CSOR W4231 Analysis of Algorithms, I

Tong Wu, tw2906

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

Example:

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

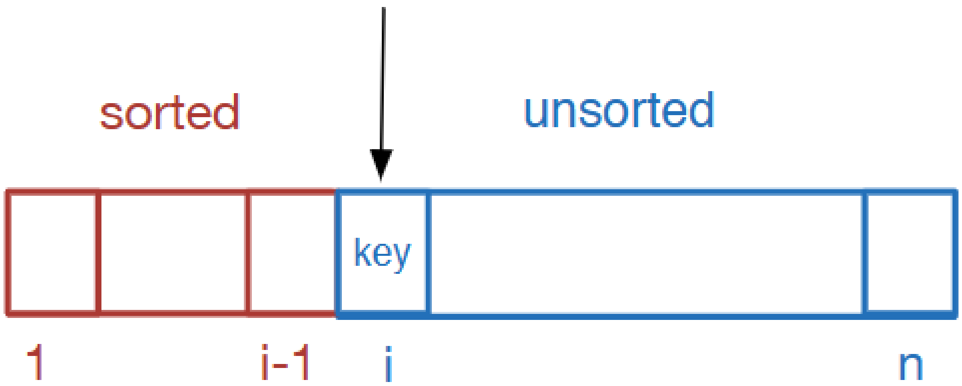
- Output: $A = \{2, 3, 5, 6, 8, 9\}$

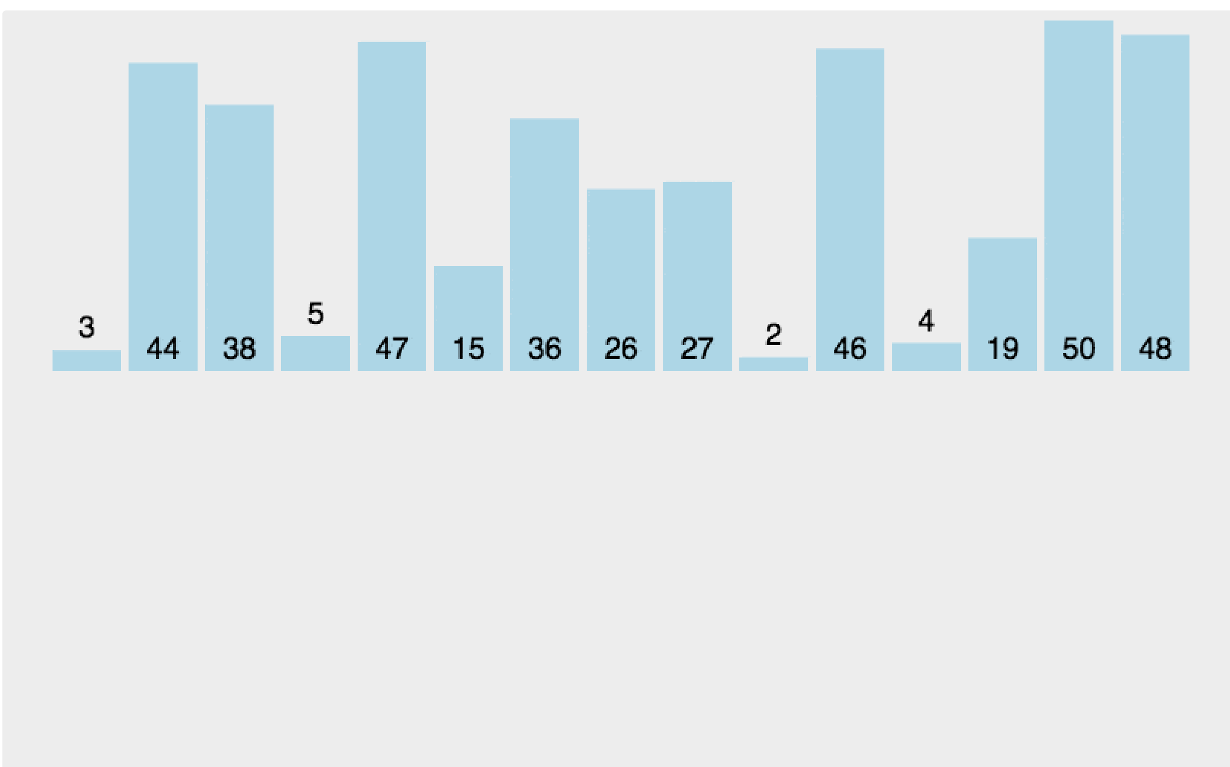
The *data structure* should use to represent the list of output is:

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

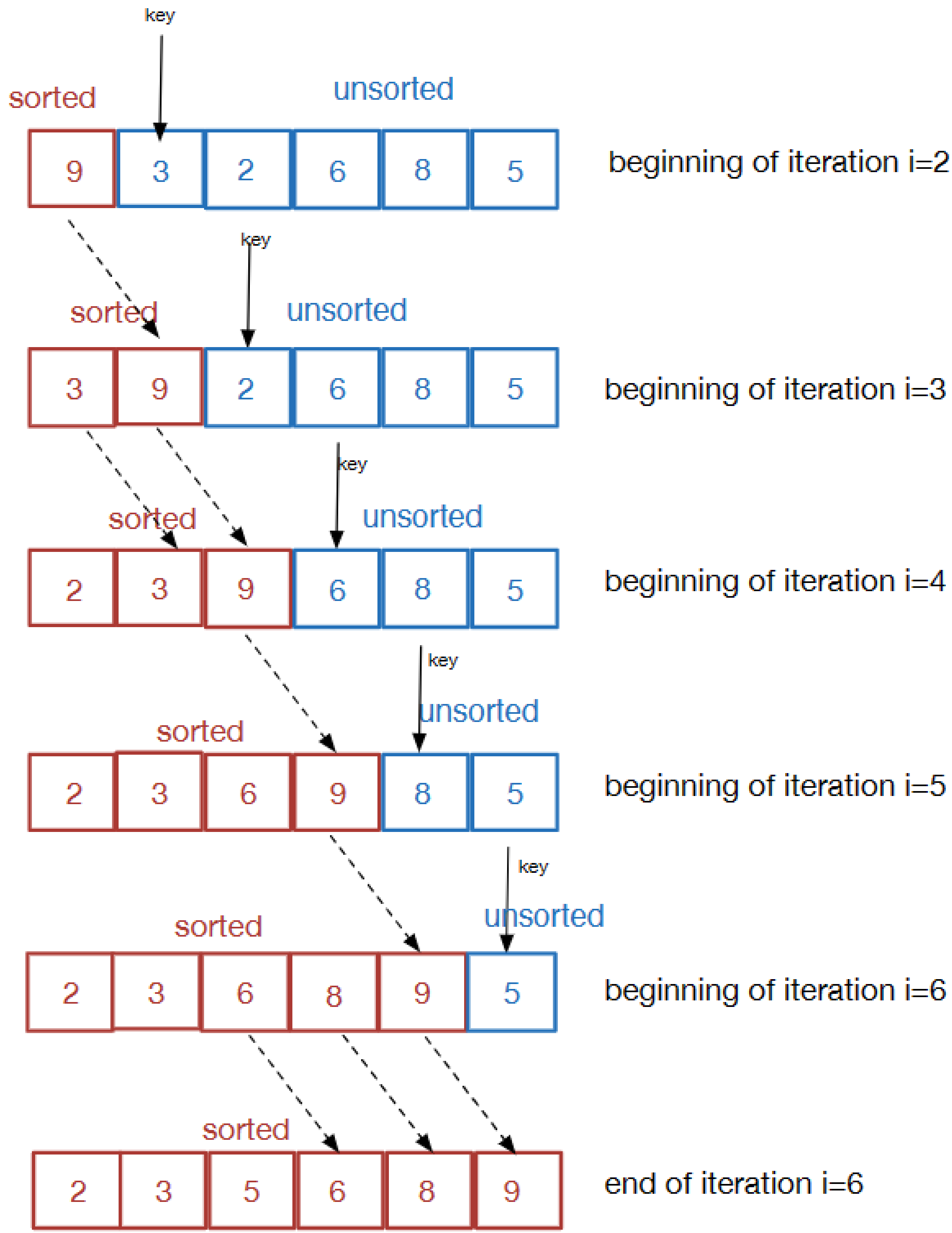
- If $x \le key$, insert key after x

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$x \le key$，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

```
insertion-sort(A)
    for i=2 to n do
        key = A[i]
        // Insert A[i] into the sorted subarray A[1,i-1]
        j = i - 1
        while j>0 and A[j]>key do
            A[j+1] = A[j]
            j = j - 1
        end while
        A[j+1] = key
    end for
```

C++

```
void insertion_sort(int arr[],int len){
    for(int i=1;i<len;i++){
        int key=arr[i];
        int j=i-1;
        while((j>=0) && (key<arr[j])){
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
}
```

Python

```
def insertionSort(arr):
    for i in range(len(arr)):
        preIndex = i-1
        current = arr[i]
        while preIndex >= 0 and arr[preIndex] > current:
            arr[preIndex+1] = arr[preIndex]
            preIndex-=1
        arr[preIndex+1] = current
    return arr
```

Java

```
public class InsertSort implements IArraySort {
    @Override
    public int[] sort(int[] sourceArray) throws Exception {
        // 对 arr 进行拷贝，不改变参数内容
        int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
        // 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
        for (int i = 1; i < arr.length; i++) {
```

```
            // 记录要插入的数据
            int tmp = arr[i];
            // 从已经排序的序列最右边的开始比较，找到比其小的数
            int j = i;
            while (j > 0 && tmp < arr[j - 1]) {
                arr[j] = arr[j - 1];
                j--;
            }
            // 存在比其小的数，插入
            if (j != i) {
                arr[j] = tmp;
            }
        }
        return arr;
    }
}
```

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 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 $A[l] \leq key$
    2. all elements in $A[l + 1, j]$ are pushed one position to the right with their order preserved

