

Pros and cons:

Space-efficient.

Worst-case performance miserable; must be careful not to let the load factor grow beyond 0.9.

Faster than binary search and linear search.

Clustering is a major problem: The collision handling strategy leads to clusters of contiguous cells being occupied.

Deletion is almost impossible.

* Double hashing:

Use a second hash function *s* to determine an offset to be used in probing for a free cell. By this way we mean, if *h*(*k*) is occupied, next try (*h*(*k*) + *s*(*k*)) mod *m*, then (*h*(*k*) + 2*s*(*k*)) mod *m*, and so on.

* Rehashing:
* The standard approach to avoiding performance deterioration in hashing is to keep track of the load factor and to rehash when it reaches, say, 0.9.
* Rehashing means allocating a larger hash table (typically twice the current size), revisiting each item, calculating its hash address in the new table, and inserting it.

1. **Heap (Priority Queue)**

* Priority queue:

A priority queue is a set (or pool) of elements. An element is injected into the priority queue together with a priority (often the key value itself) and elements are ejected according to priority.

* Heap:

A heap is a complete binary tree which satisfies the heap condition: each child has a priority (key) which is no greater/smaller than its parent’s.

* Building a heap bottom-up: (这个过程课上有图)
* *Step 1*: Start withB B B the last parent and move backwards, in level-order. For each parent node, if the largest child is larger than the parent, swap it with the parent.
* *Step 2*: Whenever a parent is found to be out of order, let it “sift down” until both children are smaller.

**function** HEAPIFY(*H*[·], *n*)

**for** *i* ← ⎣*n*/2⎦ downto 1 **do**

*k* ← *i*

*v* ← *H*[*k*]

*heap* ← *false*

**while** not *heap* and 2 × *k* ≤ *n* **do**

*j* ← 2 × *k*

**if** *j* < *n* **then**

**if** *H*[*j*] < *H*[*j* + 1] **then**

*j* ← *j* + 1

**if** *v* ≥ *H*[*j*] **then**

*heap* ← *true*

**else**

*H*[*k*] ← *H*[*j*]

*k* ← *j*

*H*[*k*] ← *v*

time complexity: O(*n*)

* Ejecting the maximal element from a heap
* *Step 1*: Swap the root with the last item *z* in the heap, and then let *z* “sift down” to its proper place.
* *Step 2*: After this, the last element is no longer considered part of the heap, that is, *n* is decremented.

1. **Graph: direct, undirected, weighted**

概念太多懒得写了，自己看屁屁踢去

**PART III: Sorting Algorithms**

1. **Selection Sort**

**function** SELSORT(*A*[·], *n*)

**for** *i* ← 0 to *n* – 2 **do**

*min* ← *i*

**for** *j* ← *i* + 1 to *n* – 1 **do**

**if** *A*[*j*] < *A*[*min*] **then**

*min* ← *j*

*t* ← *A*[*i*]

*A*[*i*] ← *A*[*min*]

*A*[*min*]← *t*

1. **Insertion Sort**

**function** INSERTIONSORT(*A*[·], *n*)

**for** *i* ← 1 to *n* – 1 **do**

*v* ← *A*[*i*]

*j* ← *i* – 1

**while** *j* ≥ 0 and *v* < *A*[*j*] **do**

*A*[*j* + 1] ← *A*[*j*]

*j* ← *j* – 1

*A*[*j* + 1] ← *v*

1. **Shellsort**

* *Step 1*: Think of the array as an interleaving of *k* lists
* *Step 2*: Sort each list separately using insertion sort
* *Step 3*: Then sort the resulting entire array using a final pass of insertion sort

1. **Mergesort**

**function** MERGESORT(*A*[·], *n*)

**if** *n* > 1 **then**

**for** *i* ← 0 to ⎣*n*/2⎦ – 1 **do**

*B*[*i*] ← *A*[*i*]

**for** *i* ← 0 to ⎡*n*/2⎤ – 1 **do**

*C*[*i*] ← *A*[⎣*n*/2⎦ + *i*]

MERGESORT(*B*, ⎣*n*/2⎦)

MERGESORT(*C*, ⎡*n*/2⎤)

MERGE(*B*, ⎣*n*/2⎦, *C*, ⎡*n*/2⎤, *A*)

