

CSOR W4231 Analysis of Algorithms, I

Tong Wu, tw2906

CSOR W4231 Analysis of Algorithms, I

0. Course Overview

- 0.1 Algorithms

- 0.2 Efficient Algorithms

- 0.2.1 Running time

1. Insertion Sort and Efficient Algorithms

- 1.1 Sorting problem

- 1.2 Insertion sort

- 1.2.1 Example of insertion sort

- 1.2.2 Code

- 1.3 Analysis of algorithms

- 1.4 Analysis of insertion sort

- 1.4.1 Correctness of insertion-sort

- 1.4.2 Running time $T(n)$ of insertion-sort

- 1.4.3 Worst-case analysis

- 1.5 Efficiency of algorithms

- 1.5.1 Efficiency of insertion-sort and the brute force solution

- 1.5.2 Attempt 1

- 1.6 Running time in terms of # primitive steps

- 1.7 Conclusion

2. Asymptotic Notation, Mergesort and Recurrences

- 2.1 Asymptotic Notation

- 2.1.1 Asymptotic upper bounds: Big- O notation

- 2.1.2 Asymptotic lower bounds: Big- Ω notation

- 2.1.3 Asymptotic tight bounds: Big- Θ notation

- 2.1.4 Asymptotic upper bounds that are not tight: little o

- 2.1.5 Asymptotic lower bounds that are not tight: little ω

- 2.1.6 Relationship between asymptotic notations

- 2.1.7 Basic rules for omitting low order terms from functions

- 2.1.8 Properties of asymptotic growth rates

- 2.2 The Divide & Conquer Principle

- 2.3 Merge sort

- 2.3.1 Codes

- 2.3.2 Analysis of mergesort

- 2.4 Solving recurrences and running time of mergesort

- 2.4.1 Solving recurrences, method 1: recursion trees

- 2.4.2 A general recurrence and its solution

- 2.4.3 Solving recurrences, method 2: Master theorem

- 2.4.4 Solving recurrences, method 3: the substitution method

- 2.5 Conclusion

3. Divide & conquer algorithms: fast int/matrix multiplication

- 3.1 Binary search

- 3.1.1 Pseudo Code for Binary Search

- 3.1.2 Binary Search Running Time – Linear

- 3.1.3 Binary Search Running Time – Sublinear

- 3.1.4 Conclusion

- 3.1.5 Binary Search from EEEN30002 Numerical Analysis

- 3.2 Integer multiplication

- 3.2.1 A first divide & conquer approach

4. Quicksort and balls-in-bins

5. Graphs, Breadth-First Search (BFS) and its applications

5.1 Graphs

- 5.1.1 Definitions of graphs
- 5.1.2 Trees and tree properties
- 5.1.3 Matching and bipartite graphs
- 5.1.4 Degree Theorem
- 5.1.5 Running time of graph algorithms

5.2 Representing graphs

- 5.2.1 Adjacency matrix
- 5.2.2 Adjacency list
- 5.2.3 Adjacency list vs. Adjacency matrix

5.3 Breadth-first search (BFS)

5.4 Applications of BFS

6 Depth-first search (DFS) and its applications

6.1 Depth-first search (DFS)

- 6.1.1 DFS vs BFS
- 6.1.2 Directed graphs: classification of edges

7 NP Class and SAT

0. Course Overview

0.1 Algorithms

An algorithm is a **well-defined** computational procedure that transforms the **input** (a set of values) into the **output** (a new set of values).

The desired I/O relationship is specified by the statement of the **computational problem** for which the algorithm is designed.

An algorithm is **correct** if, for every input, it **halts** with the correct output.

算法是一个被明确定义的计算过程。它将一组输入值转换为一组新的输出值。

期望的输入输出关系由设计算法的计算问题所声明指定。

对于每个输入，都能以正确的输出作为停止，则算法是正确的。

0.2 Efficient Algorithms

Efficiency is related to the resources an algorithm uses: **time**, **space**

- How much time/space are used?
- How do they scale as the input size grows?

0.2.1 Running time

Running time = number of **primitive computational steps** performed; typically these are

1. arithmetic operations: add, subtract, multiply, divide **fixed-size integers**
2. data movement operations: load, store, copy
3. control operations: branching, subroutine call and return

运行时间=所执行的原始计算步骤的数量；通常这些步骤是：

1. 算术操作：加、减、乘、除固定大小的整数
2. 数据移动操作：加载、存储、复制
3. 控制操作：分支、子程序调用和返回

1. Insertion Sort and Efficient Algorithms

1.1 Sorting problem

- Input: A list A of n integers x_1, \dots, x_n
- Output: A permutation (排列组合) x'_1, x'_2, \dots, x'_n of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \dots \leq x'_n$

Example:

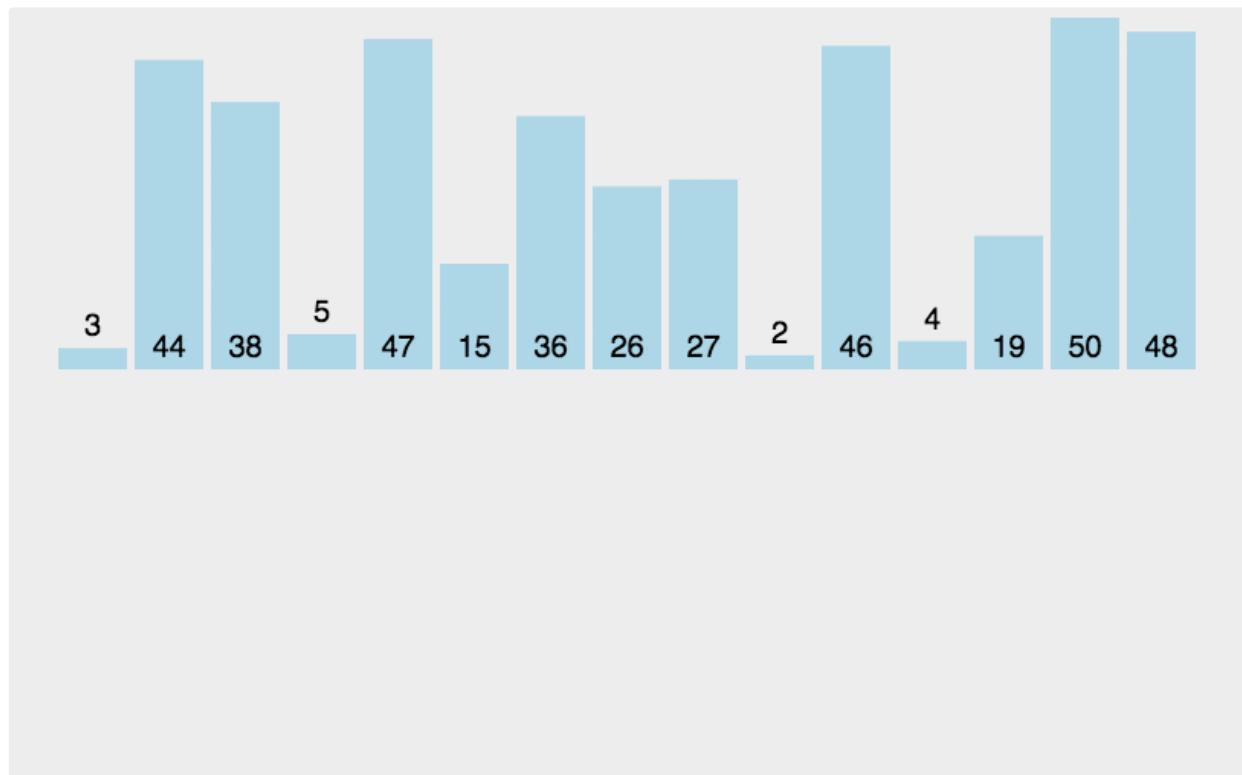
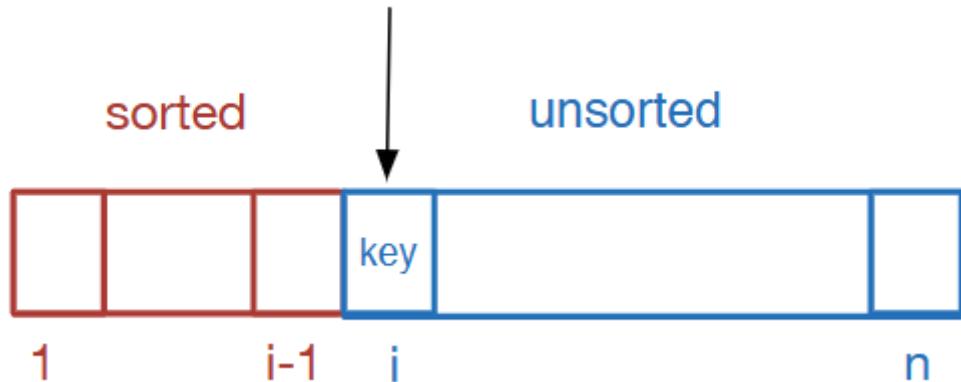
- Input: $n = 6$, $A = \{9, 3, 2, 6, 8, 5\}$
- Output: $A = \{2, 3, 5, 6, 8, 9\}$

The *data structure* should use to represent the list of output is:

Array: collection of items of the same data type

- allows for random access
- “zero” indexed in C++ and Java

1.2 Insertion sort



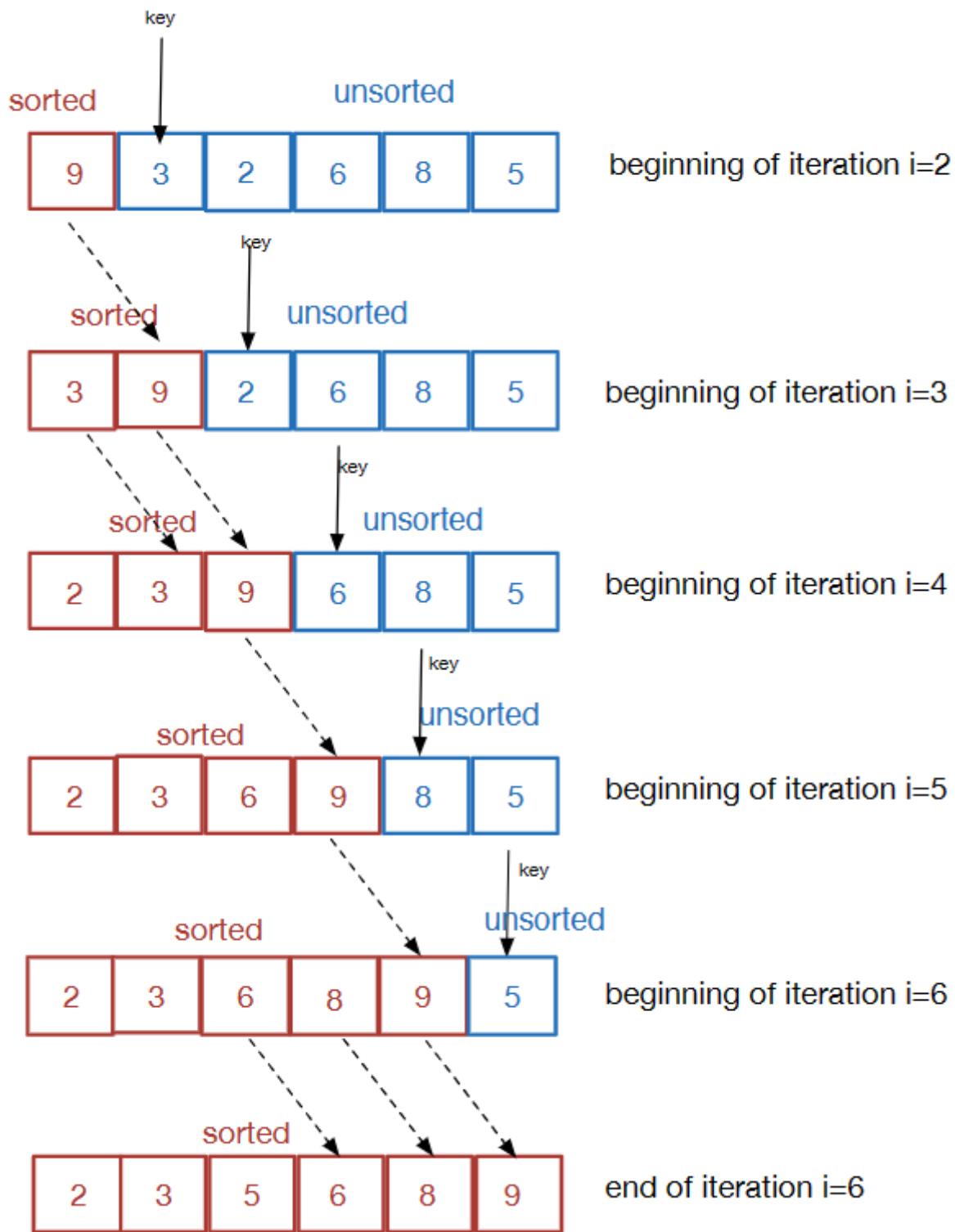
1. Start with a (trivially) sorted subarray of size 1 consisting of $A[1]$
2. Increase the size of the sorted subarray by 1, by inserting the next element A , call it **key**, in the correct position in the **sorted** subarray to its left.
 - Compare key with every element x in the sorted subarray to the left of key, starting from the right.