The statement is true for i+1

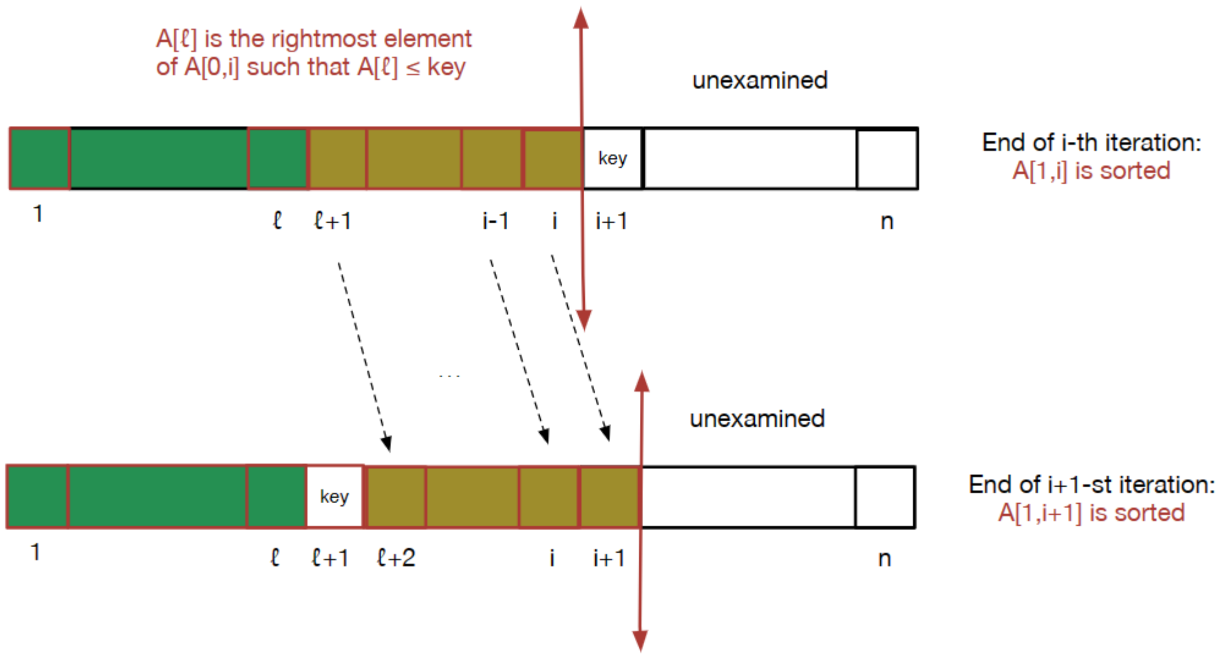
插入排序的正确性

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

```
insertion-sort(A)
    for i=2 to n do
        key = A[i]
        // Insert A[i] into the sorted subarray A[1,i-1]
        j = i - 1
        while j>0 and A[j]>key do
            A[j+1] = A[j]
            j = j - 1
        end while
        A[j+1] = key
    end for
```

for $i = 2$ to n **do** — line 1

  key $= A[i]$ — line 2

  //Insert $A[i]$ into the sorted subarray $A[1, i-1]$

  $j = i - 1$ — line 3

  while $j > 0$ and $A[j] >$ key **do** — line 4

    $A[j+1] = A[j]$ — line 5

    $j = j - 1$ — line 6

  end while

  $A[j+1] =$ key — line 7

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: $n! \approx \left(\frac{n}{e}\right)^n$
  - 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 know 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 to 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.

