INT102: Algorithmic Foundations And Problem Solving

0 Introduction & Pseudo Code

0.1 Assessments

2 assessments (20% of the final mark)
Assessment 1: week 6 - week 7 (10%)
Assessment 2: week 12 – week 13 (10%)

Final Examination (80% of the final mark)
A written examination
Date: TBA.

Closed Book Exam

0.2 What is an algorithm?

---- Algorithm design is a foundation for efficient and effective programs
---- Algorithms + Data Structures = Programs

0.3 How to represent algorithms?

*---- We use **Pseudo Code** to represent algorithms.*
---- Pseudo Code is similar to programming language and more like English.

---- Statement :

```
begin
    variable = expression
end
```

---- Conditional pattern :

```
if condition then
  statement
else
  statement
```

---- Loop pattern:

```
for  var  =  start_value  to  end_value  do
  statement

while condition do      # condition to continue the loop
  statement

repeat
  statement
until condition         # condition to stop the loop
```

---- Example : Computing sum of the first n numbers

input n
sum = 0
for i = 1 to n do
begin
  sum = sum + i
end
output sum

 # we still run the loop at i = n but not at i = n-1

suppose n = 4

iteration	i	sum
before		0
1	1	1
2	2	3
3	3	6
4	4	10
end	5	

1 Algorithm efficiency: Asymptotic Analysis 渐进式分析

1.1 Example: Find the minimum among 3 numbers

```
input a, b, c
if (a <= b) then
    if (a <= c) then
        output a
    else
        output c
else
    if (b <= c) then
        output b
    else
        output c
```

There is an important operation in this pseudo code : comparison . It takes 2 comparisons. And then we extend to a question : Find the minimum among n numbers

1.2 Example: Find the minimum among n numbers

```
input a[0], a[1], ..., a[n-1]
min = 0
i = 1
while (i < n) do
begin
    if (a[i] < a[min]) then
        min = i
    i = i + 1
end
output a[min]
```

From the pseudo code , we can see that it takes n-1 comparisons.

*1.3 If the amount of data handled matches speed increase? **No !***

If we doubled the input size, how much longer would the algorithm take?
If we trebled the input size, how much longer would it take?

- *We will answer the question upper with an assumption:*

---- an algorithm takes n^2 comparisons to sort n numbers
---- we need 1 sec to do 25 comparisons

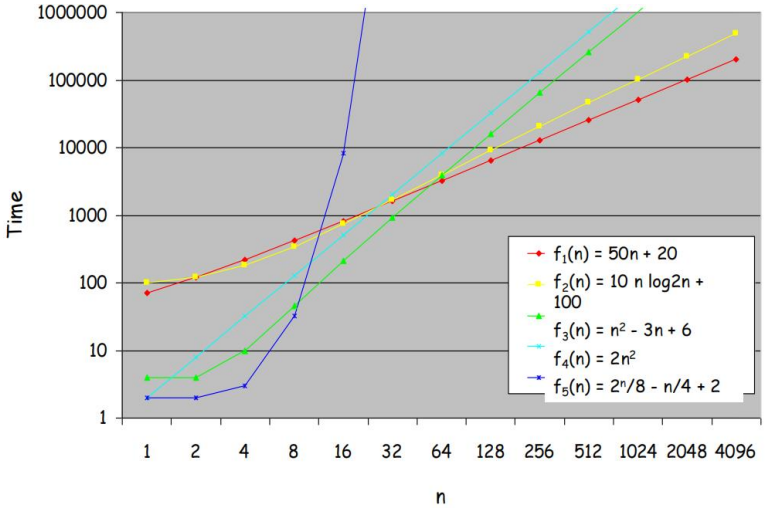
• *if computing operation speed increases by factor of 100, which means using 1 sec, we can now perform 100x25 comparisons, so we can sort 50 numbers (compared with previous 5 numbers). With 100 times speedup in operation, we only sort 10 times more numbers .*

1.4 Which algorithm is the fastest? Consider a problem that can be solved by 5 algorithms A1, A2, A3, A4, A5 using different number of operations.

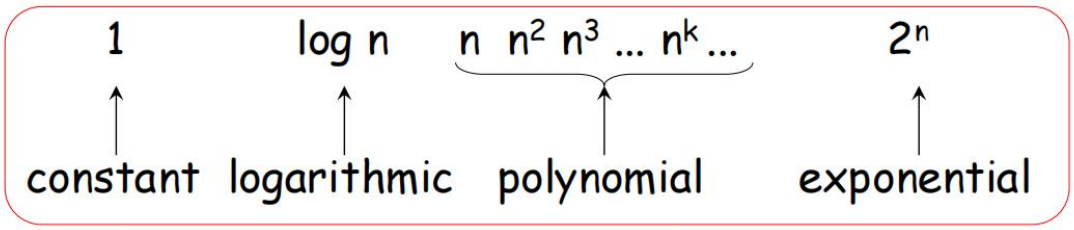
$f_1(n) = 50n + 20$ $f_2(n) = 10\ n \log_2 n + 100$

$f_3(n) = n^2 - 3n + 6$ $f_4(n) = 2n^2$

$f_5(n) = 2^n/8 - n/4 + 2$



There is huge difference between functions involving powers of n (e.g., n, n log n, n^2 called polynomial functions) and functions involving powering by n (e.g., 2^n, called exponential functions)



• *O(1) << O(log n) << O(n^1/2) < O(n^2) < O(n^k) << O(2^n) < O(n!)*

• *For example, f(n) = 2n^3 + 5n^2 + 4n + 7*
The term with the highest power is 2n^3.
The growth rate of f(n) is dominated by n^3.

*1.5 Big-O notation abstracts away constants and lower-order terms, focusing on the primary factor that affects the runtime or space requirement as the input size approaches infinity. Mathematically, define f(n) = O(g(n)) , where exists a constant c and n' such that f(n) <= c * g(n) for all n > n' , which means they have the*

*same **order of magnitude*** 数量级

Examples :
2n^3 = O(n^3)
3n^2 = O(n^2)
2n log n = O(n log n)
n^3 + n^2 = O(n^3)

2 Search and Sort

2.1 Linear Search in a sequence of numbers

```
Input:  a[0], a[1], …, a[n-1]  and  X
i = 0
while i < n do
begin
    if X == a[i] then
        output "Found!" and stop
    else
        i = i+1
end
output "Not Found!"
```

The upper is an algorithm that inputs a sequence of n numbers a[0], a[1], …, a[n-1] and a number X and outputs whether X is in the sequence or not .

The time complexity of this algorithm is not stable : O(1) ~ O(n)
Best case: X is the no.1 in list , 1 comparison, O(1)
Worst case: X is the last no. or X is not found, n comparisons, O(n)

2.2 Using binary search to improve the efficiency in a sorted sequence of numbers

*If the numbers are **pre-sorted**, then we can improve the time complexity of searching by **binary search**.*

```
Input:  sorted sequence  a[0], a[1], …, a[n-1]  and  X
first=0, last=n-1
while (first <= last) do
begin
    mid = (first+last) \ 2          #   \ get the int number from lower bound
    if (X == a[mid])
        report "Found!" & stop
    else
        if (X < a[mid])
            last = mid-1
        else
            first = mid+1
end
report "Not Found!"
```

The time complexity of this algorithm is not stable : O(1) ~ O(logn)

Best case: X is the number in the middle => 1 comparison, ***O(1)***

Worst case:at most logn or [logn]+1 (n=16/17) comparisons, ***O(logn)*** *(Every comparison reduces the amount of numbers by at least half : 16=> 8 => 4 => 2 => 1 , 4 comparison or 17 => 9=> 5 => 3 => 2 => 1 , 5 comparison)*

As ***O(log n) << O(n)*** *, therefore binary search is more efficient than linear search.*

2.3 String Matching

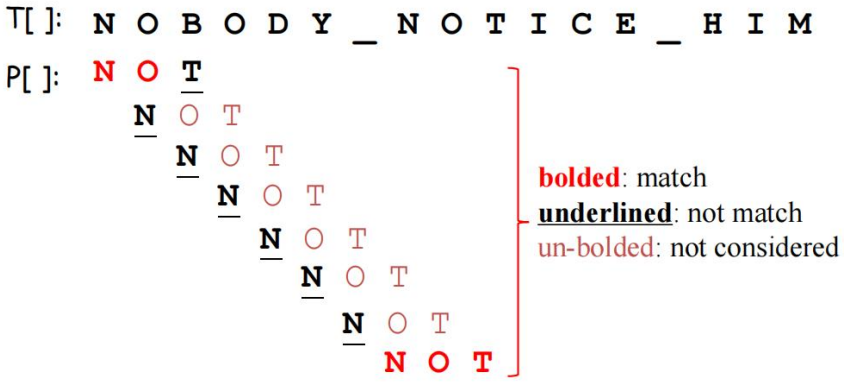
Given a string of ***n*** *characters called the* ***text*** *and a string of m characters (m<=n) called the* ***pattern****. We want to determine if the text contains a substring matching the pattern.*

Example

text: N O B O D Y _ N O T I C E _ H I M

pattern: N O T

substring: N O B O D Y _ N O T I C E _ H I M




```
for i = 0 to n-m do
begin
    j = 0
    while (j < m && P[j]==T[i+j]) do
        j = j + 1
    if (j == m) then
        report "found!" & stop
end
report "Not found!"
```

The time complexity of this algorithm is not stable : O(m) ~ O(nm)
Best case:pattern appears in the beginning of the text, O(m)
Worst case:pattern appears at the end of the text or pattern does not exist, O(nm)

2.4 Selection Sort in a sequence of numbers 选择排序

The Object of Sorting : Arrange the n numbers into ascending order , like from smallest to largest

```
for i = 0 to n-2 do
begin
    min = i
    for j = i+1 to n-1 do
        if a[j] < a[min] then
            min = j
    swap a[i] and a[min]            # swap the value of in index i and index min
end
```

The time complexity of this algorithm is stable : O(n^2)

2.5 Bubble Sort in a sequence of numbers 冒泡排序

```
for i = 0 to n-2 do
    for j = n-1 downto i+1 do
        if (a[j] < a[j-1]
            swap a[j] & a[j-1]
```

As each time it ensures that the index i is in order , so the algorithm is reasonable.
The time complexity of this algorithm is stable : O(n^2)

2.5 Insertion Sort in a sequence of numbers 插入排序

```
for i = 1 to n-1 do
begin
    tmp = a[i]
    j = 0
    while (a[j] < tmp) && (j < i) do
        j = j + 1
    move the value of a[j], a[j+1] ,..., a[i-1] to index j+1, j+2 ,..., i in order
    a[j] = tmp
end
```

e.g. 9,6,4,8,1,5,3
---- 6,9,4,8,1,5,3
---- 4,6,9,8,1,5,3
---- 4,6,8,9,1,5,3
---- 1,4,6,8,9,5,3
---- 1,4,5,6,8,9,3
---- 1,3,4,5,6,8,9

As each time it ensures that the number from index 0 to i is in order , so the algorithm is reasonable with the i being optimized .
The time complexity of this algorithm is stable : O(n^2)

The time complexity of the fastest comparison-based sorting algorithm: O(nlogn)

2.6 Exponential time algorithms

2.6.1 Traveling Salesman Problem(TSP) There are n cities. Find the shortest tour from a particular city that visit each city exactly once and return to the city where it started. For particular cities v1 , we have a tour with start v1 and the end v1 ,with v2 , v3 ,..,vN in the tour.