**function** MERGE(*B*[·], *p, C*[·], *q*, *A*[·])

*i* ← 0; *j* ← 0; *k* ← 0

**while** *i* < *p* and *j* < *q* **do**

**if** *B*[*i*] ≤ *C*[*j*] **then**

*A*[*k*] ← *B*[*i*]

*i* ← *i* + 1

**else**

*A*[*k*] ← *C*[*i*]

*j* ← *j* + 1

*k* ← *k* + 1

**if** *i* = *p* **then**

copy *C*[*j*]..*C*[*q* – 1] to *A*[*k*]..*A*[*p* + *q* – 1]

**else**

copy *B*[*i*]..*B*[*p* – 1] to *A*[*k*]..*A*[*p* + *q* – 1]

1. **Quicksort**

**function** QUICKSORT(*A*[·], *lo*, *hi*)

**if** *lo* < *hi* **then**

*s*← HOAREPARTITION(*A*, *lo*, *hi*)

QUICKSORT(*A*, *lo*, *s –* 1)

QUICKSORT(*A*, *s +* 1, *hi*)

**function** HOAREPARTITION(*A*[·], *lo*, *hi*)

*p* ← *A*[*lo*]; *i* ← *lo*; *j* ← *hi*

**repeat**

**while** *i* < *hi* and *A*[*i*] ≤ *p* **do**

*i* ← *i* + 1

**while** *j* ≥ *lo* and *A*[*j*] > *p* **do**

*j* ← *j* – 1

*swap*(*A*[*i*], *A*[*j*])

**until** *i* ≥ *j*

*swap*(*A*[*i*], *A*[*j*])

*swap*(*A*[*lo*], *A*[*j*])

**return** *j*

1. **Heapsort**

* *Step 1*: Turn *H* into a heap.
* *Step 2*: Apply the eject operation *n* – 1 times.

1. **Distribution Counting Sort**

**function** DISTRIBUTIONCOUNTINGSORT(*A*[·], *l, u*)

**for** *j* ← 0 to *u* – *l* **do**

*D*[*j*] ← 0

**for** *i* ← 0 to *n* – 1 **do**

*D*[*A*[*i*] – *l*] ← *D*[*A*[*i*] – *l*] + 1

**for** *j* ← 1 to *u* – *l* **do**

*D*[*j*] ← *D*[*j – 1*] + *D*[*j*]

**for** *i* ← *n* – 1 downto 0 **do**

*j* ← *A*[*i*] – *l*

*S*[*D*[*j*] – 1] ← *A*[*i*]

*D*[*j*] ← *D*[*j*] – 1

**return** *S*

这些排序算法里面，好像只有quicksort最不喜欢已经排好的数列

**PART IV: Search Algorithms**

1. **Linear Search**

**function** LINSEARCH(*A*[·], *x*, *n*)

*j* ← 0

**while** *j* < *n* **do**

**if** *A*[*j*] = *x* **then**

**return** *j*

*j* ← *j* + 1

**return** -1

**function** LINSEARCH(*head*, *x*)

*p* ← *head*

**while** *p* ≠ null **do**

**if** *p.*val = *x* **then**

**return** *p*

*p* ← *p.*next

**return** -1

1. **Binary Search**

**function** BINSEARCH(*A*[·], *lo*, *hi*, *key*)

**while** *lo* ≤ *hi* **do**

*mid* ← ⎣*lo* + (*hi* – *lo*)/2⎦

**if** *A*[*mid*] = *key* **then**

**return** *mid*

**if** *A*[*mid*] > *key* **then**

*hi* ← *mid* – 1

**else**

*lo* ← *mid* + 1

**return**-1

**function** BINSEARCH(*A*[·], *lo*, *hi*, *key*)

**if** *lo* > *hi* **then**

**return** -1

*mid* ← ⎣*lo* + (*hi* – *lo*)/2⎦

**if** *A*[*mid*] = *key* **then**

**return** *mid*

**if** *A*[*mid*] > *key* **then**

**return** BINSEARCH(*A*, *lo*, *mid* – 1, *key*)

**else**

**return** BINSEARCH(*A*, *mid* + 1, *hi*, *key*)

1. **Interpolation Search**
2. **Quick Select**

**function** QUICKSELECT(*A*[·], *lo*, *hi, k*)

*s* ← LOMUTOPARTITION(*A*, *lo*, *hi*)

