一、算法简介

- 术语定义

算法(algorithm)：一系列明确的指令，用于在有限的时间内为任何合法输入获得所需的输出

原文：a sequence of unambiguous instructions for obtaining a required output for any legitimate input in a finite amount of time

基本操作(basic operation)：

执行得最多的那个/段指令，如果有 if，那么 if 被执行最多。for 不算(除非 for 有 if)

- **算法关注**：正确性（Correctness）、时间效率（Time Efficiency）、空间效率（Space Efficiency）

- **算法效率**取决于
  1. 输入的数据大小
  2. 被执行的指令个数

- **西格玛函数算计算次数**：已在离散数学学过，不再赘述

- **常用的效率增长比较**

$\log_2 N < N < N*\log_2 N < N\text{^}2 < N\text{^}3 < 2\text{^}N < N!$

- **常用的领域**

logbN：　把数据分成多个部分，每轮循环只处理其中的一个部分
N：　数据有几个，处理几次
NlogN：把数据分成多个部分，每轮循环处理其中的一个到几个部分
N^2:：循环内还有一个循环
N^3：内循环里还有一个循环
2^N：处理一个集合的全部子集
N!：　处理一个集合的全部排列

- **比较算法快慢时**：先把所有常数系数变成 1，只留下 N 和基数

1. $50n^3 + 20n + 4 \quad \in O(n^3)$
2. $4n^2 + 10 \quad \in O(n^2)$
3. $n(2n + 1) \quad \in O(n^2)$
4. $3\log n + 1 \quad \in O(\log n)$
5. $3\log n + n \quad \in O(n)$
6. $1 + \log 6 \quad \in O(1)$
7. $5! + 3^2 \quad \in O(1)$

- 三种不同的标注方式

  Big-O   execution will take at MOST that long

  Big-Ω   execution will take at LEAST that long

  Big-Θ   execution will take THAT long

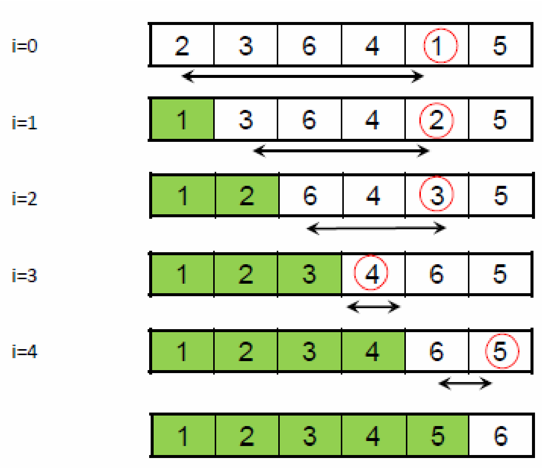
- 例子

  - $10n$ O(n)
  - $5n^2 + 20$ $O(n^2)$
  - $10000n + 2^n$ $O(2^n)$
  - log(n) * (1 + n)   O(nlog(n))

二、穷举算法

- 选择排序(selection sort, 复杂度 **n^2**)

  从未排序的部分找到最下的数字，放到已排序的末尾



ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort

//Input: An array $A[0..n-1]$ of orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 0$ **to** $n-2$ **do**

    $min \leftarrow i$

    for $j \leftarrow i+1$ **to** $n-1$ **do**

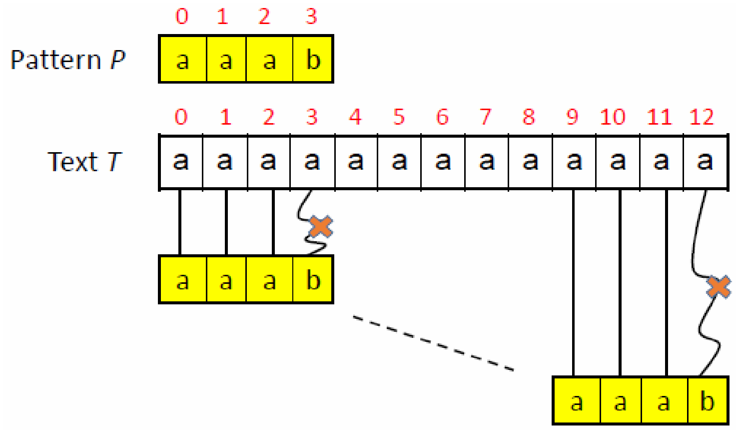
        if $A[j] < A[min]$

            $min \leftarrow j$

    swap $A[i]$ and $A[min]$

- 冒泡排序(**bubble sort**, 复杂度 **n^2**)：略

- 字符串匹配（下图为最坏情况）



There are m comparisons for each shift in the worst case (inner loop)
There are n-m+1 shifts (outer loop)
So, the worst-case running time is: O((n-m+1)*m)

- 销售员问题

路径数量：要用排列来解决 (n-1)! / 2 之所以除以二，是因为有两条路距离相等

```
for each permutation P of cities
    for each city i in P
        length ←length + weight(i,i+1)
    if length < min
        min ←length
        minroute←P
return minroute
```

- 背包问题

怎么做才能不让小偷背包超重，又能拿到最贵的几件货？
需要生成全部子集，也就是 2^n 个子集，然后逐一对比

- 人员分配问题

每个人分配到不同工作时，成本也不同，如何才能降低成本？也是用排列来解决 n!