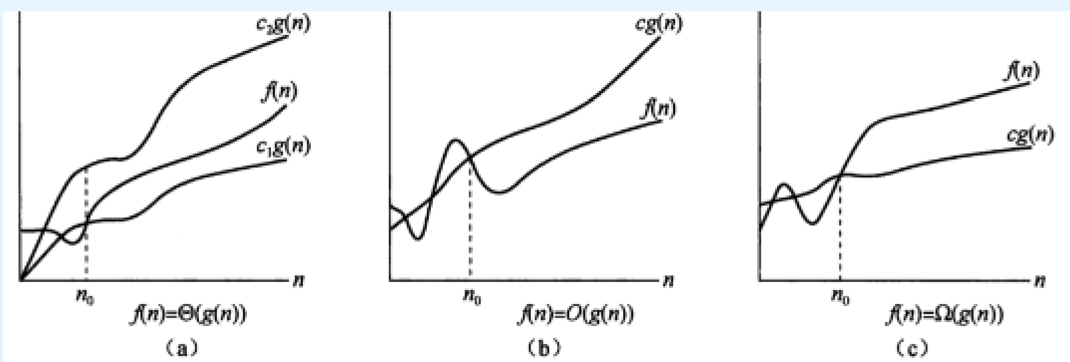
渐进符号 （Asymptotic Notation）

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。**

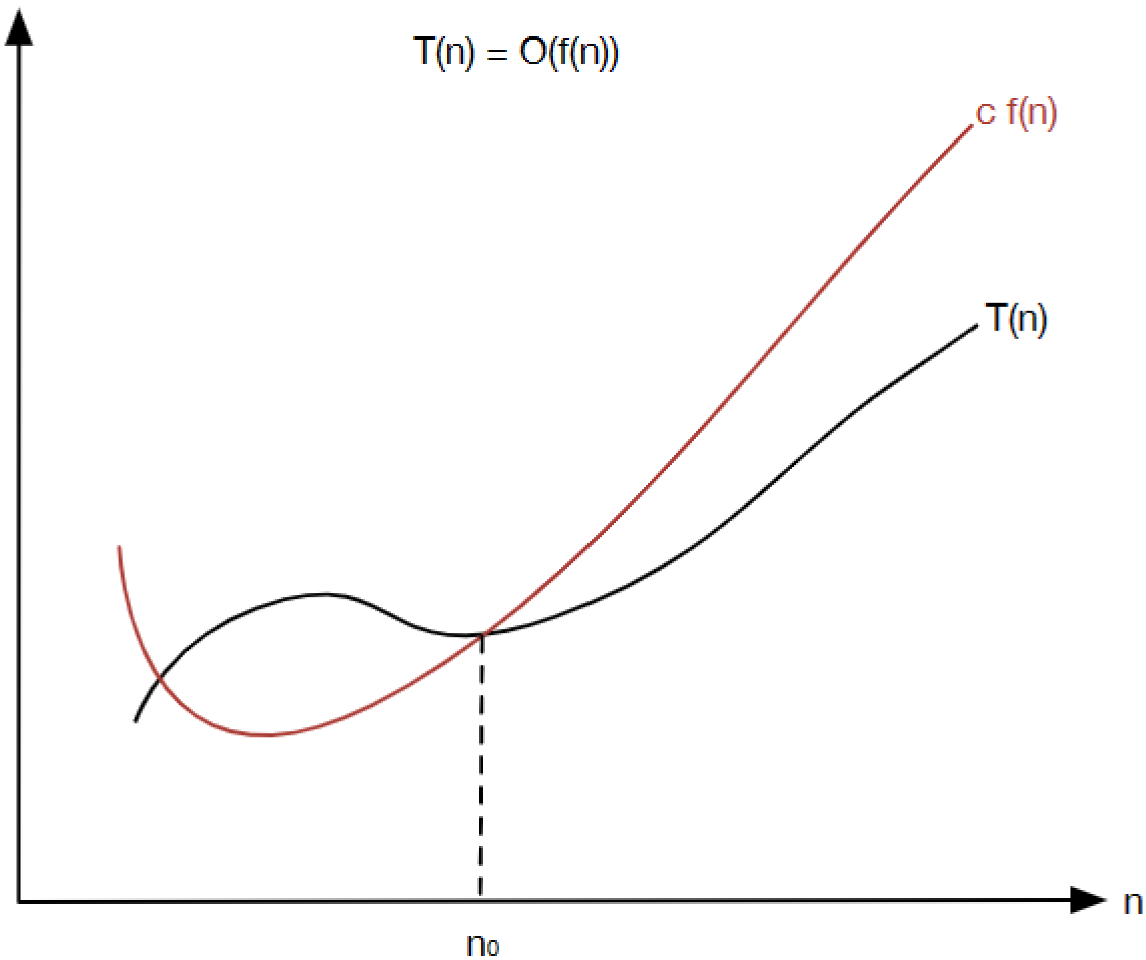
当输入规模n足够大时，我们可以忽略硬件差异带来的影响，只需要关注当输入规模无限增加时，**算法的运行时间是如何随着输入规模的变大而增加。**

