

BMI3 Lecture 1.2

Algorithms and Complexity

Wanlu Liu
ZJU-UoE Institute

Wanlulu@intl.zju.edu.cn

2022/09/15

BMI3 Week1 2022

Learning objectives for this lecture

- Describe the principles of algorithm complexity
- Formulate different sorting algorithms
- Understand and compare different algorithm design techniques

Fast vs. Slow algorithms

Iterative  Recursive

FIBONACCI(n)

```
1   $F_1 \leftarrow 1$   
2   $F_2 \leftarrow 1$   
3  for  $i \leftarrow 3$  to  $n$   
4       $F_i \leftarrow F_{i-1} + F_{i-2}$   
5  return  $F_n$ 
```

RECURSIVEFIBONACCI(n)

```
1  if  $n=1$  or  $n=2$   
2      return 1  
3  else  
4       $a \leftarrow \text{RECURSIVEFIBONACCI}(n-1)$   
5       $b \leftarrow \text{RECURSIVEFIBONACCI}(n-2)$   
6      return  $a+b$ 
```



Guess which algorithms is faster?

of operations in an algorithms

ArrayMax

Output the maximal element in an array.

Input: An array A with n elements.

Output: the maximal element within the array A .

Pseudocode

ArrayMax(A, n)

1	currentMax $\leftarrow a_1$	1
2	for $i \leftarrow 1$ to $n-1$	n
3	if $a_i > \text{currentMax}$	$(n-1)$
4	currentMax $\leftarrow a_i$	$(n-1)$
5	return currentMax	1

operations

Total $3n$



15 mins

2022 BMI3 Week 1.2 - Session 1 - ArrayMax

[编辑试卷](#)

[编程题 1](#)

标号	标题	分数	提交通过率
7-1	Array Max	30	0/0(0.00%)

The complexity for ArrayMax is $O(n)$, a **linear** complexity algorithm

Algorithms complexity

- Computer scientists use the *Big-O* notation to describe concisely the running time of an algorithm.
 - If we say the running time of an algorithm is quadratic, or $O(n^2)$, it means the running time of the algorithm on an input of size n is limited by a quadratic function of n .
 - The limit may be $99.7n^2$ or $0.001n^2$ or $5n^2+3.2n+9993$
- $f(n)=O(n^2)$ mean that the function $f(n)$ does not grow faster than a function with a leading term of cn^2 , for a suitable choice of the constant c .
- **Definition** A function $f(x)$ is $O(g(x))$ if there are positive real constants c and x_0 such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$

Algorithms complexity – Big O notation

- **Definition** A function $f(x)$ is $O(g(x))$ if there are positive real constants c and x_0 such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$
- $g(x)$ is an **asymptotic upper bound** for $f(n)$
- **Examples:**

