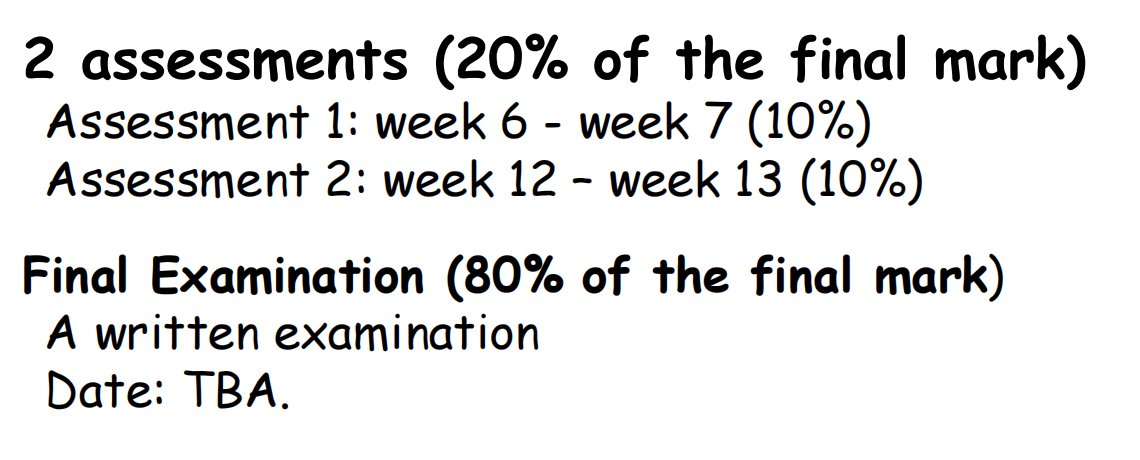
***INT102: Algorithmic Foundations And Problem Solving***

*0 Introduction & Pseudo Code*

*0.1 Assessments*

**

***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***

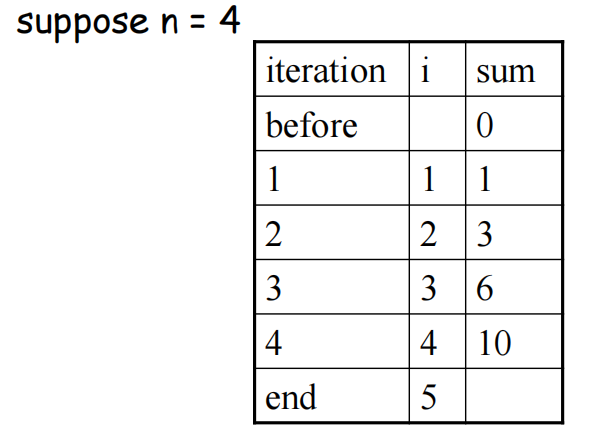
***begin***

***sum = sum + i***

***end***

***output sum***

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

******

*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?*

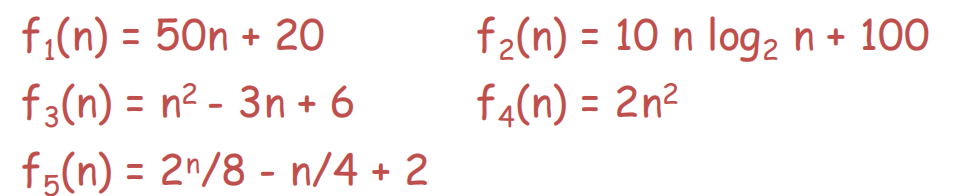
*• We will answer the question upper with an assumption:*

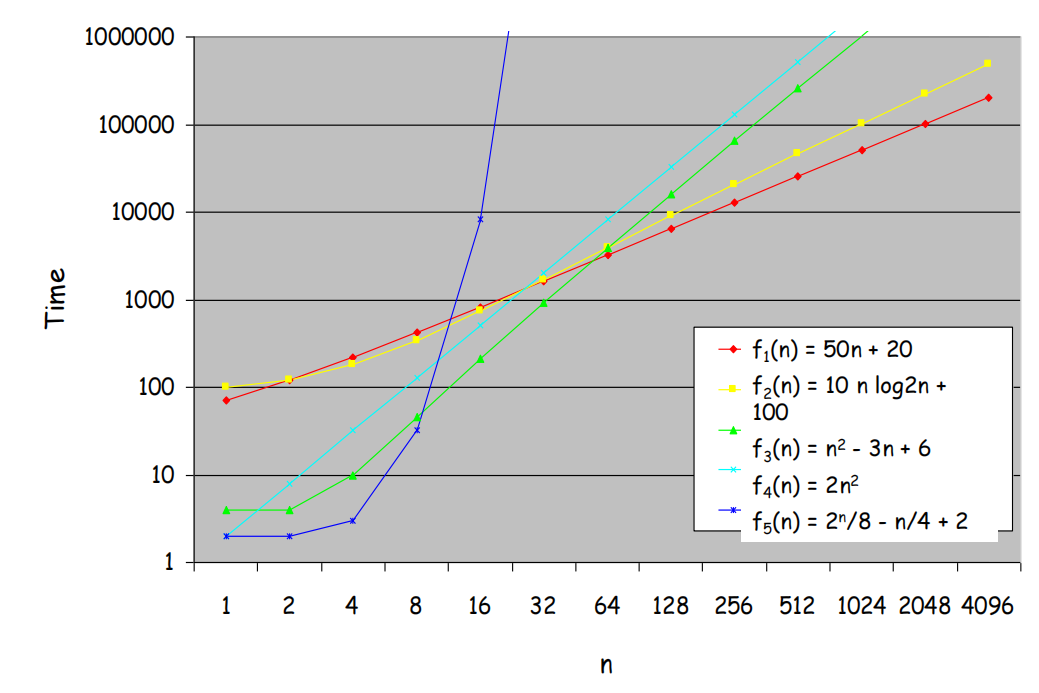
*---- an algorithm takes n^2 comparisons to sort n numbers*