因此定义了$\mathrm{O}, \Omega, \Theta$等渐进符号，下图是对他们的图像描述。



2.1.1 Asymptotic upper bounds: Big-O notation

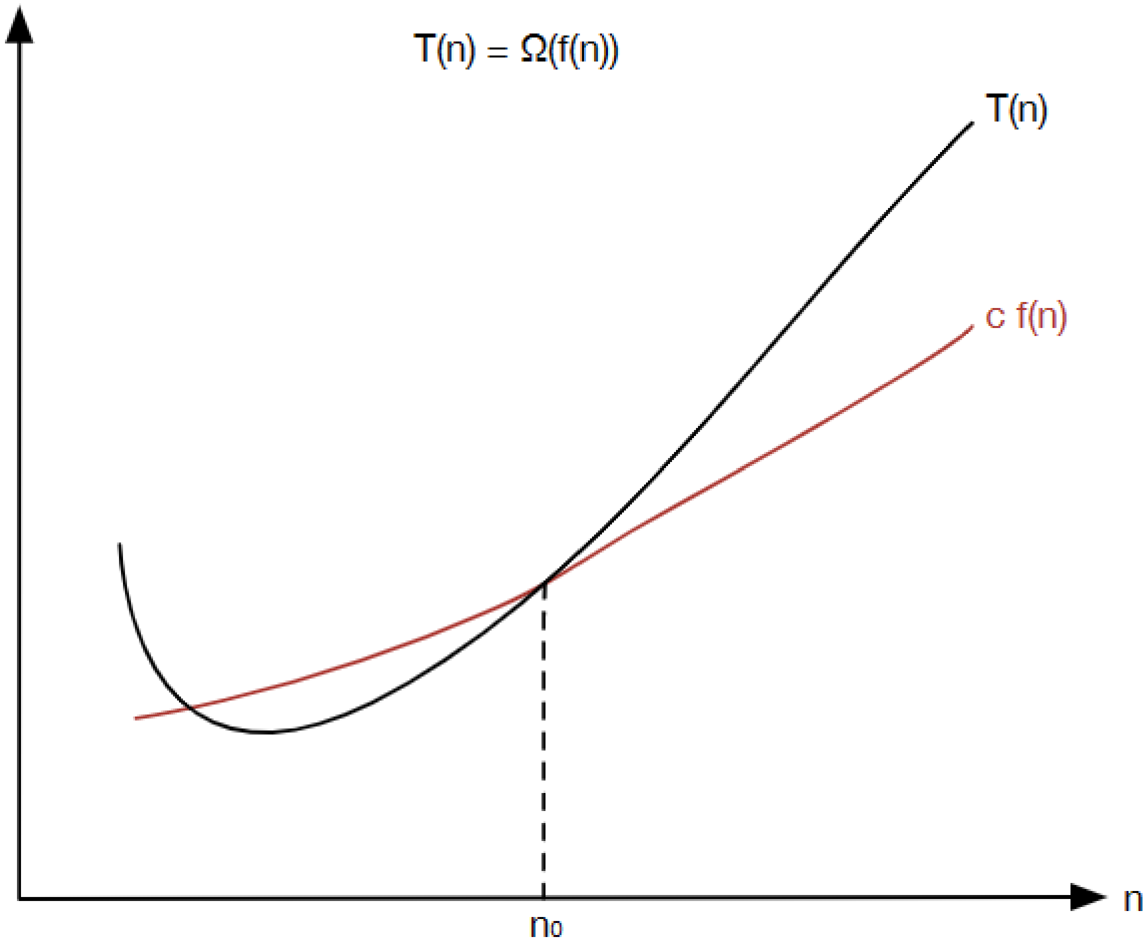
We say that $T(n)=\mathrm{O}(f(n))$ if there exist constants $c>0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \times f(n)$



> 大O记号
>
> 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 $T(n) \geq c \times f(n)$



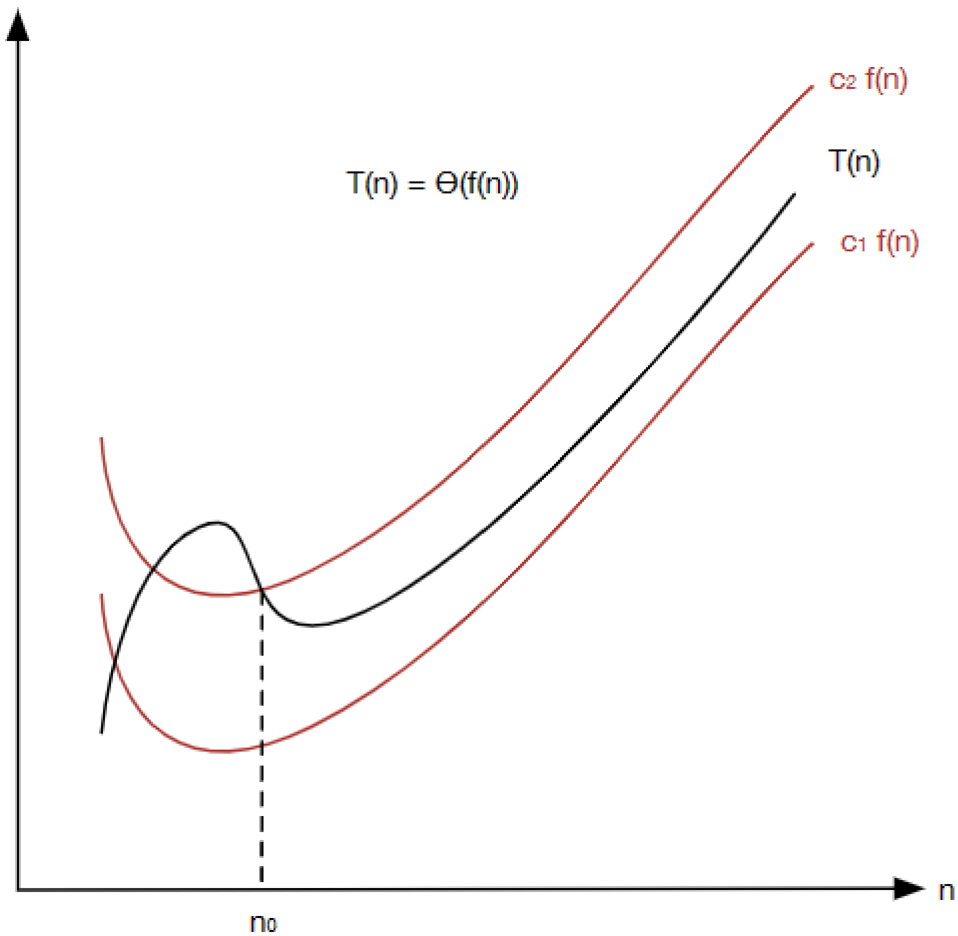
大Ω记号

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$c_1 \times f(n) \le T(n) \le c_2 \times f(n)$。

相当于“=”