$$n^2 = O(n^2)$$

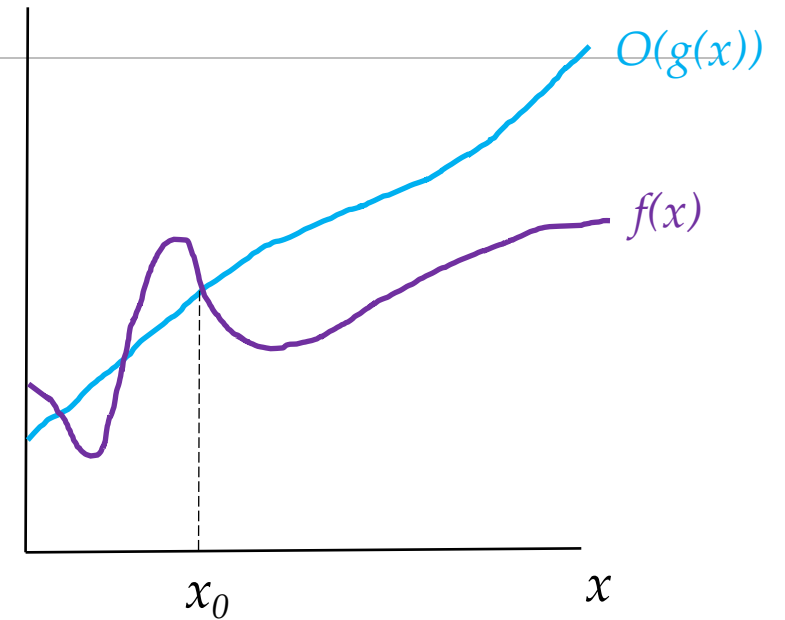
$$n^2 + n = O(n^2)$$

$$n^2 + 1000n = O(n^2)$$

$$5230n^2 + 1000n = O(n^2)$$

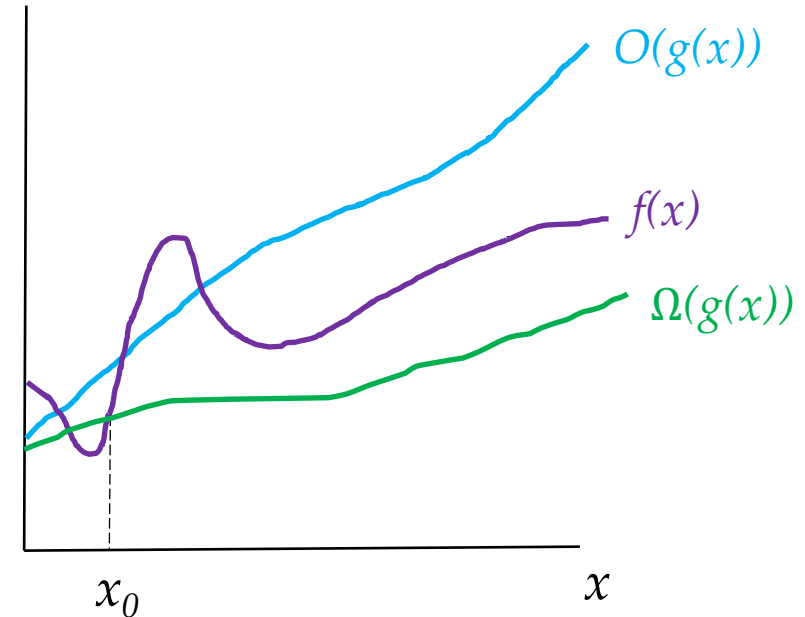
$$10n^2 + 4n + 2 = O(n^2) \text{ because } 10n^2 + 4n + 2 \leq 11n^2 \text{ for all } n \geq 5$$

$$6 \times 2^n + n^2 = O(2^n) \text{ because } 6 \times 2^n + n^2 \leq 7 \times 2^n \text{ for all } n \geq 4$$



Algorithms complexity – Big Omega, Big Theta notations

- **Definition** A function $f(x)$ is $\Omega(g(x))$ if there are positive real constants c and x_0 such that $f(x) \geq cg(x)$ for all values of $x \geq x_0$
- f is said to grow “**faster**” than g
- $g(x)$ is an **asymptotic lower bound** for $f(x)$
- If $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$, then we know very precisely how $f(x)$ grows with respect to $g(x)$. We call this the Θ relationship.
- **Definition** A function $f(x)$ is $\Theta(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$