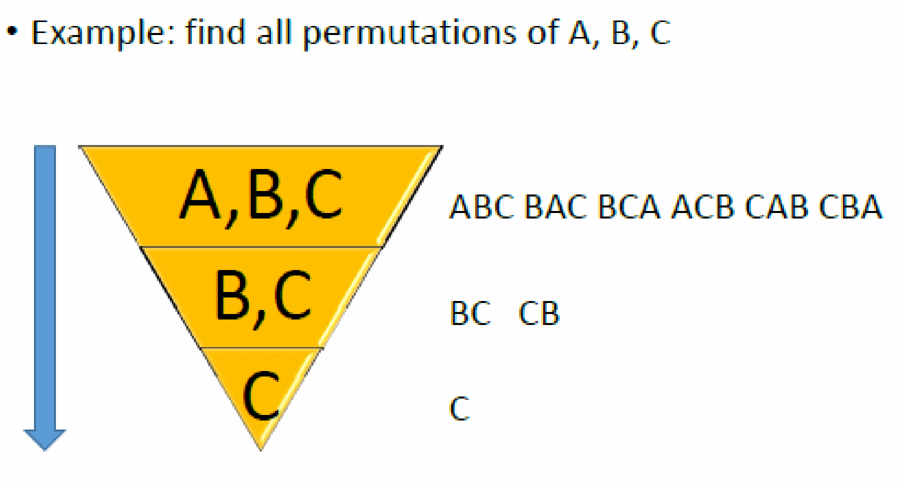
```
for each permutation P of job assignments
    totalcost←sum of the job costs for P
    if totalcost< mincost
        mincost←totalcost
        minperm←P
return minperm
```

三、减治法(Decrease and Conquer algorithms)

- 概念：仅知道一个更小输入的答案，用这个答案来求当前输入的答案。每轮循环里减去一个常量。例如：用递归(从上倒下)或循环(从下到上)求斐波那契数列。

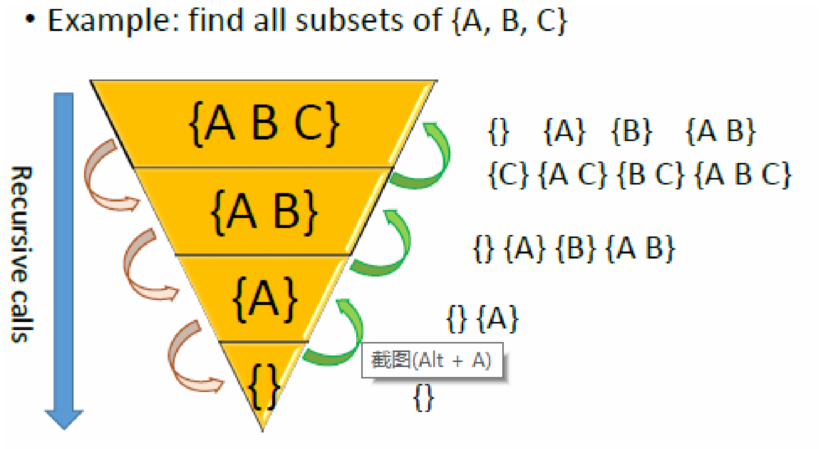
- 插入排序(Insertion Sort 复杂度 **n^2**)
  从未排序的部分里找到第一个元素，插入到已排序部分里的合适位置

- 生成排列 (Generate Permutation)




```
generatePerms(a1, a2, ..., an)
    if n > 1
        smallerPerms= generatePerms(a1, a2, ..., an-1)
        initialize allPermsto {}
        for each p in smallerPerms
        insert anbefore a1and add to allPerms
        for i = 1 to n-1
            insert anafter aiand add to allPerms
        return allPerms
```

- 生成子集 (Generate Subsets)




```
generateSubsets(a1, a2, ..., an)
  if n > 0
    subsets = generateSubsets(a1, a2, ..., an-1)
    for each subset s in subsets
      clone s to s'
      insert anto s'
      add s' to subsets
    return subsets
```

- 每次减少一个常量
  1. 减少一半的二分法(binary search)
  2. 求一个数的 n 次幂：18 -> 9 + 9 -> 1 + 4 + 4 -> 2*2 -> 1 + 1
  3. 找金币为：把金币平均分两堆，称重

- 每次减少的数量不定，例如用欧几里得算法求最大公约数 GCD

```
GCD (m, n)
  if ((m % n) = 0)
    return n
  else
    return GCD(n, m % n)
```

四、分治法(Decrease and Conquer algorithms)

- 把一个问题分为不同的子问题，每次解决其中一个或多个问题

- 影响效率的元素：T(n)= aT(n/b)+ F(n)：
  1. 每一轮要处理的子问题个：数 a
  2. 每个子问题的输入大小：n/b (总共要被分为 b 个部分)
  3. 这一轮要处理的其他子问题的总消耗：F(n)

1) If $n^{\log_b a} < F(n)$, $T(n) \in O(F(n))$

2) If $n^{\log_b a} > F(n)$, $T(n) \in O(n^{\log_b a})$

3) If $n^{\log_b a} = F(n)$, $T(n) \in O(n^{\log_b a} \log n)$

Example 1: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

a = 4, b = 2, F(n) = n → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2; F(n) = n → $T(n) \in O(n^2)$