Equivalent definition

$T(n) = \Theta(f(n))$ if $T(n) = \mathrm{O}(f(n))$ 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 $n_0 \ge 0$ such that for all $n \ge n_0$, we have $T(n) < c \times f(n)$

- Intuitively, T(n) becomes **insignificant** relative to f(n) as $n \to \infty$
- Proof by showing that $lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (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“$<$”

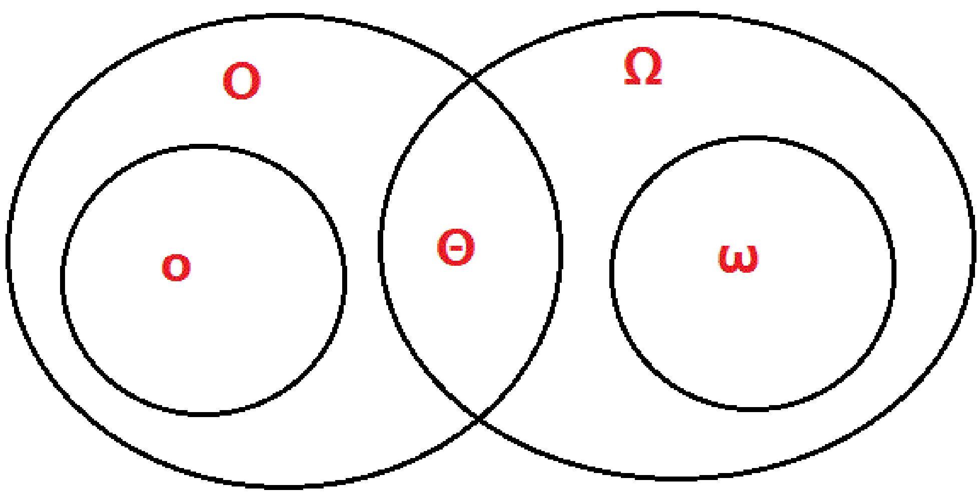
2.1.5 Asymptotic lower bounds that are not tight: little ω

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 $n \to \infty$
- $T(n) = \omega(f(n))$ implies that $lim_{n \to \infty} \frac{T(n)}{f(n)} = \infty$, 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"<="
(3)o（小欧）	非紧的上界。	相当于"<"
(4)Ω（大欧米伽）	下界。	相当于">="
(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 $log_3 n$

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 $f_1, f_2, \ldots, f_k, h$ be functions such that for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$.
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 logn 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(nlogn)。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)。

