Algorithms

1. Depending on programming language: A[0] up to A[n-1], or A[1] up to A[n].
2. Stack

   Last-in-first-out (LIFO).

   Operations: CreateStack Push Pop Top(to see what is on the top) EmptyStack?
3. Queue

   First-in-first-out (FIFO).

   Operations: CreateQueue Enqueue Dequeue Head EmptyQueue?
4. Time complexity

   for some c and n_0. A more common approach uses

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g \\ c & \text{implies } t \text{ and } g \text{ have same order of growth} \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g \end{cases}$$

   So, for example, if ln is the natural logarithm then

$$\begin{aligned} \log_2 n &= O(\ln n) \\ \ln n &= O(\log_2 n) \end{aligned}$$

   Also note that since $\log n^c = c \cdot \log n$, we have, for all constants c,

$$\log n^c = O(\log n)$$

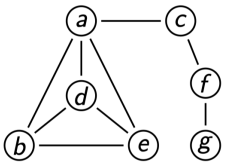
$$\begin{aligned} O(f(n)) + O(g(n)) &= O(\max\{f(n), g(n)\}) \\ c \cdot O(f(n)) &= O(f(n)) \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)). \end{aligned}$$

5. In place

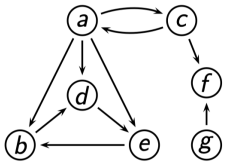
   - in-place if it does not require additional memory except, perhaps, for a few units of memory.
   - stable if it preserves the relative order of elements that have identical keys.
   - input-insensitive if its running time is fairly independent of input properties other than size.

6. Graph

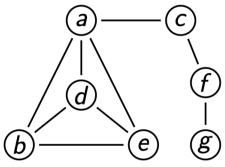
Undirected:



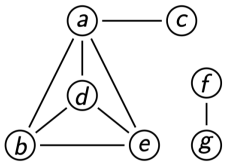
Directed:



Connected:



Not connected, two components:



If $(v, u) \in E$ then v and u are adjacent (比邻), or neighbours.

(v, u) is incident on, or connects, v and u.

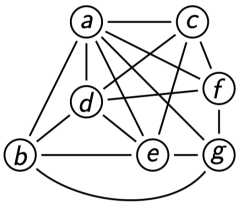
The degree of node v is the number of edges incident on v.

For directed graphs, we talk about v's in-degree (number of edges going to v) and its out-degree (number of edges going from v).

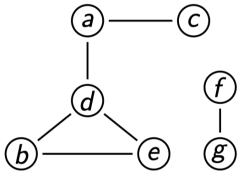
A simple path is one that has no repeated nodes.

A cycle is a simple path, except that $v_0 = v_k$, that is, the last node is the same as the first node.

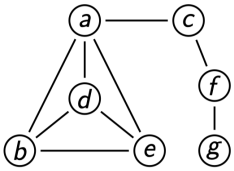
Dense (密度大的):



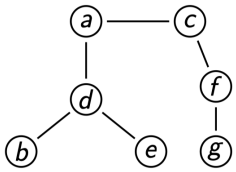
Sparse (稀少的):



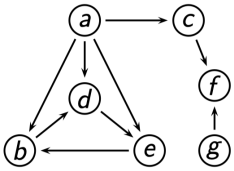
Cyclic:



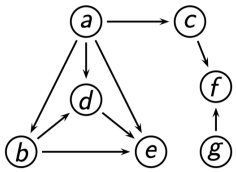
Acyclic (actually, a tree):



Directed cyclic:



Directed acyclic (a dag):



7. Insertion Sort

Insertion Sort

Sorting an array:

```
function InsertionSort(A[0..n − 1])
    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
```

Shellsort

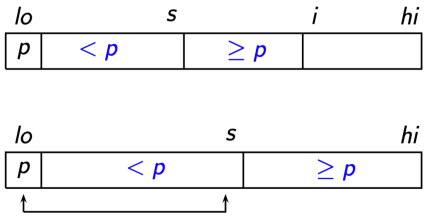
This is a version of insertion sort which aims at moving out-of-order elements over bigger distances (not just to the neighbouring cell as in insertion sort).