Exhaustive search approach: *Find all tours and compute the tour length to find the shortest among them .* 暴力搜索算法 ***O((n-1)!)***

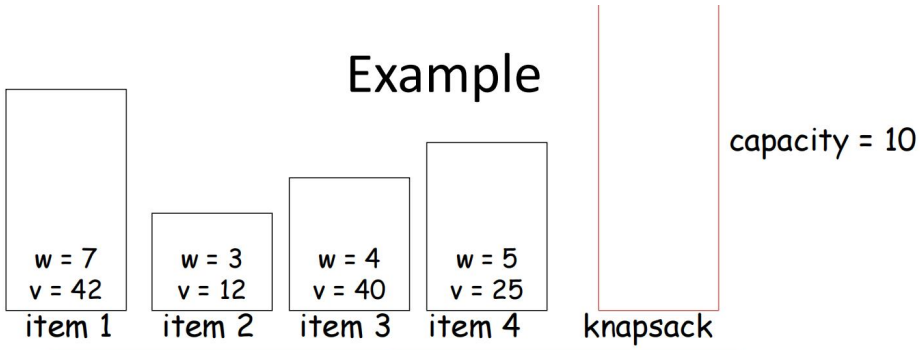
Traveling Salesman Problem（TSP）旅行商问题是：给定一组城市和城市之间的路径及其对应的距离或成本，旅行商需要从某个城市出发，访问每个城市恰好一次，最后返回起始城市。目标是找到访问所有城市的最短路径或最低成本。
TSP 不仅要找到一个访问所有城市的回路，还要使回路的总距离或成本最小。TSP 通常在带权完全图中定义，因为每对城市之间都可能存在直接路径。

Hamiltonian Circuit 汉密尔顿回路是：在一个图中，寻找一条路径，这条路径访问每个顶点恰好一次，并且回到起始顶点。
Hamiltonian Circuit 只需找到一个访问所有顶点的回路，不考虑回路的权重。这个问题可以在任何无向图中定义，不要求图是完全的。

2.6.2 Knapsack Problem Given n items with weights w1, w2, …, wn and values v1, v2, …, vn and a knapsack(背包) with capacity W. Find the most valuable subset of items that can fit into the knapsack.
Application: 运输机在不超过其容量的情况下将最有价值的物品运送到偏远的地点.

Exhaustive search approach: Try every subset of the set of n given items, compute total weight of each subset and compute total value of those subsets that do NOT exceed knapsack's capacity to find the most valuable subset of items. ***O(2^n)***

Example



subset	total weight	total value
ϕ	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1,2}	10	54
{1,3}	11	N/A
{1,4}	12	N/A
{2,3}	7	52
{2,4}	8	37
{3,4}	**9**	**65**
{1,2,3}	14	N/A
{1,2,4}	15	N/A
{1,3,4}	16	N/A
{2,3,4}	12	N/A
{1,2,3,4}	19	N/A

Max_Weight = 20

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

从 $\{I_0, I_1, I_2\}$ 取 Max = 10+4+9 (3+8+9 ≤ 20). I_0, I_1, I_2

但 $\{I_0, I_1, I_2, I_3\}$ 取 则为 I_0, I_2, I_3 Max = 10+9+11 = 30 (3+8+9 ≤ 20)

B[k, w] → max total value of S_k subset with weight w

$$B[k,w] = \begin{cases} B[k-1, w], & W_k > W \\ \max\{B[k-1,w], B[k-1, W-W_k]+Value_k\}, & else \end{cases}$$

Another Example

Max_Weight = 5

Item	Weight	Value
I_0	2	3
I_1	3	4
I_2	4	5
I_3	5	6

⇒

i/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Max Value

How to get items? I_1, I_2

$B[2,5] = \max\{B[1,5], B[1,2]+4\}$

$= 7$

$B[2,4] = \max\{B[1,4], B[1,1]+4\} = 4$

$B[3,4] = 5$

$\max\{B[2,4], B[2,0]+5\}$ (4, 0)

O(n * Max_Weight) Compare exaustable $O(2^n)$

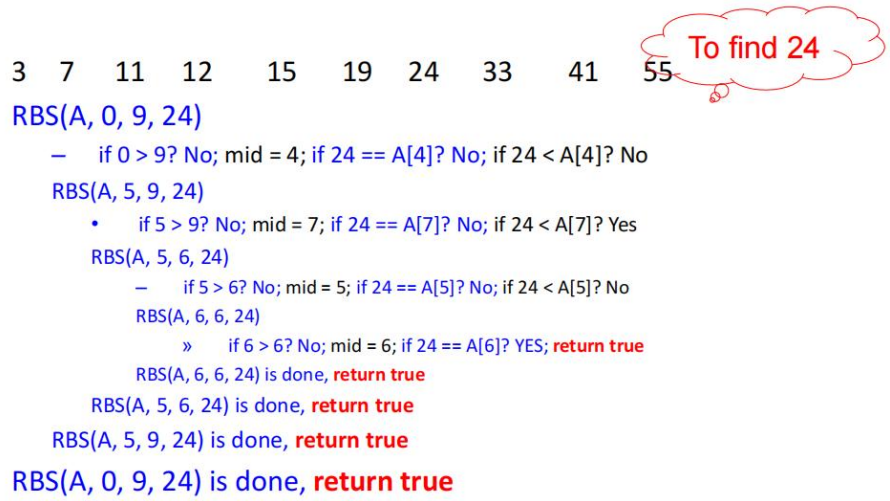
3 Divide and Conquer

Divide and Conquer is one of the best-known algorithm design techniques: A problem instance is divided into several smaller instances of the same problem, ideally of about same size. The smaller instances are solved, typically recursively. The solutions for the smaller instances are combined to get a solution to the original problem.

3.1 Recursive Binary Search (RBS) in a sorted sequence of numbers

```
RecurBinarySearch(A, first, last, X)
begin
    if (first > last) then
        return false
    mid = (first+last) \ 2          #   \ get the int number from lower bound
    if (X == A[mid]) then
        return true
    if (X < A[mid]) then
        return RecurBinarySearch(A, first, mid-1, X)
    else
        return RecurBinarySearch(A, mid+1, last, X)
end
```

Example:



Let T(n) denote the numbers of comparison operations on n numbers. We call this formula a ***recurrence.***

$$T(n) = \begin{cases} 1\ , n = 1 \\ T(n/2)\ + 1\ , n\ > 1 \end{cases}$$

The time complexity of this algorithm is unstable : O(1)~ O(logn)

If T(n) = 2T(n/2)+1 , then the time complexity is O(n).

$$T(n) = 2^k T(n/2^k) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1$$

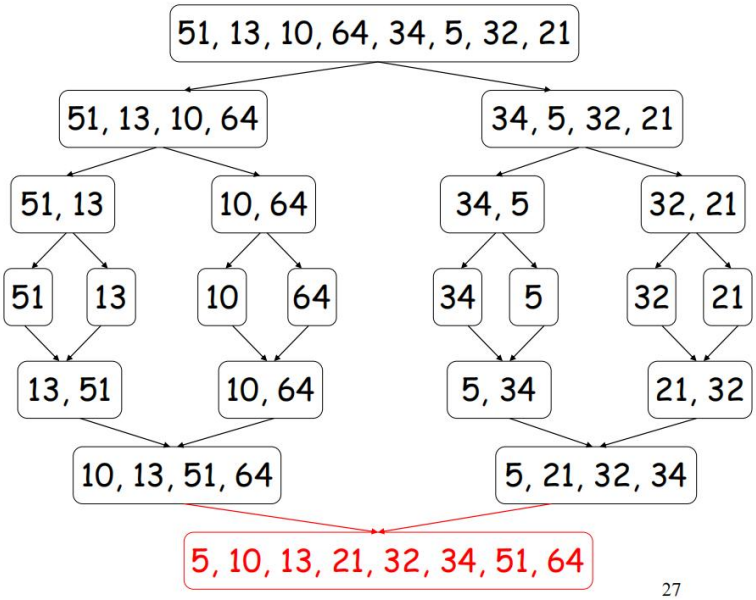
*Let n = 2^k , then T(n) = 2*2^k - 1 =2n -1 =O(n)*

If T(n) = 2T(n/2)+n , then the time complexity is O(nlogn).

*T(n) = 2^k*T(n/2^k) + kn = n + nlogn = O(nlogn)*

All of these can be proved by mathematical induction with Big-O notation.

3.2 Merge Sort is a sorting algorithm based on divide and conquer technique. It divides the sequence of n numbers into two halves and recursively sort the two halves. Merge the two sort halves into a single sorted sequence.



27

Algorithm 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
begin
    if B[i]<=C[j] then
        A[k]=B[i]
        i = i +1
    else
        A[k] = C[j]
        j = j + 1
    k = k+1
end
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]
```

Algorithm Mergesort(A[0..n-1])