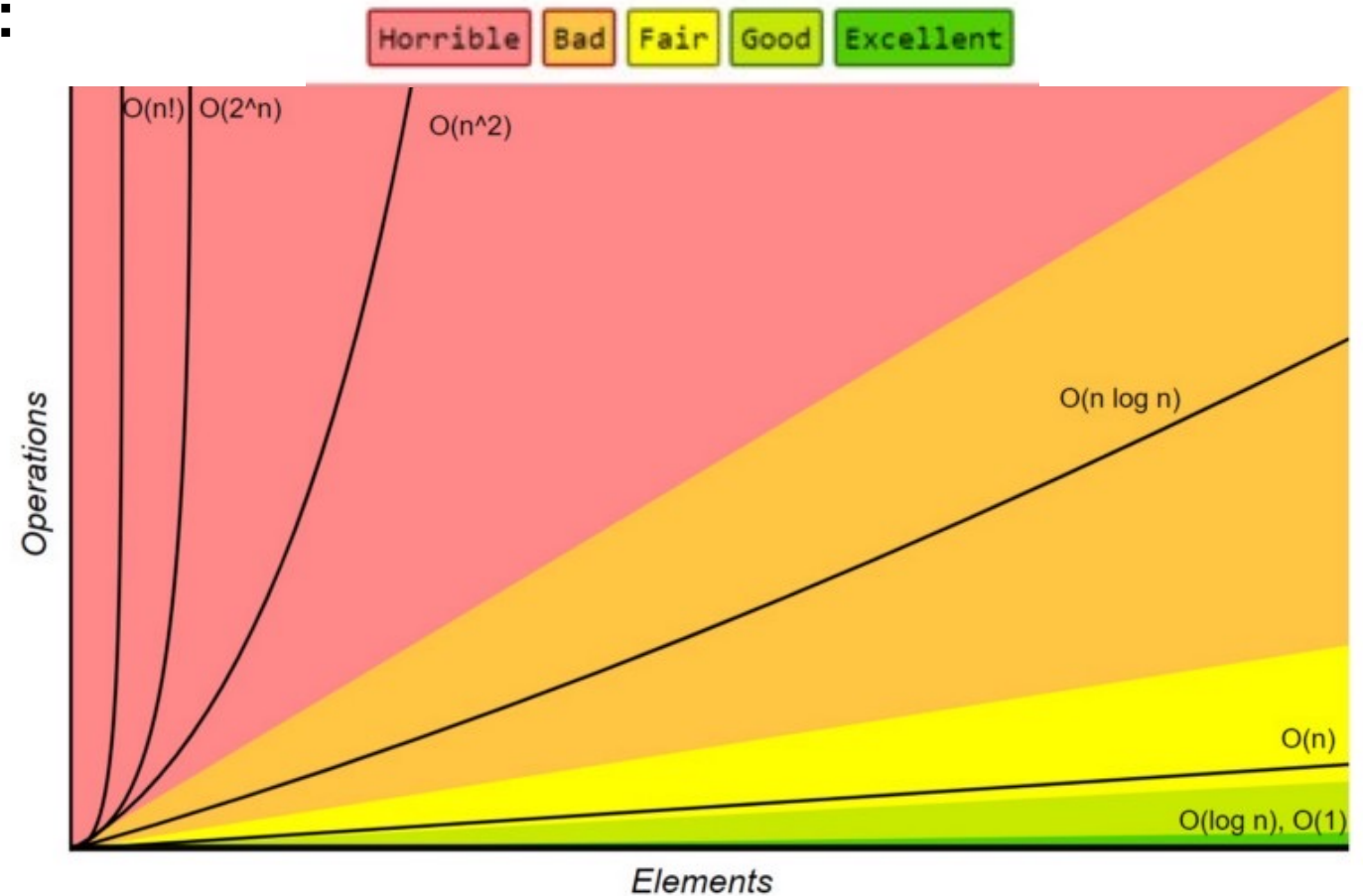


$O(g(x))$ – worst case; $\Omega(g(x))$ – best case; $\Theta(g(x))$ – average case;

Complexity of Algorithms

Algorithms complexity can be :

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: linearithmic
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential
- $O(n!)$: factorial



Complexity for Recursive Fibonacci

RECURSIVEFIBONACCI(n)

```
1  if n=1 or n=2
2    return 1
3  else
4    a ← RECURSIVEFIBONACCI(n-1)
5    b ← RECURSIVEFIBONACCI(n-2)
6    return a+b
```

Let's use $T(n)$ to denote the time complexity of **RECURSIVEFIBONACCI**(n), therefore, for $n > 1$:

$$T(n) = T(n-1) + T(n-2) + 1$$

When $n = 0$ and $n = 1$, no additions occur, thus,

$$T(0) = T(1) = 0 = O(1)$$

We only need to solve $T(n)$

Let's assume $T(n-2) \approx T(n-1)$ then,

$$T(n) = T(n-1) + T(n-1) + 1 = 2 * T(n-1) + 1$$

$$T(n) = 2 * [2 * T(n-2) + 1] + 1 = 4 * T(n-2) + 3$$

$$\dots = 8 * T(n-3) + 7$$

$$\dots = 16 * T(n-4) + 15$$

...

$$\dots = 2^k * T(n-k) + (2^k - 1)$$

$$\dots = 2^n * T(0) + (2^n - 1)$$

$$\dots = 2^n O(1) + 2^n - 1$$

$$\dots = O(2^n)$$

Complexity for Iterative Fibonacci

FIBONACCI(n)