*---- 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.*

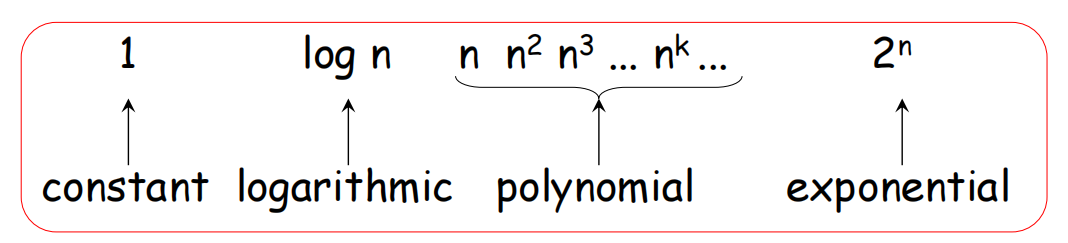
**

**

*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)***

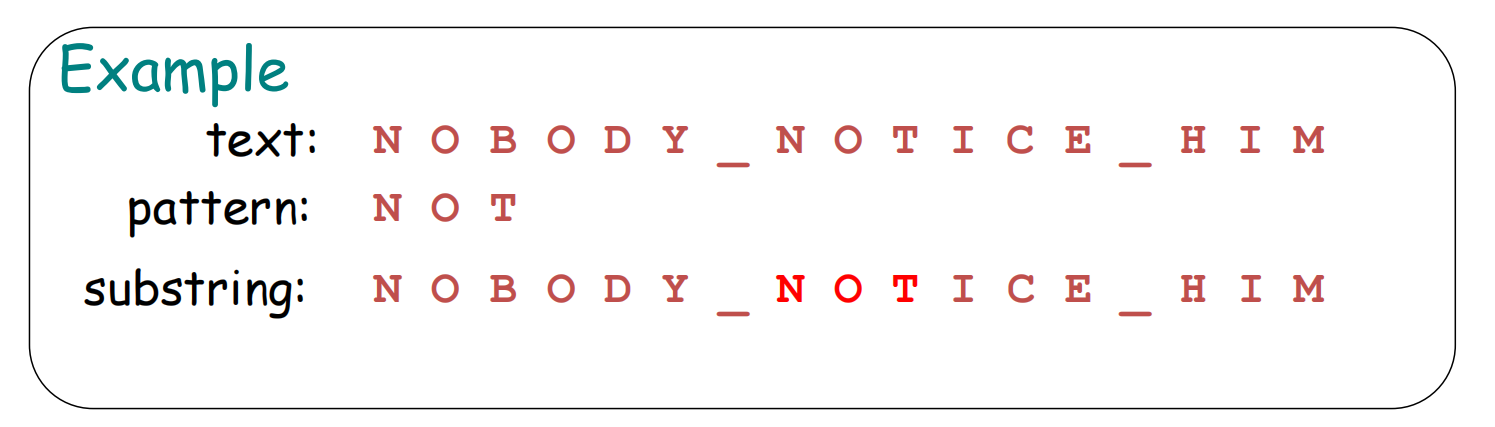
*Best case: X is the number in the middle => 1 comparison,* ***O(1)***

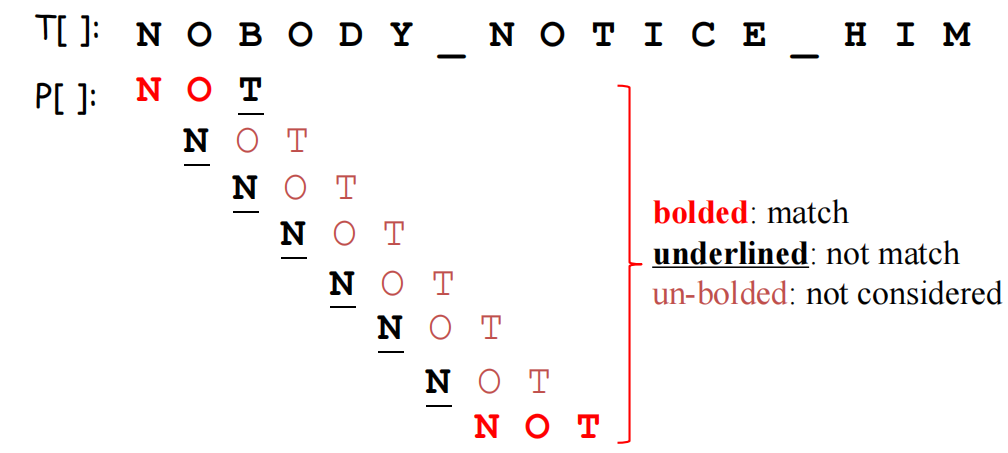
*Worst case:at most logn or [ logn ]+1 (n=16/17) comparisons,* ***O(logn)*** *( Every comparison reduces the amount of numbers by at least half : 16=> 8 => 4 => 2 => 1 , 4 comparison or 17 => 9=> 5 => 3 => 2 => 1 , 5 comparison )*

*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.*

**

**

***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。

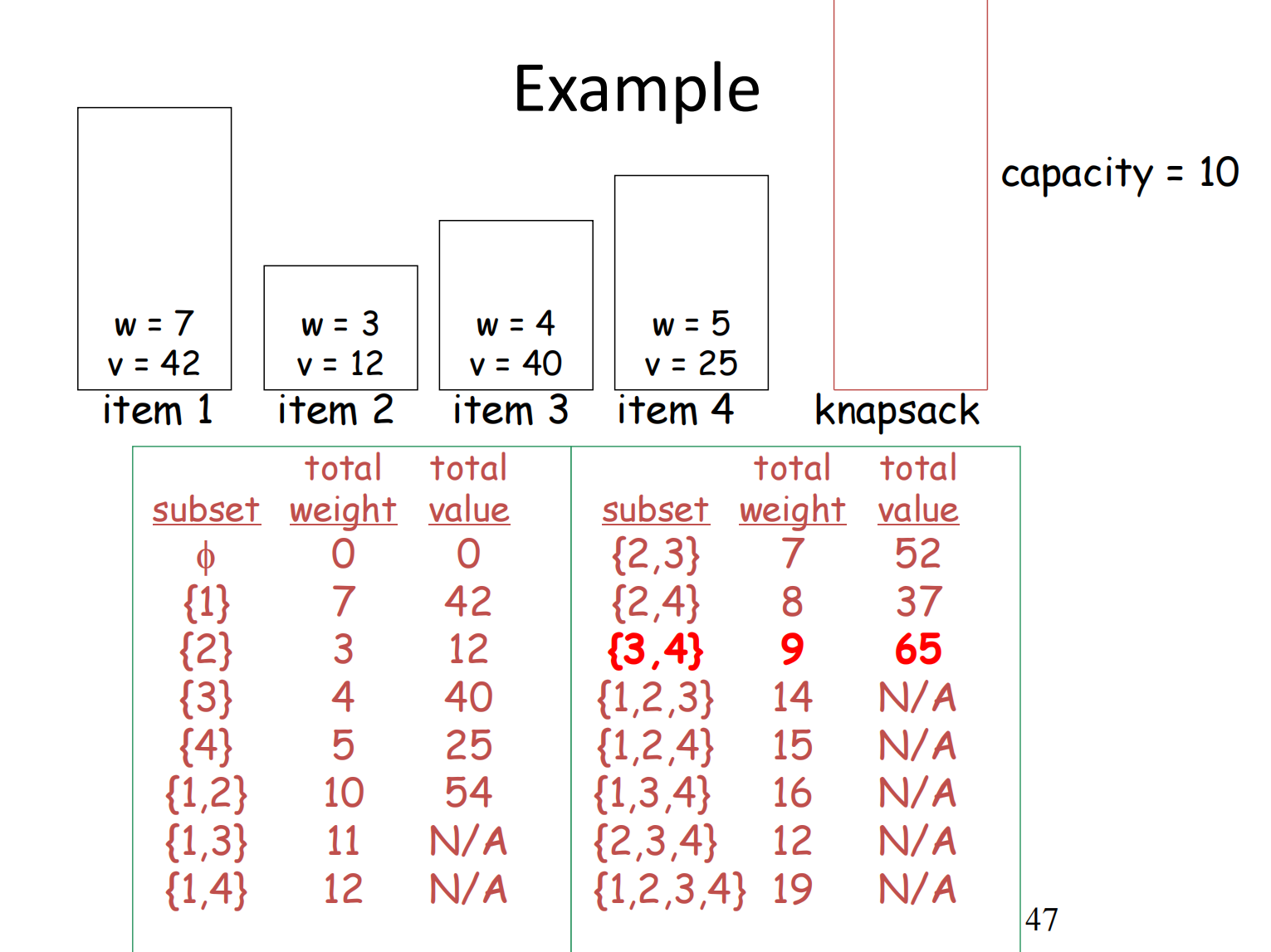
Hamiltonian Circuit汉密尔顿回路是：在一个图中，寻找一条路径，这条路径访问每个顶点恰好一次，并且回到起始顶点。