Example 2: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

a = 4, b = 2, F(n) = n^2 → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2; F(n) = n^2 → $T(n) \in O(n^2 \log n)$

Example 3: $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

a = 4, b = 2, F(n) = n^3 → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2; F(n) = n^3 → $T(n) \in O(n^3)$

- 合并排序 (Merge Sort 复杂度 **nLogN**)
  每次都把当前的数组分成两半，并对这两半调用递归，递归后把两部分元素按大小更新到原数组里。所以这个算法分为两部分，一部分是拆分和递归，另一部分是排序。

ALGORITHM *Mergesort*($A[0..n-1]$)
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2\rfloor - 1]$ to $B[0..\lfloor n/2\rfloor - 1]$
        copy $A[\lfloor n/2\rfloor..n-1]$ to $C[0..\lceil n/2\rceil - 1]$
        Mergesort($B[0..\lfloor n/2\rfloor - 1]$)
        Mergesort($C[0..\lceil n/2\rceil - 1]$)
        Merge(B, C, A)

ALGORITHM *Merge*($B[0..p-1], C[0..q-1], A[0..p+q-1]$)
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ **and** $j < q$ **do**
        if $B[i] \le C[j]$
            $A[k] \leftarrow B[i]$; $i \leftarrow i+1$
        else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
        $k \leftarrow k+1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$

- 二叉树求高度

ALGORITHM $Height(T)$

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

- 二叉树求叶子数

Algorithm $LeafCounter(T)$
//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0 //empty tree
else if $T_L = \varnothing$ **and** $T_R = \varnothing$ **return** 1 //one-node tree
else return $LeafCounter(T_L) + LeafCounter(T_R)$ //general case

五、变治法 (Transform and conquer)

- 三种主要类型
  1. 实例化简(instance simplification)：变换为同样问题的一个更简单或更方便的实例
  2. 改变表现(representation change)：变换为同样实例的不同表现
  3. 问题化简(problem reduction)：变换为一个算法已知的问题

- 实例化简的例子
  1. 检测数组里的数字是否都唯一(distinctness)
     暴力法：O(n^2)
     先用合并排序，再逐个检测：O(nlogn) + O(n) = O(nlogn)

ALGORITHM $PresortElementUniqueness(A[0..n-1])$
//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise

sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

2. 计算数组里出现最多的数字(mode)
   暴力法：O(n^2)
   创建计数数组 O(n^2)，从计数数组找到最大值 O(n)，所以结果是 O(n^2)

   先用合并排序，再逐个检测：O(nlogn) + O(n) = O(nlogn)

```
ALGORITHM  PresortMode(A[0..n − 1])
    //Computes the mode of an array by sorting it first
    //Input: An array A[0..n − 1] of orderable elements
    //Output: The array's mode

    sort the array A
    i ← 0                      //current run begins at position i
    modefrequency ← 0          //highest frequency seen so far
    while i ≤ n − 1 do
        runlength ← 1;  runvalue ← A[i]
        while i + runlength ≤ n − 1 and A[i + runlength] = runvalue
            runlength ← runlength + 1
        if runlength > modefrequency
            modefrequency ← runlength;  modevalue ← runvalue
        i ← i + runlength
    return modevalue
```

3. 在数组里查找特定数字
   直接搜是 O(n)，用二叉树是 O(logn)，但是二叉树必须要一个已排序的数组
   先排序再用二分法：O(nlogn) + O(logn) = O(nlogn)

- 改变表现的例子：堆和堆排序

用一般的方法从数组里插值和删除最大值
ArrayList：插入 O(1)，删除最大值 O(n)
SortedArrayList：插入 O(n+logn) = O(n)，删除最大值 O(1)

堆(Heap)的定义：一个二叉树，除最后一层外，每一层都被从左到右填满，且父元素都大于子元素。**堆的高度一定不会超过 log2N**

用堆来实现插值和删除最大值
插入：O(logn)
删除：O(logn)

这里我们用数组来实现堆，元素的索引是 i，那么其父元素的索引是 i/2，它的两个子元素的索引是 2i 以及 2i+1

插入的实现 O(logN)

1. 先插到数组末尾
2. 然后逐级向上比较，如果插入的这个元素比父元素大，则交换位置

删除最大元素 O(logN)

1. 把数组的首尾元素调换位置
2. 删除末尾的元素，再把 root 跟子元素里较大的那个作比较，逐级下移

从一个数组构造一个 Heap O(nlogn)

N 个父节点（其实是约等于 N/2 个），每个父节点向下比较约 logN 次

1. 假设数组已经是一个 Heap
2. 找到最右侧的父元素，与其下方的子元素中较大的那个对比和换位
3. 往左边再找一个父元素，与其下方的子元素较大的那个对比和换位
4. 最下一层的父元素处理完成后，处理上一级父元素（这时要逐级向下对比）
5. 循环往复，直到所有的父元素都已经处理完毕

堆排序

1. 先构造一个堆 O(nlogn)
2. 逐个从堆中删除最大值 O(nlogn)，并放到新数组内 O(1)
3. 最后效率 O(nlogn)

六、时空权衡(Space/Time tradeoffs)