**if** *s* – *lo* = *k* – 1 **then**

**return** *A*[*s*]

**else**

**if** *s* – *lo* > *k* – 1 **then**

QUICKSELECT(*A*, *lo*, *s – 1, k*)

**else**

QUICKSELECT(*A*, *s +* 1, *hi,* (*k* – 1) – (*s* – *lo*))

**function** LOMUTOPARTITION(*A*[·], *lo*, *hi*)

*p* ← *A*[*lo*]

*s* ← *lo*

**for** *i* ← *lo* + 1 to *hi* **do**

**if** *A*[*i*] < *p* **then**

*s* ← *s* + 1

*swap*(*A*[*s*], *A*[*i*])

*swap*(*A*[*lo*], *A*[*s*])

**return** *s*

**PART V: Graph Algorithms**

1. **Depth-First Search** (这个除了会写顺序外还要会画DFS tree和stack)

**function** DFS(<*V*, *E*>)

mark each node in *V* with 0

*count* ← 0

**for** each *v* in *V* **do**

**if** *v* is marked with 0 **then**

DFSEXPLORE(*v*)

**function** DFSEXPLORE(*v*)

*count* ← *count* + 1

mark *v* with *count*

**for** each edge (*v*, *w*) **do**

**if** *w* is marked with 0 **then**

DFSEXPLORE(*w*)

1. **Breath-First Search** (这个除了会写顺序外还要会画BFS tree)

**function** BFS(<*V*, *E*>)

mark each node in *V* with 0

*count* ← 0, *init*(*queue*)

**for** each *v* in *V* **do**

**if** *v* is marked with 0 **then**

*count* ← *count* + 1

mark *v* with *count*

*inject*(*queue*, *v*)

**while** *queue* is non-empty **do**

*u* ← *eject*(*queue*)

**for** each edge (*u*, *w*) **do**

**if** *w* is marked with 0 **then**

*count* ← *count* + 1

mark w with *count*

*inject*(*queue*, *w*)

1. **Topological Sorting**

* Algorithm 1:
* *Step 1*: Perform DFS and note the order in which nodes are popped off the stack
* *Step 2*: List the nodes in the reverse of that order
* Algorithm 2:

Repeatedly select a random source (a node with no incoming edges), list it, and remove it from the graph (including removing its outgoing edges)

1. **\*Cyclicity**

**function** ISCYCLIC(<*V*, *E*>)

mark each node in *V* with 0

**for** each *v* in *V* **do**

**if** v is marked with 0 **then**

**if** DFSEXPLORE(*v*) = True **then**

**return** True

**return** False

**function** DFSEXPLORE(*v*)

mark *v* with 1

**for** each edge (*v*, *w*) **do**

**if** *w* is marked with 0 **then**

**if** DFSEXLORE(*w*) **then**

**return** True

**else**

**if** (*v*, *w*) is a back edge **then**

**return** True

**return** False

1. **\*Connectedness**

**function** CONNECTEDNESS(<*V*, *E*>)

mark each node in *V* with 0

*component* ← 1

**for** each *v* in *V* **do**

**if** *v* is marked with 0 **then**

DFSEXPLORE(*v*)

*component* ← *component* + 1

**function** DFSEXPLORE(*v*)

mark *v* with component

**for** each edge (*v*, *w*) **do**

**if** *w* is marked with 0 **then**

DFSEXPLORE(*w*)

1. **Warshall’s Algorithm**

Warshall’s algorithm computes the transitive closure of a binary relation (or a directed graph), presented as a matrix. An edge (*a*, *z*) is in the transitive closure of graph *G* iff there is a path in *G* from *a* to *z*.

* *Step 1*: Is there a path from node *i* to node *j* using only nodes that are no larger than some *k* as “stepping stones”?
* *Step 2*: Such a path either uses node *k* as a stepping stone, or it doesn’t;
* *Step 3*: There is such a path if and only if we can step from I to j using only nodes ≤ *k* – 1, or step from I to k using only nodes ≤ *k* – 1, and then step from k to j using only nodes ≤ *k* – 1.