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 )***



*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***

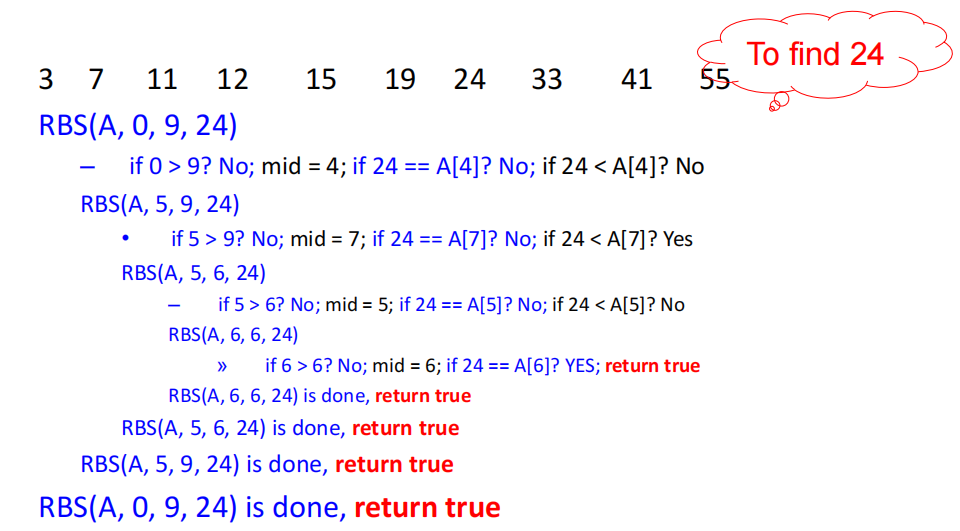
***return RecurBinarySearch(A, first, mid-1, X)***

***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) =*

***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).*

*1717176777892*

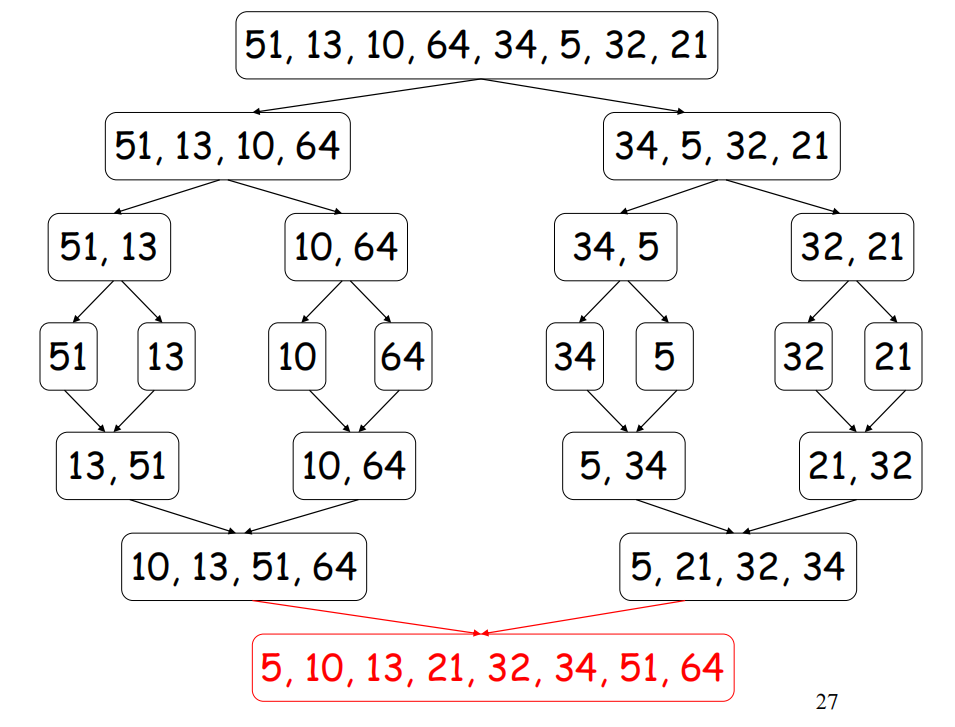
*Let n = 2^k , then T(n) = 2\*2^k - 1 =2n -1 =O(n)*

*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.*

**

***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) =*

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

***The time complexity of this algorithm is stable : O(nlogn)***

*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.)*

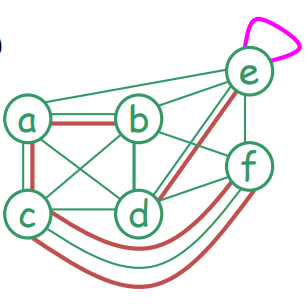
*4.1 Undirected graphs*

*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*

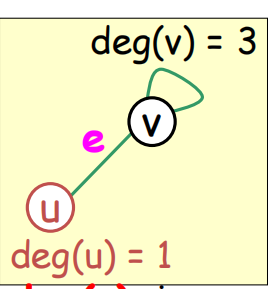
*---- u and v are said to be* ***adjacent***(相邻的) *and called* ***neighbors*** *of each other.*