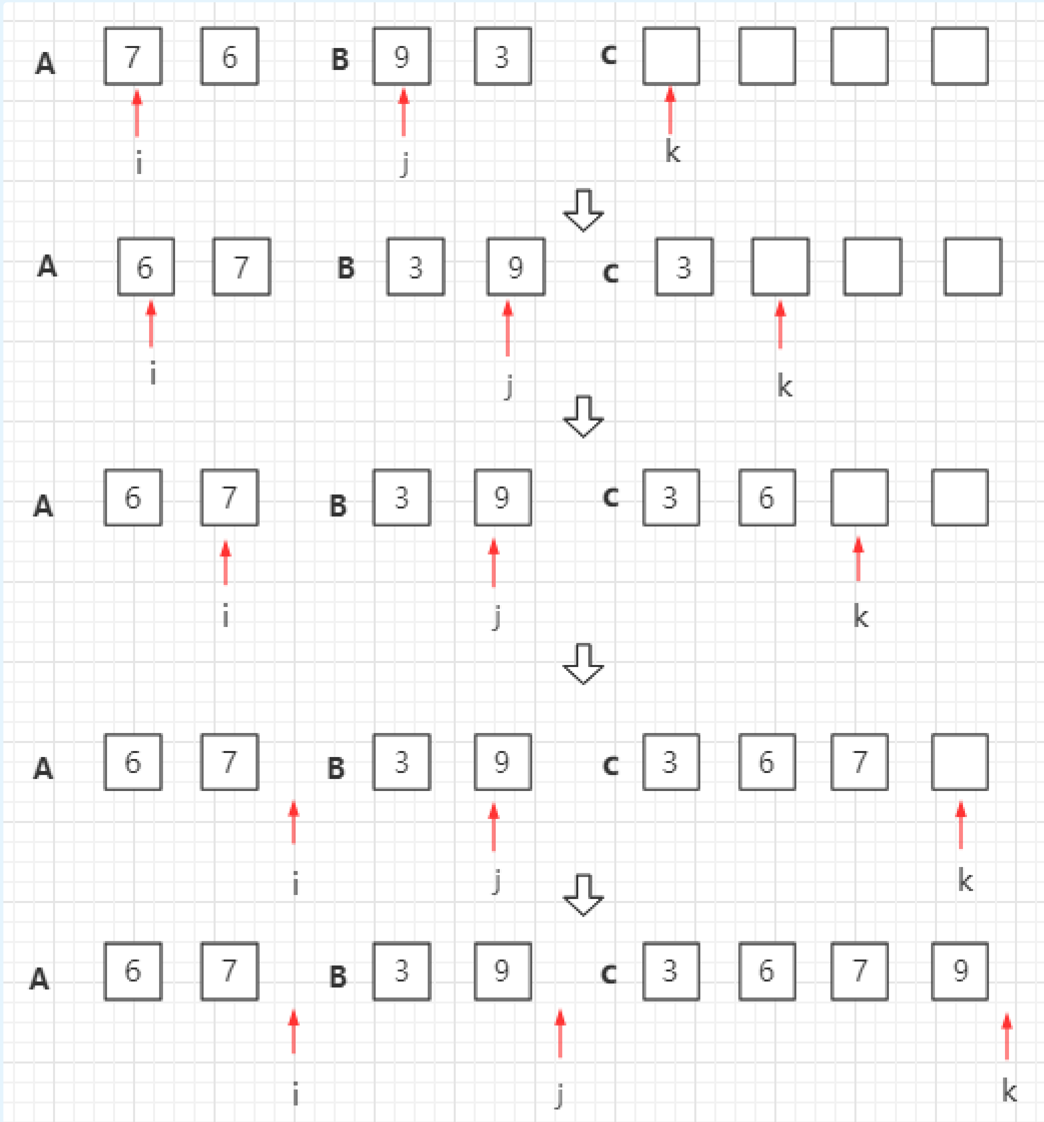
归并排序适用于数据量大，并且对稳定性有要求的场景。

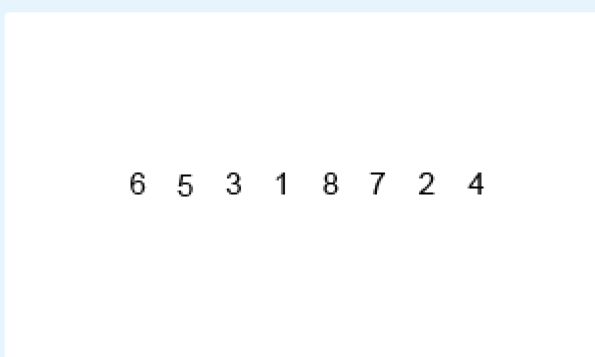
过程：

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。





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

```
merge(A, left, right, mid)
    L = A[left, mid]
    R = A[mid+1, right]
    Maintain two pointers p_l, p_r, initialised to point to the first elements of L,R, repectively
    while both lists are non-empty 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
mergesort(A, left, right)
    if right == left then return
    end if
    mid = left + lower_bound((right-left)/2) //下取整(right-left)/2
    mergesort(A, left, mid)
    mergesort(A, mid+1, right)
    merge(A, left, right, mid)
```

C++

```
// C++ program for Merge Sort
#include <iostream>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
        int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
        *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
```

```
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne
        && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]
            <= rightArray[indexOfSubArrayTwo]) {
            array[indexOfMergedArray]
                = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
        else {
            array[indexOfMergedArray]
                = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
```

```cpp
{
    if (begin >= end)
        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
    for (auto i = 0; i < size; i++)
        cout << A[i] << " ";
}

// Driver code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    auto arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";
    printArray(arr, arr_size);
    return 0;
}
```

Python

```python

# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

```

```
        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list


def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()


# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

Java

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

```java
class MergeSort {
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int[n1];
        int R[] = new int[n2];

        /*Copy data to temp arrays*/
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
```

```java
            }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r) {
            // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };

        System.out.println("Given Array");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

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.

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

     -> 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.

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

   -> merge takes $\mathrm{O}(n)$ 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 subprobelm 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 theorm

Theorem 6 (Master Theorem)