1 $F_1 \leftarrow 1$

2 $F_2 \leftarrow 1$

3 **for** $i \leftarrow 3$ **to** n

4 $F_i \leftarrow F_{i-1} + F_{i-2}$

5 **return** F_n

Let's use $T(n)$ to denote the time complexity of FIBONACCI(n), therefore, for $n > 1$:

$$T(n) = (n - 1) - 2 = O(n - 3) = O(n)$$

When $n = 0$ and $n = 1$, no additions occur, thus,

$$T(0) = T(1) = 0 = O(1)$$

Therefore, the time complexity is

$$O(1) + O(1) + O(n) = O(n)$$

The Sorting Problem

Sorting Problem:

Sort a list of integers

Input: A list of n distinct integers $a = (a_1, a_2, \dots, a_n)$

Output: Sorted list of integers, that is, a reordering $b = (b_1, b_2, \dots, b_n)$ of integers from a such that $b_1 < b_2 < \dots < b_n$

- **Computational complexity** (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical sorting algorithms, a good behavior is $O(n \log n)$ and a bad behavior is $O(n^2)$.
- **Recursion:** Some algorithms are either recursive or non-recursive.
- **Stability:** Stable sorting algorithms maintain the relative order of records with equal values.
- Whether or not they are a **comparison sort**. A comparison sort examines the data only by comparing two elements with a comparison operator.
- **Adaptability:** Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.
- General methods: **insertion, exchange, selection, merging**, etc.

The Sorting Problem

Algorithm	Best	Average	Worst
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Radix Sort	$\Omega(n \cdot k)$	$\Theta(n \cdot k)$	$O(nk)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$

Bubble Sort Algorithm

Intro

- **Bubble sort**, also referred to as **comparison sort**, is a simple sorting algorithm that repeatedly goes through the list, compares adjacent elements and swaps them if they are in the wrong order.
- This is the **most simplest** algorithm and inefficient at the same time.
- Yet, it is very much necessary to learn about it as it represents the basic foundations of sorting.

Algorithm: We compare adjacent elements and see if their order is wrong (i.e $a_i > a_j$ for $1 \leq i < j \leq \text{size of array}$; if array is to be in ascending order, and vice-versa). If yes, then swap them.

Bubble Sort algorithm -2

Consider the following array: 14, 33, 27, 35, 10

0	1	2	3	4
14	33	27	35	10


First Pass:

1. We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if $14 > 33$ which is false. So, **no swapping**.


0	1	2	3	4
14	33	27	35	10

2. We proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if $33 > 27$ which is true. So, we **swap Arr[1] and Arr[2]**.

0	1	2	3	4
14	33	27	35	10



0	1	2	3	4
14	27	33	35	10




3. We proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if $33 > 35$ which is false. So, **no swapping**.


0	1	2	3	4
14	27	33	35	10

4. We proceed with the fourth and fifth element i.e., Arr[3] and Arr[4]. Check if $35 > 10$ which is true. So, we swap **Arr[3] and Arr[4]**.

0	1	2	3	4
14	27	33	35	10



0	1	2	3	4
14	27	33	10	35



5. Thus, marks the end of the first pass, where the **Largest element reaches its final(last) position**.

Bubble Sort algorithm -3

Consider the following array: 14, 33, 27, 35, 10

0	1	2	3	4
14	27	33	10	35

Second Pass:

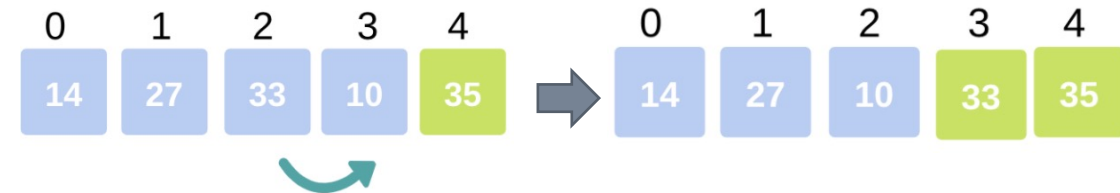
1. We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if $14 > 27$ which is false. So, **no swapping**.

0	1	2	3	4
14	27	33	10	35

2. We now proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if $27 > 33$ which is false. So, **no swapping**.

0	1	2	3	4
14	27	33	10	35

-
3. We now proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if $33 > 10$ which is true. So, we **swap Arr[2] and Arr[3]**.