*---- 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)*

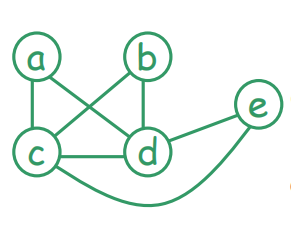
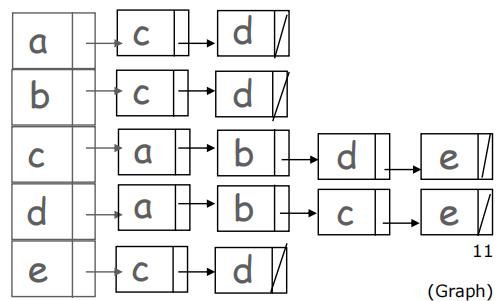
**

***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.*

*  *

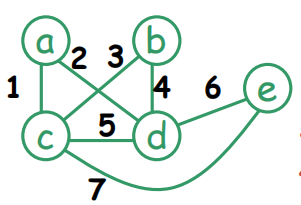
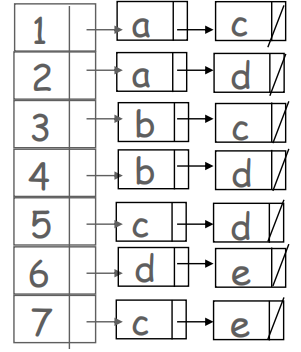
Example 1 (a) Adjacency matrix (b) Adjacency list

***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*

*  *

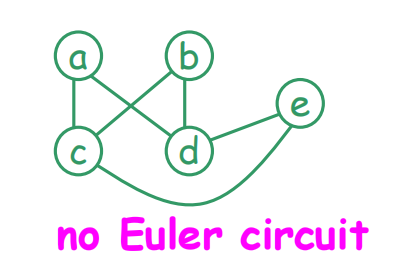
Example 2 (a) Incidence matrix (b) Incidence list

*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!*

**

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

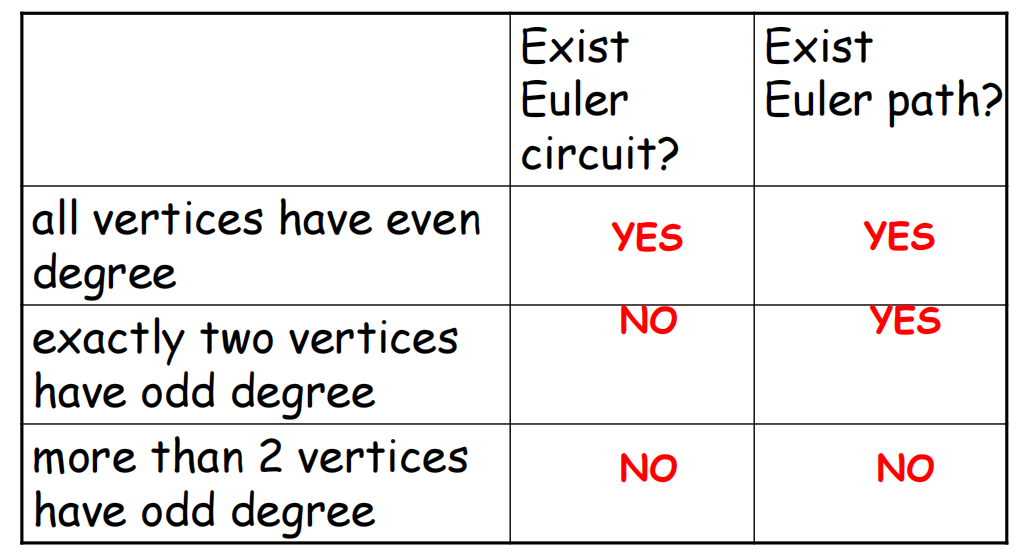
*An undirected graph G is said to be* ***connected*** *if there is a* ***path*** *between every pair of vertices. Even if the graph is* ***connected****, there may be no Euler circuit either. (Upper figure) If G is* ***not connected****, there is no single circuit to visit all edges or vertices.*

