# Data Structure & Algorithm Analysis

> CS203 DSAA 2020fall

# Lecture1 Introduction

## 1.Algorithms

A well defined **sequence of steps** for solving a computational problem

- It produces the **correct output**
- It uses **basic steps** / defined operations
- It finishes in **finite time**

## 2.Data Structure

A way of **organizing data objects** for efficient usage

Building blocks for **designing algorithms**

| Common Data Structures | 常用数据结构 |
|---|---|
| Array | 数组 |
| Linked List | 链表 |
| Stack | 栈 |
| Queue | 队列 |
| Hash Table | 哈希表 |
| Heap | 堆 |

They solve different question with different **time complexity**

But it depends on the frequency of **operations** used in your algorithm

> 数据结构就是以不同的形式存储数据，利用不同的数据结构可以构建不同时间复杂度的算法，因为数据需要有规律的存储在计算机中，所以需要挑选某种数据结构存储，在使用某种算法计算分析，以此大大减小时间复杂度和空间复杂度

---

Let S be a set of items, and x be a search key

### Useful operations on a set S

- Search(S , x)： search whether x appears in S
- Insert(S , x): insert item x into S
- Delete(S , x): remove item x from S

## 3.Recursive Problem-Solving

The recursive case

- Reduces the overall problem to one or more simpler problems of the same kind
- Make recursive calls to solve the simpler problems

### Template of a Recursive Method

```
    recursiveMethod(parameters){
        if(stopping condition){
            //handle the base case
        }else{
            //recursive cae
            //possibly do something here
            recursiveMethod(Modified parameters);
            //possibly do something here
        }
    }
```

## 4.Supplement

Computer only contains **two type of data type** (totally)

- text file
- executable file

> 在做算法题的时候，第一件事情是分析数据大小，时间复杂度（时间限制）和空间复杂度（创建变量和算法迭代的次数），以此来大概推算使用什么算法和什么时间复杂度解决问题

# Lecture2 Algorithm Analysis

## 1.RAM Computational Model

### Data Loading

- CPU load data from RAM(GB level)
- RAM load data from Hard Disk(TB level)

### Cost

- I/O cost: Read data from Hard Disk to memory
- CPU cost: Read data from memory to CPU to do computation

[^]:  I/O cost is much larger than CPU cost

### RAM computational Model

- Loading data from memory
- Putting the data in Register
- ALU calculates data from Register following the instructions
- ALU offers output

## 2.Memory

A **finite sequence of cells**, each cell has the **same number of bits**

Every cell has an address: the first cell of memory has address 0, the second cell has address 1, and so on

Store the information for **immediate use** in a computer

It is a **computer hardware** device

## 3.CPU

Contains **a fixed number of registers**

### Basic operations

- **Initialization**: Set a register to a fixed value
- **Arithmetic**(ALU): Take integers a, b sorted in two registers, calculate on of {+,-,*,/} and store the result in a register
- **Comparison**: Take integers a, b sorted in two registers, compare them, and learn which of {a<b, a=b, a>b} is true

> 我们在考虑算法的复杂度中，就是考虑CPU的这些基本操作的数量大小

> 计算时间限制1s≈10^8个基本操作，在计算机中，定义一个时间戳为一个循环（cycle）所需要花的时间

> i = i + 1 或 j = i + 1 只都算一次运算

## 4.Algorithm Analysis

### Cost analysis

- How many time my algorithm will cost
- How many space my algorithm will cost

### Correctness analysis

- Whether my algorithm is right in all situations

### Example

Give integer n, calculate 1+2+3+...+n

```
int a = 0;//initialization 1
int b = n;//initialization 1
int sum = 0;//initialization 1
for(;;){
    if(a > b){//comparision n
        break;
    }
    sum += a;//arithmetic n
    a++;//arithmetic n
}
//total time complexity 3n + 3
```

# Lecture3 Worse Case Analysis

# 1.Worst-Case Running Time

The **largest running time** of the algorithm on **all the inputs** of the same size n