4. Thus marks the end of second pass where the **second largest element in the array has occupied its correct position**.
-

Bubble Sort algorithm -4

Consider the following array: 14, 33, 27, 35, 10

0	1	2	3	4
14	27	10	33	35

Third Pass:

1. After the third pass, the **third largest element will be at the third last position** in the array.

0	1	2	3	4
14	10	27	33	35

i-th Pass:

1. After the ith pass, the **ith largest element will be at the ith last position** in the array.
-

n-th Pass:

1. After the nth pass, the **nth largest element(smallest element) will be at nth last position(1st position)** in the array, where 'n' is the size of the array.

After doing all the passes, we can easily see the array will be sorted.

0	1	2	3	4
10	14	27	33	35

Pseudocode for Bubble Sort



Write pseudocode &
implement in python3
40 mins

2022 BMI3 Week 1.2 – Session 2 – BubbleSort

 编辑

 编程题 1

标号	标题	分数	提交通过率
7-1	Bubble Sort	50	0/0(0.00%)

Insertion Sort algorithm

Intro

- Insertion sort is the sorting mechanism where the sorted array is built having one item at a time.
- The array elements are compared with each other sequentially and then arranged simultaneously in some particular order.
- The analogy can be understood from the style we arrange a deck of cards.
- This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

Working

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the element in the array is at their apt position.

Insertion Sort algorithm -2

Consider the following array: 25, 17, 31, 13, 2

First Iteration:

1. Compare 25 with 17. The comparison shows $17 < 25$.
2. Hence swap 17 and 25.

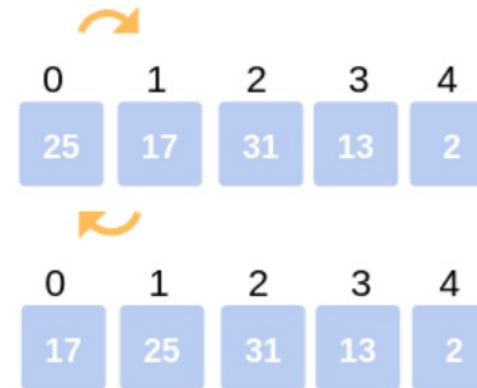
The array now looks like: 17, 25, 31, 13, 2

Second Iteration:

1. Begin with the second element (25), but it was already swapped on for the correct position, so we move ahead to the next element.
2. Now hold on to the third element (31) and compare with the ones preceding it.
3. Since $31 > 25$, no swapping takes place.
4. Also, $31 > 17$, no swapping takes place and 31 remains at its position.

The array now looks like: 17, 25, 31, 13, 2

First Iteration



Second Iteration



Insertion Sort algorithm -3

Array after 2nd iteration: **17, 25, 31, 13, 2**

Third Iteration:

1. Start the following Iteration with the fourth element (13), and compare it with its preceding elements.
2. Since $13 < 31$, we swap the two.

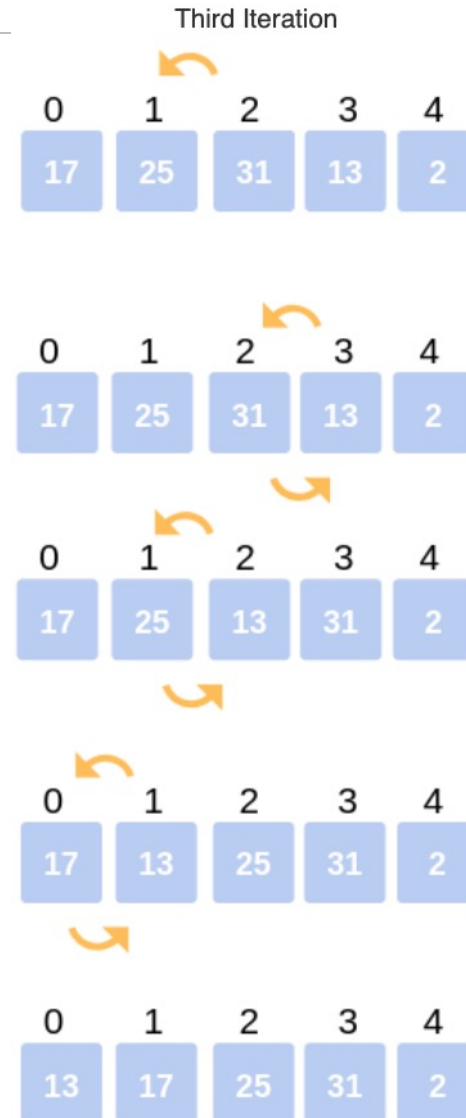
Array now becomes: **17, 25, 13, 31, 2**.