```
if n > 1 then
begin
    copy  A[0..  n\2]  to  B[0..  n\2]
    copy  A[n\2..n-1]  to  C[0..  n-1-n\2]
    Mergesort(B[0..  n\2])
    Mergesort(C[0..  n-1-n\2])
    Merge(B, C, A)
end
```

Let T(n) denote the numbers of comparison operations on n numbers.

$$T(n) = \begin{cases} 1\ , n = 1 \\ 2\ T(n/2)\ + n\ , n\ > 1 \end{cases}$$

Understanding why T(n) $= 2\ T(n/2)\ + n$. We do Mergesort function twice so we have $\underline{2\ T(n/2)}$, in merge function we can consider it as make sure one position in one operations so we need $\underline{n}$ operations to make sure n position(while n-1 is also okay) .

<u>The time complexity of this algorithm is stable : O(nlogn)</u>

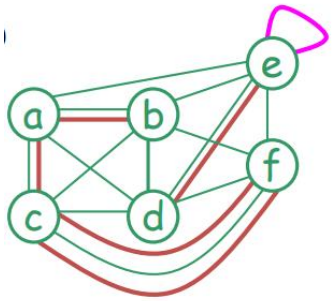
4 Graph Theory

*An **undirected** graph G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an **unordered** pair of vertices. (E.g: {b,c} & {c,b} refer to the same edge)*

*A **directed graph** G=(V,E) consists of a set of vertices V and a set of edges E. Each edge is an **ordered** pair of vertices. (E.g: (b,c) refer to an edge from b to c.)*

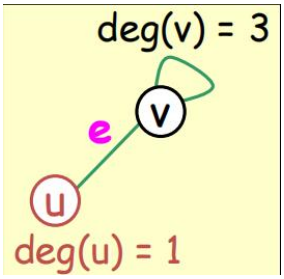
4.1 Undirected graphs

Undirected graphs Type:
*---- simple graph: at most one edge between two vertices, no self loop (i.e., an edge from a vertex to itself) **Green***
*---- multigraph: allows more than one edge between two vertices **Green + Red***
*---- pseudograph: allows a self loop **Green + Purple***

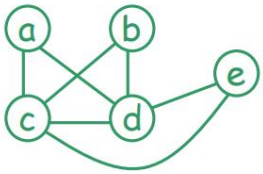


In an undirected graph G, suppose that e = {u, v} is an edge of G
*---- u and v are said to be **adjacent**(相邻的) and called **neighbors** of each other.*
*---- u and v are called **endpoints** of e.*
*---- e is said to be **incident** (相关联) with u and v.*
---- e is said to connect u and v.

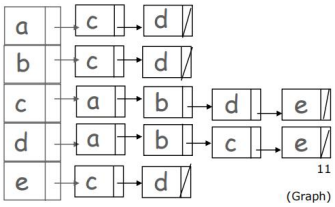
***The degree of a vertex v**, denoted by **deg(v)**, is the number of edges incident with it (a loop contributes twice to the degree)*



***Adjacency matrix** M for a simple undirected graph with n vertices:*
---- M is an n x n matrix
---- M(i, j) = 1 if vertex i and vertex j are adjacent
***Adjacency list**: each vertex has a list of vertices to which it is adjacent.*



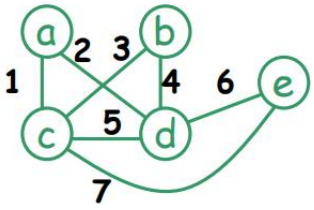
	a	b	c	d	e
a	0	0	1	1	0
b	0	0	1	1	0
c	1	1	0	1	1
d	1	1	1	0	1
e	0	0	1	1	0



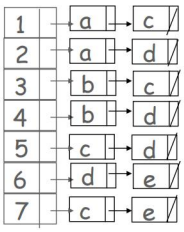
Example 1 (a) Adjacency matrix (b) Adjacency list

Incidence matrix *M for a simple undirected graph with n vertices and m edges:*
---- M is an m x n matrix
---- M(i, j) = 1 if edge i and vertex j are incidence

Incidence list: *each edge has a list of vertices to which it is incident with*



	a	b	c	d	e
1	1	0	1	0	0
2	1	0	0	1	0
3	0	1	1	0	0
4	0	1	0	1	0
5	0	0	1	1	0
6	0	0	0	1	1
7	0	0	1	0	1



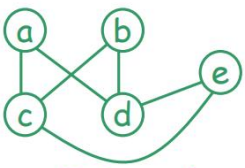
Example 2 (a) Incidence matrix (b) Incidence list

4.2 Euler circuit and Euler path

*---- In an undirected graph, a **path** from a vertex u to a vertex v is a sequence of edges e1= {u, x1}, e2= {x1, x2}, …en= {xn-1, v}, where n≥1. The length of this path is n. If u = v, this path is called a **circuit**.*

A simple circuit *visits an edge at most once, which means there does not exist a->b->c->b phenomenon.* ***The Euler circuit*** *requires travelling all edges in the graph and each edge can only be visited once , and then return to the starting point. In this process, vertices can be visited repeatedly.*

Does every graph has an Euler circuit ? No!



no Euler circuit

4.3 How to determine whether there is an Euler circuit in a graph?

*An undirected graph G is said to be **connected** if there is a **path** between every pair of vertices. Even if the graph is **connected**, there may be no Euler circuit either. (Upper figure) If G is **not connected**, there is no single circuit to visit all edges or vertices.*

***Theorem 1：** G is a **connected graph**, then it contains an Euler circuit if and only if every vertex has even degree.*

*Let G be an undirected graph. An **Euler path** requires travelling all edges in the graph and each edge can only be visited once.*

Theorem 2：** An undirected graph contains an Euler path if it is **connected** and contains exactly **two** vertices of **odd degree** or **every vertex has even degree.

如果是一个完整连接的无向图，是否存在所有节点都是偶数但是欧拉路径起点和最终点不同情况？不存在，首先我们需要为什么恰好俩个就存在欧拉路径，相当于从欧拉回路删除一条线，导致起点和终点不同，现在对与起始点不管后面回来几次都会出去相同次数，最后不返回那么他这个点上的 degree 就是奇数不是偶数所有不存在这种情况。

	Exist Euler circuit?	Exist Euler path?
all vertices have even degree	YES	YES
exactly two vertices have odd degree	NO	YES
more than 2 vertices have odd degree	NO	NO

4.4 Hamiltonian circuit and Hamiltonian path

*Let G be an undirected graph. **A Hamiltonian circuit** is a circuit containing every **vertex** of G exactly once. **A Hamiltonian path** is a path containing every **vertex** of G exactly once. Note that a Hamiltonian circuit or path may NOT visit all edges.*

Unlike the case of Euler circuits / paths, determining whether a graph contains a Hamiltonian circuit (path) is a very difficult problem. (For a graph with n vertices, there are n! possible permutations of the vertices to check if they form a Hamiltonian circuit or path , NP-hard)

4.5 Breadth-first search (BFS) 广度优先搜索是一种图搜索算法，用于遍历或搜索树或图的数据结构。它从一个起始节点开始，首先访问该节点的所有相邻节点，然后再访问这些相邻节点的相邻节点，以此类推，直到访问完所有节点。

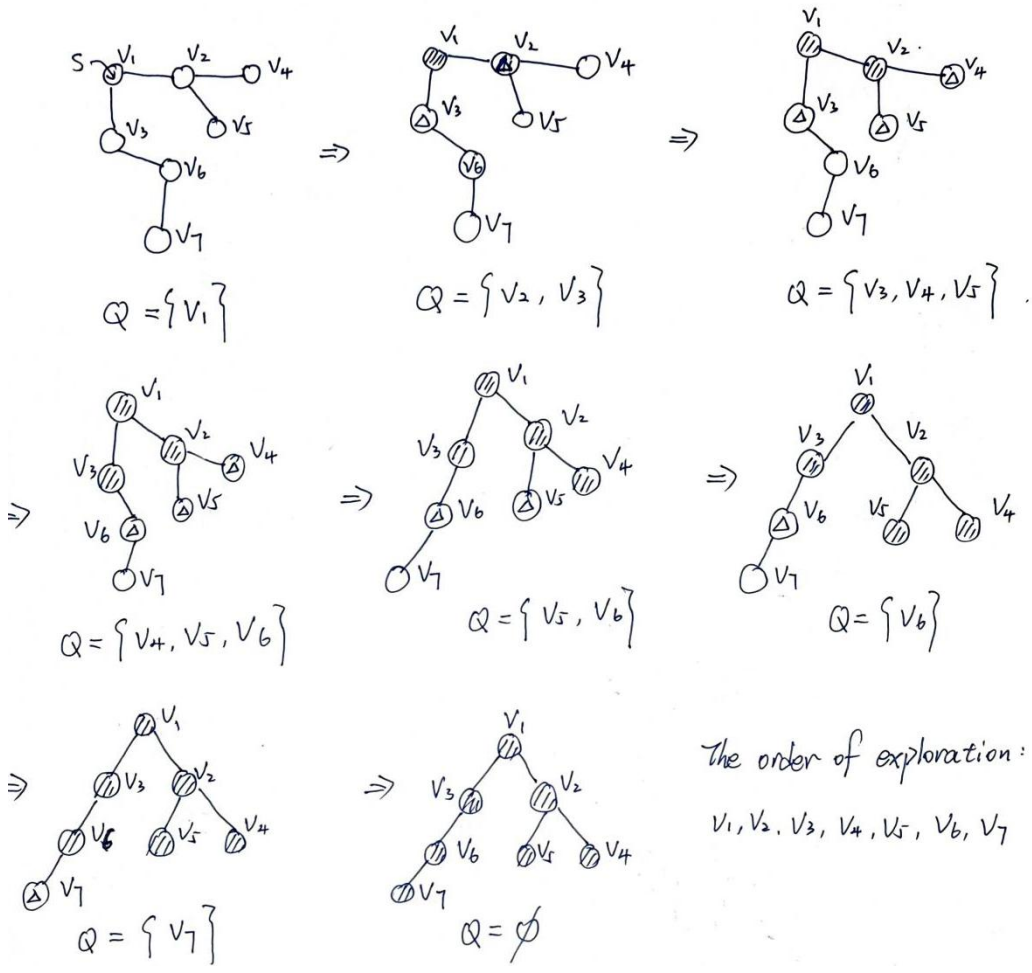
Object : get the order of exploration

```
start point s
for each vertex u in V\s
    color u to white
Q = { }          # Q is a queue
Put s into Q

while Q is not ∅

    remove a vertex from Q
    for each v in Adj(u)
        if v is white then
            Color the v into gray
            Put v into Q
    color u to black
```

Example :

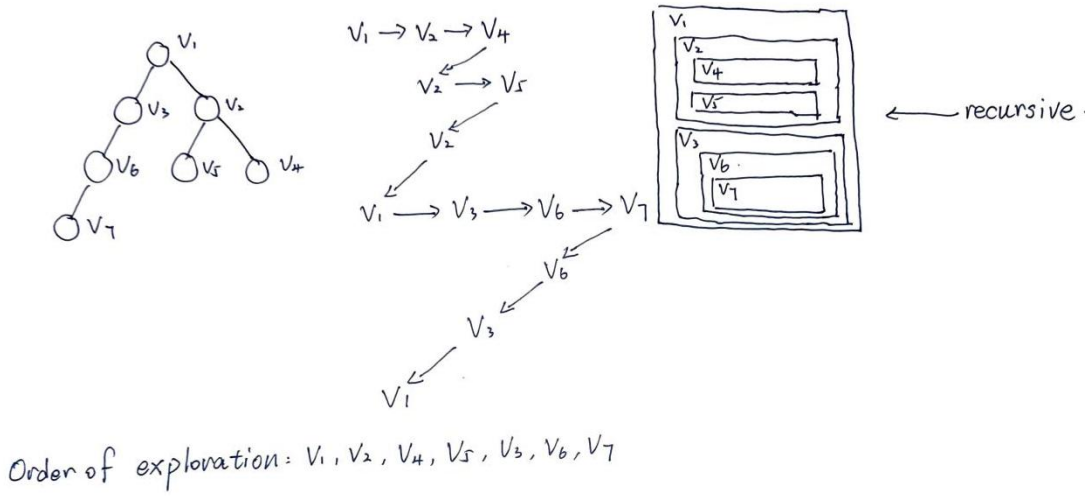


4.6 Depth First Search (DFS)

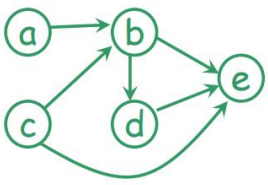
Algorithm DFS(vertex v)

```
visit v
for each unvisited neighbor w of v do
begin
    DFS(w)
end
```

Example :



4.7 Directed graph



E = { (a,b), (b,d), (b,e), (c,b), (c,e), (d,e) }

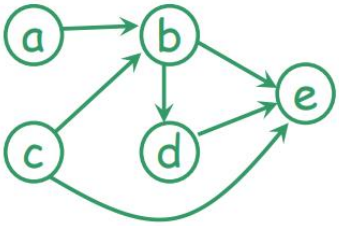
N.B. (a,b) is in E, but (b,a) is NOT

	in-deg(v)	out-deg(v)
a	0	1
b	2	2
c	0	2
d	1	1
e	3	0
sum:	6	6

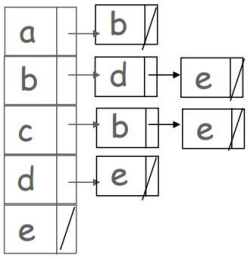
*The **in-degree** of a vertex v is the number of edges leading to the vertex v.*
*The **out-degree** of a vertex v is the number of edges leading away from the vertex v.*
The sum of in-degree and out-degree is always equal.

***Adjacency matrix** M for a directed graph with n vertices:*
---- M is an n x n matrix
---- M(i, j) = 1 if (i,j) is an edge

***Adjacency list:** each vertex u has a list of vertices pointed to by an edge leading away from u.*



	a	b	c	d	e
a	0	1	0	0	0
b	0	0	0	1	1
c	0	1	0	0	1
d	0	0	0	0	1
e	0	0	0	0	0



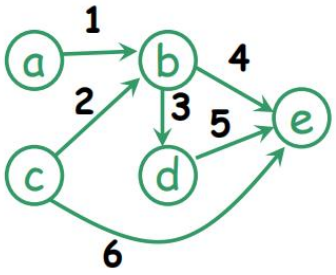
Example 3 (a) Adjacency matrix (b) Adjacency list

Incidence matrix *M for a directed graph with n vertices and m edges is an m x n matrix*

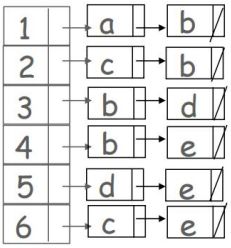
---- M(i, j) = 1 if edge i is leading away from vertex j

---- M(i, j)= -1 if edge i is leading to vertex j

Incidence list: *each edge has a list of two vertices (leading away is 1st and leading to is 2nd)*



	a	b	c	d	e
1	1	-1	0	0	0
2	0	-1	1	0	0
3	0	1	0	-1	0
4	0	1	0	0	-1
5	0	0	0	1	-1
6	0	0	1	0	-1



Example 4 (a) Incidence matrix (b) Incidence list

4.7 Tree An undirected graph G=(V,E) is a tree if G is connected and contains no cycles.

---- There is exactly one path between any two vertices in G

---- G is connected and removal of one edge disconnects G

---- G is acyclic(contains no cycles) and adding one edge creates a cycle

---- G is connected and n = m + 1 (where |V|=n, |E|=m) Reason: every vertex has a branch to its upper vertex except the highest vertex .

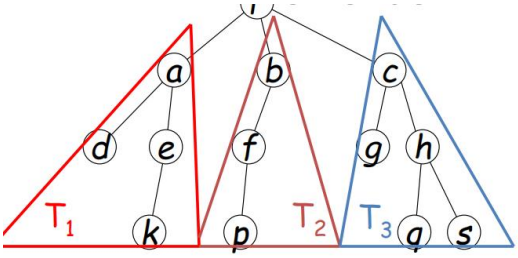
The highest vertex is called the ***root.***

A vertex u may have some ***children*** *directly below it, u is called the* ***parent*** *of its* ***children.***

Degree of a vertex *is the number of children it has.*

Degree of a tree *is the max degree of all vertices.*

A vertex with no child is called a ***leaf.*** *All others are called* ***internal vertices.***



T1, T2, …, Tk are called ***subtrees of T.***

4.8 Binary tree

---- a tree of degree at most TWO

---- the two subtrees are called ***left subtree*** *and* ***right subtree***

Three common ways to traverse 遍历 a binary tree:

---- preorder traversal 前序遍历