# 2.Asymptotic Analysis (渐进分析)

Running time of two algorithms, with input size n

$$Algorithm1: f(n) = 4n + 1$$
$$Algorithm2: g(n) = 8log_2n + 10$$

In computer science we **ignore all constants**, but only worry about the dominating term

## Big-O notation

Let f(n) and g(n) be two function of n

We say that f(n) grows asymptotically no faster than g(n) if there is a constant c1 > 0 such that

$$f(n) \leq c_1 \times g(n)$$

holds for all n ≥ c2, We denote this by

$$f(n) = O(g(n))$$

| Complexity | Name | Algorithm |
|---|---|---|
| O(1) | Constant time | Compare two numbers |
| O(log n) | Logarithmic | Binary Search |
| O(n) | Linear time | Search |
| O(n log n) | | Merge Sort |
| O(n^2) | Quadratic | Selection Sort |
| O(n^3) | Cubic | Matrix Multiplication |
| O(2^n) | Exponential | Brute-force Search on Boolean satisfiability |
| O(n!) | Factorial | Brute-force Search on Travelling Salesman |

## Big-Ω notation

Let f(n) and g(n) be two function of n

We say that f(n) grows asymptotically no slower than g(n) if there is a constant c1 > 0 such that

$$f(n) \geq c_1 \times g(n)$$

holds for all n ≥ c2, We denote this by

$$f(n) = \Omega(g(n))$$

# Binary Search Algorithm (log n)

An array **A** of **n** integers have been sorted in ascending order.

Design an algorithm to determine whether given value **t** exists **A**.

We utilize the fact that array **A** has been sorted in ascending order. Let us compare **t** to the element **x** in the middle of **A**

- If t = A[n/2], we have found t
- If t < A[n/2], we can ignore A[n/2+1] to A[n]
- If t > A[n/2], we can ignore A[0] to A[n/2]

In the 2nd and 3nd cases, we have at most n/2 element. Then repeat the above.

```java
public static boolean BSA(int[] arr, int num){
    boolean exist = false;
    //original location
    int left = 0;
    int right = arr.length - 1;
    boolean flag = true;
    while(flag){
        int mid = left + (right - left) / 2;
        if(arr[mid] == num){
            exist = true;
            break;//remind letting the loop break!!
        }else if(num < arr[mid]){
            right = mid - 1;
        }else if(num > arr[mid]){
            left = mid + 1;
        }
        //break situation
        if(left > right){
            flag = false;
        }
    }
    return exist;
}
```

Suppose that there are h iterations in total  it holds that h is the smallest integer satisfying that

$$n/2^h < 1$$

Then

$$h > log_2 n \rightarrow h = 1 + log_2 n$$

The worst case time of binary search is at most

$$g(n) = 2 + 8(1 + log_2 n)$$

# Question

Function A is an implementation of binary search algorithm to find the largest integer k in an ascending size-n array Arr. If does not exist in Arr, return -1. But there are bugs in the code, please find them and fix them.

```c
int A(int Arr[], int v){
```

```
    int min = 0;
    int max = Arr.length;
    int mid;
    while(min < max){
        mid = (max + min) / 2;
        if(mid < k){
            min = mid;
        }else{
            max = mid - 1;
        }
        if(Arr[max] == k){
            return max;
        }else{
            return -1;
        }
    }
}
```

## Answer

```
    public static int A(int[] Arr, int k){
        int min = 0;
        int max = Arr.length;
        int mid;
        while(true){
            mid = min + (max - min) / 2;
            if(Arr[mid] <= k){
                min = mid;
            }else{
                max = mid;
            }
            if(min + 1 == max){
                break;
            }
        }
        if(min == 0){
            return -1;
        }
        return min;
    }
```

> **二分法版式**
>
> 1.循环直接插旗子用while(true)
>
> 2.分析最终逼近的是可行域的最大值还是最小值
>
> > 2.1最大值 左边往右边逼近 假想 <= 实际 min = mid
>
> > 2.2最小值 右边往左边逼近 假想 >= 实际 max = mid
>
> 3.旗倒条件 min + 1 == max → break
>
> 4.返回值（找到的情况）
>
> > 4.1 最大值 返回min
>
> > 4.2 最小值 返回max