98 14 55 31 44 83 25 77 47 57 49 52 72 29 64 26 33 89 38 32 94 17

8. Quick Select

Lomuto Partitioning

```
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
```



Finding the kth Smallest Element

Here is how we can use partitioning to find the kth smallest element.

```
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))
```

9. Interpolation Search

we perform a linear interpolation between the points $(lo, A[lo])$ and $(hi, A[hi])$. That is, we use

$$m \leftarrow lo + \left\lfloor \frac{k - A[lo]}{A[hi] - A[lo]} (hi - lo) \right\rfloor$$

Interpolation search has average-case complexity $O(\log \log n)$.

10. Master Therom

For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d), d \geq 0$, the recurrence

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

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

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

11. Merge Sort

```
function MERGESORT(A[0..n − 1])
    if n > 1 then
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        MERGESORT(B[0..⌊n/2⌋ − 1])
        MERGESORT(C[0..⌈n/2⌉ − 1])
        MERGE(B, C, A)

function MERGE(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    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[j]
            j ← j + 1
        k ← k + 1
    if i = p then
        copy C[j..q − 1] to A[k..p + q − 1]
    else
        copy B[i..p − 1] to A[k..p + q − 1]
```

12. Quick Sort

```
function QUICKSORT(A[lo..hi])
    if lo < hi then
        s ← PARTITION(A[lo..hi])
        QUICKSORT(A[lo..s − 1])
        QUICKSORT(A[s + 1..hi])
```

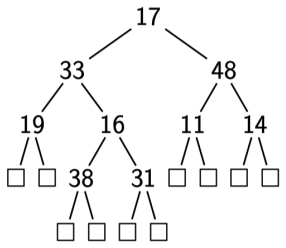
```
function PARTITION(A[lo..hi])
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j]) — undo the last swap
    swap(A[lo], A[j]) — bring pivot to its correct position
    return j
```

The best case happens when the pivot is the median; that results in two sub-tasks of equal size.

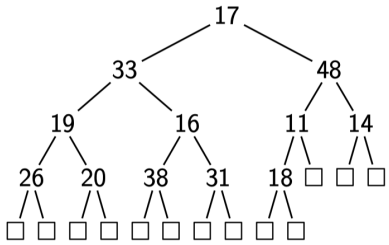
$$C_{best}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C_{best}(n/2) + n & \text{otherwise} \end{cases}$$

The worst case happens if the array is already sorted.

13. Full and complete



A full binary tree: Each node has 0 or 2 children.



A complete tree: Each level filled left to right.

14. Height

```
function HEIGHT(T)
    if T is empty then
        return −1
    else
        return max(HEIGHT(T_left), HEIGHT(T_right)) + 1
```

15. Traversal

Preorder traversal visits the root, then the left subtree, and finally the right subtree.

Inorder traversal visits the left subtree, then the root, and finally the right subtree.

Postorder traversal visits the left subtree, the right subtree, and finally the root.

Level-order traversal visits the nodes, level by level, starting from the root.

16. Heap

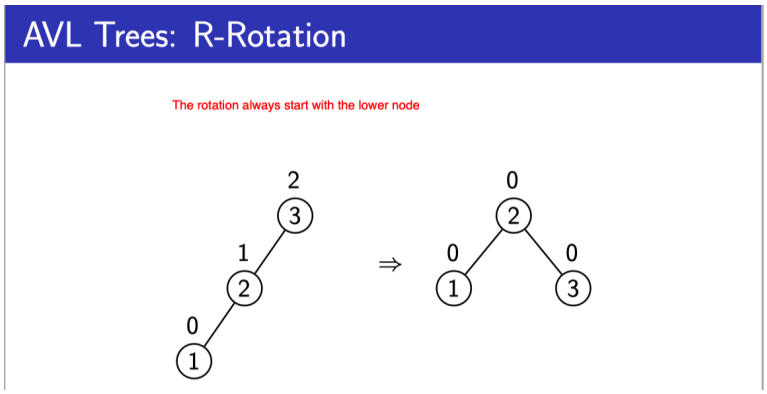
A heap is a complete binary tree which satisfies the heap condition:

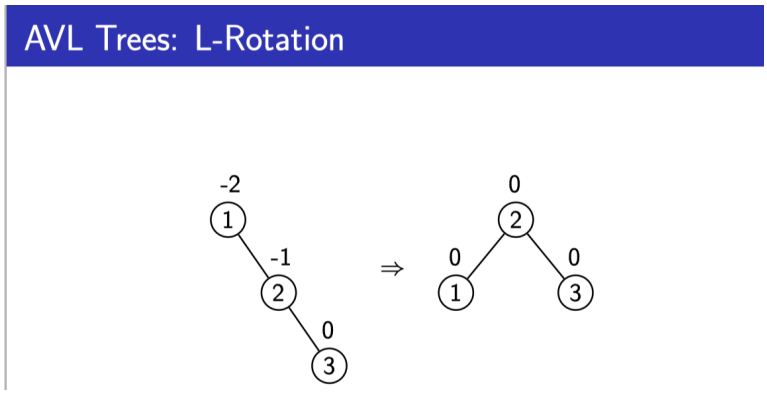
Each child has a priority (key) which is no greater than its parent's.

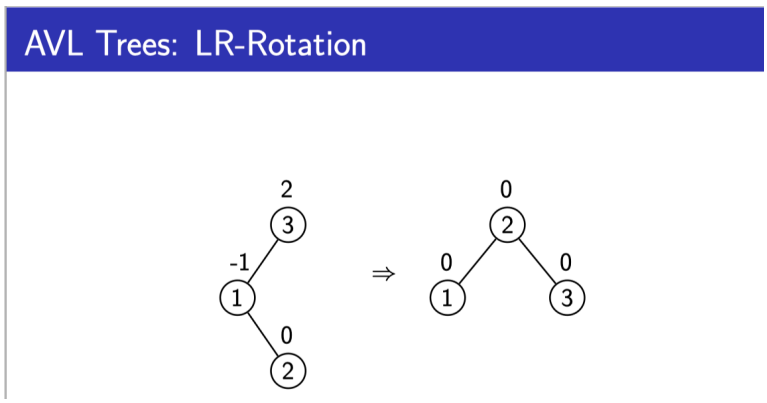
17. Ejecting a Maximal Element from a Heap

    swap the root with the last item z in the heap, and then let z "sift down" to its proper place.

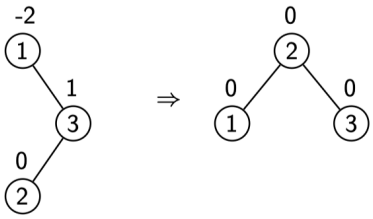
18. AVL tree







AVL Trees: RL-Rotation



19. Sorting by counting

For example, suppose all keys are single digits:

6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

Then we can, in a single linear scan, count the occurrences of each key in array A and store the result in a small table:

key	0	1	2	3	4	5	6	7	8	9
Occ	1	4	2	5	0	4	2	2	3	1

Now use a second linear scan to make the counts cumulative: 累计

key	0	1	2	3	4	5	6	7	8	9
Occ	1	5	7	12	12	16	18	20	23	24

Place the first record (with key 6) in $S[18]$ and decrement $Occ[6]$ (so that the next '6' will go into slot 17), and so on.

for $i \leftarrow 1$ **to** n **do**
    $S[Occ[A[i]]] \leftarrow A[i]$
    $Occ[A[i]] \leftarrow Occ[A[i]] - 1$

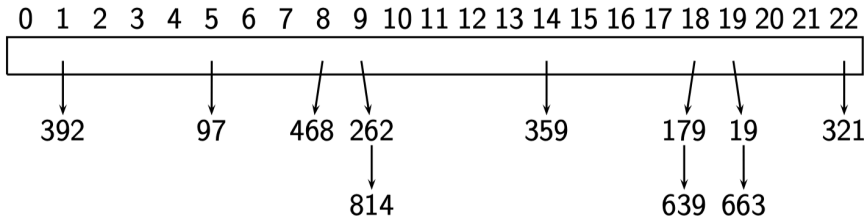
20. Horspool's string search

21. Hash table

Operation: find insert lookup (search and insert if not there) initialize delete rehash
Separate Chaining

Separate Chaining

Element k of the hash table is a list of keys with the hash value k.