- If $x > key$, move x one position to the right
 - If $x \leq key$, insert key after x
3. Repeat Step 2. until the sorted subarray has size n .

1. 从一个有序序列开始, 即将第一个元素看作一个有序序列(sorted subarray), 称作为数组A[1], 其余的被称为未排序序列(unsorted subarray)
2. 将此有序序列的大小增加1, 方法是将下一个元素A(key), 插入到有序数组的适当位置。
 - 将key和每个有序序列中的元素x进行比较, 从最右边的数字开始。
 - 如果 $x > key$, 将 x 向右移一格
 - 如果 $x \leq key$, 将key插入在x的后面
3. 重复第二步, 直到未排序序列为0

1.2.1 Example of insertion sort

$n = 6, A = \{9, 3, 2, 6, 8, 5\}$



1.2.2 Code

Pseudo Code

Let A be an array of n integers

```

1 insertion-sort(A)
2     for i=2 to n do
3         key = A[i]
4             // Insert A[i] into the sorted subarray A[1,i-1]
5             j = i - 1
6             while j>0 and A[j]>key do
7                 A[j+1] = A[j]
8                 j = j - 1
9             end while
10            A[j+1] = key
11        end for

```

C++

```

1 void insertion_sort(int arr[],int len){
2     for(int i=1;i<len;i++){
3         int key=arr[i];
4         int j=i-1;
5         while((j>=0) && (key<arr[j])){
6             arr[j+1]=arr[j];
7             j--;
8         }
9         arr[j+1]=key;
10    }
11 }

```

Python

```

1 def insertionSort(arr):
2     for i in range(len(arr)):
3         preIndex = i-1
4         current = arr[i]
5         while preIndex >= 0 and arr[preIndex] > current:
6             arr[preIndex+1] = arr[preIndex]
7             preIndex-=1
8             arr[preIndex+1] = current
9     return arr

```

Java

```

1 public class InsertSort implements IArraySort {
2     @Override
3     public int[] sort(int[] sourceArray) throws Exception {
4         // 对 arr 进行拷贝, 不改变参数内容
5         int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
6         // 从下标为1的元素开始选择合适的位置插入, 因为下标为0的只有一个元素, 默认是有序的
7         for (int i = 1; i < arr.length; i++) {
8             // 记录要插入的数据
9             int tmp = arr[i];
10            // 从已经排序的序列最右边的开始比较, 找到比其小的数
11            int j = i;

```

```

12         while (j > 0 && tmp < arr[j - 1]) {
13             arr[j] = arr[j - 1];
14             j--;
15         }
16         // 存在比其小的数，插入
17         if (j != i) {
18             arr[j] = tmp;
19         }
20     }
21     return arr;
22 }
23 }
```

1.3 Analysis of algorithms

- Correctness: formal proof often by induction
- Running time: number of primitive computational steps
 - Not the same as time it takes to execute the algorithm
 - We want to measure that is independent of hardware
 - We want to know how running time scales with the size of the input
- Space: how much space is required by the algorithm

正确性：通常通过归纳法进行正式证明

运行时间：原始计算步骤的数量

- 与执行算法的时间不一样
- 我们希望测量的是独立于硬件的时间
- 我们想知道运行时间是如何随输入的大小而变化的

空间：算法所需的空间有多大

1.4 Analysis of insertion sort

Notation: $A[i, j]$ is the subarray of A that starts at position i and ends at position j

- **Correctness:** follows from the key observation that after loop i , the subarray $A[1, i]$ is sorted.
- **Running time:** number of primitive computational steps
- **Space:** in place algorithm (at most a constant number of elements of A are stored outside A at any time)

1.4.1 Correctness of insertion-sort

Notation: $A[i, j]$ is the subarray of A that starts at position i and ends at position j

Minor change in the pseudo code: in line 1, start from $i = 1$ rather than $i = 2$.

Claim 1.

Let $n \geq 1$ be a positive integer. For all $1 \leq i \leq n$, after the i -th loop, the subarray $A[1, i]$ is sorted.

Correctness of insertion-sort follows if we show Claim 1.

Proof of claim 1

By induction on i

- Base case: $i = 1$, trivial
- Induction hypothesis: assume that the statement is true for some $1 \leq i \leq n$
- Inductive step: show it true for $i + 1$
 - In loop $i + 1$, element $\text{key} = A[i + 1]$ is inserted into $A[1, i]$. By the induction hypothesis, $A[1, i]$ is sorted. Since
 1. key is inserted after the last element $A[l]$ such that $1 \leq l \leq i$ and $A[l] \leq \text{key}$
 2. all elements in $A[l + 1, i]$ are pushed one position to the right with their order preserved

The statement is true for $i+1$

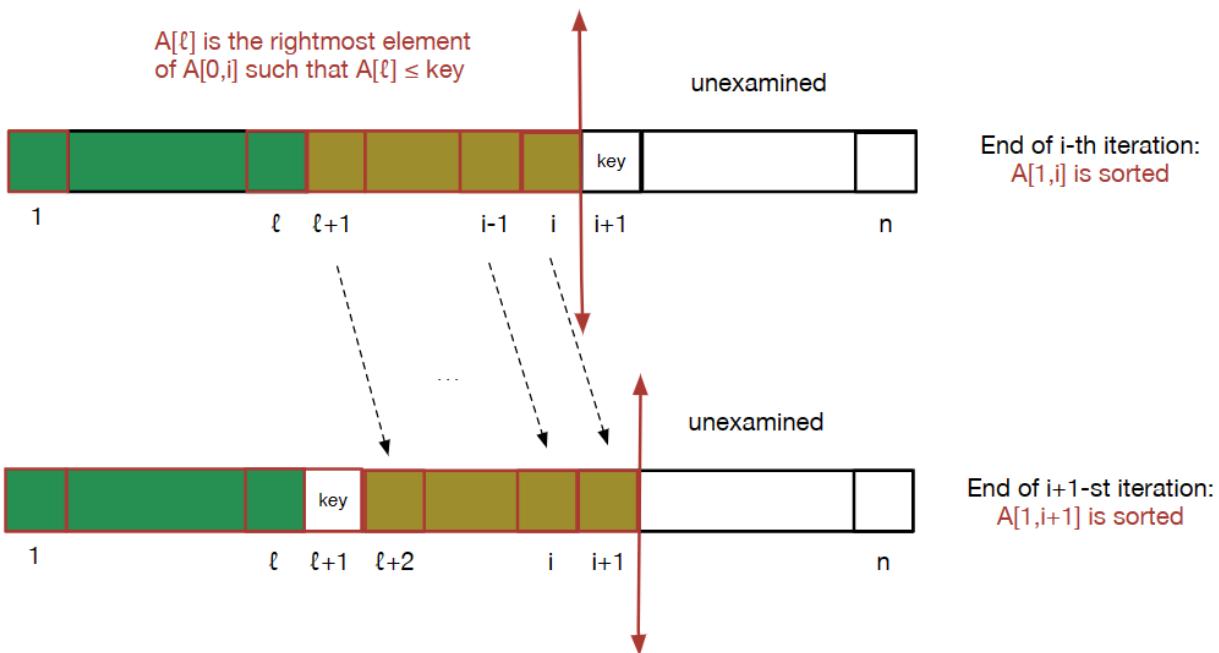
插入排序的正确性

基本情况: $i=1$

诱导假设: 假设当 $1 \leq i \leq n$, 插入排序是正确的

诱导步骤:

- 在循环 $i+1$ 中, 元素 $\text{key}=A[i+1]$ 被插入到 $A[1, i]$ 。根据诱导假设, $A[1, n]$ 是有序的, 因为:
 - key 是被插入最后一个小于等于 key 的元素后面
 - 所有在此被插入的元素后面的元素都被向右移动了一个单位, 他们的顺序保持不变
- 故该陈述是正确的



1.4.2 Running time $T(n)$ of insertion-sort

```

1  insertion-sort(A)
2      for i=2 to n do
3          key = A[i]
4          // Insert A[i] into the sorted subarray A[1..i-1]
5          j = i - 1
6          while j>0 and A[j]>key do
7              A[j+1] = A[j]
8              j = j - 1
9          end while
10         A[j+1] = key
11     end for

```

for $i = 2$ to n do	line 1
key = $A[i]$	line 2
//Insert $A[i]$ into the sorted subarray $A[1..i-1]$	line 3
$j = i - 1$	line 4
while $j > 0$ and $A[j] > \text{key}$ do	line 5
$A[j + 1] = A[j]$	line 6
$j = j - 1$	line 7
end while	
$A[j + 1] = \text{key}$	
end for	

- For $2 \leq i \leq n$, let $t_i =$ number of times that line 4 is executed. Then

$$T(n) = n + 3(n-1) + \sum_{i=2}^n t_i + 2 \sum_{i=2}^n (t_i - 1) = 3 \sum_{i=2}^n t_i + 2n - 1$$

第一项 n 代表执行line 1, 即for循环的次数。由于从2到n一共需要循环 $n-1$ 次, for循环额外需要一次判断以退出循环, 所以共 n 次。

第二项 $3(n-1)$ 代表执行line 2, 3, 7的次数。

第三项 $\sum_{i=2}^n t_i$ 代表执行line 4的次数, 其中 t_i 代表在每次loop中line 4执行的次数。

第四项 $2 \sum_{i=2}^n (t_i - 1)$ 代表执行line 5, 6的次数。

- Best-case running time:

- In each loop i , the line 4 only execute once. 即line 5, 6不运行
- So the total number of time that line 4 is executed is $n-1$
- $3(n-1) + 2n - 1 = 5n - 4$

- Worst-case running time:

- $T(n) = \frac{3n^2}{2} + \frac{7n}{2} - 4$

1.4.3 Worst-case analysis

Worst-case running time: largest possible running time of the algorithm over all inputs of a given size n

Why worst-case analysis?

- It gives well-defined computable bounds
- Average-case analysis can be tricky: how do we generate a “random” instance?

1.5 Efficiency of algorithms

1.5.1 Efficiency of insertion-sort and the brute force solution

Compare to brute force solution (蛮力解):

- At each step, generate a new permutation of the n integers
- If sorted, stop and output the permutation

Worst-case analysis: generate $n!$ permutations. Is brute force solution efficient?

- Efficiency relates to the performance of the algorithm as n grows.
- Stirling’s approximation formula: $n! \approx (\frac{n}{e})^n$
 - For $n = 10$, generate $3.67^{10} \geq 2^{10}$ permutations.
 - For $n = 100$, generate $36.7^{100} \geq 2^{700}$ permutations.