```
PreorderTraversal(node):
    if node is not null:
        visit(node)
        PreorderTraversal(node.left)
        PreorderTraversal(node.right)
```

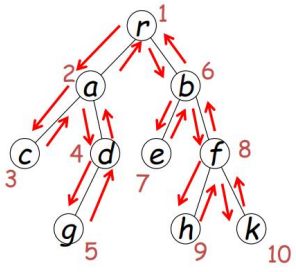
---- inorder traversal 中序遍历

```
InorderTraversal(node):
    if node is not null:
        InorderTraversal(node.left)
        visit(node)
        InorderTraversal(node.right)
```

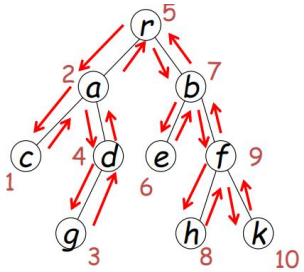
---- postorder traversal 后序遍历

```
PostorderTraversal(node):
    if node is not null:
        PostorderTraversal(node.left)
        PostorderTraversal(node.right)
        visit(node)
```

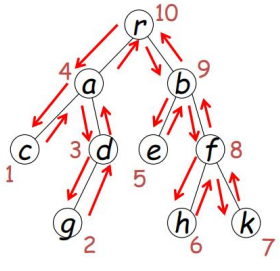
Example 1: 这个红线没用只看遍历顺序



(a) preorder traversal

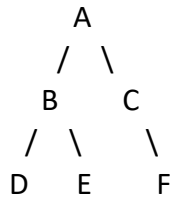


(b) inorder traversal



(c) postorder traversal

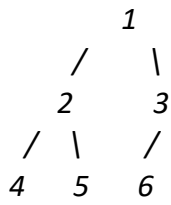
Example 2:



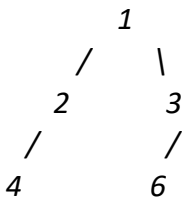
---- Preorder Traversal: A, B, D, E, C, F
---- Inorder Traversal : D, B, E, A, C, F
---- Postorder Traversal: D, E, B, F, C, A

4.9 Heap 堆

---- ***Complete tree*** 完全二叉树*: a binary tree and all layers except the last one are full, the last layer is arranged from left to right without space.*



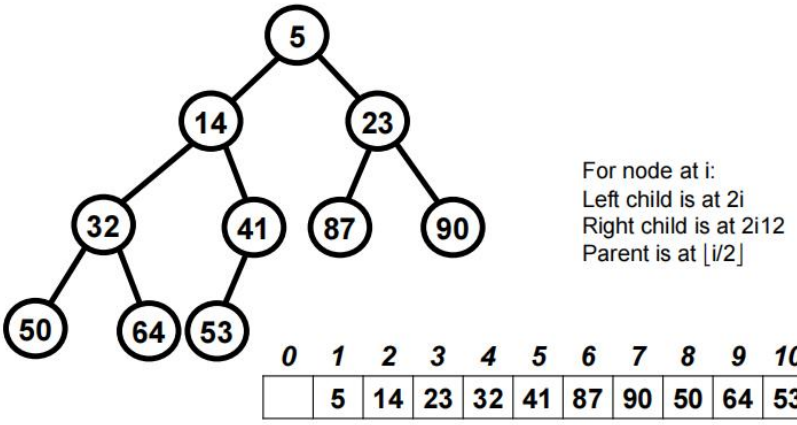
(a) complete tree



(b) not complete tree

---- ***Min-Heap*** 最小堆：每个节点的值都小于或等于其子节点的值并且是完全二叉树
---- ***Max-Heap*** 最大堆：每个节点的值都大于或等于其子节点的值并且是完全二叉树

---- ***Storage of a heap*** *： Use an array to hold the data. Store the root in position 1. We won't use index 0 for this implementation. For any node in position i, its left child (if any) is in position 2i ,its right child (if any) is in position 2i + 1 ,its parent (if any) is in position i\2 (use integer division) .*



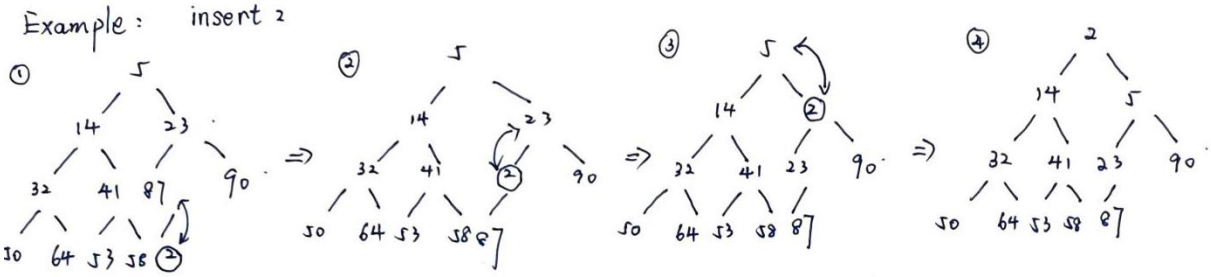
---- 插入新元素到最小堆的算法 ***O(1)~ O(logn)***

1 将新元素放在数组的下一个可用位置：如果当前堆的数组表示为 [- , 1, 3, 2, 7, 6, 4, 5]，插入的新元素是 0，那么将 0 放在数组的下一个位置，结果为 [- , 1, 3, 2, 7, 6, 4, 5, 0]。

2 比较新元素与其父节点的大小：如果新元素比其父节点小，则交换这两个元素。

在数组表示中，对于位置 i 的节点，其父节点的位置是 (i-1)//2。

3 重复这个过程直到满足以下条件之一：新元素的父节点比新元素小或等于新元素或者新元素到达根节点（数组索引 1）。



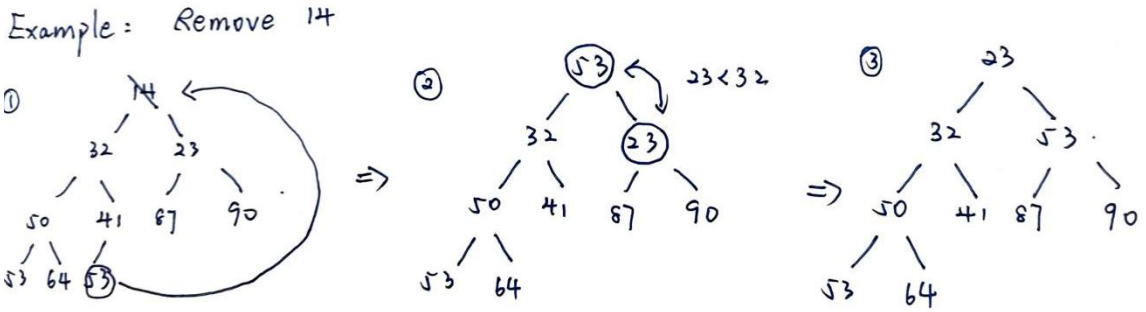
---- 移除元素在最小堆的算法 ***O(1)~ O(logn)***

1 保存根节点的值

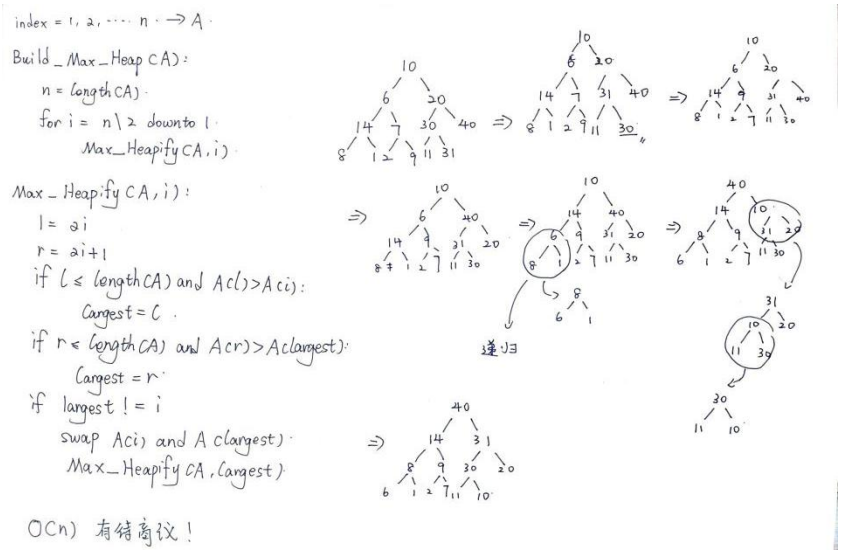
2 将最后一个元素移动到根节点位置

3 比较根节点与其子节点：如果根节点大于其子节点，**选择较小的子节点进行交换。**例如 8 与子节点 3 和 2 比较，选择较小的 2 进行交换。

4 继续调整，直到堆的性质恢复



----构造堆



5 *Greedy Method*

---- At every step, make the best move you can make. Keep going until you're done.
Greedy algorithm does NOT always return the best solution.(Knapsack Problem)

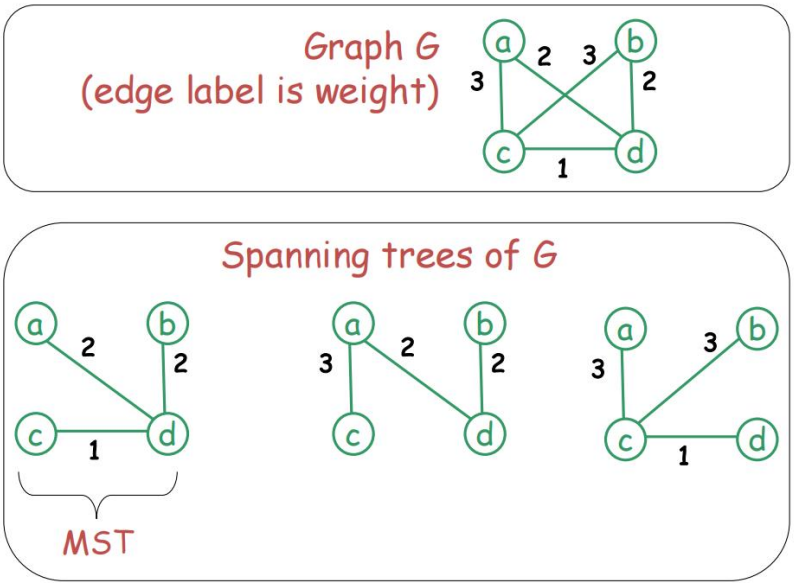
5.1 Minimum Spanning tree (MST) 最小生成树

在一给定的无向图 $G=(V,E)$ 中，(u,v) 代表连接顶点 u 与顶点 v 的边（即 $(u,v)\in E$），而 $w(u,v)$ 代表此边的权重，若存在 T 为 E 的子集（即 $T\subseteq E$）且 (V, T) 为树，使得:

$$w(T)=\sum_{(u,v)\in T} w(u,v)$$

的 w(T) 最小，则此 T 为 G 的最小生成树。

注意这里一定要数据结构是树



为什么 MST 是左边第一幅画？1+2+2=5 最小

5.2 Cut Property 割集性质 *Let G=(V,E) be a connected, undirected graph with real-valued weights on edges. Let A be a subset of E that forms a minimum spanning tree of G, and let S and V-S be two disjoint subsets of V. Let e be the minimum weight edge that has one endpoint in S and the other endpoint in V-S. Then, e is part of the minimum spanning tree A of G.*

我们可以通过反证法来证明这个定理。假设 e 不是最小生成树 A 的一部分，由于 A 是一个最小生成树，它一定连通所有顶点。现在加入边 e 到 A 中，由于 e 一端在 S 中，另一端在 V-S 中，加入 e 会在 A 中形成一个环。在这个环中，必然存在另一条边 e'，其一端点在 S 中，另一端点在 V-S 中。由于 e 是 S 和 V-S 之间权重最小的边，边 e 的权重小于或等于边 e ' 的权重。移除边 e' 并保留边 e，得到的子图仍然是一个连通无环图，并且其权重不大于原生成树,矛盾。

5.3 How to solve Minimum Spanning tree Problems ?

5.3.1 Prim's algorithm

// Given a weighted connected graph G=(V,E)

pick a vertex v0 in V

VT = { v0 }

ET= {}

For i = 1 to |V|-1 do

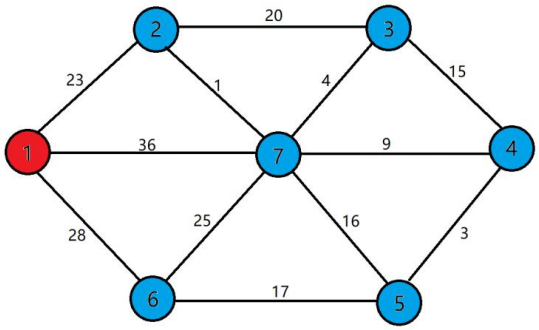
pick an edge e =(v*, u*) with minimum weight among all the edges (v, u) such that v is in VT and u is in V-VT

VT = VT U { u* }

ET = ET U { e*}

Return ET

Example:



VT ={ 1 } , ET= { } i= 1 ~6

I = 1 VT ={ 1, 2 } , ET= { 23 }

I = 2 VT ={ 1, 2, 7 } , ET= { 23, 1 }

I = 3 VT ={ 1, 2, 7 ,3 } , ET= { 23, 1, 4 }

I = 4 VT ={ 1, 2, 7 , 3, 4 } , ET= { 23, 1, 4 , 9 }

I = 5 VT ={ 1, 2, 7 , 3, 4, 5 } , ET= { 23, 1, 4 , 9, 3 }

I = 6 VT ={ 1, 2, 7 , 3, 4, 5, 6 } , ET= { 23, 1, 4 , 9, 3, 17 }

---- Does Prim's algorithm always yield a minimum spanning tree? 是的，Prim 算法总是会生成一个最小生成树. Prim 算法的正确性可以通过割集性质保证。***O(|E|log|V|)***

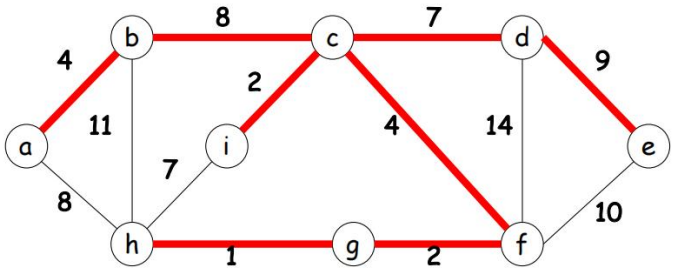
```
MST-PRIM(G, w, root)    //G=(V,E) and a root vertex
1.  for each u in V
2.      do key[u] <- ∞
3.         π[u] <- NIL
4.  key[root] <- 0
5.  Q <- V   //implement by min-heap, O(|V|)
6.  while Q is not empty
7.          do u <- Extract-min(Q)    // O(lg(|V|) by heap
8.             for each v in Adj[u]
9.             do if v in Q and w(u,v) <key[v]
10.                then π[v] <- u
11.                       key[v] = w(u,v)
```

5.3.2 Kruskal's algorithm

// Given an undirected connected graph G=(V,E)
pick an edge e in E with minimum weight
T = { e } and E' = E – { e }
while E' ≠ {} do
begin
    pick an edge e in E' with minimum weight
    if adding e to T does not form cycle then
        T = T U { e }
    E' = E' – { e }
end

Time complexity: O(E log E)

(h,g)	1
(i,c)	2
(g,f)	2
(a,b)	4
(c,f)	4
(c,d)	7
~~(h,i)~~	~~7~~
(b,c)	8
~~(a,h)~~	~~8~~
(d,e)	9
~~(f,e)~~	~~10~~
~~(b,h)~~	~~11~~
~~(d,f)~~	~~14~~



MST is found when all edges are examined

46

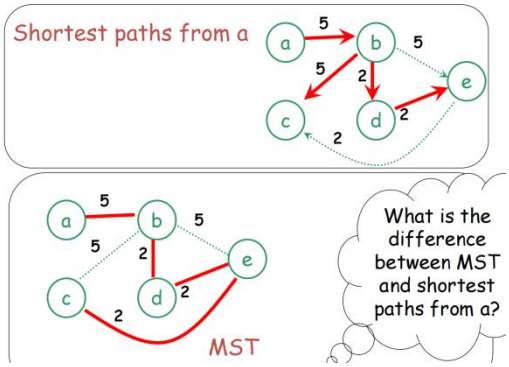
此题不只有一解在 8 的选择上

5.4 How to solve shortest-paths?

Single-source shortest-paths
---- Consider a (un)directed connected graph G with the edges labeled by weight
---- Given a particular vertex called the source to Find shortest paths from the source to all other vertices (shortest path means the total weight of the path is the smallest)
    ----Only nonnegative edge weights: Dijkstra's algorithm
    ----Allow negative edge weights: Bellman-Ford algorithm

All-pairs shortest paths: Find shortest path between each pair of vertices.
---- Floyd's algorithm

5.4.1 Dijkstra's algorithm



Shortest Path : 5 + 10 +7 +7 =29/ MST: 5 + 7 + 9 + 11 = 32 > 29

// Given a graph G=(V,E) and a source vertex s
for every vertex v in the graph do
    set d(v) = +oo and p(v) = null
set d(s) = 0 and VT = {}
while V - VT ≠ {} do *// there is still some vertex left*
begin
    choose the vertex u in V - VT with minimum d(u) *// DISTANCE*
    set VT = VT U { u }
     for every vertex v in V - VT that is a neighbour of u do
        if d(u) + w(u,v) < d(v) then *// a shorter path is found*
            set d(v) = d(u) + w(u,v) and p(v) = u
End

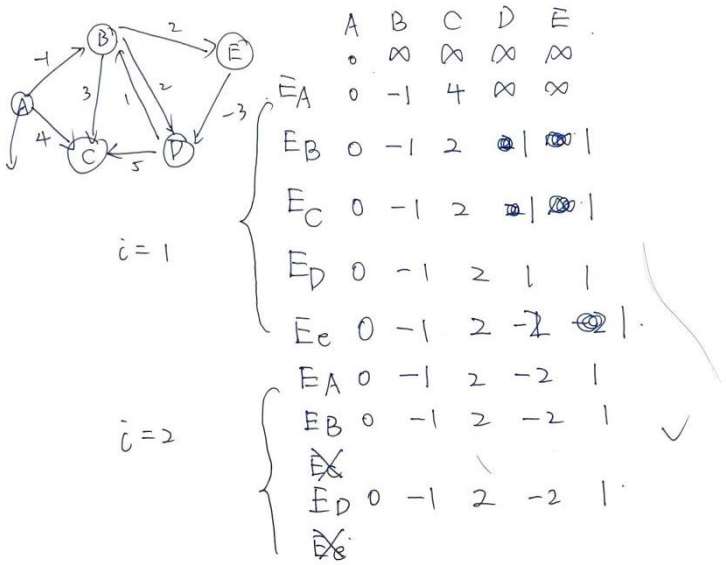
If G is represented by adjacency list and priority queue is implemented by min heap, then the answer is ***O(|E|log|V|)***

5.4.2 Bellman-Ford Algorithm

Algorithm Bellman-Ford(G=(V, E), s)
//input: a graph G=(V,E) with a source vertex s
//output: an array d[0..|V|-1], indexed with V, d[v] is the
//length of shortest path from s to v
    d[s] ← 0
    for each v belongs to V - {s}
        do d[v] ← +oo
    for i ← 1 to |V|- 1
     do for each edge (u, v) belongs to E
        do if d[v] > d[u] + w(u, v)
            then d[v] ← d[u] + w(u, v)
    for each edge (u, v) belongs to E
        do if d[v] > d[u] + w(u, v)
            then report that a negative-weight cycle exists

Dijlstra’s Algorithm does not work with graphs with negative weighted edges. If a graph G = (V, E) contains a negative weight cycle, then some shortest paths may not exist.

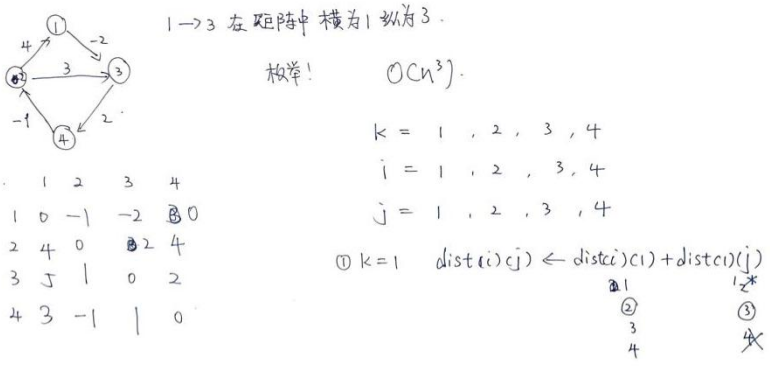
Time complexity: O(VE)



5.4.3 Floyd’s Algorithm