- **例子**

  更少内存但是更多运行时间的例子：冒泡排序，选择排序

  更多内存但是更少运行时间的例子：合并排序

- **输入增强(Input enhancement)**

  处理输入的数据，生成一些特定信息，用于解决后续问题

  1. **比较计数排序(CCS，Comparison Counting Sort，复杂度 O(n^2))**

     针对数组的每一个元素，统计小于该元素的元素个数，并把结果记录到一个新数组。最后根据新数组的记录(每个记录对应着老元素的新索引)，把原始数组的元素复制到结果数组里对应的索引。

```
Algorithm ComparisonCountingSort(A[0..n-1])
  for i←0 to n-2
    for j ←i+1 to n-1
      if input[i] < input[j]
        Count[j]++
      else
        Count[i]++
  for i←0 to n-1
    output[Count[i]] ←input[i]
  return output
```

2. 分布计数排序(DCS，Distribution Counting Sort，复杂度 O(n))

数组里的元素全部来自于一个已知的集合，把各种元素出现的次数记录到新数组里，之后计算各种元素最后出现的位置，最后放到结果数组的对应位置

```
Algo DistributionCountingSort(A[0.. n-1])
  for j←0 to u-l do        // u - l 是最大元素减去最小元素，即记录表 C 大小
    C[j] ←0      // C 内所有的元素计数器归零
  for i←0 to n-1 do
     C[A[i]-l] ←C[A[i]-l] + 1    // A[i]-l 即当前元素在 C 内的索引，这句是元素计数
  for j←1 to u-l do
    C[ j] ←C[ j-1] + C[ j]    // 计算每个元素在结果数组中最后应出现的位置
  for i←n-1 down to 0 do
    j←A[i]-l
    S[C[ j] -1] ←A[i]   // 按第三个循环的结果，把原始数据放到新数组对应位置
    C[ j] ←C[ j]  - 1
  return S
```

3. 字符串查找 (Horspool 算法, 复杂度 O(nm))

假设有一个长度为 m 的 pattern(即待搜索的 keyword)，可能出现在这个 pattern 里的字符一共有 A-Z 总共 26 个。

1. 先构造一个长度为 26 的数组，所有元素初始化为 m，然后遍历 pattern 的第一个到倒数第二个字符，更新 A-Z 数组里的对应内容，以计算出一个“当需要右移时，具体需要移动多少个字符”的表。算法如下：

```
ShiftTable(P[0 …m-1])
  for i ← 0 to tableSize -1
    Table[i] ← m
  for j ← 0 to m -1
    table[ P[j] ] ← m-1-j
  Return table
```

2. 把 pattern 的最后一位与文本 text 的位置对齐，从右往左逐个对比 pattern 的字符，一旦遇到不匹配的字符，就从 table 里找到“该向右把 pattern 移动多少位”，然后移动 pattern，从新开始对比。算法如下：

```
HorspoolMatching(P[0…m-1], T[0…n-1])
  i ← m-1
  while I <= n-1
    k←0
    while k <= m-1 and P[m-1-k] = T[i-k]
      k ← k+1
    if k = m
      return i – m + 1
    else
      i ← i + Table[ T[i] ]
```

- **预构造(pre-structuring)**

  用更多的空间(space)来加快数据的访问。

  有序数组来存取数据

  查找：O(logn)　　插入/删除: O(n)

  无序数组存取数据

  插入 O(1)　　　查找/删除 O(n)

  散列(hashing)

  1. 每个元素有一个唯一的 key
  2. 用一个大的数组，称为哈希表(hash table)
  3. 用一个散列函数把 key 映射到哈希表 f(key) = index，

  常用取余(mod)法

  1. 如果是数字，直接取余，把 value 放到 hash 表内
  2. 如果不是数字，先转换成数字，再映射到哈希表，例如把字母的 ord 相加

```
h ←0                    // input is a string S of length s
for i←0 to s-1 do       // ci is the char in ithposnof S
   h ←h + ord(ci)      // ord(ci) is the relative posn
                        // of ciin the alphabet
 hashcode←h mod numBuckets      // map sum of posnsinto range
```

  冲突的解决

  当两个 key 算出来的 index 一致时，目前有两种解决办法

  1. 分离链(Separate Chaining)

     hash 表内不存值，而是存一个指针，类似链表，最差情况下它就是个链表
  2. 闭散列(Close Hashing)

     计算出的 index 指向的行已经有值时，自动往下检测空闲的行，如到表尾则回表头

  散列的效率全部是 O(1)，但依赖于实现，所以 hash 函数很重要

  例如：每次插入有冲突，则是 O(n)

  一个好的哈希函数应该：

  1. 将密钥均匀地分布在桶上
  2. 为相似的数据生成非常不同的哈希码

  一个改进版的字符串 Hash 函数

```
alpha ←|alphabet|// size of the alphabet used
h ←0
for i←0 to s-1 do
  h ←h + (ascii(ci) * alpha^(i))
code ←h mod numBuckets
```

七、基础数据结构和图

- 数组 Array
  优点：每个元素都可以在常量时间内访问（随机访问）
  缺点：长度固定；插入和删除需要批量移动元素

- 链表 Linked List
  优点：大小不固定；插入和删除都方便
  缺点：无法随机访问

- 栈 Stack
  特性：后进先出(LIFO, Last-in-first-out)
  操作：插入 push，删除 pop，获取最上层的元素 peek

- **抽象数据类型**
  定义：数据结构 ＋ 操作
  例如：Priority Queue(Insert/pop) 和 Stack(push/pop/peek)