> 5.返回值找不到
>
> > 5.1 最大值 min = 原始设定值 返回-1
>
> > 5.2 最小值 max = 原始设定值 返回-1

> 注意一个问题，我们在二分法使用中通常会保证二分查找的**数组是有序**的，当然我们在实际题目考虑中的时候，通常使用时间序列找的时候默认有序，如果是查找某个实际数组就需要保证它是有序的了

# Lecture4 Sorting Algorithm

## 1.Sorting Problem Description

> Comparison + Swap

> We will not introduce Shell Sort

- Input: an array A[1...n] with n integers
- Output: a sorted array A (in ascending order)

## Selection Sort Algorithm (n^2)

```
    public static int[] SSA(int[] arr){
        int[] a1 = arr;
        for(int i = 0; i < arr.length - 1; i++){
            int k = i;
            for(int j = i + 1; j < arr.length; j++){
                if(a1[k] > a1[j]){
                    k = j;
                }
            }
            int temp;
            temp = a1[i];
            a1[i] = a1[k];
            a1[k] = temp;
        }
        return a1;
    }
```

> Selection Sort的实现是从第1位开始，遍历后面的找到是否有一个比第1位上小的数，交换两者，遍历将每一位排好
>
> 第1位，遍历全部，找到最小的那一个，把它放在第1位
>
> 第2位，遍历除第1位的全部，找到最小的那一个，把它放在第2位

### Time Complexity Analysis——SSA

```
for integer i←1 to n-1                  Cost:O(n)
    k←i                                 Cost:O(n)
    for integer j←i+1 to n              Cost:O(n²)
        if  A[k] > A[j]  then           Cost:O(n²)
            k←j                         Cost:O(n²)
    swap A[i] and A[k]                  Cost:O(n)

```

# ▯Insertion Sort Algorithm (n^2)

## Idea

- One input each iteration, growing a sorted output list
- Remove one element from input data
- Find the location it belongs within the sorted list
- Repeat until no input elements remain

```
    public static int[] ISA(int[] arr){
        for(int i = 1; i < arr.length; i++){
            for(int j = i; j >= 1; j--){
                if(arr[j - 1] > arr[j]){
                    int temp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = temp;
                }
            }
        }
        return arr;
    }
```

# ▯Bubble Sort(n^2)

## Idea

For each pass

- Compare the pair of adjacent item
- Swap them if they are in the wrong answer

Repeat the pass through until no swaps are needed

```
    public static int[] BSA(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 1; j < arr.length; j++){
                if(arr[j - 1] > arr[j]){
                    int temp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = temp;
                }
            }
        }
        return arr;
    }
```

```
    public static int[] BSA2(int[] arr) {
        boolean flag = true;
        while (flag) {
            int count = 0;
            for (int j = 1; j < arr.length; j++) {
                if (arr[j - 1] > arr[j]) {
                    int temp = arr[j];
                    arr[j] = arr[j - 1];
                    arr[j - 1] = temp;
                    count++;
```

```
                }
            }
            if (count == 0) {
                flag = false;
            }
        }
        return arr;
    }
```

> 排序的各种算法哪个是稳定的，哪个是不稳定的

# Lecture5 Merge Sort

## 1.Divide and Conquer

Divide and Conquer: an algorithmic technique

- **Divide**: divide the problem into smaller subproblems
- **Conquer**: solve each problem recursively
- **Combine**: combine the solution of subproblems into the solution of the original problem