```
let V = number of vertices in graph
let dist = V × V array of minimum distances initialized to ∞
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```

Time efficiency: O(n^3)



5.4.4 *Warshall 's Algorithm* Computes the transitive closure of a relation

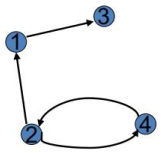
ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: O(n^3)

Rule 1 If an element in row i and column j is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

Rule 2 If an element in row i and column j is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$



$R^{(0)} =$

0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0

$R^{(1)} =$

0	0	1	0
1	0	**1**	1
0	0	0	0
0	1	0	0

$R^{(2)} =$

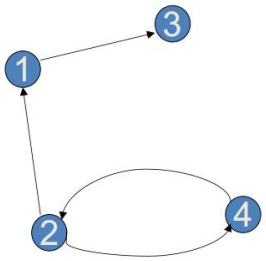
0	0	1	0
1	0	1	1
0	0	0	0
1	1	**1**	**1**

$R^{(3)} =$

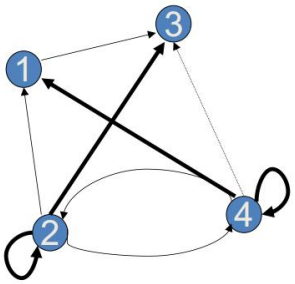
0	0	1	0
1	0	1	1
0	0	0	0
1	1	1	1

$R^{(4)} =$

0	0	1	0
1	**1**	1	1
0	0	0	0
1	1	1	1



0	0	1	0
1	0	0	1
0	0	0	0
0	1	0	0



0	0	1	0
1	1	1	1
0	0	0	0
1	1	1	1

6 Dynamic programming

6.1 Fibonacci numbers

```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

---- Optimization 1:

```
Procedure F(n)
    if (v[n] < 0) then
        v[n] = F(n-1)+F(n-2)
    return v[n]

Main
    set v[0] = v[1] = 1
    for i = 2 to n do
        v[i] = -1
     output F(n)
```

在 F(n) 中,如果 v[n] 已经计算过,则直接返回,避免了重复计算.否则,递归地计算 F(n-1) 和 F(n-2), 并将结果存储在 v[n] 中。

---- Optimization 2:

```
Procedure F(n)
    Set A[0] = A[1] = 1
    for i = 2 to n do
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