- 队列 Queue
  特性：先进先出(FIFO, First-in-first-out)
  操作：入队 enqueue / 出队 dequeue / 获取最前面的元素 peek

- 集合 Set
  特性：集合内的元素不能重复
  操作：添加 Add / 删除 Remove / 检查 Check / 遍历 Iterate over
  例子：HashSet 用 Hash 表做的 Set，访问 O(1)，无序
  　　　TreeSet 用 BinaryTree 做 Set，访问 O(logn)，有序

- 映射 Map
  作用：用 key 在 hash table 里查找，并返回对应的 value。常用于 hashtable / hashmap。

- 树 Tree
  特性：有连接，但无闭环(acyclic)。用树来存储数据往往可以加速数据访问速度。

  邻接矩阵和邻接列表：用不同的方式表达节点之间的关系(关系就是边 edge)

  邻接矩阵的元素个数是 n^2
  邻接列表的元素个数是 2 * | E |，最坏情况下 E 就是 n 选 2：n(n-1) / 2

- 特殊的图：
  连通图：任意两个节点之间有路到达
  二分图：节点分在两个不同的集合，边只在两个集合的点之间
  有环图：图中间至少有一个环
  无环图：图里没有环
  树：有连接，但无闭环(acyclic)

- 图的算法

图的遍历

DFS(Depth-first Search，复杂度 O(n^2)，离散数学里已经学过，注意栈的使用):
深度优先算法。没有被用到的边称为 back edge

```
Algorithm Depth_First_Search(Graph G)
  // Graph G = {V,E}
  initialize visited to false for all vertices
  for each vertex v in V
    if v has not been visited
    dfs_helper(v)

function dfs_helper(Vertex v)
  visit node v
  for each vertex w in V adjacent to v
    if w has not been visited
      dfs_helper(w)
```

DFS 用处

1. 生成一个图的子图
2. 找到两个节点间的路径
3. 迷宫寻路
4. 检测图是否有环
5. 找到图里全部的 component
6. 搜索问题的状态空间以获得解决办法(AI)

DFS 的效率

1. O(|V|2) for adjacency matrix
2. O(|V|+|E|) for adjacency lists

BFS(Breadth-first Search)，离散数学里也学过
广度优先算法。不用 stack 而用 queue 来检测下一个要检测的节点。
没有被使用到的边称为“cross-edges”。
效率与 DFS 相同，但遍历节点的顺序不一致。

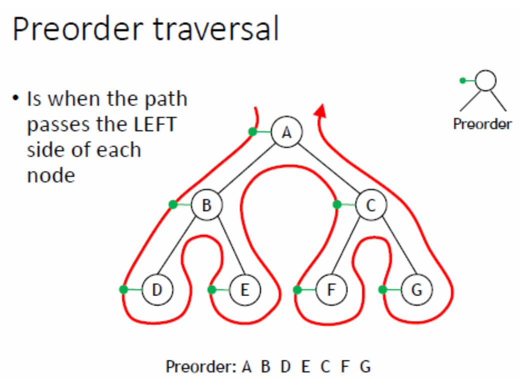
```
Algorithm Breadth_First_Search(Graph G)
  // Graph G = {V,E}
  initialize visited to false for all vertices
  for each vertex v in V
    if v has not been visited
      bfs_helper(v)

function bfs_helper(Vertex v)
  visit node v
  initialize a queue Q
  add v to Q
  while Q is not empty
    for each w adjacent to Q.head
      if w has not been visited
        visit node w
        add w to Q
    Q.dequeue()
```

BFS 的用途
与 DFS 一致，在部分问题上甚至优于 DFS

- 树的遍历

常用的有三种，如果先遍历右侧，还能再来三种
前序遍历(Pre-order)：中左右
中序遍历(In-order)：左中右
后序遍历(Post-order)：左右中

一种新的思考方式



只要有中序(必须)的列表加上前序/后续的列表(二选一)，就可以重构树

- 一些例子
  1. 生成一张图的生成树：则用 BFS 生成的树，两点之间路径较短
  2. 迷宫寻路：BFS 路线总和最短，DFS 走的回头路最少，BFS 回到父节点次数太多
  3. 寻找最短路径：BFS 生成的路径最短，DFS 可找到路径但不一定最短(先 BFS 再 DFS)

八、拓扑排序

- **DAGs** (Directed acyclic graphs，有向非循环图)
  一种有向图，这种图里，任何一个顶点出发的路径，不可能回到其本身（没有路径环）

  拓扑排序问题
  当有一系列任务，且某些任务依赖于其他任务时，需要对任务来排序，并放到一个队列

  拓扑排序算法一：DFS
  1. 用条件来构造并验证一个图，每个任务是一个顶点，每一个约束是一条边
  2. 从任何一个节点开始 DFS
  3. 到达死胡同(Dead end)时，遍历死胡同的顺序，就是拓扑顺序的“逆顺序”
  4. 到达死胡同后，把死胡同的这个节点放到列表 L 里，然后返回到父节点查看其他子节点（当有多个父节点时，随便选择一个）
  5. 循环执行操作 4，直到所有的节点被遍历完成
  6. 把列表 L 逆序排列，得到结果