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

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

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** $1, 2, \ldots, n-1$

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**
  - ```
    merge(A, left, right, mid)
        L = A[left, mid]
        R = A[mid+1, right]
        Maintain two pointers p_l, p_r, initialised to point to the first elements of L,R, repectively
        while both lists are non-empty 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
    mergesort(A, left, right)
        if right == left then return
        end if
        mid = left + lower_bound((right-left)/2) //下取整(right-left)/2
        mergesort(A, left, mid)
        mergesort(A, mid+1, right)
        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

From EEEN30002 Numerical Analysis

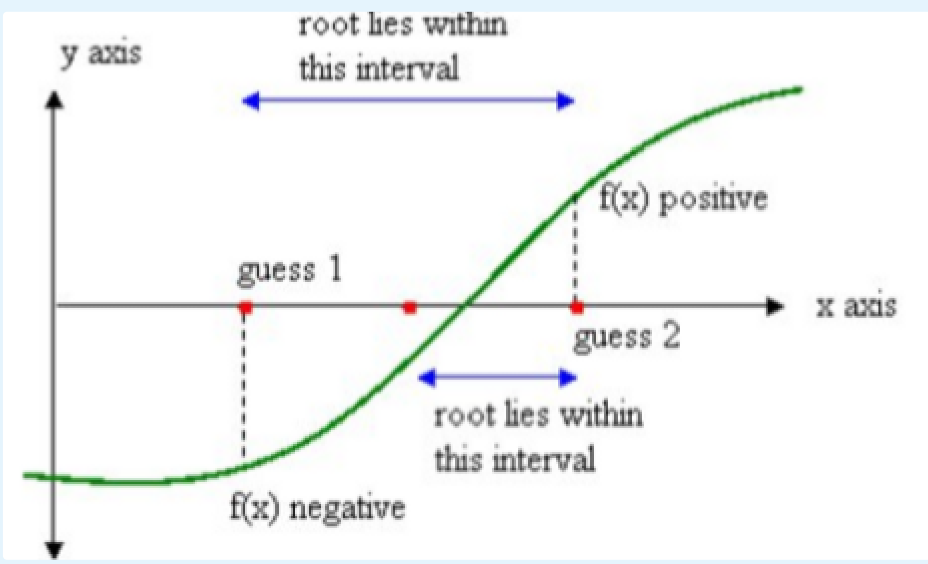
💡 将复杂的非线性方程问题简化，将非线性方程分成几个区间，对每个区间分别求解，选取一个近似区间使用迭代法逼近真实解

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$x_m = (a+b)/2$



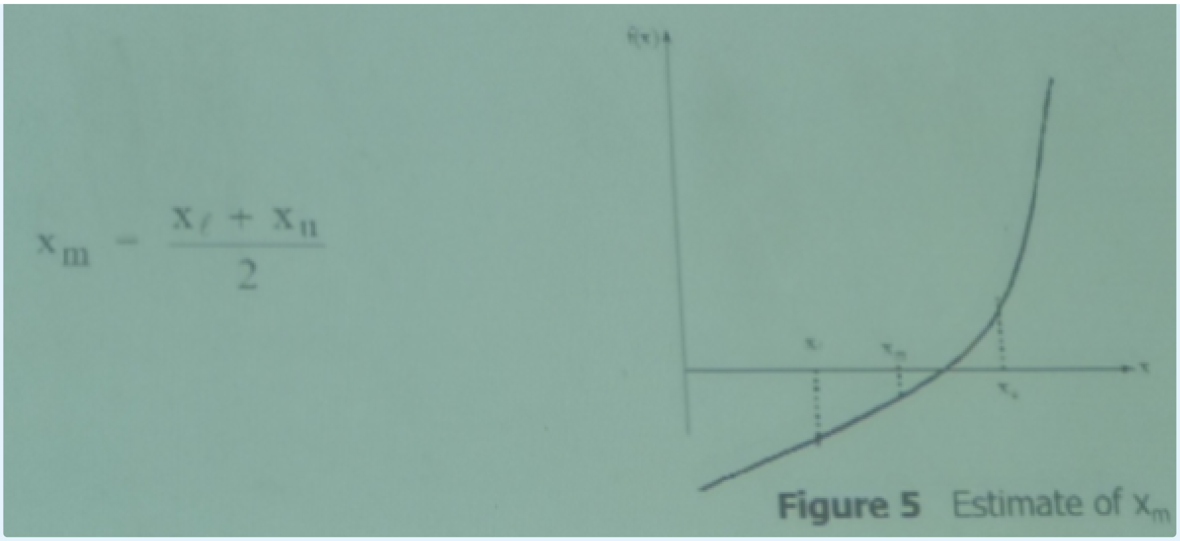


Figure 5 Estimate of x_m

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$e_k <= 1/2(a_{prev} - b_{prev}) = (1/2)^{k+1} * (a_0 - b_o)$

💡 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$(1/2)^{k+1} * (a_0 - b_o)$，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epsilon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epsilon值判断需要多少个循环才能满足条件

假设我们需要$e_k < epsilon$

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

可得$k + 1 > (ln(a_0 - b_0) - ln(epsilon))/(ln(2))$

MATLAB

```matlab
%%% bisection algorithm to find sqrt(2)

xmin = 0;
xmax = 2;
fmin = xmin^2-2;
fmax = xmax^2-2;

%%% choose epsilon
epsilon = 10^-3/2;

for k = 0 : ceil((log(2)-log(epsilon))/log(2) -1),

    xhat = (xmin+xmax)/2;
    fhat = xhat^2-2;

    disp([k xmin xmax xhat abs(xhat-sqrt(2)) (xmax-xmin)/2 fmin fmax fhat])

    if fhat*fmin > 0,
        xmin = xhat; fmin = fhat;
    else
        xmax = xhat; fmax = fhat;
    end

end
```

C++

```cpp
#include <iostream>
using namespace std;
#define EP 0.01
// An example function whose solution is determined using
// Bisection Method. The function is x^3 - x^2 + 2
double solution(double x) {
   return x*x*x - x*x + 2;
}
// Prints root of solution(x) with error in EPSILON
void bisection(double a, double b) {
   if (solution(a) * solution(b) >= 0) {
      cout << "You have not assumed right a and b\\n";
      return;
   }
   double c = a;
   while ((b-a) >= EP) {
      // Find middle point
      c = (a+b)/2;
      // Check if middle point is root
```

```
        if (solution(c) == 0.0)
           break;
         // Decide the side to repeat the steps
        else if (solution(c)*solution(a) < 0)
           b = c;
        else
           a = c;
    }
    cout << "The value of root is : " << c;
}
 // main function
int main() {
   double a =-500, b = 100;
   bisection(a, b);
   return 0;
}
```