If G’s adjacency matrix is A then we can express the recurrence relation as

*R*0*ij*= *A*[*i*, *j*]

*Rkij* = *Rk* – 1*ij* or (*Rk* – 1*ik*and *Rk* – 1*kj*)

这道题目最好考前多画几遍！

**function** WARSHALL(*A*[·,·], *n*)

*R*[·,·, 0] ← *A*

**for** *k* ← 1 to *n* **do**

**for** *i* ← 1 to *n* **do**

**for** *j* ← 1 to *n* **do**

*R*[*i*, *j*, *k*] ← *R*[*i*, *j*, *k –* 1] or (*R*[*i*, *k*, *k –* 1] and *R*[*k*, *j*, *k –* 1])

**return** *R*[·,·, *n*]

**function** WARSHALL(*A*[·,·], *n*)

**for** *k* ← 1 to *n* **do**

**for** *i* ← 1 to *n* **do**

**if** *A*[*i*, *k*] **then**

**for** *j* ← 1 to *n* **do**

**if** *A*[*k*, *j*] **then**

*A*[*i*, *j*] ← 1

1. **Floyd’s Algorithm**

Floyd’s algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.

原理同上！要学会画表！

**function** FLOYD(*W*[·,·], *n*)

*D* ← *W*

**for** *k* ← 1 to *n* **do**

**for** *i* ← 1 to *n* **do**

**for** *j* ← 1 to *n* **do**

*D*[*i*, *j*] ← *min*(*D*[*i*, *j*], *D*[*i*, *k*] + *D*[*k*, *j*])

**return** *D*

Sub-structure property:

Shortest-path problems have this property. For example, if {*x*1, *x*2, …, *x*i, …, *x*n} is a shortest path from *x*1 to *x*n then {*x*1, *x*2, …, *x*i} is a shortest path from *x*1 to *x*n. Longest-path problems don’t have that property.

1. **Prim’s Algorithm**

A spanning tree of a graph <*V*, *E*> is a tree <*V*, *E*’> where *E*’ is a subset of *E*. Prim’s algorithm constructs a sequence of subtrees T, by adding to the latest tree the closest node not currently on it.

同上上！屁屁踢上有一个例子，你们把bc边去掉再画一下，有惊喜

**function** PRIM(<*V*, *E*>)

**for** each *v* in *V* **do**

*cost*[*v*] ← ∞

*prev*[*v*] ← *nil*

pick initial node *v*0

*cost*[*v*0] ← 0

*Q* ← INITPRIORITYQUEUE(*V*)

**while** *Q* is non-empty **do**

*u* ← EJECTMIN(*Q*)

**for** each (*u*, *w*) in *E* **do**

**if** *weight*(*u*, *w*) < *cost*[*w*] **then**

*cost*[*w*] ← *weight*(*u*, *w*)

*prev*[*w*] ← *u*

UPDATE(*Q*, *w*, *cost*[w])

1. **Dijkstra’s Algorithm**

Dijkstra’s algorithm is also a shortest-path algorithm for (directed or undirected) weighted graphs. It finds all shortest paths from a fixed start node.

同上上上

**function** DIJKSTRA(<*V*, *E*>, *v*0)

**for** each *v* in *V* **do**

*dist*[*v*] ← ∞

*prev*[*v*] ← *nil*

*dist*[*v*0] ← 0

*Q* ← INITPRIORITYQUEUE(*V*)

**while** *Q* is non-empty **do**

*u* ← EJECTMIN(*Q*)

**for** each (*u*, *w*) in *E* **do**

**if** *dist*[*u*] + *weight*(*u*, *w*) < *dist*[*w*] **then**

*dist*[*w*] ← *dist*[*u*] + *weight*(*u*, *w*)

*prev*[*w*] ← *u*

UPDATE(*Q*, *w*, *dist*[w])