***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。

**

*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)***

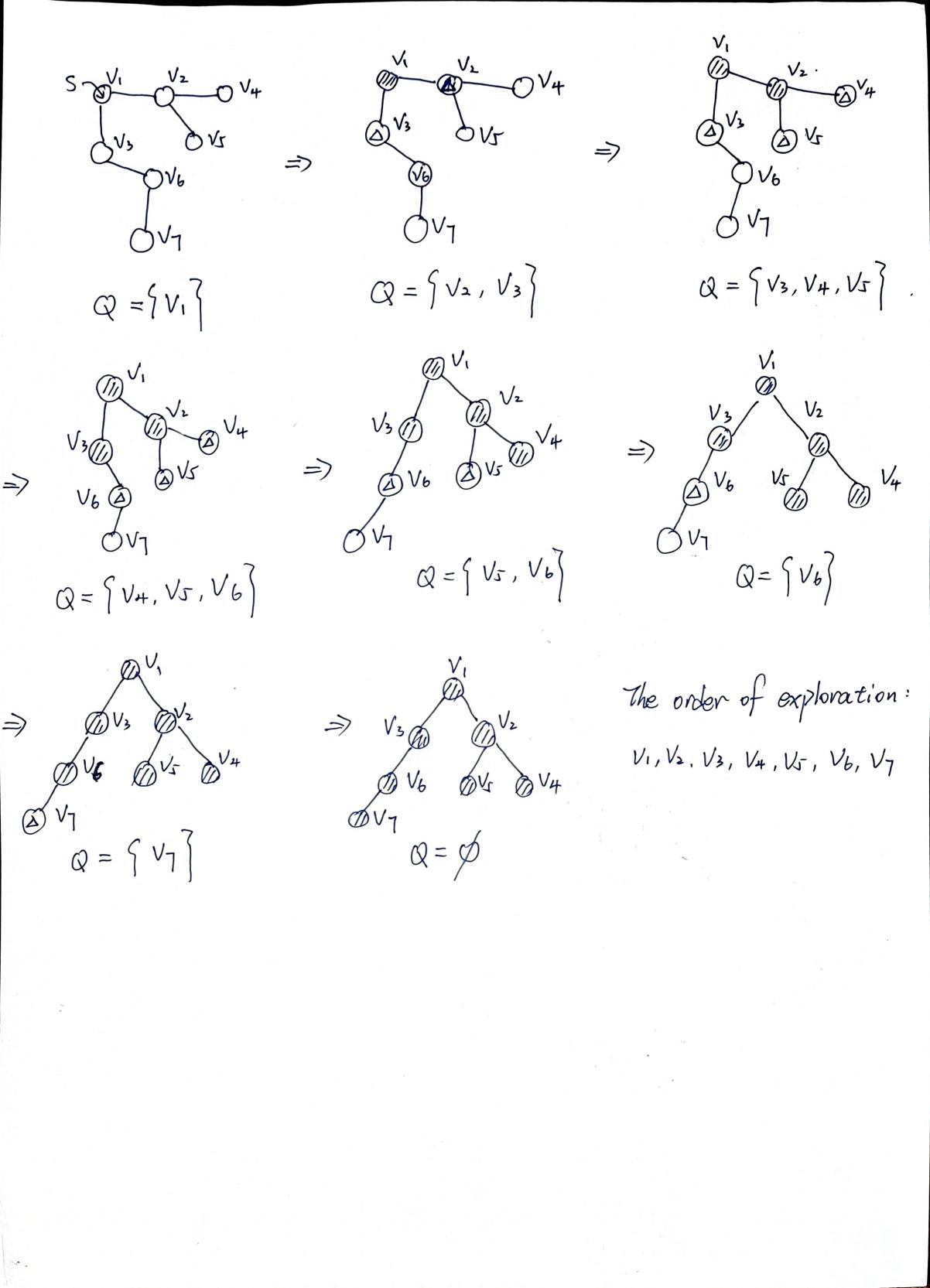
***if v is white then***

***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***

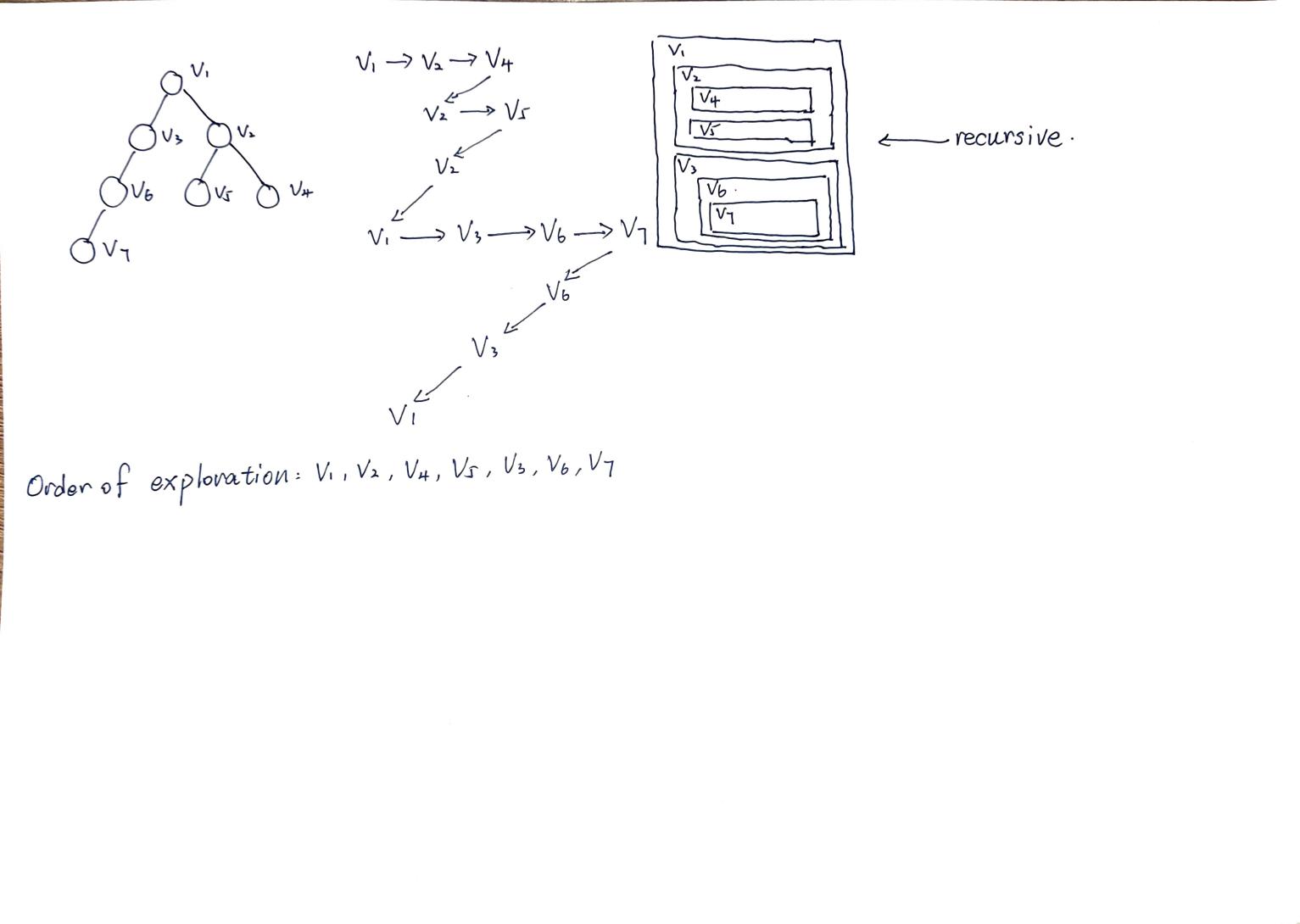
***for each unvisited neighbor w of v do***

***begin***

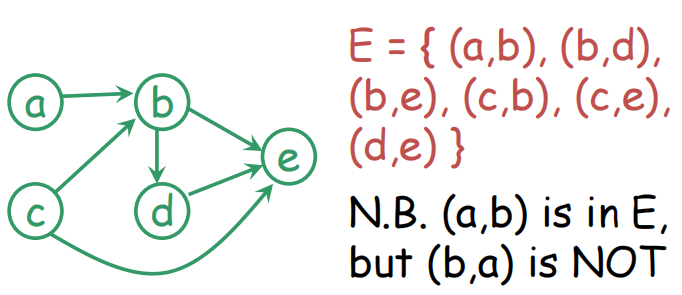
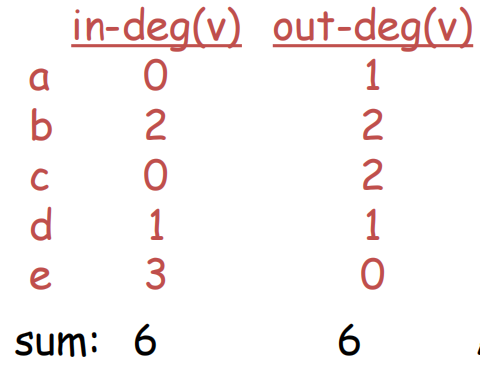
***DFS(w)***