  拓扑排序算法二：减治法
  1. 找到一个入度为 0 的顶点
  2. 删除这个顶点和所有连到它的边
  3. 重复 1 和 2，直到删除全部顶点
  4. 拓扑顺序与删除顶点的顺序一致
  5. 如果删除的过程中找不到一个入度为 0 的定点，则这个图不是 DAG

  减治法的实现
  1. 用一个 SET 来存储所有的入度为 0 的点
  2. 用一个有序列表存储被删除的点

```
Algorithm TopoSort(G)
  create an empty ArrayList A
  create an empty TreeSet Candidates
  add all v with inDegree=0 to Candidates
  while Candidates is not empty
    v = Candidates.first()
    add v to A
    for each vertex w adjacent to v
      remove edge (v,w) from G
    if w has inDegree=0
      add w to Candidates
    remove vertex v from G
  if there are no vertices remaining in G
```

```
   solution is in A
else
   no solution exists
```

贪心算法

例如用最少的硬币数量进行找零

```
Algorithm MakeChange(N)
   sum = 0
   coins = {} // set of coins to be returned
   while sum < N do
      choose the largest coin X with value <= (N-sum)
      sum += X.value
      coins += {X}
   return coins
END
```

注意：贪心算法不一定总是能给出最优解，只能给出一个近似的最优解

最优问题：找到一个(接近)最优的解
决策问题：有没有解

对于一个背包问题，用贪心算法只能从价值最大的开始拿，然后再看下一个能拿的价值最大的是多少，而不是先产生全部的组合，再来判断哪个最优。

贪心算法的每一步的选择都需要满足：

1. **可行的(feasible)**：它必须满足问题的约束
2. **局部最优(local optimal)**：它是当前步骤中所有可行选择中的最佳局部选择(可能要提前排序)
3. **不可取消(irrevocable)**：选择一旦做出，在算法的后面步骤中就无法改变了

- **最小生成树(Minimum Spanning Trees, MST)**

1. 没有封闭的环
2. 包括所有的顶点
3. 所有的边的权重加起来最小
4. 最小生成树可能不止一个

Prim 算法

1. 选择一个顶点 A，做为树的一个节点
2. 找到连接到 Tree 所有顶点里权重最小的边 E，这个边 E 连接了顶点 B，且 B 不在 Tree 里
3. 把 B 和 E 加入到 Tree 里
4. 重复 2 和 3 直到所有的顶点和边都加入到了 Tree

(算法见下页)

```
Algorithm Prim(G)
   VT←{v0}
   ET←∅
   for i←0 to |V|-1 do
      find a min-weight edge e=(u,v) from E where u is in VT(in the tree) and v is in
            V-VT(not yet in the tree)
      VT←VT ∪{v}
      ET←ET ∪{e}
return T = (VT, ET)
```