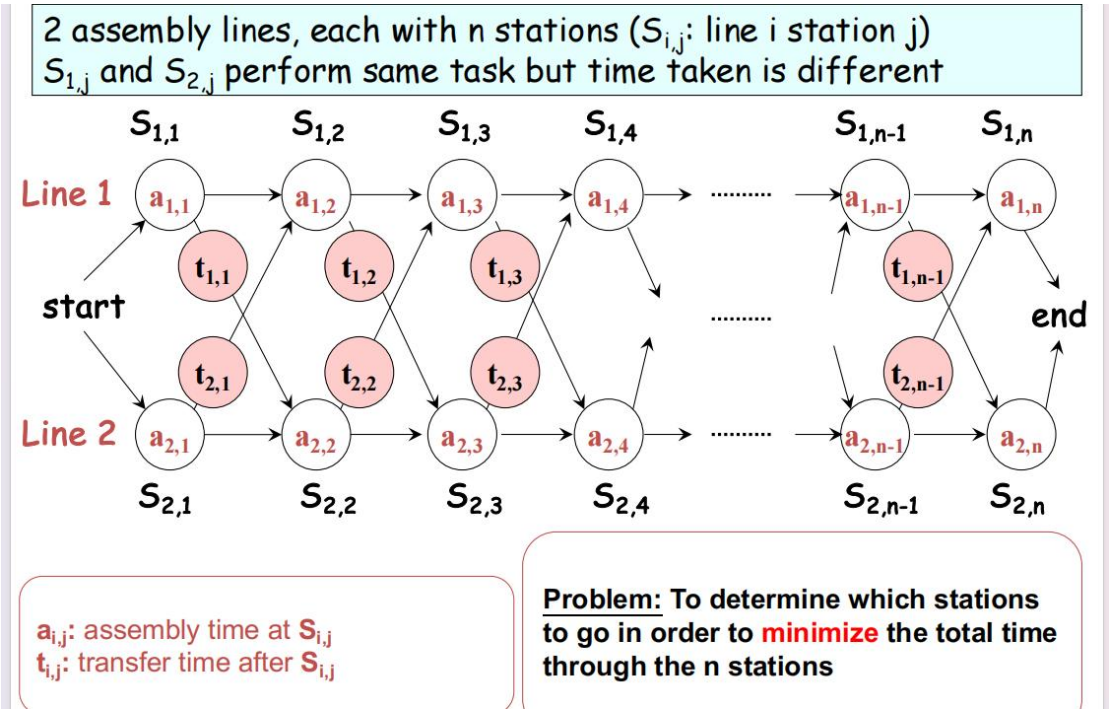
通过使用动态规划方法,每个斐波那契数只计算一次,不需要像原来俩种方法进行递归运算,我们显著减少了计算斐波那契数列的时间复杂度, 从指数级 O(2^n) 降低到线性级 O(n), 大大提高了效率.

6.2 Assembly line scheduling

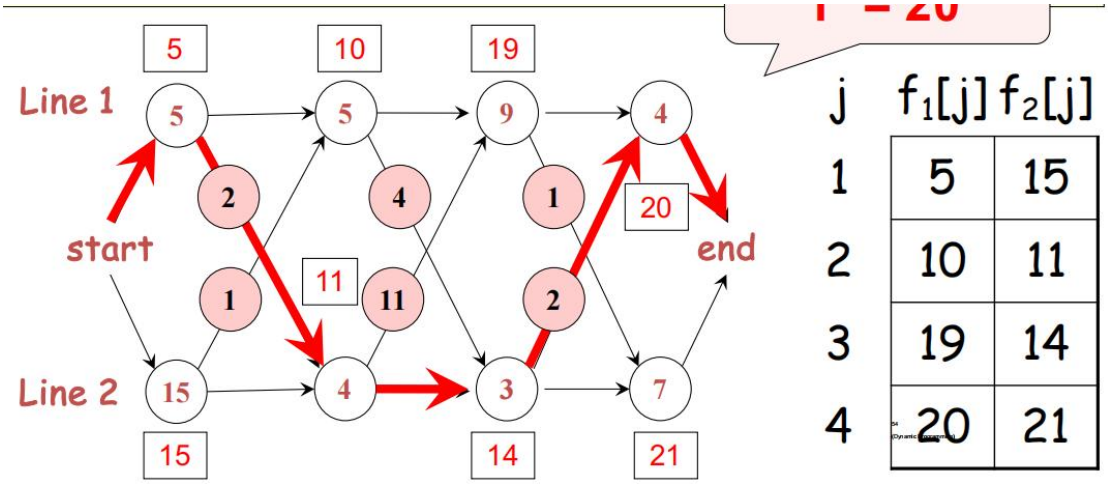
$$f_1[j] = \begin{cases} a_{1,1} & \text{if } j=1, \\ \min\left(f_1[j-1]+a_{1,j} \ , \ f_2[j-1]+t_{2,j-1}+a_{1,j} \right) & \text{if } j>1 \end{cases}$$

$$f_2[j] = \begin{cases} a_{2,1} & \text{if } j=1, \\ \min\left(f_2[j-1]+a_{2,j} \ , \ f_1[j-1]+t_{1,j-1}+a_{2,j} \right) & \text{if } j>1 \end{cases}$$

$$f^* = \min\left(f_1[n] \ , \ f_2[n] \right)$$



Time complexity is O(n).



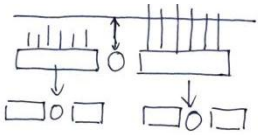
3 or more lines Overall time complexity: O(n*m^2)

7 Space and Time Trade-offs

Two types of space-for-time algorithms: Input-enhancement and Pre-structuring.

7.1 Input enhancement

: preprocess the input (or its part) to store some info to be used later in solving the problem. Two algorithms: Distribution counting sort And Horspool's algorithm for string searching

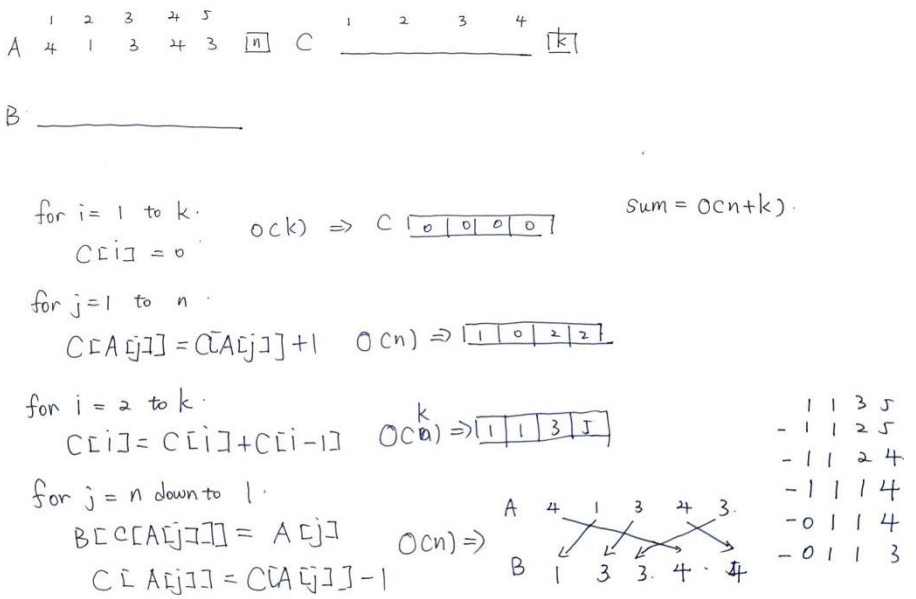
Quick Sort:  choose one (lefts <one< rights) O(n).

$$T(n) = 2T(\frac{n}{2}) + O(n) \quad \text{let } O(n) \le pn$$
$$= 4T(\frac{n}{4}) + 2O(\frac{n}{2}) + O(n)$$
$$\le 4T(\frac{n}{4}) + 2p\frac{n}{2} + pn$$
$$= 4T(\frac{n}{4}) + 2pn$$
$$= 2^k T(\frac{n}{2^k}) + kpn \qquad n = 2^k \Rightarrow k = \log_2 n$$
$$= n \cdot 1 + \log n \cdot p \cdot n \le pn\log n = O(n\log n)$$

*All the sorting algorithms we have seen so far are **comparison sorts** : only use comparisons to determine the relative order of elements like Selection sort, bubble sort, insertion sort, merge sort.The best worst-case running time that we've seen for comparison sorting is **O (n log n).***

---- Is O(nlogn) the best we can do? Yes, as long as we use comparison sorts . In fact, we can prove that any comparison sorting takes at least O(nlogn) time in the worst case.

7.1.1 Distribution counting sort



No comparisons between elements.
---- Input: A [1 . . n], where A [j] ∈ {1, 2, …, k }.
---- Output: B [1 . . n], sorted.
---- Auxiliary storage 辅助存储器: C [1 . . k].

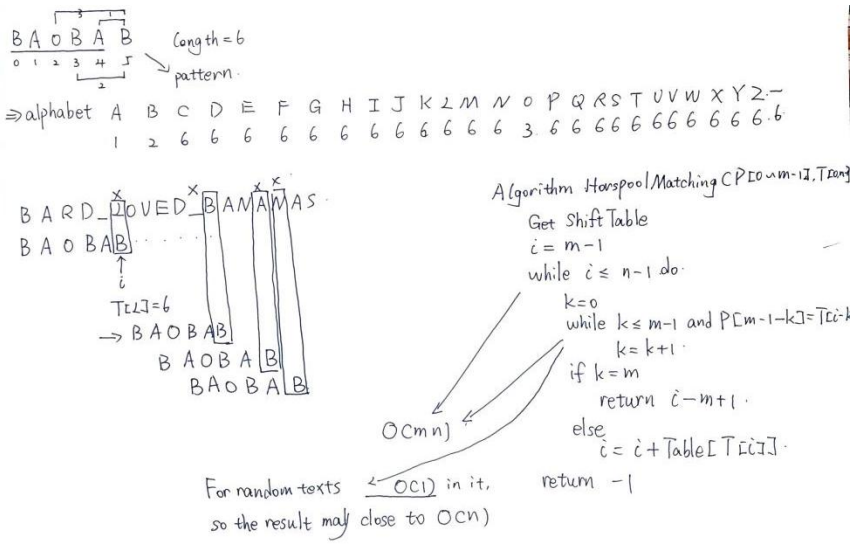
In the distributed computing setting, the counting sort takes O(n) time. Counting sort is not a comparison sort.