***end***

*Example :*

******

*4.7 Directed graph*

* *

*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.*

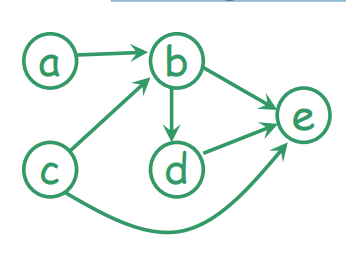
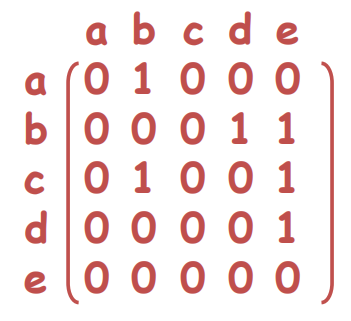
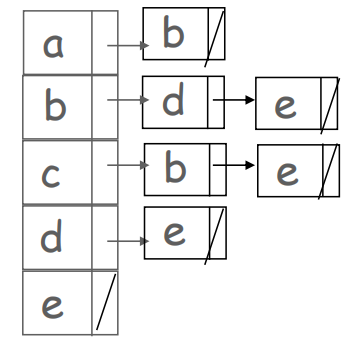
*The sum of in-degree and out-degree is always equal.*

***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.*

*  *

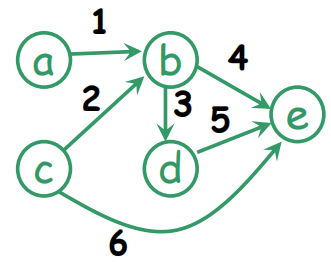
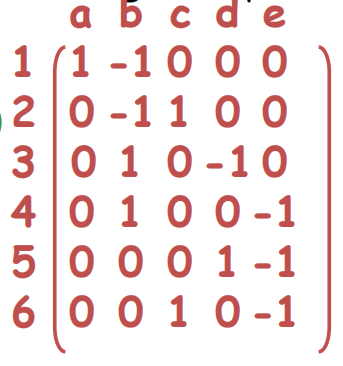
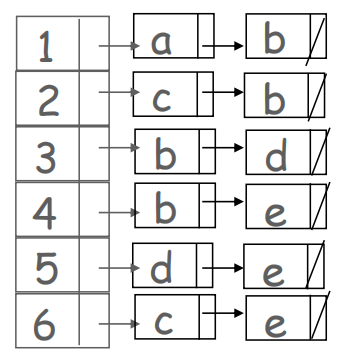
Example 3 (a) Adjacency matrix (b) Adjacency list

***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)*

*  *

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 .*

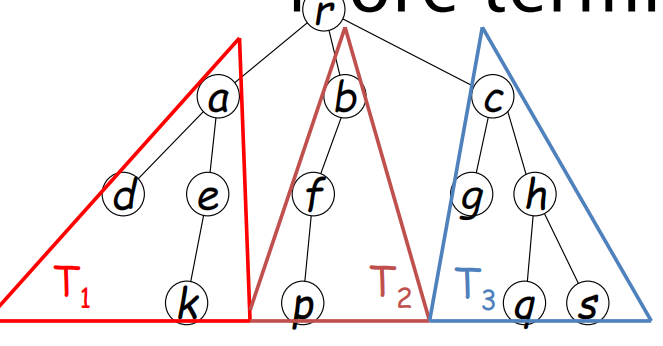
*The highest vertex is called the* ***root.***

*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):***

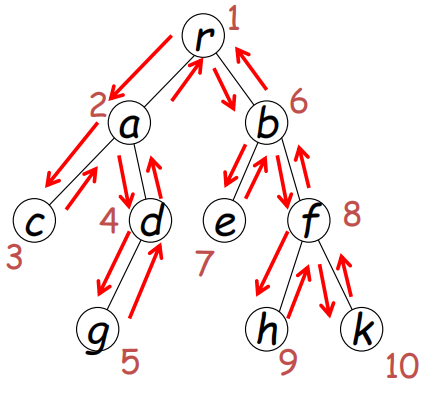
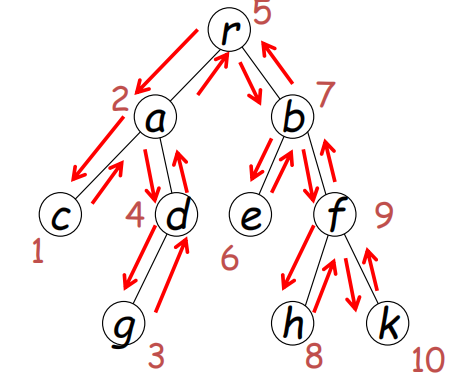
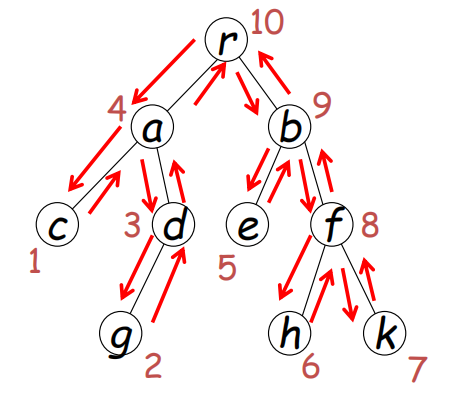
***if node is not null:***

***PostorderTraversal(node.left)***

***PostorderTraversal(node.right)***

***visit(node)***

*Example 1:* **这个红线没用只看遍历顺序**

*  *

1. preorder traversal (b) inorder traversal (c) postorder traversal

*Example 2:*

A

/ \

B C

/ \ \

D E F

*---- 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.*

*1 1*

*/ \ / \*

*2 3 2 3*

*/ \ / / /*

*4 5 6 4 6*

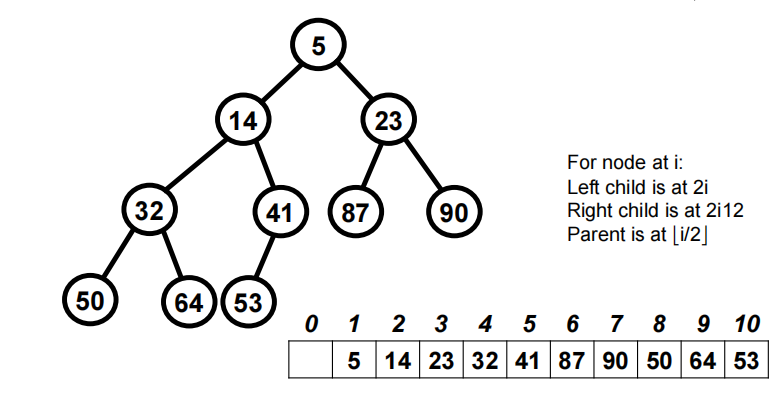
1. *complete tree (b) not complete tree*

