**O(NlogN)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Best | Average | Worst | Space | Place | Stable |
| Selection | N2 | N2 | N2 | 1 | In | N |
| Insertion | N | N2 | N2 | 1 | In | Y |
| Bubble | N | N2 | N2 | 1 | In | Y |
| Quick | NlogN | NlogN | N2 | logN | In | N |
| Merge | NlogN | NlogN | NlogN | N | Out | Y |
| Heap | NlogN | NlogN | NlogN | 1 | In | N |
| Counting | N+k | N+k | N2 | k | Out | Y |
| Radix | Nk | Nk | Nk | N+k | Out | Y |
| Bucket | N+k | N+k | N2 | N | Out | Y |

Quick need logN for recursion stack.

1. opti ; randomly choose pivot

2. worst case : always pick the biggest/smallest as the pivot, decayed to insertion sort

Counting

malloc跟range一樣大小的array. 2.另外malloc output array.

Radix

把每一位數用countingSort來sort

Bucket

每一bucket放某一位數相同的值, bucket內自已sort

**O(logN)**

BST

AVLT

1. 節點數相同時候，由於AVL樹時嚴格平衡的，它比紅黑樹高度要小。

2. AVL is faster to search, Red-Black faster to insert

3. If insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred"

RBT

Definition

1. root是黑色

2.每一個leaf是黑色

3.紅色node的兩個child是黑色

4.從任意node走到其descendant leave所經過的黑色node數量相等"

Characteristics

1.插入、刪除、搜索操作的複雜度都是logN

2. 插入最多兩次旋轉，刪除最多三次旋轉。

3. 藉由控制顏色，能夠保證在RBT中，最長path(路徑)不會超過最短path的兩倍.

4. 紅黑樹的典型應用是關聯數組，linux內核中的用戶態地址空間管理使用了紅黑樹。另外，java中的TreeSet類的底層也是紅黑樹來實現的。

5. If application involves many frequent insertions and deletions, then Red Black trees should be preferred.

**O(1)**

Hash

Definition

1. Use hash function that converts a given key to a smaller number and uses the small number as index in a table called hash table.

2. Hash func ensures the keys are well distributed.

Characteristics

1. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

2. Collision handling

a. separate chaining

Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in worst case.

4) Uses extra space for links.

b. open addressing

If hash(x) is full

(a). linear probing : hash(x)+i

(b). quadratic probing : hash(x)+i^2

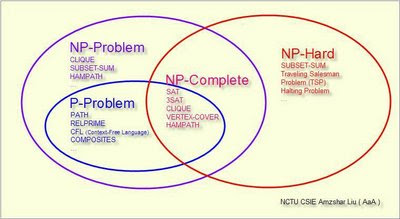
(c). double hashing : hash(x)+i\*hash2(x)"

3. Cache performance & Clustering problem"

4. std::map - RB Tree

5. std::unordered\_map - Hash Table"

=============================================================



A Decision Problem L is

**P**

Solved by a deterministic Turing machine in polynomial time.

Ex: Shortest Path Problem, Longest Path in a DAG(Directed Acyclic Graph)

**NP**

1. Solved by a non-deterministic Turing Machine in polynomial time

NDTM:may have a set of rules that prescribes more than one action for a given situation

2. Solution can be verified in polynomial time

**NP-Complete**

1. L is in NP

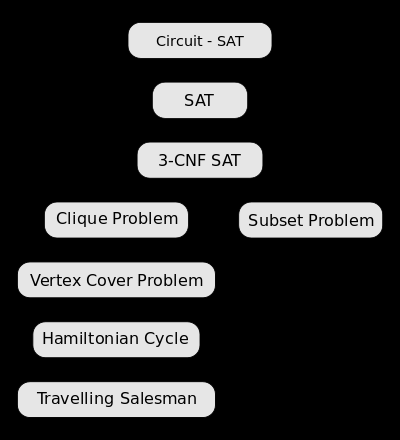
2. Every problem in NP is reducible to L in polynomial time

Ex:

1st NPC problem : SAT (Boolean satisfiability problem)

對一個確定的邏輯電路，是否存在一種輸入使輸出為true

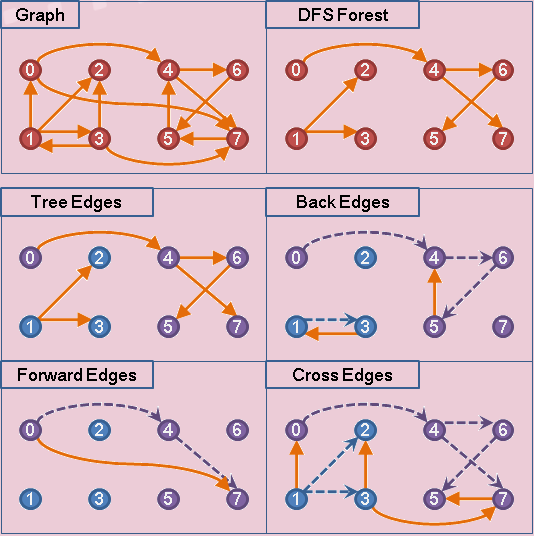
ex:



**NP-Hard**

1. Every problem in NP is reducible to L in polynomial time
2. CAN NOT be verified in polynomial time
3. Optimization Problem

ex : Go, Turing Halting Problem, Longest Path in a **general graph**



**Graph**

Objects & Points

1. Vertex a = new Vertex(1); // vertex 1

2. Vertex b = new Vertex(2); //vertex 2

3. Edge e = new Edge(a,b, 30); // init an edge between a and b with weight 30

Pros:

1. Saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space.

2. Adding a vertex is easier.

Cons:

1. Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(E).

Adjacency Matrix

1. 2D array of size V x V, V is the # of vertices in a graph.

2. Adjacency matrix for undirected graph is always symmetric.

Adding an edge – O(1);

* Deleting an edge – O(1);
* Answering the question “is there an edge between i and j” – O(1);
* Finding the successors of a given vertex – O(n);
* Finding (if exists) a path between two vertices – O(n2);

Pros:

1. Representation is easier to implement and follow.

2. Removing an edge takes O(1) time.

3. Queries like whether there is an edge from v to u can be done O(1).

Cons:

1. Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space.

2. Adding a vertex is O(V^2) time.

Adjacency List

1. An array of linked lists

2. Size of the array is equal to number of vertices.

3. An entry array[i] represents the linked list of vertices adjacent to the ith vertex.

Adding an edge – O(log(n));

1. Deleting an edge – O(log(n));
2. Answering the question “is there an edge between i and j” – O(log(n));
3. Finding the successors of a given vertex – O(k), where “k” is the length of the lists containing the successors of i;
4. Finding (if exists) a path between two vertices – O(n+m) – where m <= n;

Pros:

1. Saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space.

2. Adding a vertex is easier.

Cons:

1. Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

**Graph BFT&DFT**

1. **Shortest Path** and **Minimum Spanning Tree** for unweighted graph

2. **Cycle detection** in **undirected** graph.

3. **Ford-Fulkerson** algorithm, find the maximum flow. (BFT preferred as it reduces worst case time complexity to O(VE^2))

4. **Bipartite detection** (whose vertices can be divided into two independent sets )

5. **Path finding**, if there is a path between two vertices.

6. Find all nodes reachable from a given node

**Graph BFT**

1. **Peer to Peer Networks**. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

2. **Crawlers in Search Engines**: Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

3. **Social Networking Websites**: In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

4. **GPS Navigation systems**: Breadth First Search is used to find all neighboring locations.

5. **Broadcasting in Network**: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

6. **Garbage Collection**: Breadth First Search is used in copying garbage collection using Cheney’s algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:

**Graph DFT**

1. **Cycle detection** in **directed** graph.

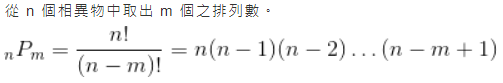
2. **Find a path** between two given vertices u and z

3. **Topological Sorting**

4. Solving puzzles with only one solution

直線排列

不可重複

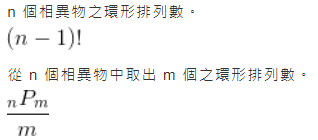


可重複



＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

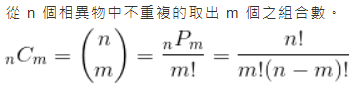
環狀排列



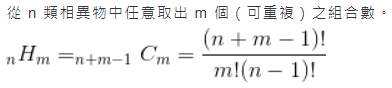
＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

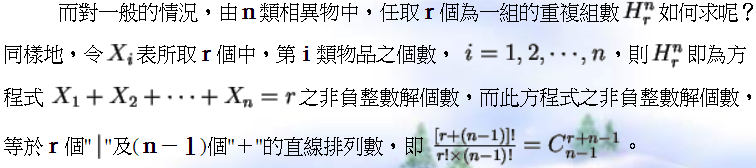
組合

不可重複



可重複





DP : 從小的答案開始計算到大的答案 or 從1,2,3…個輸入開始枚舉可能的答案，最後看被問的目標是否被枚舉到。

Search：搜索可以產生某一答案的排列組合