我好像发现了一个小秘密：prim和dj两个画表的过程都极其相似，唯一不同的地方就素prim取当前最小，dj取的是累加最小，emmmmm看上去非常有道理

**PART VI: String Manipulation Algorithms**

1. **Brute Force String Search**

**function** STRINGSEARCH(*P*[·], *m*, *T*[·], *n*)

**for** *i*← 0 to *n* – *m* **do**

*j*← 0

**while** *j* < *m*and *P*[*j*] = *T*[*i* + j] **do**

*j*← *j* + 1

**if** *j = m* **then**

**return** *i*

**return** -1

1. **Horspool’s String Search**

**function** FINDSHIFTS(*P*[·], *m*)

**for** *i* ← 0 to *alphasize* – 1 **do**

*Shift*[*i*] ← *m*

**for** *j* ← 0 to *m* – 2 **do**

*Shift*[*P*[*j*]] ← *m* – (*j* + 1)

**function** HORSPOOL(*P*[·], *m*, *T*[·], *n*)

FINDSHIFTS(*P*, *m*)

*i* ← *m* – 1

**while** *i < n* **do**

*k* ← 0

**while** *k* < *m* and *P*[*m* – 1 – k] = *T*[*i* – k] **do**

*k* ← *k* + 1

**if** *k* = *m* **then**

**return** *i* – *m* + 1

**else**

*i* ← *i* + *Shift*[*T*[*i*]]

**return** -1

注：考试可能会给一个骚气的*P*，里面有重复的字母，记得所有字母全部取最小值，比如BARBER，取B: 2, R: 3

**PART VII: Other Algorithms**

1. **Closest Pair Problem**

* Algorithm 1:

**function** CLOSESTPAIR(*X*[·], *Y*[·], *n*)

*min* ← ∞

**for** *i* ← 0 to *n* – 2 **do**

**for** *j* ← *i* + 1 to *n* – 1 **do**

*d* ← sqrt((*X*[*i*] – *X*[*j*])2 + (*Y*[*i*] – *Y*[*j*])2)

**if** *d* < *min* **then**

*min* ← *d*

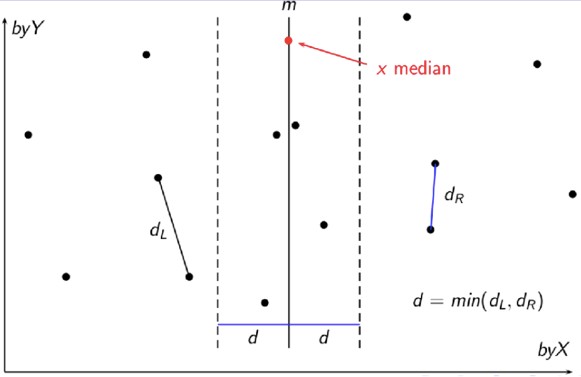
*p*1 ← *i*

*p*2 ← *j*

**return** *i*, *j*

Time complexity: O(*n*2)

* Algorithm 2:
* *Step 1*: Sort the points by *x* value and store the result in array *byX*. Also sort the points by *y* value and store the result in array *byY*;
* *Step 2*: Identify the *x* median, and recursively process the set *PL* of points with lower *x* values, as well as the set *PR* with higher *x* values;
* *Step 3*: The recursive calls will identify *dL*, the shortest distance for pairs in *PL*, and *dR*, the shortest distance for pairs in *PR*;
* *Step 4*: Let *m* be the x median and let *d* = *min*(*dL*, *dL*). This *d* is a candidate for the smallest distance;
* *Step 5*: But d may not be the global minimum - there could be some close pair whose points are on opposite sides of the median line *x* = *m*;
* *Step 6*: For candidates that may improve on *d* we only need to look at those in the band *m* – *d* ≤ *x* ≤ *m* + *d*;
* *Step 7*: So pick out, from array *byY,* each point *p* with x-coordinate *m* – *d* and *m* + *d,* and keep these in array *S.*
* *Step 8*: For each point in *S*, consider just its “close” neighbors in *S*.