*----* ***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) .*

**

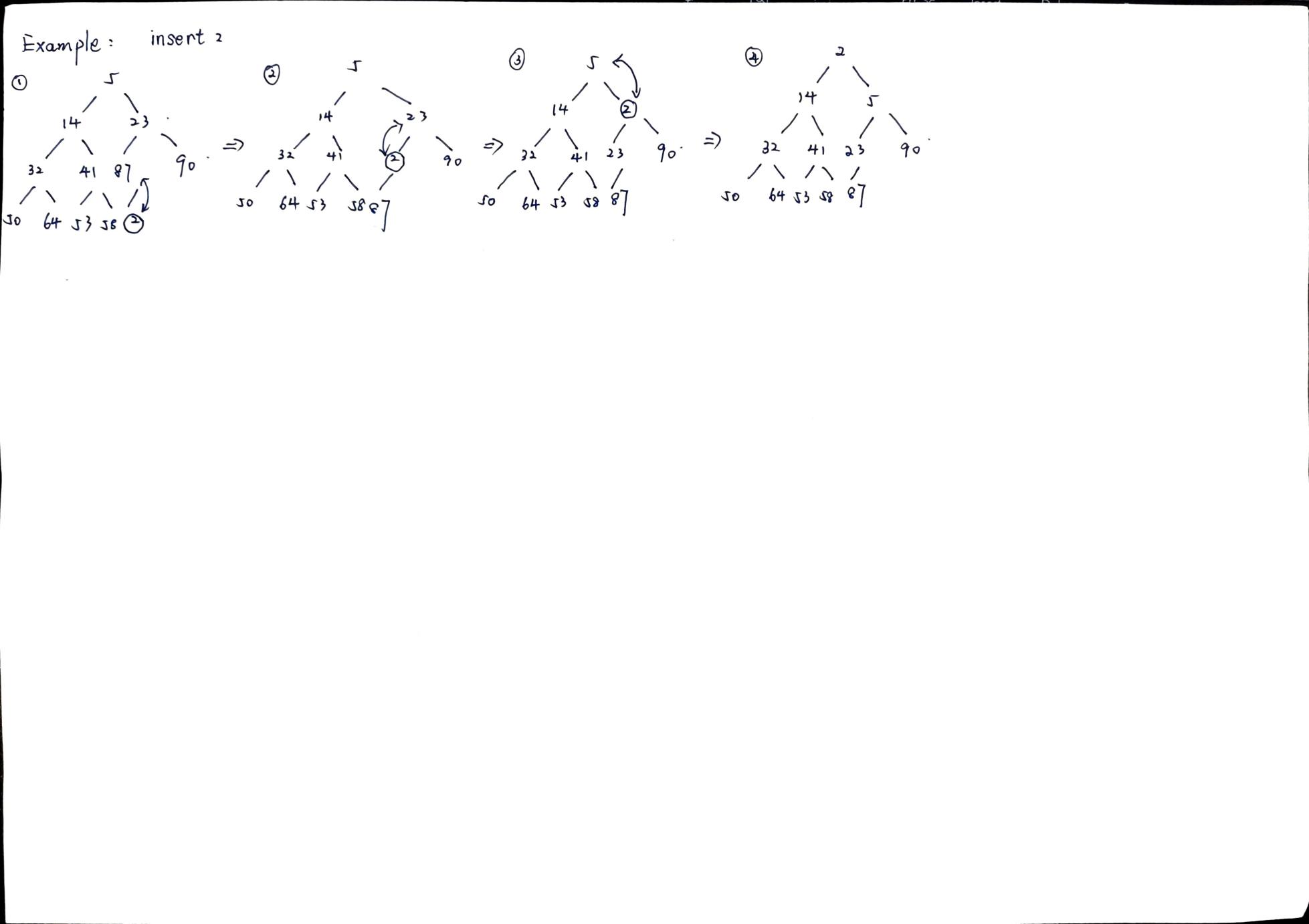
---- 插入新元素到最小堆的算法 ***O(1)~ O(logn)***

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）。



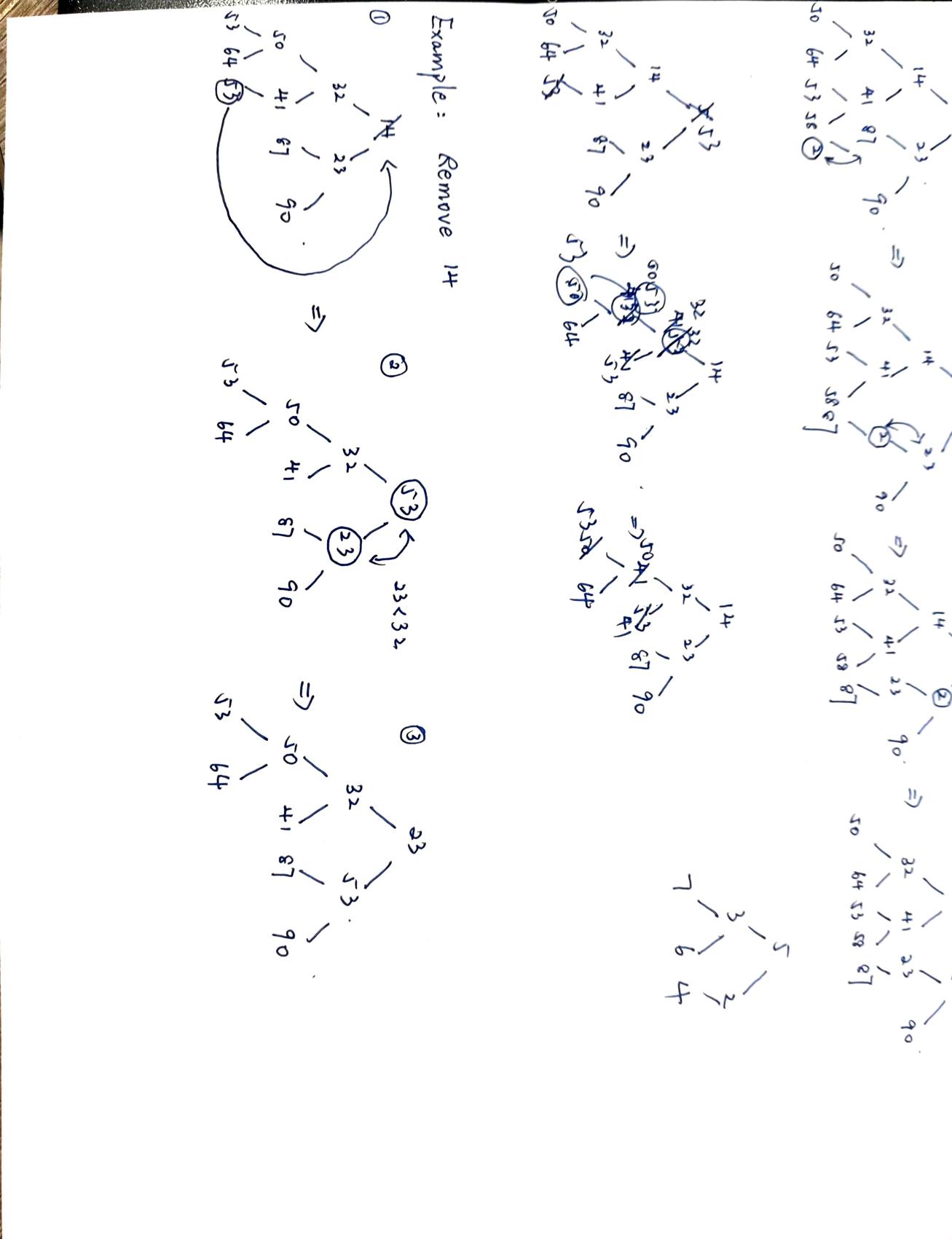
---- 移除元素在最小堆的算法 ***O(1)~ O(logn)***

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。

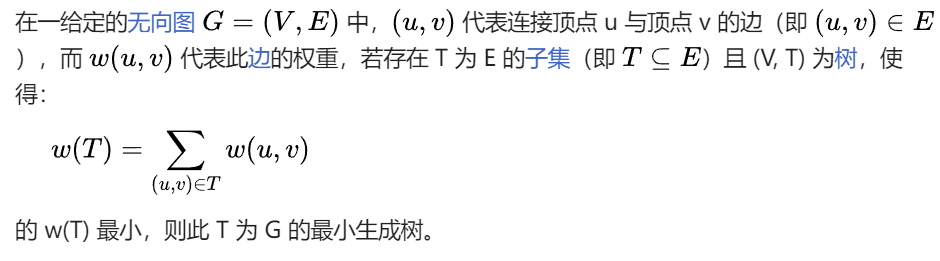
4继续调整，直到堆的性质恢复



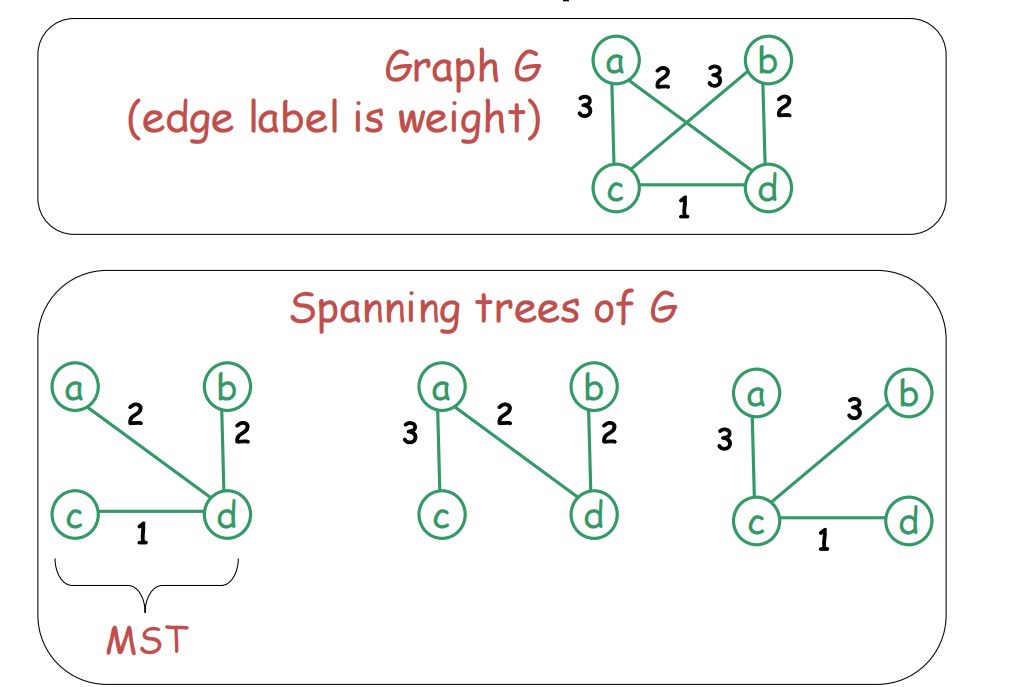