## ❗Merge Sort (n log n)

**Divide**: divide the array into two subarrays of n/2 numbers each

**Conquer**: sort two subarrays recursively

**Combine**: merge two sorted subarrays into a sorted array

```
    public static int[] MSA(int[] A){
        int n = A.length;
        int[] s1 = A;
        if(n > 1){
            //divide
            int p = n / 2;
            int[] a = Arrays.copyOfRange(A,0,p);
            int[] b = Arrays.copyOfRange(A,p,A.length);
            //conquer
            int[] c = MSA(a);
            int[] d = MSA(b);
            //combine
            s1 = Merge(c,d);
        //be care of the basic situation
        }else{
            return s1;
        }
        return s1;
    }
//This is my own code----------------------------------------
    public static int[] Merge(int[] A, int[] B){
        int[] returnList = new int[A.length + B.length];
        int countA = 0;
        int countB = 0;
        for(int i = 0; i < returnList.length; i++){
            if(A[countA] < B[countB]){
                returnList[i] = A[countA];
                if(countA == A.length - 1){
```

```java
                    for(int j = i + 1; j < returnList.length; j++){
                        returnList[j] = B[countB];
                        if(countB == B.length - 1){
                            break;
                        }
                        countB++;
                    }
                    break;
                }
                countA++;
                continue;
            }else if(A[countA] > B[countB]){
                returnList[i] = B[countB];
                if(countB == B.length - 1){
                    for(int j = i + 1; j < returnList.length; j++){
                        returnList[j] = A[countA];
                        if(countA == A.length - 1){
                            break;
                        }
                        countA++;
                    }
                    break;
                }
                countB++;
                continue;
            }else if(A[countA] == B[countB]){
                returnList[i] = B[countB];
                i++;
                returnList[i] = A[countA];
                if(countA + countB == returnList.length - 2){
                    break;
                }
                countA++;
                countB++;
            }
        }
        return returnList;
    }
//This is the reference code---------------------------------
    public static int[] Merge2(int[] A, int[]B){
        int nL = A.length;
        int nR = B.length;
        int[] returnList = new int[A.length + B.length];
        int countA = 0;
        int countB = 0;
        for(int i = 0; i < returnList.length; i++){
            if((countA <= nL - 1) && (countB > nR - 1 || A[countA] <= B[countB])){
                returnList[i] = A[countA];
                countA++;
            }else{
                returnList[i] = B[countB];
                countB++;
            }
        }
        return returnList;
    }
```

## 2. Master Theorem

Recurrence equation:

$$T(n) = aT(n/b) + f(n)$$

Let T(n) be a function that return a positive value for every integer n > 0.

We know that:

$$T(1) = O(1)$$
$$T(n) = \alpha T([\frac{n}{\beta}]) + O(n^c) for (n \geq 2)$$
$$where\ \alpha \geq 1, \beta > 1, and\ c \geq 0\ Then:$$
$$If\ log_\beta \alpha < c, then\ T(n) = O(n^c)$$
$$If\ log_\beta \alpha = c, then\ T(n) = O(n^c logn)$$
$$If\ log_\beta \alpha > c, then\ T(n) = O(n^{log_\beta \alpha})$$

For Merge Sort, we know that

$$T(n) = 2T(\frac{n}{2}) + O(n)$$
$$\alpha = 2, \beta = 2, c = 1$$
$$log_2 2 = 1 = c \rightarrow T(n) = O(nlogn)$$

# Lecture6——Quick Sort

So far in DSAA, all our algorithms are deterministic, that is, they **do not involve any randomization**

**Randomized algorithms** play an important role in Computer Science, they often simpler, and sometimes can be provably faster as well

## ❗Quick Sort (n log n~ n^2)

### Idea

**Randomly** pick an integer p in A, call it the **pivot**

Re-arrange the integers in the array A' such that

- All the integers **smaller** than p are positioned **before** p in A'
- All the integers **larger** than p are positioned **after** p in A'

Sort the part of A‘ before p recursively

Sort the part of A‘ after p recursively

### Code

Quicksort ([A|1...n], lo=1, hi=n)