这个算法是啥子哦？？？后面的伪代码看都看球不懂，我看不懂的我猜他也应该不会考

**function** CLOSESTPAIR(*P*[·], *Q*[·], *n*)

**if** *n* ≤ 3 **then**

return the minimal distance found by the brute-force algorithm

**else**

copy the first ⌈*n*/2⌉ points of *P* to array *Pl*  
 copy the same ⌈*n*/2⌉ points from *Q* to array *Ql*  
 copy the remaining ⌊*n*/2⌋ points of *P* to array *Pr*  
 copy the same ⌊*n*/2⌋ points from *Q* to array *Qr*  
 *dl* ← CLOSESTPAIR(*Pl*, *Ql,* ⌈*n*/2⌉)  
 *dr* ← CLOSESTPAIR(*Pr*, *Qr,* ⌊*n*/2⌋)  
 *d* ← *min*(*dl*, *dr*)  
 *m* ← *P*[⌈*n*/2⌉ − 1] · *x*  
 copy all the points of *Q* for which |*x* − *m*| < *d* into array *S*[0..*num* − 1]

*dminsq* ← *d*2  
 **for** *i* ← 0 to *num* − 2 **do**

*k* ← *i* + 1  
 **while** *k* ≤ *num* − 1 and (*S*[*k*] · y − *S*[*i*] · *y*)2 < *dminsq*

*dminsq* ← *min*((*S*[*k*] · *x* − *S*[*i*] · *x*)2 + (*S*[*k*] · *y* − *S*[*i*] · *y*)2, *dminsq*)

*k* ← *k* + 1

**return** *sqrt*(*dminsq*)

Time complexity: O(*n*log*n*)

1. **Fibonacci Number**

**function** FIB(*n*)

*a* ← 1

*b* ← 0

**while** *n* > 0 **do**

*t* ← *a*

*a* ← *a* + *b*

*b* ← *t*

*n* ← *n* – 1

**return** *a*

**function** FIB(*n*, *a*, *b*)

**if** *n* = 0 **then**

**return** *a*

**return** FIB(*n* – 1, *a* + *b*, *a*)

Time complexity O(n)

**function** FIB(*n*)

**for** *i* ← 0 to *n* **do**

*F*[*n*] ← 0

**if** *n* = 0 or *n* = 1 **then**

**return** 1

*result* ← *F*[*n*]

**if** *result* = 0 **then**

*result* ← FIB(*n* – 1) + FIB(*n* – 2)

*F*[*n*] ← *result*

**return** *result*

1. **Tower of Hanoi**

**function** HANOI(*n*, *init*, *aux*, *fin*)

**if** *n* > 0 **then**

HANOI(*n* – 1, *init*, *fin*, *aux*)

Move one disk from *init* to *fin*

HANOI(*n* – 1, *aux*, *init*, *fin*)

1. **Finding the mode**

**function** MODE(*A*[·], *n*)

SORT(*A*, *n*)

*i* ← 0

*maxfreq* ← 0

**while** *i* < *n* **do**

*runlength* ← 1

**while** *i* + *runlength* < *n* and *A*[*i* + *runlength*] = *A*[*i*] **do**

*runlength* ← *runlength* + 1

**if** *runlength* > *maxfreq* **then**

*maxfreq* ← *runlength*

*mode* ← *A*[*i*]

*i* ← *i* + *runlength*

**return** *mode*

1. **The Coin Row Problem**

* *Step 1*: Let the values of the coins be *v*1, *v*2,…*vn*;
* *Step 2*: Let *S*(*i*) be the sum that can be gotten by picking optimally from the first *i* coins;
* *Step 3*: Either the *i*th coins (with value *vi*) is part of the solution or it is not;
* *Step 4*: If we choose to pick it up then we cannot also pick its neighbor on the left, so the best we can achieve is *S*(*i* – 2) + *vi*;
* *Step 5*: Otherwise we leave it, and the best we can achieve is *S*(*i* – 1).

Here is saying the same thing formally, as a recurrence relation:

*S*(*n*) = *max*{*S*(*n* – 1), *S*(*n* – 2) + *vn*}

This holds for *n* > 1.

We need two base cases: *S*(0) = 0 and *S*(1) = *v*1.