*5 Greedy Method*

*---- At every step, make the best move you can make. Keep going until you’re done*

*5.1 Minimum Spanning tree (MST)* 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 ? 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***

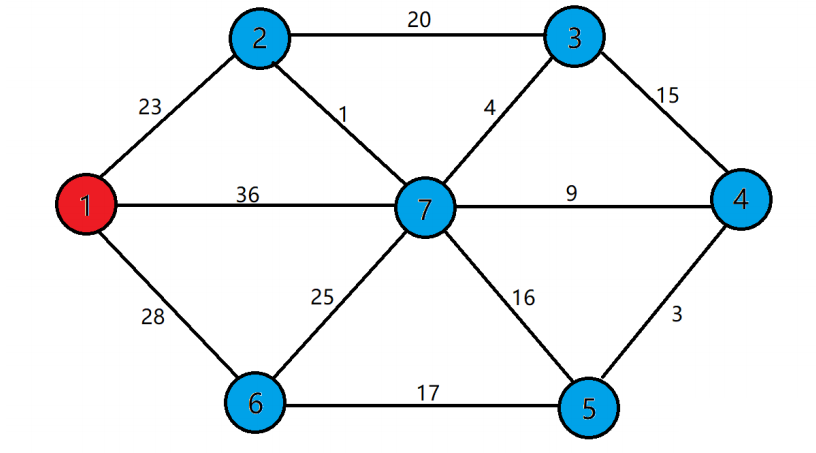
***that v is in VT and u is in V-VT***

***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。

*----Complexity of Prim’s Algorithm*