7.1.2 Horspool's Algorithm

*---- preprocesses pattern to generate a **shift table** that determines how far to shift the pattern when a mismatch occurs*
---- always makes a shift based on the text's character c aligned with the last character in the pattern according to the shift table's entry for c

Shift table *: For a pattern P[0..m-1], shift size s(c) of a letter c in the text can be precomputed as following:*
---- if c is in P[0..m-2] , s(c) =The number of characters from c 's rightmost occurrence in P[0..m-2] to the right end of the pattern P[0..m-1]
---- if c is not in P[0..m-2] s(c) =the length m of the pattern P[0..m-1] otherwise

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$
```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//         or −1 if there are no matches
ShiftTable(P[0..m − 1])   //generate Table of shifts
i ← m − 1                 //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                 //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```



8 Sequence Alignment by Dynamic Programming

8.1 Longest Common Subsequence (LCS)

---- Subsequence of X is X with some symbols left out. Z=CGTC is a subsequence of X=ACGCTAC.
---- Common subsequence Z of X and Y: a subsequence of X and also a subsequence of Y. Z=CGA is a common subsequence of both X=ACGCTAC and Y=CTGACA. 共同子序列的概念：它是两个字符串中都存在的序列，但不一定要求在两个字符串中是连续的。
---- Longest Common Subsequence (LCS): the longest one of common subsequences. Z' =CGCA is the LCS of the above X and Y.
---- LCS problem: given X=<x1, x2,…, xm> and Y=<y1, y2,…, yn>, find their LCS.
动态规划方法
为了高效地解决 LCS 问题，可以使用动态规划（DP），它将问题分解为更简单的子问题并存储它们的结果，以避免重复计算。

1. **最优子结构:**
   - 设 $X = \langle x_1, x_2, \ldots, x_m \rangle$ 和 $Y = \langle y_1, y_2, \ldots, y_n \rangle$。
   - 如果 $x_m = y_n$，则 X 和 Y 的 LCS 包含这个匹配的元素以及剩余元素 X_{m-1} 和 Y_{n-1} 的 LCS。
   - 如果 $x_m \neq y_n$，则 LCS 是 X_{m-1} 和 Y 或 X 和 Y_{n-1} 的 LCS 中较长的一个。
2. **递归解法:**
   - 定义 $c[i,j]$ 为 X 的前 i 个字符与 Y 的前 j 个字符的 LCS 的长度。
   - 递归公式为:

$$c[i,j] = \begin{cases} 0 & \text{如果 } i = 0 \text{ 或 } j = 0 \\ c[i-1,j-1] + 1 & \text{如果 } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]) & \text{如果 } x_i \neq y_j \end{cases}$$

3. **计算 LCS 的长度:**
   - 使用矩阵 $c[0..m, 0..n]$ 存储子问题的 LCS 长度。
   - 初始化矩阵的基值为零（当任一字符串为空时）。

箭头的标记方法

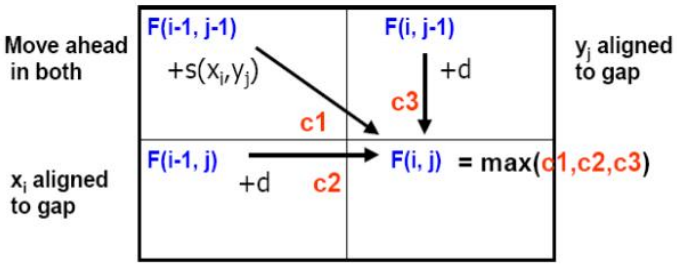
1. **对角线箭头（↖）：**
   - 当 $x_i = y_j$ 时，从 $c[i,j]$ 指向 $c[i-1,j-1]$。
   - 箭头表示字符匹配，并且当前 LCS 长度增加 1。
2. **向上箭头（↑）：**
   - 当 $x_i \neq y_j$ 且 $c[i-1,j] \geq c[i,j-1]$ 时，从 $c[i,j]$ 指向 $c[i-1,j]$。
   - 箭头表示字符 x_i 不在 LCS 中，所以考虑前一个字符。
3. **向左箭头（←）：**
   - 当 $x_i \neq y_j$ 且 $c[i,j-1] > c[i-1,j]$ 时，从 $c[i,j]$ 指向 $c[i,j-1]$。
   - 箭头表示字符 y_j 不在 LCS 中，所以考虑前一个字符。

		a	a	b
	0	0	0	0
a	0	1 ↖	1 ↖	1 ←
z	0	1 ↑	1 ↑	1 ↑
b	0	1 ↑	1 ↑	2 ↖

8.2 Pairwise Sequence Alignment Problem

DP (Global Alignment)

- Algorithm
  - Initialize: F(0,0) = 0, F(i,0) = i × d, F(0, j) = j × d *(gap penalties)*
  - Fill from top left to bottom right using recursive formula



While building the table, keep track of where optimal score came from, then reverse arrows

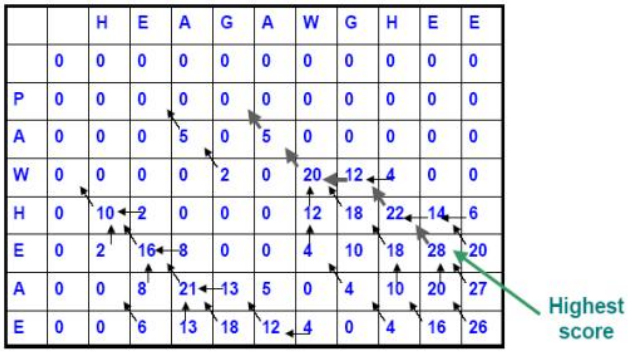
$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) & \text{c1} \\ F(i-1,j) + d & \text{c2} \\ F(i,j-1) + d & \text{c3} \end{cases}$$

DP (Local Alignment)

- Make 0 minimal score (i.e., start new alignment)
- Alignment can start / end anywhere
  - Start at highest score(s)
  - End when 0 reached

$$F(i,j) = \max \begin{cases} F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) + d \\ F(i,j-1) + d \\ 0 \leftarrow \end{cases}$$

- Traceback
  - Start at highest score and trace arrows back to first 0



		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	5	0	0	0	0	0
W	0	0	0	0	2	0	20	12	4	0	0
H	0	10	2	0	0	0	12	18	22	14	6
E	0	2	16	8	0	0	4	10	18	28	20
A	0	0	8	21	13	5	0	4	10	20	27
E	0	0	6	13	18	12	4	0	4	16	26

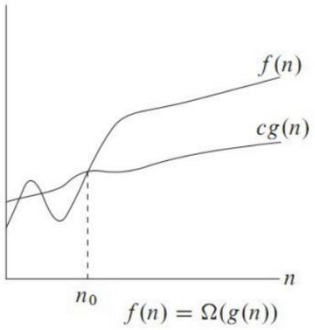
Optimal local alignment