Brute force solution is not efficient.

1.5.2 Attempt 1

Definition 3

An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Caveat: fails to discuss the scaling properties of the algorithm; if the input size grows by a constant factor, we would like the running time $T(n)$ of the algorithm to increase by a constant factor as well.

Polynomial running times: on input of size n , $T(n)$ is at most $c \times n^d$ for $c, d > 0$ constants

- polynomial running times scale well
- the smaller the exponent of the polynomial the better

Definition 4

An algorithm is efficient if it has a polynomial running time.

Caveat

- What about huge constants in front of the leading term or large exponents?

However

- Small degree polynomial running time exist for most problems that can be solved in polynomial time
- Conversely, problems for which no polynomial-time algorithm is known tend to be very hard in practice
- So we can distinguish between easy and hard problems

1.6 Running time in terms of # primitive steps

To discuss this, we need a coarser(更粗糙的) classification of running times of algorithms; exact characterisations:

- are too detailed
- do not reveal similarities between running times in an immediate way as n grows large
- are often meaningless: pseudocode steps will expand by a constant factor that depends on the hardware.

1.7 Conclusion

In Chapter 1, we:

- Introduced the problem of sorting.
- Analysed insertion-sort
 - Worst-case running time: $T(n) = \frac{3n^2}{2} + \frac{7n}{2} - 4$

- Space: in-place algorithm
- Worst-case running time analysis: a reasonable measure of algorithmic efficiency
- Defined polynomial-time algorithms as “efficient”
- Argued that detailed characterisations of running times are not convenient for understanding scalability of algorithms

2. Asymptotic Notation, Mergesort and Recurrences

2.1 Asymptotic Notation

A framework that will allow us to compare the rate of growth of different running times as the input size n grows.

- We will express the running time as a function of the number of primitive steps; the latter is a function of the input size n .
- To compare functions expressing running times, we will ignore their low-order terms and focus solely on the highest-order term.

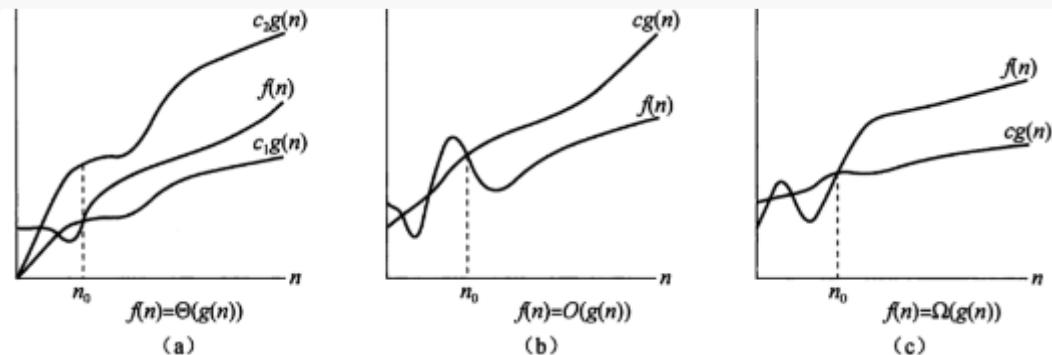
渐进符号 (Asymptotic Notation)

一个算法在编程语言的翻译后成为可执行程序。而计算机执行该算法的程序需要一定的时间，该时间的长短受很多方面的影响。因此我们不能仅仅依靠判断算法的程序执行时间来判断算法的好坏。

计算机的硬件影响是“死的”，而数据量的影响是“活的”。即计算机性能差异远不如数据量的多少所带来的对程序运行时间的影响大。**如果待处理的数据量很小，则无法区分算法的好坏，因此我们需要将待处理的数据量假设为特别大。**

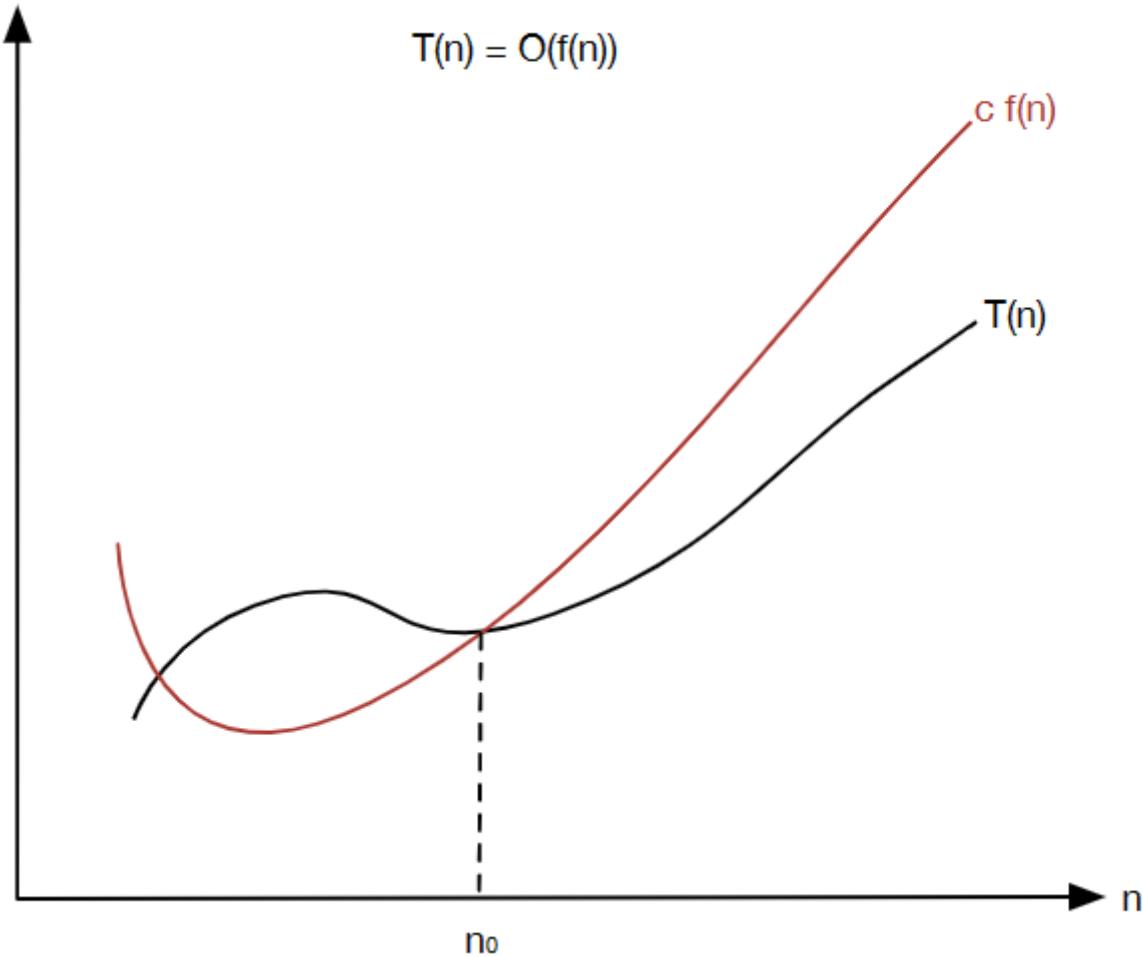
当输入规模 n 足够大时，我们可以忽略硬件差异带来的影响，只需要关注当输入规模无限增加时，**算法的运行时间是如何随着输入规模的变大而增加。**

因此定义了 O , Ω , Θ 等渐进符号，下图是对他们的图像描述。



2.1.1 Asymptotic upper bounds: Big- \mathcal{O} notation

We say that $T(n) = \mathcal{O}(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \times f(n)$



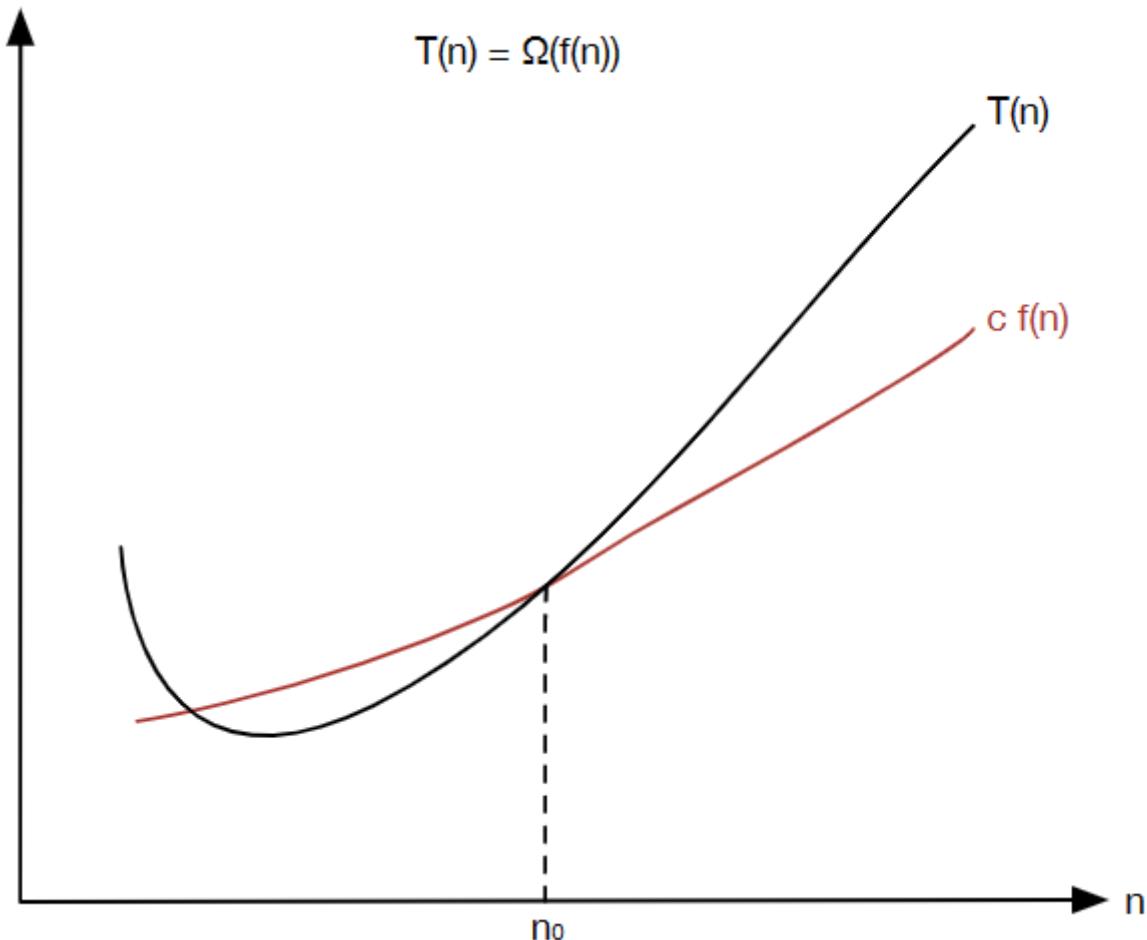
大O记号

\mathcal{O} 记号给出函数的渐近上界，即当函数在增长到一定程度时总小于等于一个特定函数的常数倍，即 $T(n) \leq c \times f(n)$ 。

相当于“ \leq ”