**function** COINROW(*C*[·], *n*)

*S*[0] ← 0

*S*[1] ← *C*[1]

**for** *i* ← 2 to *n* **do**

*S*[*i*] ← *max*(*S*[*i* – 1], *S*[*i* – 2] + *C*[*i*])

**return** *S*[*n*]

time (and space) complexity: O(*n*)

1. **The Knapsack Problem**

* *Step 1*: Let *K*(*i*, *w*) be the value of the best choice of items amongst the first *i* using knapsack capacity *w*;
* *Step 2*: Among the first *i* items we either pick item *i* or we don’t;
* *Step 3*: For a solution that excludes item I, the value of an optimal subset is simply *K*(*i –* 1, *w*);
* *Step 4*: For a solution that includes item I, apart from that item, an optimal solution contains an optimal subset of the first I – 1 items that will fit into a bag of capacity *w* – *wi*. The value of such a subset is *K*(*i –* 1, *w* – *wi*) + *vi*, provided item I fits, that is, provided *w* – *wi* ≥ 0.

Now it is easy to express the solution recursively:

*K*(*i*, *w*) = 0 if *i* = 0 or *w* = 0

Otherwise:

这道题目有个很骚气的表考前最好多画几遍!

**function** KNAPSACK(*n*, *W*)

**for** *i* ← 0 to *n* **do**

*K*[*i*, 0] ← 0

**for** *j* ← 1 to *W* **do**

*K*[0, *j*] ← 0

**for** *i* ← 1 to *n* **do**

**for** *j* ← 1 to *W* **do**

**if** *j* < *wi* **then**

*K*[*i*, *j*] ← *K*[*i* – 1, *j*]

**else**

*K*[*i*, *j*] ← max(*K*[*i* – 1, *j*], *K*[*i* – 1, *j - wi*] + *vi*)

**return** *K*[*n*, *W*]

time (and space) complexity: O(*nW*)

**PART VIII: Algorithmic Techniques**

这个地方可以出一些骚气的题目，之前那套卷子里有一个举一个stable的排序算法并求复杂度，这里可以出写一个用了divide-and-conquer的排序算法并求复杂度

1. **Brute Force Methods**

Exhaustive search for solutions

Example: selection sort, brute force string search, closest pair problem algorithm 1, DFS, BFS

1. **Decrease-and-Conquer**

The size of the problem is reduced by some constant/factor in each iteration of the algorithm.

Example:

* Decrease-by-a-constant: insertion sort, shellsort, topological sorting algorithm 2
* Decrease-by-a-factor: binary search, quick select, interpolation search

1. **Divide-and-Conquer**

* Divide the given problem instance into smaller instances;
* Solve the smaller instances recursively;
* Combine the smaller solutions to solve the original instance.

Example:

mergesort, quicksort, tree traversal, closest pair problem algorithm 2

1. **Transform-and-Conquer**

Try to make the problem easier through some type of pre-processing, typically sorting.

Example:

finding the median, uniqueness checking, finding the mode, finding the anagram.

Instance simplification: AVL trees, red-black trees, splay trees

Representational changes: 2-3 trees, 2-3-4 trees, B-trees

1. **Dynamic Programming**

Dynamic programming is a bottom-up problem solving technique. The idea is to divide the problem into smaller, overlapping ones. The results are tabulated and used to find the complete solution.

Example: Fibonacci number algorithm 3, the coin row problem, the knapsack problem.

1. **Greedy Method**

A problem solving strategy is to take the locally best choice among all feasible ones. Once we do this, our decision is irrevocable.（同上）

Example: Prim’s algorithm, Dijkstra’s algorithm

1. **Time/Space Tradeoff**

Spend space to save time

Example:

Fibonacci number algorithm 3, distribution counting sort, Horspool’s string search

**PART IX: Algorithm Analysis**

1. **Asymptotic Notation**

0 *f*(*n*) ∈ O(*g*(*n*))

limn→∞ *f*(*n*)/*g*(*n*) = c *f*(*n*) ∈ Θ(*g*(*n*))

∞ *f*(*n*) ∈ Ω(*g*(*n*))

1. **The Master Theorem**

For integer constants *a* ≥ 1 and *b* > 1, and function *f* with *f*(*n*) ∈ Θ(*nd*), the recurrence

*T*(*n*) = *aT*(*n*/*b*) + *f*(*n*)

(with *T*(1) = *c*) has solutions, and

Θ(*nd*) if *a* < *bd*

*T*(*n*) = Θ(*nd*log*n*) if *a* = *bd*

Θ(*n*log*ba*) if *a* > *bd*

Note that we also allow *a* to be greater than *b*

--------------完结撒花✿✿ヽ(°▽°)ノ✿----------------