This gives easy collision handling.

The load factor $\alpha = n/m$, where n is the number of items stored.

Number of probes in successful search $\approx 1 + \alpha/2$.

Number of probes in unsuccessful search $\approx \alpha$.

With open-addressing methods (also called closed hashing) all records are stored in the hash table itself (not in linked lists hanging off the table).

There are many methods of this type. We only discuss two:

- linear probing
- double hashing

Linear Probing

Again let m be the table size,
and n be the number of records stored.

As before, $\alpha = n/m$ is the load factor.

Average number of probes:

- Successful search: $\frac{1}{2} + \frac{1}{2(1-\alpha)}$
- Unsuccessful: $\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$

For successful search:

α	#probes
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Linear Probing Pros and Cons

Space-efficient.

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

Comparative behaviour, $m = 11113$, $n = 10000$, $\alpha = 0.9$:

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

Deletion is almost impossible.

Double Hashing

To alleviate the clustering problem in linear probing, there are better ways of resolving collisions.

One is double hashing which uses a second hash function s to determine an offset to be used in probing for a free cell.

For example, we may choose $s(k) = 1 + k \bmod 97$.

By this we mean, if $h(k)$ is occupied, next try $h(k) + s(k)$, then $h(k) + 2s(k)$, and so on.

This is another reason why it is good to have m being a prime number. That way, using $h(k)$ as the offset, we will eventually find a free cell if there is one.

Some drawbacks:

If an application calls for traversal of all items in sorted order, a hash table is no good.

Also, unless we use separate chaining, deletion is virtually impossible.

It may be hard to predict the volume of data, and rehashing is an expensive "stop-the-world" operation.

22. Dynamic Programing

23. Warshall's algorithms

```
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
```

24. Floyd's algorithms

```
function FLOYD(W[1..n, 1..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
```

25. Prim

```
function PRIM(⟨V, E⟩)
    for each v ∈ V do
        cost[v] ← ∞
        prev[v] ← nil
    pick initial node v_0
    cost[v_0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are cost values
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ E do
            if weight(u, w) < cost[w] then
                cost[w] ← weight(u, w)
                prev[w] ← u
                UPDATE(Q, w, cost[w])   ▷ rearranges priority queue
```

26. Dijkstra

```
function DIJKSTRA(⟨V, E⟩, v_0)
    for each v ∈ V do
        dist[v] ← ∞
        prev[v] ← nil
    dist[v_0] ← 0
    Q ← INITPRIORITYQUEUE(V)        ▷ priorities are distances
    while Q is non-empty do
        u ← EJECTMIN(Q)
        for each (u, w) ∈ 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])   ▷ rearranges priority queue
```

SPECIAL CARE

1. Cyclic graph can't use topological sort
2. Selection sort

```
function SELSORT(A[0..n − 1])
    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
        swap A[i] and A[min]
```

3. Use master therom must a>=1 b>1
4. Spanning tree

   Given a weighted graph, a sub-graph which is a tree with minimal weight is a minimum spanning tree for the graph.

5. Dynamic programing

   If a dynamic programming problem satisfies the optimal-substructure property, then a locally optimal solution is globally optimal.

6. An abstract data type defines what the data type can do and can have multiple different underling implementation.

An Abstract Data Type (such as a Dictionary) describes the functionality of the type (such insertion, deletion, modification, and lookup of an item) but not its underlying implementation. A Dictionary can be implemented using a range of concrete data structures, including hash tables and trees.

When considering different types of algorithms, which one of the following statements is true?

Selected Answer:	Brute-force algorithms can never be faster than a well-designed greedy algorithm for the same problem.
Answers:	Brute-force algorithms can never be faster than a well-designed greedy algorithm for the same problem.
	Divide and conquer algorithms are always faster than greedy algorithms for the same problem
	Greedy algorithms are always faster than divide and conquer algorithms for the same problem
	Greedy algorithms give good approximate answers to problems, but never the best possible answer.

7.

8.