2.1.2 Asymptotic lower bounds: Big- Ω notation

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \times f(n)$



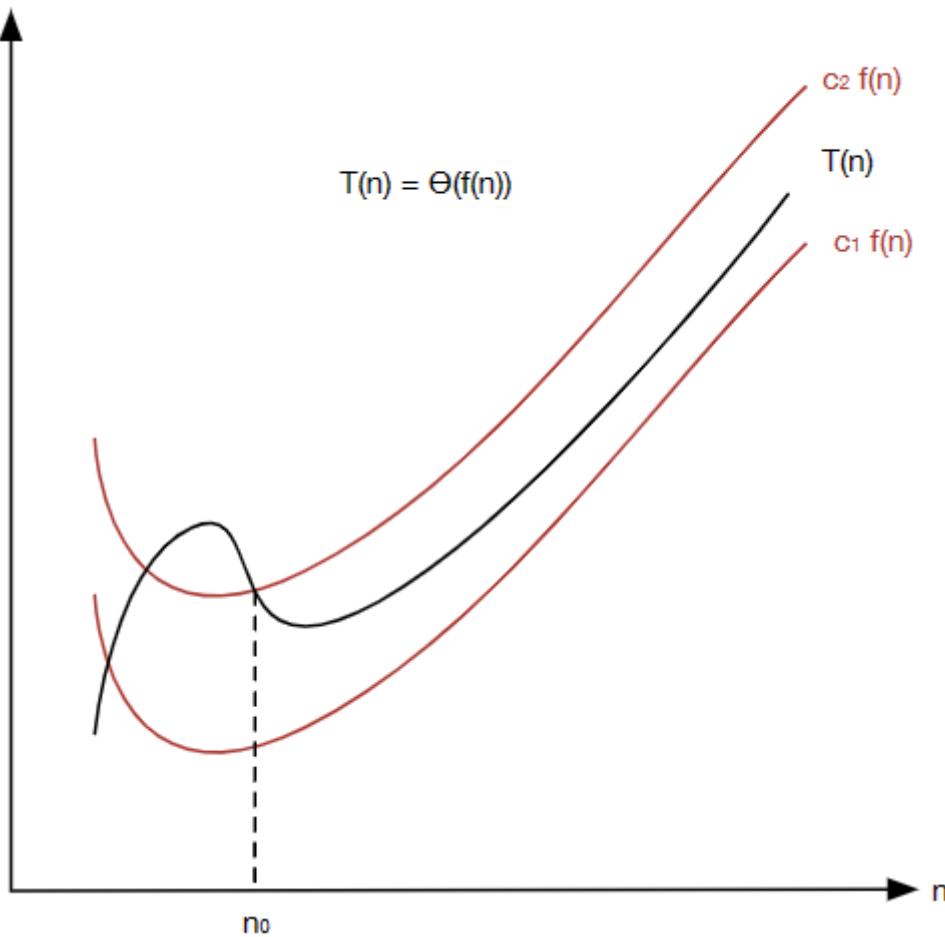
大 Ω 记号

Ω 记号给出函数的渐近下界，即当函数在增长到一定程度时总大于等于一个特定函数的常数倍，即 $T(n) \geq c \times f(n)$ 。

相当于“ \geq ”

2.1.3 Asymptotic tight bounds: Big- Θ notation

We say that $T(n) = \Theta(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$



大 Θ 记号

Θ 记号给出函数的渐近紧确界，即 $c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$ 。

相当于“=”

Equivalent definition

$$T(n) = \Theta(f(n)) \text{ if } T(n) = O(f(n)) \text{ and } T(n) = \Omega(f(n))$$

2.1.4 Asymptotic upper bounds that are not tight: little o

We say that $T(n) = o(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) < c \times f(n)$

- Intuitively, $T(n)$ becomes **insignificant** relative to $f(n)$ as $n \rightarrow \infty$
- Proof by showing that $\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists)

小 O 记号

O 记号给出函数的非紧上界，即当函数在增长到一定程度时总小于一个特定函数的常数倍，即 $T(n) < c \times f(n)$ 。

相当于“<”

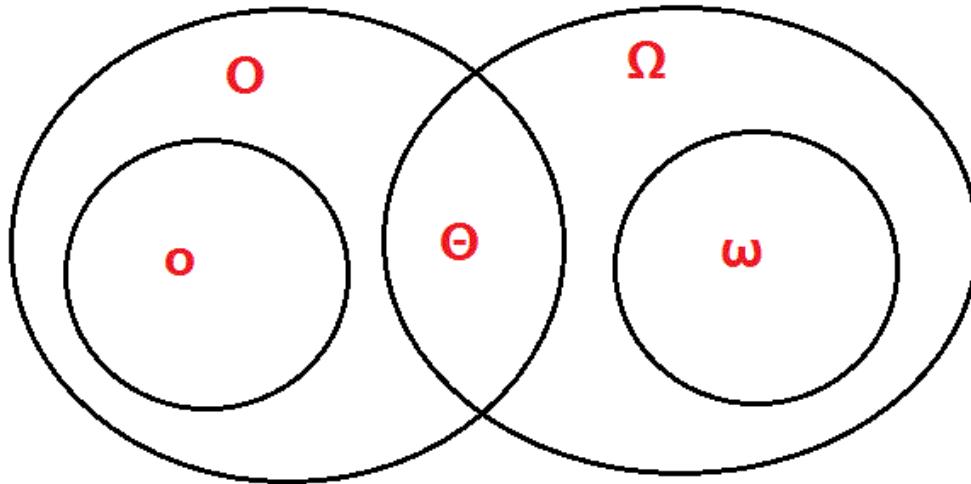
2.1.5 Asymptotic lower bounds that are not tight: little ω

We say that $T(n) = \omega(f(n))$ if, for any constant $c > 0$, there exists a constant $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) > c \times f(n)$

- Intuitively, $T(n)$ becomes arbitrary large relative to $f(n)$ as $n \rightarrow \infty$
- $T(n) = \omega(f(n))$ implies that $\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = \infty$, if the limit exists. Then $f(n) = o(T(n))$

2.1.6 Relationship between asymptotic notations

记号	含义	通俗理解
(1) Θ (西塔)	紧确界。	相当于"="
(2) O (大欧)	上界。	相当于" \leq "
(3) o (小欧)	非紧的上界。	相当于" $<$ "
(4) Ω (大欧米伽)	下界。	相当于" \geq "
(5) ω (小欧米伽)	非紧的下界。	相当于" $>$ "



2.1.7 Basic rules for omitting low order terms from functions

1. Ignore multiplicative factors: e.g., $10n^3$ becomes n^3
2. n^a dominates n^b if $a > b$: e.g., n^2 dominates n
3. Exponentials dominate polynomials: e.g., 2^n dominates n^4
4. Polynomials dominate logarithms: e.g., n dominates $\log n$

For large enough n ,

$$\log(n) < n < n \times \log(n) < n^2 < 2^n < 3^n < n^n$$

2.1.8 Properties of asymptotic growth rates

1. Transitivity

- 1.1 If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- 1.2 If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.
- 1.3 If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.

2. Sums of up to a constant number of functions

- 2.1 If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$.
- 2.2 Let k be a fixed constant, and let f_1, f_2, \dots, f_k, h be functions such that for all i , $f_i = O(h)$. Then $f_1 + f_2 + \dots + f_k = O(h)$.

3. Transpose symmetry

- $f = O(g)$ if and only if $g = \Omega(f)$.
- $f = o(g)$ if and only if $g = \omega(f)$.

2.2 The Divide & Conquer Principle

The divide & conquer principle 分裂与征服原则

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
 - Divide the input array into two lists of equal size.
- **Conquer** the subproblems by solving them recursively.
 - Sort each list recursively. (Stop when lists have size 2.)
- **Combine** the solutions to the subproblems to get the solution to the overall problem.
 - Merge the two sorted lists and output the sorted array.

2.3 Merge sort

归并排序 (Merge sort) 是建立在归并操作上的一种有效、稳定的排序算法，该算法是采用分治法(Divide and Conquer) 的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

当有 n 个记录时，需进行 $\log n$ 轮归并排序，每一轮归并，其比较次数不超过 n ，元素移动次数都是 n ，因此，归并排序的时间复杂度为 $O(n \log n)$ 。归并排序时需要和待排序记录个数相等的存储空间，所以空间复杂度为 $O(n)$ 。

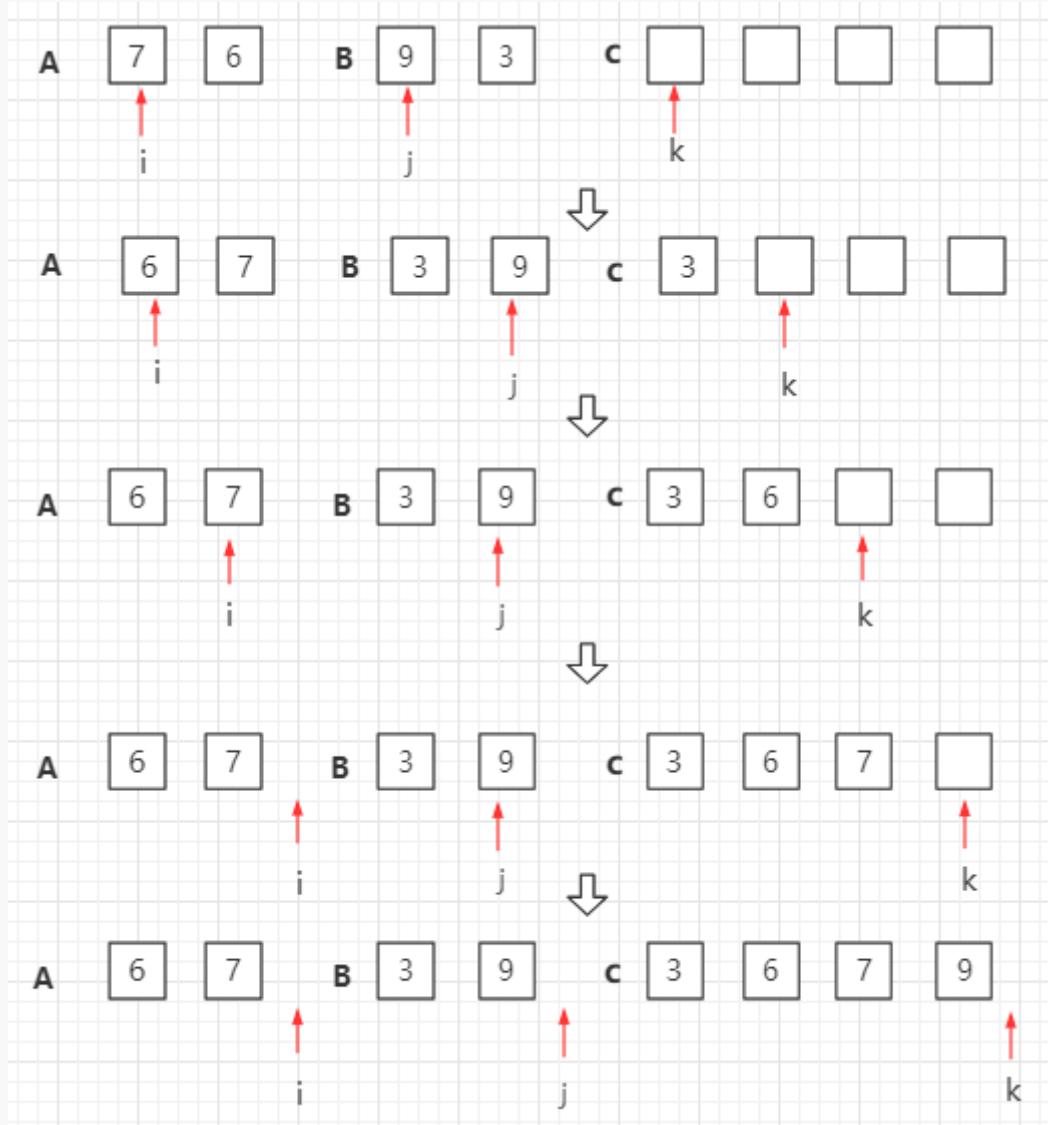
归并排序适用于数据量大，并且对稳定性有要求的场景。

过程：

归并排序是递归算法的一个实例，这个算法中基本的操作是合并两个已排序的数组，取两个输入数组 A 和 B，一个输出数组 C，以及三个计数器 i、j、k，它们初始位置置于对应数组的开始端。