对离散子集(Disjointed Subset)的操作

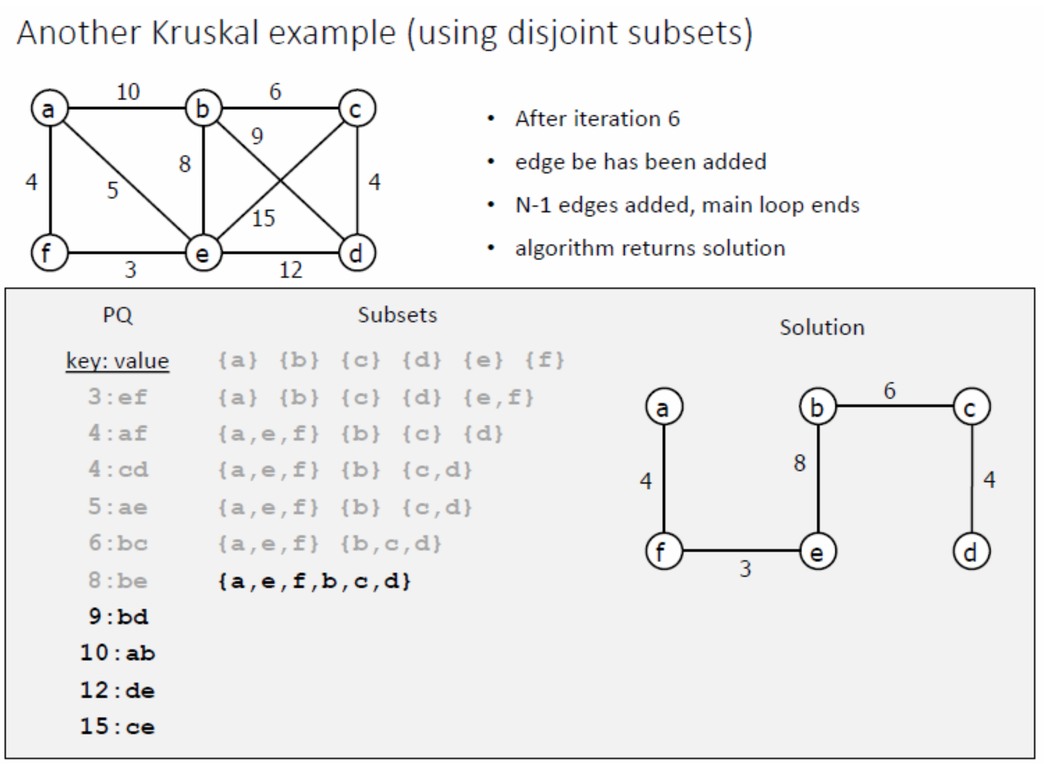
Makeset(x) – 创建指定集合 x 的离散子集，每个子集里面一个元素

Find(x) – 返回含有指定元素 x 的子集

Union(x,y) – 把含有元素 x 和元素 y 的两个子集进行合并

Kruskal's 算法

1. 用所有顶点加到一个树 T 里（只加顶点，不加边）
2. 创建一个优先级的 map 队列，key 是权重，value 是边的两个顶点
3. 把图里的每一个顶点各放到一个子集里
4. 按优先级找到图里权重最低的边，往树里加
5. 当一个边(u,v)被添加到树的时候，把 u 和 v 所在的子集合并
6. 每一个子集都是一个 Component
7. 如果 u 和 v 本身就在同一个子集，则不做任何操作，否则会产生闭环
8. 最后只会剩下一个子集，所有顶点都在这个子集里



(伪代码见下页)

```
Algorithm Kruskal(G)
  Add all vertices in G to T // add v's but don't add e's
  Create a priority queue PQ // will hold candidate edges
  Create a collection DS // disjoint subsets
  for each vertex v in G do
      DS.makeset(v)
  for each edge e in G do
      PQ.add(e.weight, e ) // PQ of edges by min weight
  while T has fewer than n-1 edges do
    (u,v)  ←PQ.removeMin() // get next smallest edge
    cu  ←DS.find(u)
    cv  ←DS.find(v)
    if cu  ≠  cv then // be sure u,vare not in
      T.addEdge(u,v) // the same subset
      DS.union(cu, cv)
  return T
```

Kruskal's 的 union-find 版本复杂度为 O(N^2 * logN)

N^2 是因为 N 个顶点全部互相连接边数最差情况为 N^2，然后对全部的边按照权重排序，得到 N^2 * LOG(N^2)，化简后得到 O(N^2 * logN)

- Prim 与 Kruskal 算法的异同

  相同点

  1. 都是解决最小生成树的问题
  2. 都是用的贪心算法

  不同点

  1. Prim 查找权重最小的边来扩展一个树
  2. Kruskal 查找权重最小且不会产生闭环的边

九、动态编程(Dynamic Programming)

- 步骤

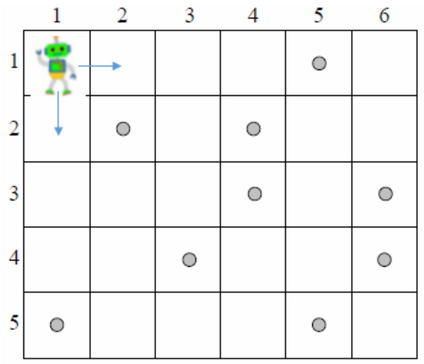
  1. 把问题划分为子问题
  2. 用刚刚划分出来的子问题来找到或表达解决方案
  3. 使用表格来记录从下而上计算出的最优值
  4. 基于步骤 1 到 3 得到最终最优解

- 斐波那契数列

  用递归法，从上而下来算，需要重复计算很多数，例如 f(5)要计算 f(4)和 f(3)，f(4)也要计算一次 f(3)。

  可以改为每次计算一个未计算的数值时，放到一个表里，以后先查表，表里有的直接返回，没有的再进行计算。那么所有数值只用计算一次。时空**效率都是 O(N)**

- 机器人捡金币：在一个 n 行 m 列的格里，只能向右或向下移动，怎样让收益最大化



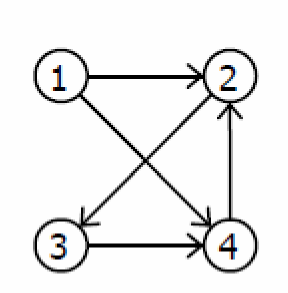
1. 创建一个 n 行 m 列的二位数组，与原始数组的大小一致
2. 依据原始数组，横向计算第一行每一格最多可以取到多少个金币
3. 从第二行到最后一行，检查每一格从左方和上方来，最多可以取到几个金币
4. 找到最后一个行的最大值，得到最终解答，**时空效率都是 Θ (nm)**

```
ALGORITHM RobotCoinCollection(C[1..n, 1..m])
  // Robot coin collection using dynamic programming
  // Input: Matrix C[1..n, 1..m] with elements equal to 1 and 0 for
  //        cells with and without coins, respectively.
  // Output: Returns the maximum collectible number of coins
  F[1, 1] ← C[1, 1]
  for j ← 2 to m do
    F[1, j] ← F[1, j - 1] + C[1, j]
  for i ← 2 to n do
    F[i, 1] ← F[i - 1, 1] + C[i, 1]
    for j ← 2 to m do
      F[i, j] ← max(F[i - 1, j], F[i, j - 1]) + C[i, j]
  return F[n, m]
```

- 动态编程基本方法
1. 弄懂最终的问题
2. 为问题创建一个可以递归定义：
   a. 子问题是什么？
   b. 子问题怎么样互相关联？
3. 确定怎么样存储子问题的结果
4. 用算法来填充 3 里面创建的数据结构

- 传递闭包 (Transitive Closure)