1. p ← partition(A, lo, hi)
2. quicksort(A, lo, p-1)
3. quicksort(A,p+1,hi)

```
    private static void quickSort(int[] arr, int leftIndex, int rightIndex)
{
        //basic situation
```

```
        if (leftIndex >= rightIndex) {
            return;
        }
        int left = leftIndex;
        int right = rightIndex;
        //select the first element as pivot
        int key = arr[left];
        //scanner each side, until left = right
        while (left < right) {
            //左移
            while (right > left && arr[right] >= key) {
                right--;
            }
            arr[left] = arr[right];

            //右移
            while (left < right && arr[left] <= key) {
                left++;
            }
            arr[right] = arr[left];
        }
        arr[left] = key;
        //divide and conquer
        quickSort(arr, leftIndex, left - 1);
        quickSort(arr, right + 1, rightIndex);
        //combine--we don't need to combine in this situation
    }
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 39 | 28 | 55 | 87 | 66 | 3 | 17 | 39* |

```
    private static void quickSort(int[] arr, int leftIndex, int rightIndex)
{
        //basic situation
        if (leftIndex >= rightIndex) {
            return;
        }
        int left = leftIndex;
        int right = rightIndex;
```

```
        //select the first element as pivot
        int key = arr[left];
        //scanner each side, until left = right
        while (left < right) {
            //左移
            while(right > left && key % 2 == 0 &&((arr[right] % 2 == 0 && arr[right] > key)|| arr[right] % 2 == 1)){
                right--;
            }
            while(right > left && key % 2 == 1 && arr[right] % 2 == 1 && arr[right] > key){
                right--;
            }
            arr[left] = arr[right];