$A[i]$ 和 $B[j]$ 中较小者拷贝到 C 中的下一个位置，相关计数器向前推进一步。

当两个输入数组有一个用完时候，则将另外一个数组中剩余部分拷贝到 C 中。



6 5 3 1 8 7 2 4

Remarks:

- Merge sort is a recursive procedure
- Initial call: mergesort(A, 1, n)

- Subroutine `merge` merges two sorted lists of size $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$ into one sorted list of size n

Intuition: to merge two sorted lists of size $n/2$ repeatedly

- compare the two items in the front of the two lists
- extract the smaller item and append it to the output
- update the front of the list from which the item was extracted

2.3.1 Codes

Pseudo Code

```

1  merge(A, left, right, mid)
2      L = A[left, mid]
3      R = A[mid+1, right]
4      Maintain two pointers p_l, p_r, initialised to point to the first elements
5      of L,R, respectively
6      while both lists are non-empty do
7          let x,y be the elements pointed to by p_l, p_r
8          compare x,y and append the smaller to the output
9          advance the pointer in the list with the smaller of x,y
10     end while
11    append the remainder of the non-empty list to the output
12 mergesort(A, left, right)
13     if right == left then return
14     end if
15     mid = left + lower_bound((right-left)/2) //下取整(right-left)/2
16     mergesort(A, left, mid)
17     mergesort(A, mid+1, right)
18     merge(A, left, right, mid)
```

C++

```

1 // C++ program for Merge Sort
2 #include <iostream>
3 using namespace std;
4
5 // Merges two subarrays of array[].
6 // First subarray is arr[begin..mid]
7 // Second subarray is arr[mid+1..end]
8 void merge(int array[], int const left, int const mid,
9             int const right)
10 {
11     auto const subArrayOne = mid - left + 1;
12     auto const subArrayTwo = right - mid;
13
14     // Create temp arrays
15     auto *leftArray = new int[subArrayOne],
16         *rightArray = new int[subArrayTwo];
17 }
```

```

18 // Copy data to temp arrays leftArray[] and rightArray[]
19 for (auto i = 0; i < subArrayOne; i++)
20     leftArray[i] = array[left + i];
21 for (auto j = 0; j < subArrayTwo; j++)
22     rightArray[j] = array[mid + 1 + j];
23
24 auto indexOfSubArrayOne
25     = 0, // Initial index of first sub-array
26     indexOfSubArrayTwo
27     = 0; // Initial index of second sub-array
28 int indexOfMergedArray
29     = left; // Initial index of merged array
30
31 // Merge the temp arrays back into array[left..right]
32 while (indexOfSubArrayOne < subArrayOne
33     && indexOfSubArrayTwo < subArrayTwo) {
34     if (leftArray[indexOfSubArrayOne]
35         <= rightArray[indexOfSubArrayTwo]) {
36         array[indexOfMergedArray]
37             = leftArray[indexOfSubArrayOne];
38         indexOfSubArrayOne++;
39     }
40     else {
41         array[indexOfMergedArray]
42             = rightArray[indexOfSubArrayTwo];
43         indexOfSubArrayTwo++;
44     }
45     indexOfMergedArray++;
46 }
47 // Copy the remaining elements of
48 // left[], if there are any
49 while (indexOfSubArrayOne < subArrayOne) {
50     array[indexOfMergedArray]
51         = leftArray[indexOfSubArrayOne];
52     indexOfSubArrayOne++;
53     indexOfMergedArray++;
54 }
55 // Copy the remaining elements of
56 // right[], if there are any
57 while (indexOfSubArrayTwo < subArrayTwo) {
58     array[indexOfMergedArray]
59         = rightArray[indexOfSubArrayTwo];
60     indexOfSubArrayTwo++;
61     indexOfMergedArray++;
62 }
63 delete[] leftArray;
64 delete[] rightArray;
65 }
66
67 // begin is for left index and end is
68 // right index of the sub-array

```

```

69 // of arr to be sorted */
70 void mergeSort(int array[], int const begin, int const end)
71 {
72     if (begin >= end)
73         return; // Returns recursively
74
75     auto mid = begin + (end - begin) / 2;
76     mergeSort(array, begin, mid);
77     mergeSort(array, mid + 1, end);
78     merge(array, begin, mid, end);
79 }
80
81 // UTILITY FUNCTIONS
82 // Function to print an array
83 void printArray(int A[], int size)
84 {
85     for (auto i = 0; i < size; i++)
86         cout << A[i] << " ";
87 }
88
89 // Driver code
90 int main()
91 {
92     int arr[] = { 12, 11, 13, 5, 6, 7 };
93     auto arr_size = sizeof(arr) / sizeof(arr[0]);
94
95     cout << "Given array is \n";
96     printArray(arr, arr_size);
97
98     mergeSort(arr, 0, arr_size - 1);
99
100    cout << "\nSorted array is \n";
101    printArray(arr, arr_size);
102    return 0;
103 }
```

Python

```

1
2 # Python program for implementation of MergeSort
3 def mergeSort(arr):
4     if len(arr) > 1:
5
6         # Finding the mid of the array
7         mid = len(arr)//2
8
9         # Dividing the array elements
10        L = arr[:mid]
11
12        # into 2 halves
13        R = arr[mid:]
```

```

14
15      # Sorting the first half
16      mergeSort(L)
17
18      # Sorting the second half
19      mergeSort(R)
20
21      i = j = k = 0
22
23      # Copy data to temp arrays L[] and R[]
24      while i < len(L) and j < len(R):
25          if L[i] <= R[j]:
26              arr[k] = L[i]
27              i += 1
28          else:
29              arr[k] = R[j]
30              j += 1
31          k += 1
32
33      # Checking if any element was left
34      while i < len(L):
35          arr[k] = L[i]
36          i += 1
37          k += 1
38
39      while j < len(R):
40          arr[k] = R[j]
41          j += 1
42          k += 1
43
44 # Code to print the list
45
46
47 def printList(arr):
48     for i in range(len(arr)):
49         print(arr[i], end=" ")
50     print()
51
52
53 # Driver Code
54 if __name__ == '__main__':
55     arr = [12, 11, 13, 5, 6, 7]
56     print("Given array is", end="\n")
57     printList(arr)
58     mergeSort(arr)
59     print("Sorted array is: ", end="\n")
60     printList(arr)

```

Java

```
1 /* Java program for Merge Sort */
```

```

2  class MergeSort {
3      // Merges two subarrays of arr[].
4      // First subarray is arr[l..m]
5      // Second subarray is arr[m+1..r]
6      void merge(int arr[], int l, int m, int r)
7      {
8          // Find sizes of two subarrays to be merged
9          int n1 = m - l + 1;
10         int n2 = r - m;
11
12         /* Create temp arrays */
13         int L[] = new int[n1];
14         int R[] = new int[n2];
15
16         /*Copy data to temp arrays*/
17         for (int i = 0; i < n1; ++i)
18             L[i] = arr[l + i];
19         for (int j = 0; j < n2; ++j)
20             R[j] = arr[m + 1 + j];
21
22         /* Merge the temp arrays */
23
24         // Initial indexes of first and second subarrays
25         int i = 0, j = 0;
26
27         // Initial index of merged subarray array
28         int k = l;
29         while (i < n1 && j < n2) {
30             if (L[i] <= R[j]) {
31                 arr[k] = L[i];
32                 i++;
33             }
34             else {
35                 arr[k] = R[j];
36                 j++;
37             }
38             k++;
39         }
40
41         /* Copy remaining elements of L[] if any */
42         while (i < n1) {
43             arr[k] = L[i];
44             i++;
45             k++;
46         }
47
48         /* Copy remaining elements of R[] if any */
49         while (j < n2) {
50             arr[k] = R[j];
51             j++;
52             k++;
53         }
54     }
55 }
```

```

53     }
54 }
55
56 // Main function that sorts arr[l..r] using
57 // merge()
58 void sort(int arr[], int l, int r)
59 {
60     if (l < r) {
61         // Find the middle point
62         int m = l + (r - l) / 2;
63
64         // Sort first and second halves
65         sort(arr, l, m);
66         sort(arr, m + 1, r);
67
68         // Merge the sorted halves
69         merge(arr, l, m, r);
70     }
71 }
72
73 /* A utility function to print array of size n */
74 static void printArray(int arr[])
75 {
76     int n = arr.length;
77     for (int i = 0; i < n; ++i)
78         System.out.print(arr[i] + " ");
79     System.out.println();
80 }
81
82 // Driver code
83 public static void main(String args[])
84 {
85     int arr[] = { 12, 11, 13, 5, 6, 7 };
86
87     System.out.println("Given Array");
88     printArray(arr);
89
90     MergeSort ob = new MergeSort();
91     ob.sort(arr, 0, arr.length - 1);
92
93     System.out.println("\nSorted array");
94     printArray(arr);
95 }
96 }
```

2.3.2 Analysis of mergesort

1. Correctness: by induction on the size of the two lists

For simplicity, assume $n = 2^k$ for integer $k \geq 0$

We will use induction on k .

- o Base case: For $k = 0$, the input consists of 1 item; mergesort return the item
- o Induction hypothesis: For $k \geq 0$, assume that mergesort correctly sorts any list of size 2^k
- o Induction step: We will show that mergesort correctly sorts any list A of size 2^{k+1}

From the pseudocode of mergesort, we have:

- Line 3: mid takes the value 2^k
- Line 4: mergesort($A, 1, 2^k$) correctly sorts the leftmost half of the input, by the induction hypothesis
- Line 5: mergesort($A, 2^k+1, 2^{k+1}$) correctly sorts the rightmost half of the input, by the induction hypothesis
- Line 6: merge correctly merges its two sorted input lists into one sorted output of size $2^k + 2^k$

\rightarrow mergesort correctly sorts any input of size 2^{k+1}

2. Running time

merge ($A, left, right, mid$)

$L = A[left, mid]$ →not a primitive computational step!

$R = A[mid + 1, right]$ →not a primitive computational step!

Maintain two pointers p_L, p_R initialized to point to the first elements of L, R , respectively

while both lists are nonempty **do**

 Let x, y be the elements pointed to by p_L, p_R

 Compare x, y and append the smaller to the output

 Advance the pointer in the list with the smaller of x, y

end while

Append the remainder of the non-empty list to the output.

- o Suppose L, R have $n/2$ elements each
- o How many iterations before all elements from both lists have been appended to the output? At most $n-1$.
- o How much work within each iteration? constant.