给定一个没有权重的有向图，判断任意两个节点之间是否可以到达



	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

1. 复制一个该图的邻接矩阵，作为 R0，作为将来的输出数据
2. 遍历所有节点，看看谁连接到了这个节点，而这个节点又连到了那些节点
3. 在当前节点的父子节点之间全部添加边
4. 在步骤 2-3 的循环里，每遍历完一个节点，结果 R 的下标加 1，如 R1、R2…

- Warshall's algorithm (就是解决上面这个问题的算法，效率为 **O(N^3)**)

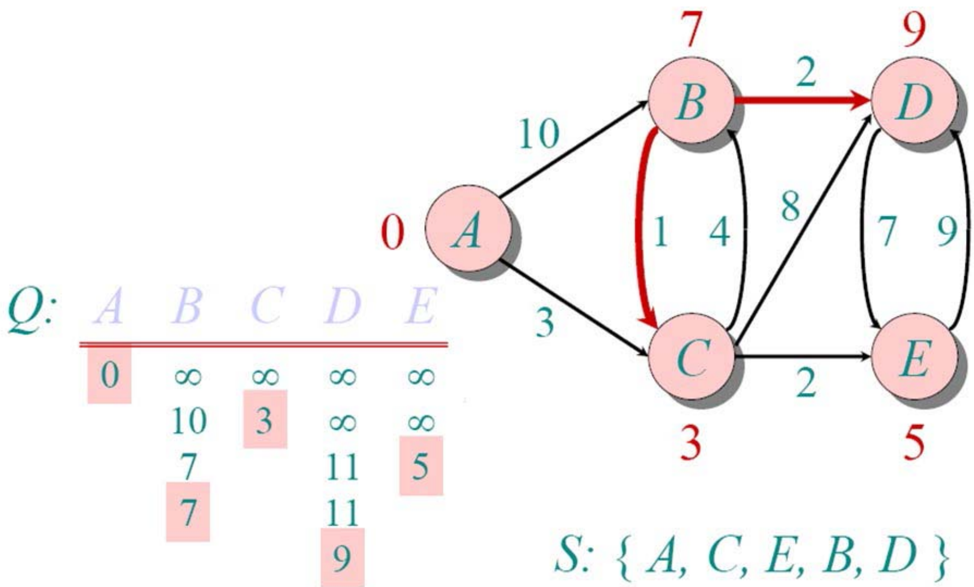
```
Warshall(G[1..n, 1..n])
  for k ←1 to n {
    for i←1 to n {
      for j ←1 to n {
        if ( G[i,k] == G[k,j] == 1 ) {
          set G[i,j] ←1
        }
      }
    }
  }
```

- 为什么 Warshall 也是动态编程
  它也是在找更简单的子问题，而且是从下往上的计算并记录

十、最短路径

- Dijkstra 算法（SSSP, Single source shortest path，单起点最短路径）
  解决的问题：计算从某个起点到图上所有其他点的最短路径
  为什么不用 BFS：因为 BFS 不能计算带权重的图

  步骤
  1. 为所有节点记录当前最短距离，先把所有距离初始化为正无穷
  2. 选择一个最近的没有被处理的节点（首次为起点），并查看它的全部邻居，如果计算出来的距离比它们先前记录的最短距离更短，那么更新这些距离**(Relaxation)**
  3. 重复 2



结果

1. 得到一个根节点为起始节点的最短路径树
2. 这是一个贪婪算法：每次都是处理离自己最近的节点
3. 因为最后的结果 pre 里包含每个节点的前一个节点，所以可以求**反向路径**
4. **输出**一个最短路径树，或者一个从起始节点到其他节点的距离列表

High-level pseudocode:

```
Initialise d and prev
Add all vertices to a PQ with distance from source as the key
While there are still vertices in PQ
    Get next vertex u from the PQ
    For each vertex v adjacent to u
        If v is still in PQ, relax v
```

```
Relax(v):
    if d[u] + w(u,v) < d[v]
        d[v] ← d[u] + w(u,v)
        prev[v] ← u
        PQ.updateKey(d[v], v)
```

与 Prim 算法的区别

1. 在 Prim 算法里，如何选择下一个节点的优先级，就是边权重
2. 在 Dijkstra 算法里，如何选择下一个节点的优先级，是边的权重加迄今到起点距离

限制

Dijkstra 算法不能处理权重为负数的图

- Floyd 算法（APSP，All-pairs shortest paths，完全最短路径）

解决的问题：计算图上任意两点的最短路径，不论是否用 Dijkstra 算法，复杂度是 O(N^3)

步骤（给予 Warshall’s 算法改造）

1. 邻接矩阵初始化时，两点间的边不用 1 而用权重表示
2. 如果不存在边，则邻接矩阵内该项目为正无穷
3. 任何定点到自身的权重都是 0
4. 如果 A 经过中间点 B 到 C 的总距离，小于 A 到 C 的直接距离，则更新 A 到 C 距离

			j		
		1	2	3	4
	1	0	1	**2**	1
i	2	∞	0	1	∞
	3	∞	∞	0	1
	4	∞	1	**2**	0

```
Floyd(D[1..n, 1..n])
    for k ←1 to n {
        for i←1 to n {
            for j ←1 to n {
                cost_thru_k←D[i,k] + D[k,j]
                if ( cost_thru_k< D[i,j] ) {
                    set D[i,j] ←cost_thru_k
                }
            }
        }
    }
```

十一、回溯法

略