```
AWGHE
AW-HE
```

Omega Notation

Big oh provides an asymptotic upper bound on a function.
Omega provides an asymptotic lower bound on a function.

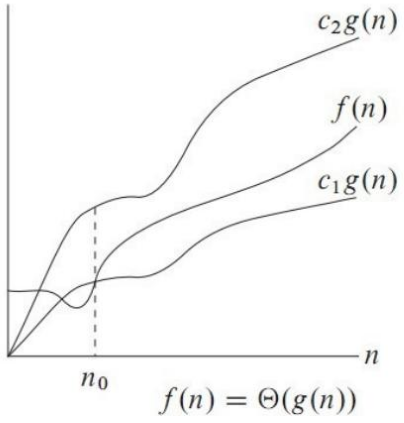
$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



Theta Notation

Theta notation is used when function *f* can be bounded both from above and below by the same function *g*

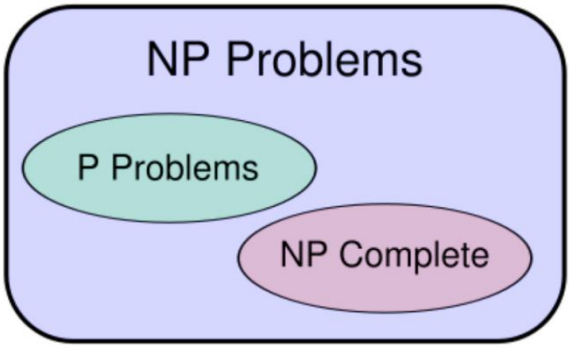
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]



4. 难计算问题

- **P 类问题**：能在多项式时间内解决的问题。
- **NP 类问题**：验证一个给定解是否正确能在多项式时间内完成的问题。
- **NP 完全问题**：如果能在多项式时间内解决一个 NP 完全问题，那么所有 NP 问题都能在多项式时间内解决。
- **NP 硬问题**：比 NP 问题更难，不能保证在多项式时间内解决。

1. **P 类问题**：能在多项式时间内解决的问题（计算和验证都在多项式时间内完成）。
2. **NP 类问题**：验证解在多项式时间内完成的问题（找到解可能需要非多项式时间）。
3. **NP 完全问题**：NP 类中最难的问题，所有 NP 问题都可以在多项式时间内规约到这个问题上。
4. **NP 硬问题**：至少与 NP 类问题一样难的问题，所有 NP 问题都可以规约到这个问题上，但这些问题不一定属于 NP 类（验证解可能不在多项式时间内完成）。



6. 经典 NP 问题示例

- **哈密顿回路问题（Hamiltonian Circuit）**：
  - **输入**：一个连通图 G。
  - **问题**：是否存在一条通过每个顶点且仅通过一次的回路。
- **0/1背包问题**：
  - **输入**：n 个物品，每个物品有整数重量和价值，背包的容量为 W。
  - **问题**：选择一部分物品使得总重量不超过 W 且总价值最大。
- **电路可满足性问题（Circuit-SAT）**：
  - **输入**：一个布尔电路。
  - **问题**：是否存在一组输入值使得电路输出值为 1。

Can All Decision Problems Be Solved By Algorithms?

- The Answer is No.
- The problems can not solved by algorithms is called undecidable problems.
- One such a problem is Halting Problem (AlanTuring 1936)

> "Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it."

COMP108

Solving/Verifying a problem

Solving a problem is different from verifying a problem

- solving: we are given an input, and then we have to FIND the solution
- verifying: in addition to the input, we are given a "certificate" and we verify whether the certificate is indeed a solution

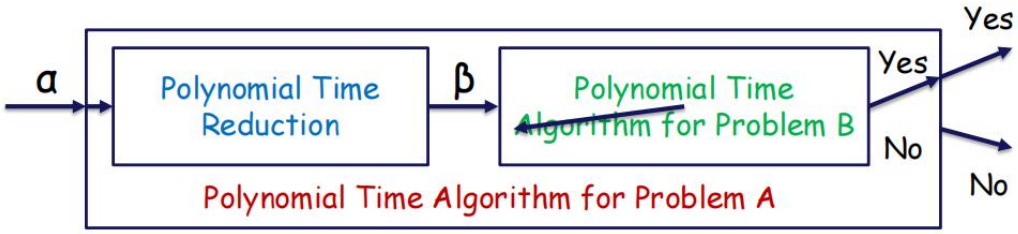
We may not know how to solve a problem efficiently, but we may know how to verify whether a candidate is actually a solution

P = NP?

问题背景：P 是 NP 的一个子集，但我们不确定 P 是否等于 NP。即，是否存在一个问题在 NP 中但不在 P 中。

多项式时间归约（Polynomial-time reduction）

- **定义**：给定两个决策问题 A 和 B，如果存在一个多项式时间归约从 A 到 B，即对于任何 A 的输入 a，我们可以在多项式时间内构造一个 B 的输入 b，使得 a 是 "是" 当且仅当 b 是 "是"，则 A 是多项式时间可归约到 B。
- **记号**：$A \leq_P B$
- **直观意义**：这意味着问题 B 至少和问题 A 一样难。



If problem A is **reducible** to problem B in polynomial time, then which problem is easier?

A ✓ Or B ?

Problem B is at least **as hard as** Problem A!

COMP108

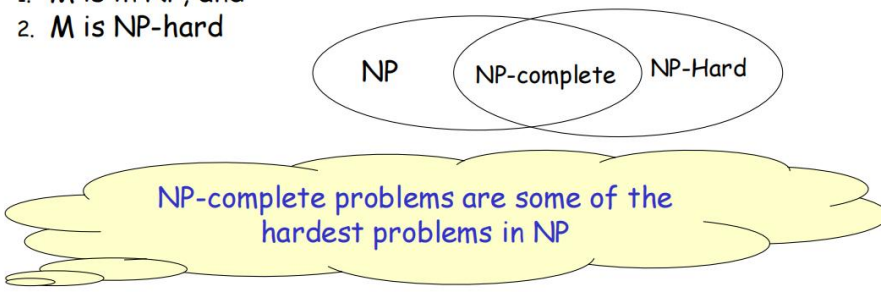
NP-hardness

A problem M is said to be **NP-hard** if every other problem in NP is polynomial time reducible to M

- intuitively, this means that M is at least as difficult as all problems in NP

M is further said to be NP-complete if

1. M is in NP, and
2. M is NP-hard



Problem A is NP-complete if

1.Problem A is **in NP**

2.For **any** Problem A' in NP, A' is **reducible** to A in polynomial time

.Class NPC: The class of all NP-complete problems, which is a subclass of NP

- The **hardest** problems in NP
- Solve **a** problem in NPC, you can solve **ALL** problems in NP

Proof of NP-Completeness

Given a Problem A, prove that A is NP-complete

Proof Scheme 1

- Show Problem A is in NP (easier part)
- For all Problems in NP, reduce them to A in **polynomial time**
  - ❖ This has been done for 3SAT, the first NP-complete problem

Proof Scheme 2

- Show Problem A is in NP (easier part)
- For arbitrary problem A' in NPC, reduce A' to A in **polynomial time**
  - ❖ This would be much easier

经典 NP 完全问题

- **Cook-Levin 定理**：电路可满足性问题（Circuit-SAT）是第一个被证明为 NP 完全的问题。
- **其他 NP 完全问题**：
  - 哈密顿回路问题（Hamiltonian Circuit）
  - 0/1 背包问题
  - CNF-SAT 和 3-SAT（合取范式可满足性问题）
  - 顶点覆盖问题（Vertex Cover）

CNF-SAT and 3-SAT

CNF-SAT

- Input: a Boolean formula in CNF
- Question: Is there an assignment of Boolean values to its variables so that the formula evaluates to true? (i.e., the formula is satisfiable)

3-SAT

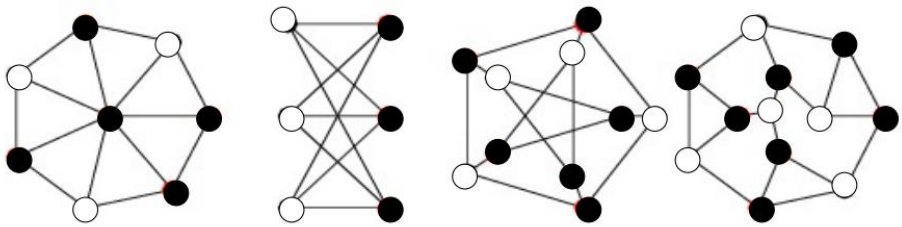
- Input: a Boolean formula in CNF in which each clause has exactly 3 literals

CNF-SAT and 3-SAT are NP-complete

Vertex Cover

Given a graph G = (V,E)

A **vertex cover** is a subset $C \subseteq V$ such that for every edge (v,w) in E, **$v \in C$ or $w \in C$**



some graphs and their vertex cover (shaded vertices)

Vertex Cover

The optimisation problem is to find as **small** a vertex cover as possible

Vertex Cover is the **decision** problem that takes a graph **G** and an integer **k** and asks whether there is a vertex cover for G containing **at most** k vertices

Vertex Cover is NP-complete

10 Coping with the Limitations of Algorithm Power

在处理困难的组合问题（例如 NP 难问题）时，有两种主要的方法：

1. 使用能够保证准确解决问题的策略，但不保证在多项式时间内找到解。
2. 使用近似算法，在多项式时间内找到一个近似（次优）解。

精确解决策略

1. 穷举搜索（暴力法）

- **特点**：遍历所有可能的解。
- **应用**：仅适用于规模较小的问题实例。

2. 回溯法

- **特点**：通过消除一些不必要的情况来减少搜索空间。
- **应用**：对许多实例在合理时间内求解，但最坏情况下仍然是指数时间复杂度。

3. 分支定界法（Branch-and-Bound）

- **特点**：进一步优化回溯法，适用于优化问题。
- **应用**：计算状态空间树中每个节点的目标函数的边界值，使用边界值来裁剪非优解的分支。

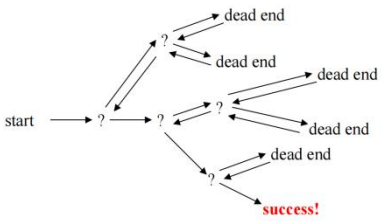
4. 动态规划（Dynamic Programming）

- **特点**：存储子问题的解以避免重复计算。
- **应用**：适用于一些特定问题，例如背包问题。

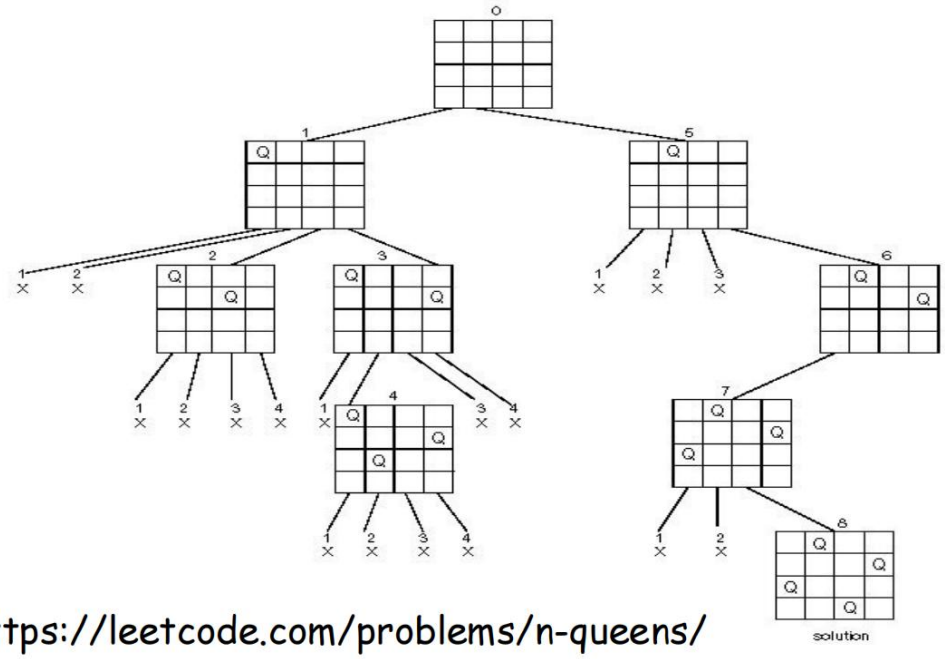
回溯法

- **构建状态空间树**：树的节点表示部分解，边表示扩展部分解的选择。
- **使用深度优先搜索探索状态空间树。**
- **剪枝**：停止探索不能导致解的节点的子树，并回溯到该节点的父节点继续搜索。

Backtracking (animation)



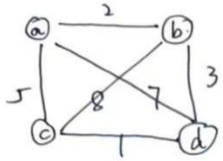
State-Space Tree of the 4-Queens Problem



https://leetcode.com/problems/n-queens/

The traveling salesperson optimization problem

- Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimal cost.
- It is NP-complete
- We try to avoid n! exhaustive search by the branch-and-bound technique.

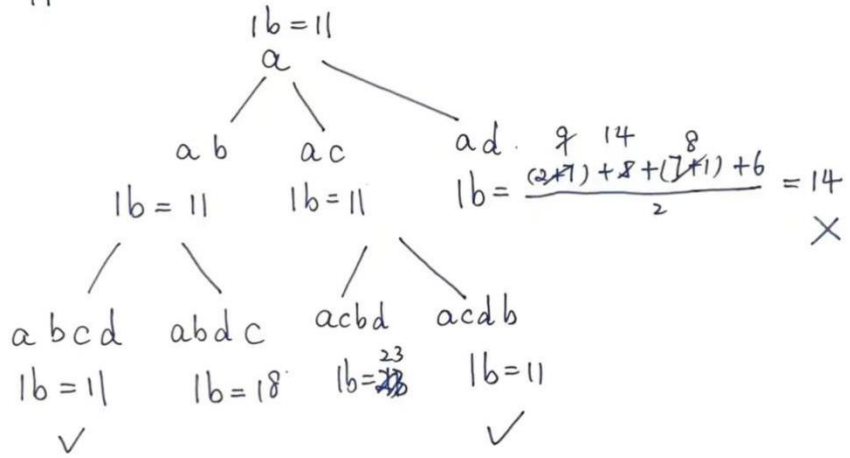


TSP $a \to \cdots \to a$

lb = 每一个点进+出理论最小距离

$= \lceil (2+5) + (2+3) + (3+1) + (5+1) \rceil / 2$ → 进出被算2遍.

$= 11$



11 Approximation Algorithms

2. NP 完全性

- **动机**：对于 NP 完全问题，我们假设它们没有多项式时间算法。
- **问题**：我们能否找到有效的近似算法?

3. 应对 NP 难度的方法

- **暴力算法**：
  - 发展巧妙的枚举策略。
  - 保证找到最优解，但不保证运行时间。
- **启发式算法 (Heuristics)**：
  - 发展直观的算法。
  - 保证在多项式时间内运行，但不保证解的质量。
- **近似算法 (Approximation Algorithms)**：
  - 保证在多项式时间内运行。
  - 保证找到 "高质量" 的解，例如在最优解的 1% 以内。
  - 挑战：在不知道最优解的情况下证明解的质量接近最优。

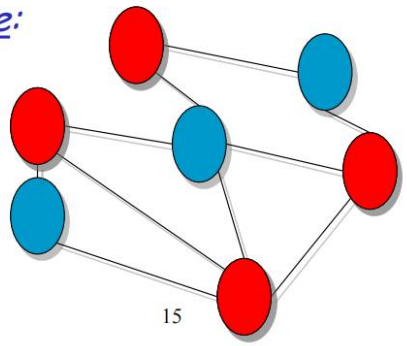
4. 近似算法的定义

- **近似比率 (Accuracy Ratio)**：
  - **最小化问题**：$r(s_a) = \frac{f(s_a)}{f(s^*)}$
    - $f(s_a)$ 是近似算法给出的解的目标函数值。
    - $f(s^*)$ 是最优解的目标函数值。
  - **最大化问题**：$r(s_a) = \frac{f(s^*)}{f(s_a)}$
- **性能比率 (Performance Ratio)**：
  - 如果存在 $c \geq 1$，使得对于所有实例 $r(s_a) \leq c$，则该算法称为 c-近似算法。
  - 所有实例中满足此条件的最小 c 称为算法的性能比率 R_A。

VERTEX-COVER

- Instance: an undirected graph $G=(V,E)$.
- Problem: find a set $C \subseteq V$ of minimal size s.t. for any $(u,v) \in E$, either $u \in C$ or $v \in C$.

Example:



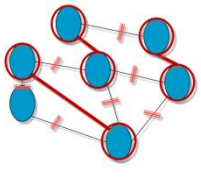
- $C \leftarrow \phi$
- $E' \leftarrow E$
- **while** $E' \neq \phi$ **do**
  - let (u,v) be an arbitrary edge of E'
  - $C \leftarrow C \cup \{u,v\}$
  - remove from E' every edge incident to either u or v

$O(n^2)$

- return C

Remove O(E) while O(E) 整体 O(E^2)

Demo



旅行商问题 (TSP) 的其他近似算法

旅行商问题（TSP）是一个经典的 NP 完全问题，寻找从一个起点出发，访问每个城市一次并返回起点的最短路径。由于找到最优解的计算复杂度很高，我们通常使用近似算法来找到接近最优解的路径。以下是几种常见的 TSP 近似算法。

1. 最近邻算法 (Nearest-Neighbor Algorithm)

算法步骤：

1. 从某个城市开始，选择尚未访问的最近的城市。
2. 重复步骤 1，直到访问所有城市。
3. 返回起始城市。

优点：简单，易于实现。
缺点：对于某些图，近似比率可能很差。

2. 两次绕树算法 (Twice-Around-the-Tree Algorithm)

算法步骤：

1. 构造图的最小生成树（MST）（例如使用 Kruskal 或 Prim 算法）。
2. 从任意顶点开始，沿着树两次行走，返回到起始顶点。
3. 对路径进行简化，避免多次访问同一个顶点。

优点：生成的路径长度不超过最优解的两倍。
缺点：实现较为复杂。

3. 多片段启发式算法 (Multifragment Heuristic)

算法步骤：

1. 将所有边按升序排列。
2. 选择下一最小的边，确保不创建度为 3 的顶点或长度小于 n 的环。
3. 重复步骤 2，直到构造出一个包含所有顶点的完整路径。

优点：生成的路径通常较短，接近最优解。
缺点：实现复杂度高。

TSP: Euclidean Instances

- ***Euclidean instances*** of the TSP problem obey the natural geometry of a 2D map.
  - triangle inequality: $d[i,j] \leq d[i,k] + d[k,j]$
  - symmetry: $d[i,j] = d[j]$
- For Euclidean instances, the nearest neighbor algorithm satisfies:

  $r(s_a) \leq \frac{1}{2} (\log_2 n + 1)$, where n = # cities
  - (still not a c-approximation algorithm)