\rightarrow merge takes $O(n)$ time to merge L, R

3. Space

- Extra $\Theta(n)$ space to store L, R (the output of merge is stored directly in A)

2.4 Solving recurrences and running time of mergesort

解决递归问题和合并排序的运行时间

2.4.1 Solving recurrences, method 1: recursion trees

The recursion trees (递归树) consists of three steps:

1. Analyse the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum the work spent over all levels of recursion

2.4.2 A general recurrence and its solution

The running time of many recursive algorithms can be expressed by the following recurrence

$$T(n) = aT(n/b) + cn^k, \text{ for } a, c > 0, b > 1, k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
 - b is the factor by which the size of each subproblem shrinks
- > at level i, there are a^i subproblems, each of size n/b^i
- > each subproblem at level i requires $c(n/b^i)^k$ work
- The height of the tree is $\log_b n$ levels
- > Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum_{i=0}^{\log_b n} (\frac{a}{b^k})^i$

2.4.3 Solving recurrences, method 2: Master theorem

Theorem 6 (Master Theorem)

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ for some constants $a > 0, b > 1, k \geq 0$, then

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

2.4.4 Solving recurrences, method 3: the substitution method

The technique consists of two steps

1. Guess a bound
2. Use (strong) induction to prove that the guess is correct

1. Simple induction: the induction step at n requires that the inductive hypothesis holds at step $n - 1$
 2. Strong induction: is just a variant of simple induction where the induction step at n requires that inductive hypothesis holds at all previous steps $1, 2, \dots, n - 1$

2.5 Conclusion

In Chapter 2, we discussed:

- **Asymptotic notation** ($O, \Omega, \Theta, o, \omega$)
- **The divide & conquer principle**
 - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
 - Divide the input array into two lists of equal size.
 - **Conquer** the subproblems by solving them recursively.
 - Sort each list recursively. (Stop when lists have size 2.)
 - **Combine** the solutions to the subproblems to get the solution to the overall problem.
 - Merge the two sorted lists and output the sorted array.
- **Application: mergesort**

```

o   1 merge(A, left, right, mid)
    2     L = A[left, mid]
    3     R = A[mid+1, right]
    4     Maintain two pointers p_l, p_r, initialised to point to the first
        elements of L,R, respectively
    5     while both lists are non-empty do
        6         let x,y be the elements pointed to by p_l, p_r
        7         compare x,y and append the smaller to the output
        8         advance the pointer in the list with the smaller of x,y
    9     end while
    10    append the remainder of the non-empty list to the output
    11 mergesort(A, left, right)
    12    if right == left then return
    13    end if
    14    mid = left + lower_bound((right-left)/2) //下取整(right-left)/2
    15    mergesort(A, left, mid)
    16    mergesort(A, mid+1, right)
  
```

17 | merge(A, left, right, mid)

- Solving recurrences

- Recursion trees
- Master theorem
- Substitution method

3. Divide & conquer algorithms: fast int/matrix multiplication

3.1 Binary search

- Input:
 1. sorted list A of n integers
 2. integer x
- Output:
 - index j such that $1 \leq j \leq n$ and $A[j] = x$, or
 - no if x is not in A

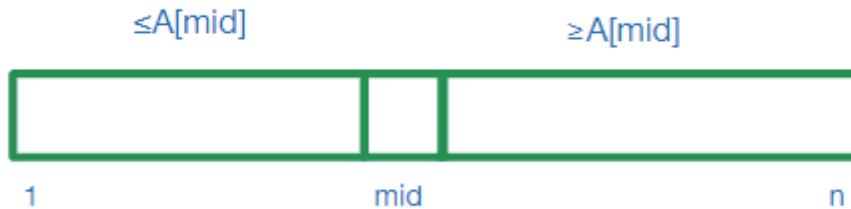
Example: $A = [0, 2, 3, 5, 6, 7, 9, 11, 13]$, $n = 9$, $x = 7$

Idea: use the fact that the array is **sorted** and probe specific entries in the array

First, probe the middle entry, let $mid = \lceil n/2 \rceil$.

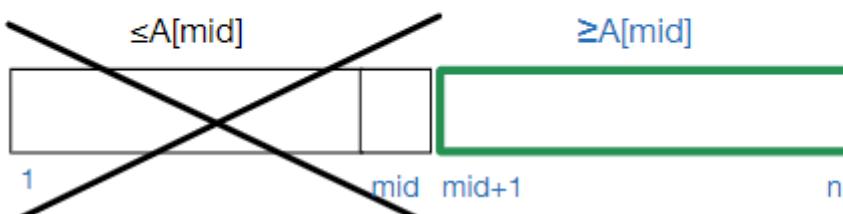
- If $x == A[mid]$, return mid
- If $x < A[mid]$, looking for x in range $A[1, mid - 1]$
- Else if $x > A[mid]$, looking for x in range $A[mid + 1, n]$

Initially, the entire array is “active”, that is, x might be anywhere in the array.



Suppose $x > A[\text{mid}]$.

Then the active area of the array, where x might be, is to the right of mid.



3.1.1 Pseudo Code for Binary Search

```

1 | binarysearch(A, left, right)
2 |   mid = left + (right - left)/2e
3 |   if x == A[mid] then
4 |     return mid
5 |   else if right == left then
6 |     return no
7 |   else if x > A[mid] then
8 |     left = mid + 1
9 |   else right = mid - 1
10|  end if

```

3.1.2 Binary Search Running Time – Linear

Observation: At each step there is a region of A where x could be and we shrink the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements
- If n is even, then we are throwing away at least $n/2$ elements

Hence the recurrence for the running time is:

$$T(n) \leq T(n/2) + O(1)$$

3.1.3 Binary Search Running Time – Sublinear

Here are two ways to argue about the running time:

1. Master theorem:

- If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ for some constants $a > 0, b > 1, k \geq 0$, then

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

- $b = 2, a = 1, k = 0 \rightarrow T(n) = O(\log n)$

2. We can reason as follows: starting with an array of size n ,

- After k probes, the array has size at most $\frac{n}{2^k}$ (every time we probe an entry, the active portion of array halves)
- After $k = \log n$ probes, the array has **constant size**. We can now search **linearly** for x in the constant size array
- We spend **constant** work to halve the array. Thus the total work spent is $O(\log n)$

3.1.4 Conclusion

- The right data structure can improve the running time of the algorithm significantly
 - What if we used a linked list to store the input?
 - Arrays allow for random access of their elements: given an index, we can read any entry in time $O(1)$ (constant time)
- In general, we obtain running time $O(\log n)$ when the algorithm does a constant amount of work to throw away a constant fraction of the input.

3.1.5 Binary Search from EEEN30002 Numerical Analysis

From EEEN30002 Numerical Analysis

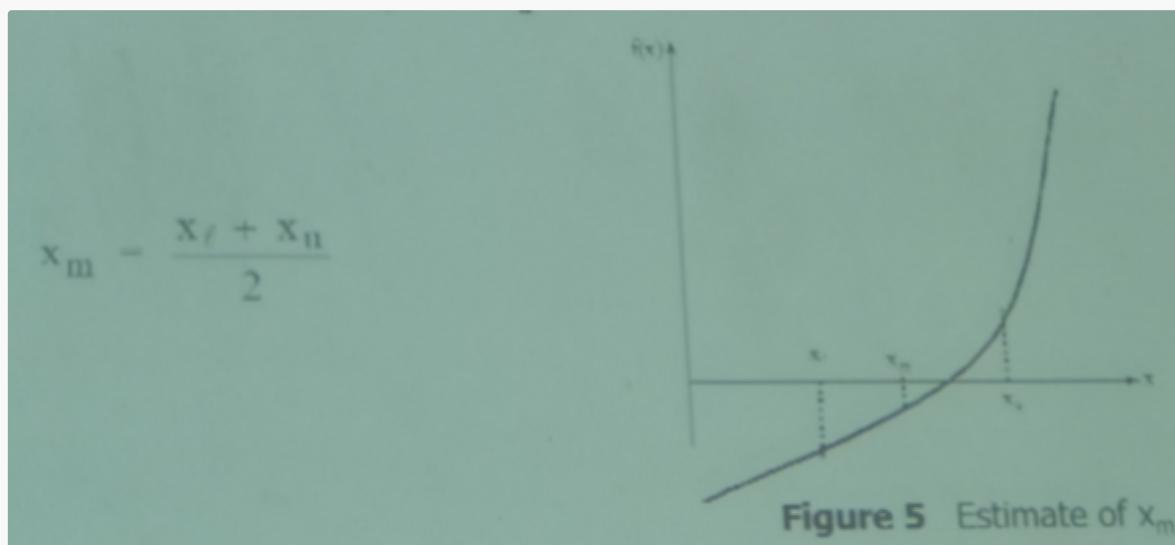
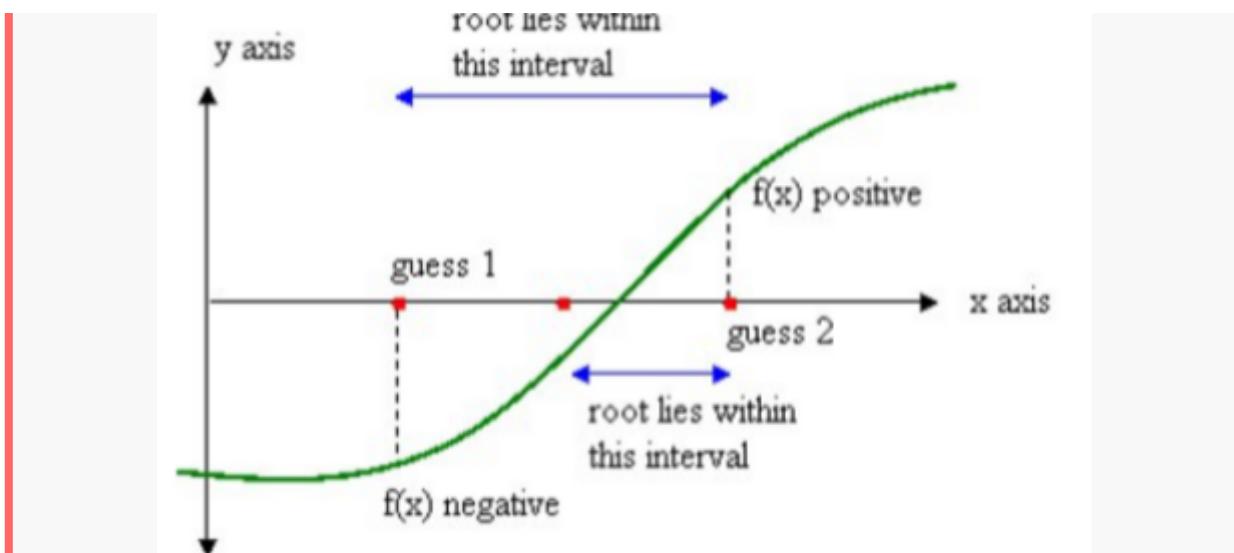
💡 将复杂的非线性方程问题简化，将非线性方程分成几个区间，对每个区间分别求解，选取一个近似区间使用迭代法逼近真实解

步骤1

在函数上选取两个点 a, b , 确保 $f(a) * f(b) < 0$ 。即 ab 两点在函数图像上的分布为：一个在函数根的左侧，一个在函数根的右侧。

步骤2

设近似跟 $x_m = (a + b)/2$



步骤3

判断：

- 如果 $f(a) * f(x_m) < 0$, 则函数的真实解在a和m之间
 - $a = a; b = x_m$
- 如果 $f(a) * f(x_m) > 0$, 则函数的真实解在m和b之间
 - $a = x_m; b = b$
- 如果 $f(a) * f(x_m) = 0$, 则函数的真实解为a, 停止算法

步骤4

寻找新的近似根 $x_m = (a + b)/2$

计算绝对相对估计误差 $e_k \leq 1/2(a_{prev} - b_{prev}) = (1/2)^{k+1} * (a_0 - b_0)$

💡 a_{prev}, b_{prev} 为 a 和 b 的上次的值, a_0, b_0 为 a 和 b 第一次估计的值

绝对误差 e_k 应该小于上一次计算出的误差，同时应该等于 $(1/2)^{k+1} * (a_0 - b_0)$ ，因为二分，所以误差值可以计算

步骤5

将绝对相对误差 e_k 和事先设定的 ϵ 值做比较，如果 $e_k < \epsilon$ ，则算法停止，否则算法继续

根据初始的ab和 ϵ 值判断需要多少个循环才能满足条件

假设我们需要 $e_k < \epsilon$

$$(1/2)^{k+1} * (a_0 - b_0) < \epsilon$$

$$\text{可得 } k + 1 > (\ln(a_0 - b_0) - \ln(\epsilon)) / (\ln(2))$$

MATLAB

```

1 %% bisection algorithm to find sqrt(2)
2
3 xmin = 0;
4 xmax = 2;
5 fmin = xmin^2-2;
6 fmax = xmax^2-2;
7
8 %% choose epsilon
9 epsilon = 10^-3/2;
10
11 for k = 0 : ceil((log(2)-log(epsilon))/log(2) -1),
12
13     xhat = (xmin+xmax)/2;
14     fhat = xhat^2-2;
15
16     disp([k xmin xmax xhat abs(xhat-sqrt(2)) (xmax-xmin)/2 fmin fmax fhat])
17
18     if fhat*fmin > 0,
19         xmin = xhat; fmin = fhat;
20     else
21         xmax = xhat; fmax = fhat;
22     end
23
24 end

```

C++

```

1 #include <iostream>
2 using namespace std;
3 #define EP 0.01
4 // An example function whose solution is determined using

```

```

5 // Bisection Method. The function is x^3 - x^2 + 2
6 double solution(double x) {
7     return x*x*x - x*x + 2;
8 }
9 // Prints root of solution(x) with error in EPSILON
10 void bisection(double a, double b) {
11     if (solution(a) * solution(b) >= 0) {
12         cout << "You have not assumed right a and b\n";
13         return;
14     }
15     double c = a;
16     while ((b-a) >= EP) {
17         // Find middle point
18         c = (a+b)/2;
19         // Check if middle point is root
20         if (solution(c) == 0.0)
21             break;
22         // Decide the side to repeat the steps
23         else if (solution(c)*solution(a) < 0)
24             b = c;
25         else
26             a = c;
27     }
28     cout << "The value of root is : " << c;
29 }
30 // main function
31 int main() {
32     double a = -500, b = 100;
33     bisection(a, b);
34     return 0;
35 }
```

3.2 Integer multiplication

How we multiply two integers x and y ?