            //右移
            while (left < right && arr[left] <= key) {
                left++;
            }
            arr[right] = arr[left];
        }
        arr[left] = key;
        //divide and conquer
        quickSort(arr, leftIndex, left - 1);
        quickSort(arr, right + 1, rightIndex);
        //combine--we don't need to combine in this situation
    }
```

# 1.Time complexity analysis

## Best case

在最优情况下，Partition每次都划分得很均匀，如果排序n个关键字，其递归树的深度就为 [log2n]+1（[x] 表示不大于 x 的最大整数），即仅需递归 log2n 次，需要时间为T（n）的话，第一次Partition应该是需要对整个数组扫描一遍，做n次比较。然后，获得的枢轴将数组一分为二，那么各自还需要T（n/2）的时间（注意是最好情况，所以平分两半）。于是不断地划分下去，就有了下面的不等式推断：

$$
\begin{aligned}
&T(n) = 2T(\tfrac{n}{2}) + n \\
&T(n) = 2(2T(\tfrac{n}{4} + \tfrac{n}{2}) + n = 4T(\tfrac{n}{4}) + 2n \\
&T(n) = 4(2T(\tfrac{n}{8} + \tfrac{n}{4}) + n = 8T(\tfrac{n}{4}) + 3n \\
&\ldots\ldots \\
&T(n) = nT(1) + (log(n)) \times n = O(n \times log(n))
\end{aligned}
$$

这说明，在最优的情况下，快速排序算法的时间复杂度为O(nlogn)。

## Worst case

最糟糕情况下的快排，当待排序的序列为正序或逆序排列时，且每次划分只得到一个比上一次划分少一个记录的子序列，注意另一个为空。如果递归树画出来，它就是一棵斜树。此时需要执行n-1次递归调用，且第i次划分需要经过n-i次关键字的比较才能找到第i个记录，也就是枢轴的位置，因此比较次数为

$$\sum_{i=1}^{n-1}(n-i) = n-1+n-2+\ldots+1 = \frac{n(n-1)}{2}$$

**此时Quick Sort会退化成Bubble Sort**：将最小的或者最大的冒泡出来

### General case

最后来看一下一般情况，平均的情况，设枢轴的关键字应该在第k的位置（1≤k≤n），那么：

$$T(n) = \frac{1}{n}\sum_{k=1}^{n}(T(k-1)+T(n-k))+n = \frac{2}{n}\sum_{k=0}^{n-1}T(k)+n$$

## 2.Time complexity analysis——advanced

### Worst case analysis

假设T(n)是最坏情况下快排在输入规模为n的数据集合上所花费的时间，则有递归式

$$T(n) = max\{T(q)+T(n-q-1)\}+O(n)$$

因为分治生成的两个子问题的规模加总为n-1，所以参数q的变化范围是0-n-1

猜测$T(n) \le cn^2$成立，其中c为常数，将此式带入递归式中，得到

$$\begin{aligned} &T(n) \le \max_{0\le q\le n-1}(cq^2+c(n-q-1)^2)+O(n) \\ &= c\cdot\max_{0\le q\le n-1}(q^2+(n-q-1)^2)+O(n) \end{aligned}$$

表达式$q^2+(n-q-1)^2$在参数取值区间0≤q≤n-1的端点取到最大值，即

$$T(n) \le cn^2 - c(2n-1)+O(n) \le cn^2$$

因此我们可以选择一个足够大的常数c，使得c(2n-1)项能够显著大于O(n)项，所以有$T(n)=O(n^2)$

### Expected running time

定义

$$\begin{aligned} &1.\text{定义数组}A\text{中的各个元素重新命名为}z_1,z_2,\ldots,z_n \\ &\text{其中}z_i\text{是数组}A\text{中第}i\text{小的元素} \\ &2.\text{定义}Z_{ij}=\{z_i,z_{i+1},\ldots,z_j\}\text{为}z_i\text{到}z_j\text{之间元素的集合} \\ &3.\text{定义}X_{ij}=I\{z_i\text{和}z_j\text{进行比较一次}\} \end{aligned}$$

我们考虑的是比较操作是否在算法执行过程中的任意时间发生，而不是局限在循环的一次迭代或者分治中的一次调用是否发生，因为每一对元素最多被比较一次，所以我们很容易刻画出算法的总比较次数

$$X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}$$

对上式两边取期望，再利用期望值的线性特性可以得到

$$E(X) = E[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}Pr\{z_i\text{和}z_j\text{进行比较}\}$$
$$Pr\text{表示}z_i\text{和}z_j\text{进行比较的概率}$$

在随机快排中，主元的选择是随机的，让我们考虑两个元素何时不会进行比较的情况

假设快排中的一个输入，它由数字1到10所构成，顺序任意，并且假设第一个主元是7，那么分治的第一次调用将把这些数字划分成两个集合：{1,2,3,4,5,6}和{8,9,10}，在这一过程中，主元7要和其他所有元素进行比较，但是第一个集合中的任何一个元素都不会与第二个集合中的任何元素比较

通常我们假设每个元素的值是互异的，因此一旦满足z_i<x<z_j的主元x被选择后，我们就知道z_i和z_j以后再也不可能进行比较了，另一种，如果z_i在Z_ij中的所有其他元素之前被选为主元，那么z_i将与Z_ij中除了它自身以外的所有元素进行比较。

因此，z_i和z_j会进行比较，当且仅当Z_ij中被选为主元的第一个元素是z_i或者z_j

在Z_ij中某个元素被选为主元之前，整个集合Z_ij的元素都属于某一划分的同一分区，因此Z_ij中的任何元素都会等可能的被首先选为主元，因为集合Z_ij中有j-i+1个元素，并且主元的选择是随机且独立的，所以任何元素被首先选为主元的概率为1/(j-i+1))，于是我们有：

$$Pr\{z_i和\ z_j进行比较\} = \frac{2}{j-i+1}$$

故整和起来有c

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

在求这个累加和的时候

$$\begin{aligned}
&令\ k = j - i \\
&E(X) = \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \textstyle\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n-1} O(logn) = O(nlogn) \\
&调和级数：\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \ logn, when\ \ n \to \infty
\end{aligned}$$

# Debug

## 拿到题目分析限定时间所需要的时间复杂度和空间复杂度

- 1s = 10^8 次运算

## 数据类型

拿到题目开始就计算一下最大输出值是不是爆`int`了

如果爆了就直接改用`long`

除了`数组大小`和`test case`以外的所有变量都改成`long`！！