- Elementary school method: compute a partial product by multiplying every digit of y separately with x and then add up all the partial products.

- Remark: this method works the same in base 10 or base 2

Examples: $(12)_{10} \cdot (11)_{10}$ and $(1100)_2 \cdot (1011)_2$

$$\begin{array}{r}
 \begin{array}{r}
 12 \\
 \times 11 \\
 \hline
 12 \\
 + 12 \\
 \hline
 132
 \end{array}
 \qquad
 \begin{array}{r}
 1100 \\
 \times 1011 \\
 \hline
 1100 \\
 1100 \\
 0000 \\
 + 1100 \\
 \hline
 10000100
 \end{array}
 \end{array}$$

Elementary algorithm running time:

- $O(n)$ time to compute a partial product
- $O(n)$ time to combine it in a running sum of all partial products so far
- Hence the total number of operations is $O(n^2)$

3.2.1 A first divide & conquer approach

4. Quicksort and balls-in-bins

5. Graphs, Breadth-First Search (BFS) and its applications

5.1 Graphs

5.1.1 Definitions of graphs

A directed graph consists of a finite set V of vertices (or nodes) and a set E of directed edges. A directed edge is an ordered pair of vertices (u, v)

- In mathematical terms, a directed graph $G = (V, E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V

- An undirected graph is the special case of a directed graph where $(u, v) \in E$ if and only if $(v, u) \in E$. In this case, an edge may be indicated as the unordered pair u, v
- Notational conventions: $|V| = n$, $|E| = m$

Undirected graphs:

$$\deg(v) \triangleq \text{number of edges incident to } v$$

Directed graphs:

$$\text{indeg}(v) \triangleq \text{number of edges entering } v$$

$$\text{outdeg}(v) \triangleq \text{number of edges leaving } v$$

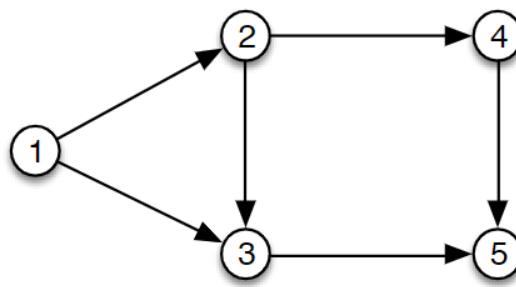
Example:

Circles denote **vertices** (nodes).

Lines denote **edges** connecting vertices.

Arrows on lines indicate the direction along which the edge may be traversed.

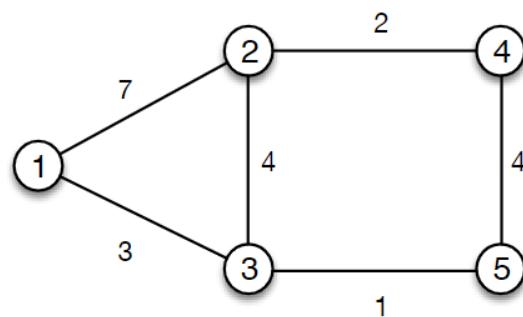
A directed, unweighted graph G
(default edge weight $w(e) = 1$)



$$\text{indeg}(1) = 0 \\ \text{indeg}(3) = 2$$

$$\text{outdeg}(1) = 2 \\ \text{outdeg}(3) = 1$$

An undirected, weighted graph G'



$$\deg(1) = 2 \\ \deg(3) = 3$$

Examples of graphs (networks)

- Transportation networks: e.g., nodes are cities, edges (potentially **weighted**) are highways connecting the cities
- Information networks: e.g., the WWW can be modeled as a directed graph
- Wireless networks: nodes are devices sitting at locations in physical space and there is an edge from u to v if v is close enough to u to hear from it.
- Social networks: e.g., nodes are people, edges represent friendship
- Dependency networks: e.g., given a list of functions in a large program, find an order to test the functions

- A **path** is a sequence of vertices (x_1, x_2, \dots, x_n) such that consecutive vertices are adjacent, that is, there exists an edge $(x_i, x_{i+1}) \in E$ for all $1 \leq i \leq n-1$
 - Example: $(1, 2, 3, 2, 4)$ in G' is a path
- A **path is simple** when all vertices are distinct
 - Example: $(1, 2, 4)$ in G' is a simple path
- A **simple cycle** is a simple path that ends where it starts
 - Example: $(1, 2, 3, 1)$ in G' is a simple cycle
- The distance from u to v , denoted by $dist(u, v)$, is the length of the shortest path from u to v
 - Unweighted graphs:
$$\text{length of path } P \triangleq \text{number of edges on } P$$
 - Weighted graphs: Not yet covered
 - Shortest path from u to v : a path of minimum length among all paths from u to v
- An undirected graph is **connected** when there is a path between every pair of vertices
 - In example, graph G is connected
- The **connected component** of a node u is the set of all nodes in the graph reachable by a path from u
 - In example, the connected component of 1 in G' is $1, 2, 3, 4, 5$
- A directed graph is **strongly connected** if for every pair of vertices u, v , there is a path from u to v and from v to u
- The **strongly connected component** of a node u in a directed graph is the set of nodes v in the graph such that there is a path from u to v and from v to u
 - In example, the strongly connected component of 1 in G is 1

图论:

图G定义为V和E的集合 $G=\{V, E\}$, 其中V表示图中的所有的顶点集合, E表示的是G中的所有的边的集合。图按照E中的元素是否有方向, 分为有向图和无向图。

5.1.2 Trees and tree properties

- A **tree** is a **connected acyclic** (连通的无周期性) graph (undirected graphs). Or; A **rooted** graph such that there is a unique path from the root to any other vertex (all graphs)

- A tree is the most widely used special type of graph: it is the minimal connected graph
- Let G be an undirected graph. Any two of the following properties imply the third property, and that G is a tree
 - G is connected
 - G is acyclic
 - $|E| = |V| - 1$

5.1.3 Matching and bipartite graphs

Bipartite graphs

Bipartite graphs: (两分图) vertices can be split into two subsets such that there are no edges between vertices in the same subset

- Applications: social networks, coding theory
- Notation: $G = (X \cup Y, E)$, where $X \cup Y$ is the set of vertices in G and every edge in E has one endpoint in X and one endpoint in Y

Matching

Matching: a subset of the edges where every node appears at most once

Goal: find a one-to-one matching (also called, a perfect matching) of people to jobs, if one exists

5.1.4 Degree Theorem

In any graph, the sum of the degrees of all vertices is equal to twice the number of the edges

Proof:

Every edge is incident to two vertices, thus contributes twice to the total sum of the degrees. (Summing the degrees of all vertices simply counts all instances of some edge being incident to some vertex)

5.1.5 Running time of graph algorithms

Input: graph $G = (V, E), |V| = n, |E| = m$

- Linear graph algorithms run in $O(n + m)$ time
 - Lower bound on m (assume connected graphs)
 - Upper bound on m (assume simple graphs)
- More general running times: the best performance is determined by the relationship between n and m

- For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

5.2 Representing graphs

5.2.1 Adjacency matrix

We want to represent a graph $G = (V, E)$, $|V| = n$, $|E| = m$

Adjacency matrix for G : an $n \times m$ matrix A such that

$$A[i, j] = \begin{cases} 1, & \text{if edge } (i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Space required for adjacency matrix A : $\Theta(n^2)$

Space requirements can be improved if the graph is

- undirected: A is symmetric \rightarrow only store its upper triangle
- unweighted: only need 1 bit per entry

Pro and Cons

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ by its adjacency matrix has the following pro and cons

Advantages:

- check whether edge $e \in E$ in constant time
- easy to adapt if the graph is weighted
- suitable for dense graphs where $m = \Theta(n^2)$

Drawbacks:

- requires $\Omega(n^2)$ space even if G is sparse ($m = o(n^2)$)
- does not allow for linear time algorithms in sparse graphs (at least when all matrix entries must be examined)

5.2.2 Adjacency list

An alternative representation for graph $G = (V, E)$, $|V| = n$, $|E| = m$ is as follows.

Adjacency list: recall that vertex j is **adjacent** to vertex i if $(i, j) \in E$; then the adjacency list for vertex i is simply the list of vertices adjacent to vertex i .

The adjacency list representation of a graph consists of an array A with n entries such that $A[i]$ points to the adjacency list of vertex i .

Need

- ▶ an array of n pointers: $O(n)$ space; plus
 - ▶ the sum of the lengths of all adjacency lists:
 - ▶ **directed** G : maintain the list of vertices with incoming edges from v and the list of vertices with outgoing edges to v .
 - ▶ length of adjacency lists of $v = \text{outdeg}(v) + \text{indeg}(v)$
 - ▶ length of all adjacency lists = $\sum_v \text{outdeg}(v) + \text{indeg}(v) = 2m$
 - ▶ **undirected** G : maintain the list of vertices adjacent to v
 - ▶ length of adjacency list of $v = \deg(v)$
 - ▶ length of all adjacency lists = $\sum_v \deg(v) = 2m$.
- ⇒ Total space: $O(n + m)$

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ using adjacency lists has the following pros/cons.

Advantages:

- ▶ allocates no unnecessary space: $O(n + m)$ space to represent a graph on n vertices and m edges
- ▶ suitable for linear or near-linear time algorithms

Drawbacks:

- ▶ searching for an edge can take $O(n)$ time

5.2.3 Adjacency list vs. Adjacency matrix

We prefer **adjacency matrix** when

- We need determine quickly whether an edge is in the graph
- The graph is dense
- The graph is small (it is a simpler representation)

We use **adjacency list** otherwise

5.3 Breadth-first search (BFS)

广度优先搜索算法 (Breadth-First-Search)

简单的说，BFS是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止。

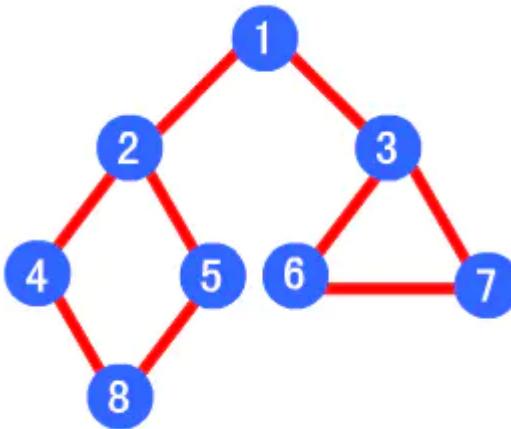
BFS同样属于盲目搜索。

一般用队列数据结构来辅助实现BFS算法。

算法步骤：

1. 首先将根节点放入队列中。
2. 从队列中取出第一个节点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果。否则将它所有尚未检验过的直接子节点加入队列中。
3. 若队列为空，表示整张图都检查过了一一亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。
4. 重复步骤2。

如下图，其广度优先算法的遍历顺序为：1->2->3->4->5->6->7->8



5.4 Applications of BFS

6 Depth-first search (DFS) and its applications

6.1 Depth-first search (DFS)

Depth-first search (DFS): starting from a vertex s , explore the graph as deeply as possible, then backtrack.

1. Try the first edge out of s , towards some node v
2. Continue from v until you reach a **dead end**, that is a node whose neighbours have all been explored
3. **Backtrack** to the first node with an unexplored neighbour and repeat step 2

深度优先搜索算法 (Depth-First-Search)

深度优先搜索算法 (Depth-First-Search)，是搜索算法的一种。它沿着树的深度遍历树的节点，尽可能深的搜索树的分支。

当节点 v 的所有边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。一般用堆数据结构来辅助实现DFS算法。

深度优先遍历图算法步骤：

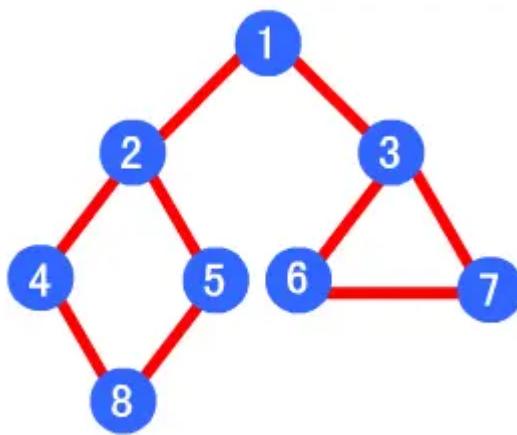
1. 访问顶点 v ；

2. 依次从 v 的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和 v 有路径相通的顶点都被访问；
3. 若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

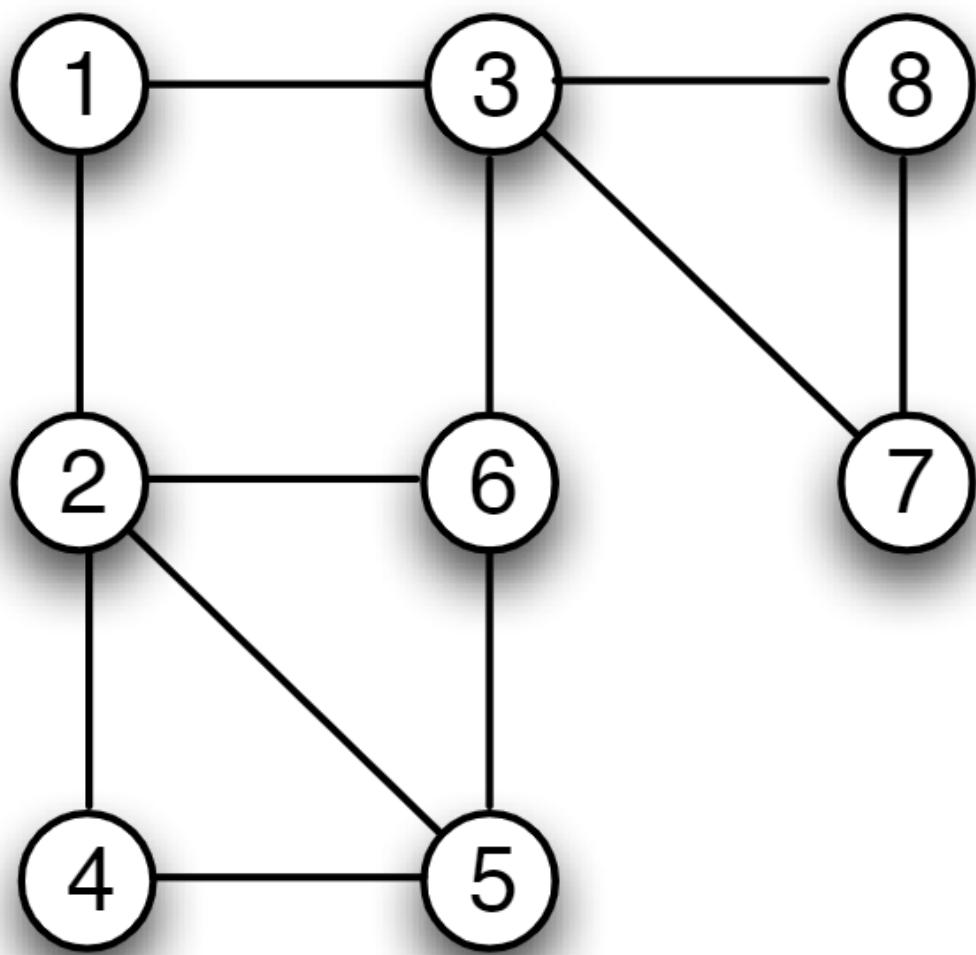
DFS 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一邻接顶点 w_1 ；再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2 ；然后再从 w_2 出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。

接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

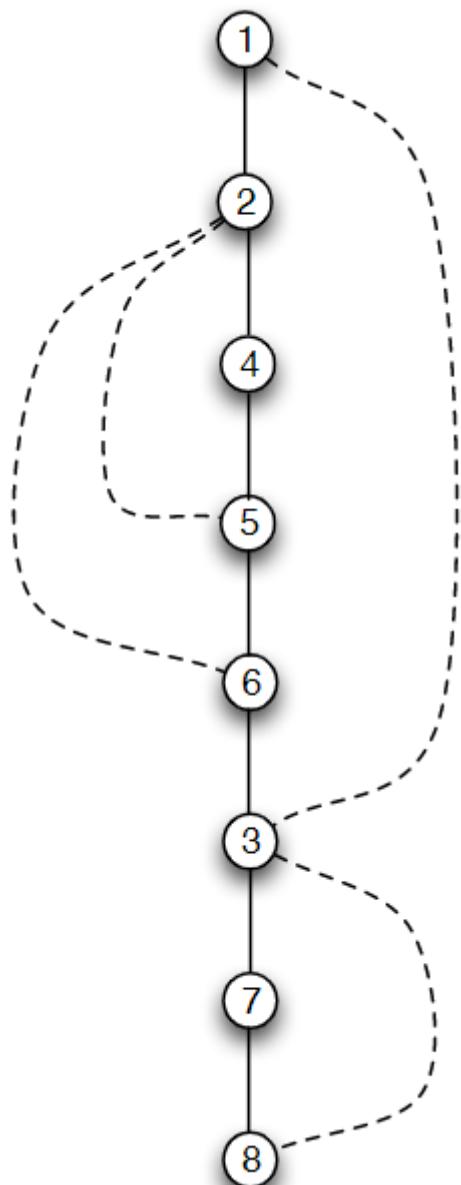
例如下图，其深度优先遍历顺序为 $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7$



An undirected graph G_1

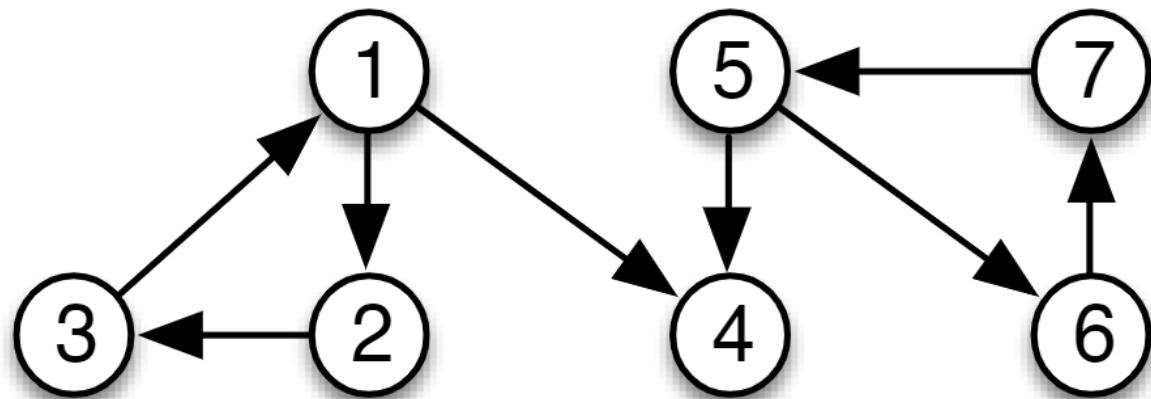


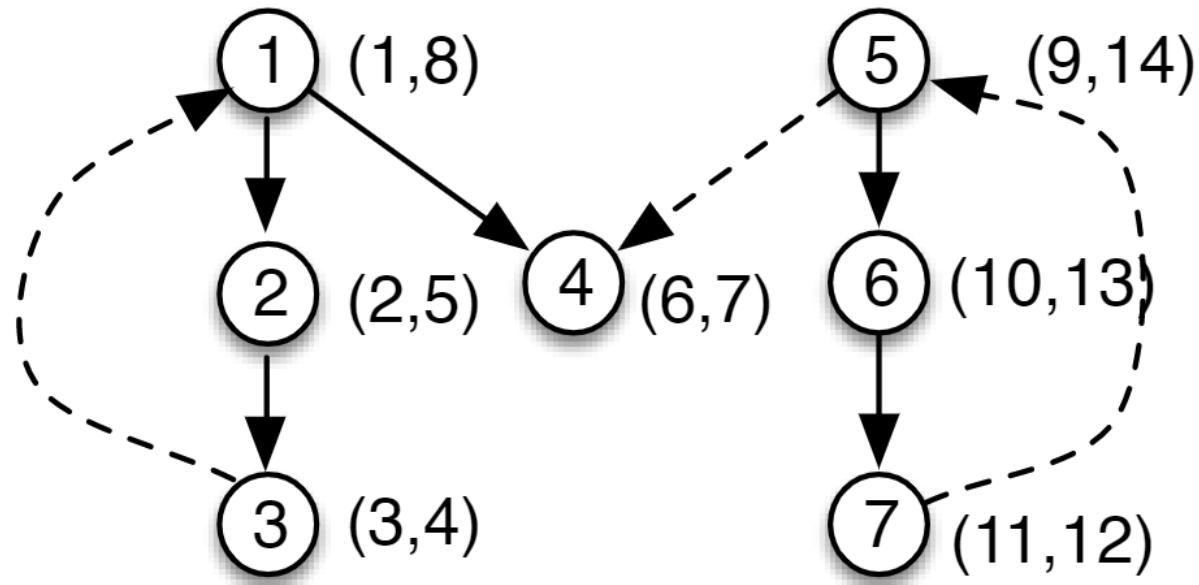
The DFS tree for the undirected graph G_1



Dashed edge belong to the graph but not to the DFS tree. Ties are broken by considering nodes by increasing index

A directed graph G_1



The DFS forest for the directed graph G_1 

Dashed edges belong to G_1 but not to the trees in the DFS forest. (start, finish) intervals appear to the right of every node.

Pseudo Code

```

1  DFS(G = (V, E))
2      for u ∈ V do
3          explored[u] = 0
4      end for
5      for u ∈ V do
6          if explored[u] == 0 then Search(u)
7          end if
8      end for
9  Search(u)
10     previsit(u)
11     explored[u] = 1
12     for (u, v) ∈ E do
13         if explored[v] == 0 then Search(v)
14         end if
15     end for
16     postvisit(u)

```

6.1.1 DFS vs BFS

Similarities

- Linear-time algorithms that essentially can be used to perform the same tasks

Differences

- DFS is more impulsive: when it discovers an explored node, it moves on to exploring it right away; BFS defers exploring until all nodes in the layer have been discovered

- DFS is naturally recursive and implemented using a *stack*
 - A stack is a LIFO (Last-In-First-Out) data structure implemented as a doubly linked list: `insert` (push) / extract (pop) the top element requires $O(1)$ time

6.1.2 Directed graphs: classification of edges

7 NP Class and SAT