3. But there still exist elements that we haven't yet compared with 13. Now the comparison takes place between 25 and 13. Since, $13 < 25$, we swap the two.

The array becomes **17, 13, 25, 31, 2**.

4. The last comparison for the iteration is now between 17 and 13. Since $13 < 17$, we swap the two.

The array becomes **13, 17, 25, 31, 2**.

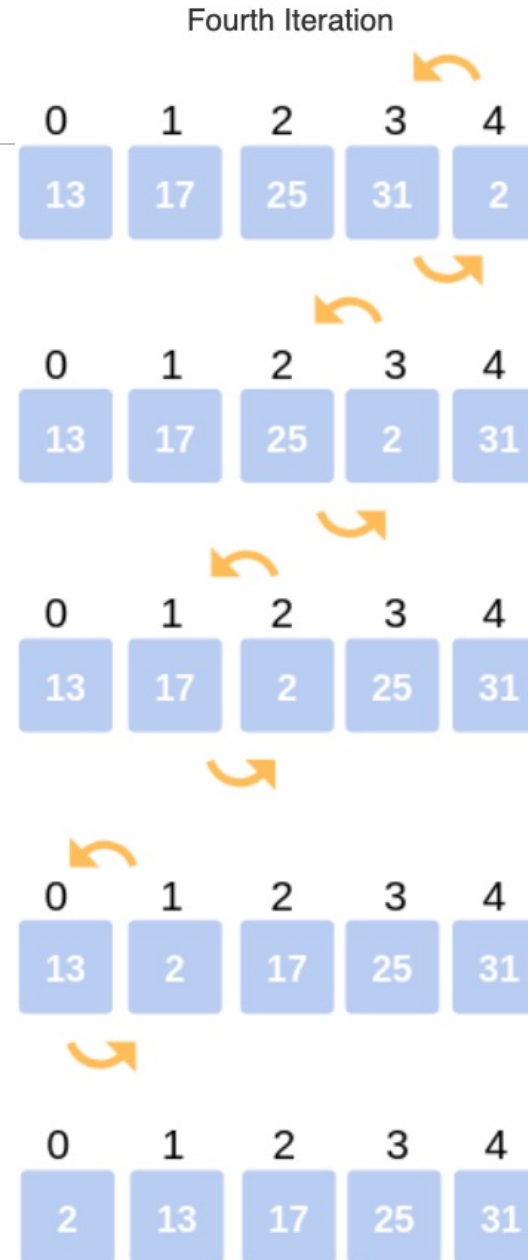


Insertion Sort algorithm -4

Array after 3rd iteration: **13, 17, 25, 31, 2**.

Fourth Iteration:

1. The last iteration calls for the comparison of the last element (2), with all the preceding elements and make the appropriate swapping between elements.
2. Since, $2 < 31$. Swap 2 and 31.
Array now becomes: **13, 17, 25, 2, 31**.
3. Compare 2 with 25, 17, 13. Since, $2 < 25$. Swap 25 and 2.
Array now becomes: **13, 17, 2, 25, 31**.
4. Compare 2 with 17 and 13. Since, $2 < 17$. Swap 2 and 17.
Array now becomes: **13, 2, 17, 25, 31**.
5. The last comparison for the Iteration is to compare 2 with 13. Since $2 < 13$. Swap 2 and 13.
The array now becomes: **2, 13, 17, 25, 31**.



Pseudocode for Insertion Sort



Write pseudocode &
implement in python3
40 mins

2022 BMI3 Week 1.2 – Session 3 – InsertionSort

 编辑试卷

 编程题 1

标号	标题	分数	提交通过率
7-1	Insertion Sort	50	0/0(0.00%)

Quick Sort Algorithms

Intro

- **Quick sort** is a **Divide and conquer based** algorithm that picks a **pivot** element from the given array and **partition**(rearranges) the array in such a way that
 - elements in subarray **before** pivot are **less than or equal to the pivot**
 - element in subarray **after** pivot are **greater than pivot** (but not necessarily in sorted order).
- **Fast & widely used in our daily life**

Working

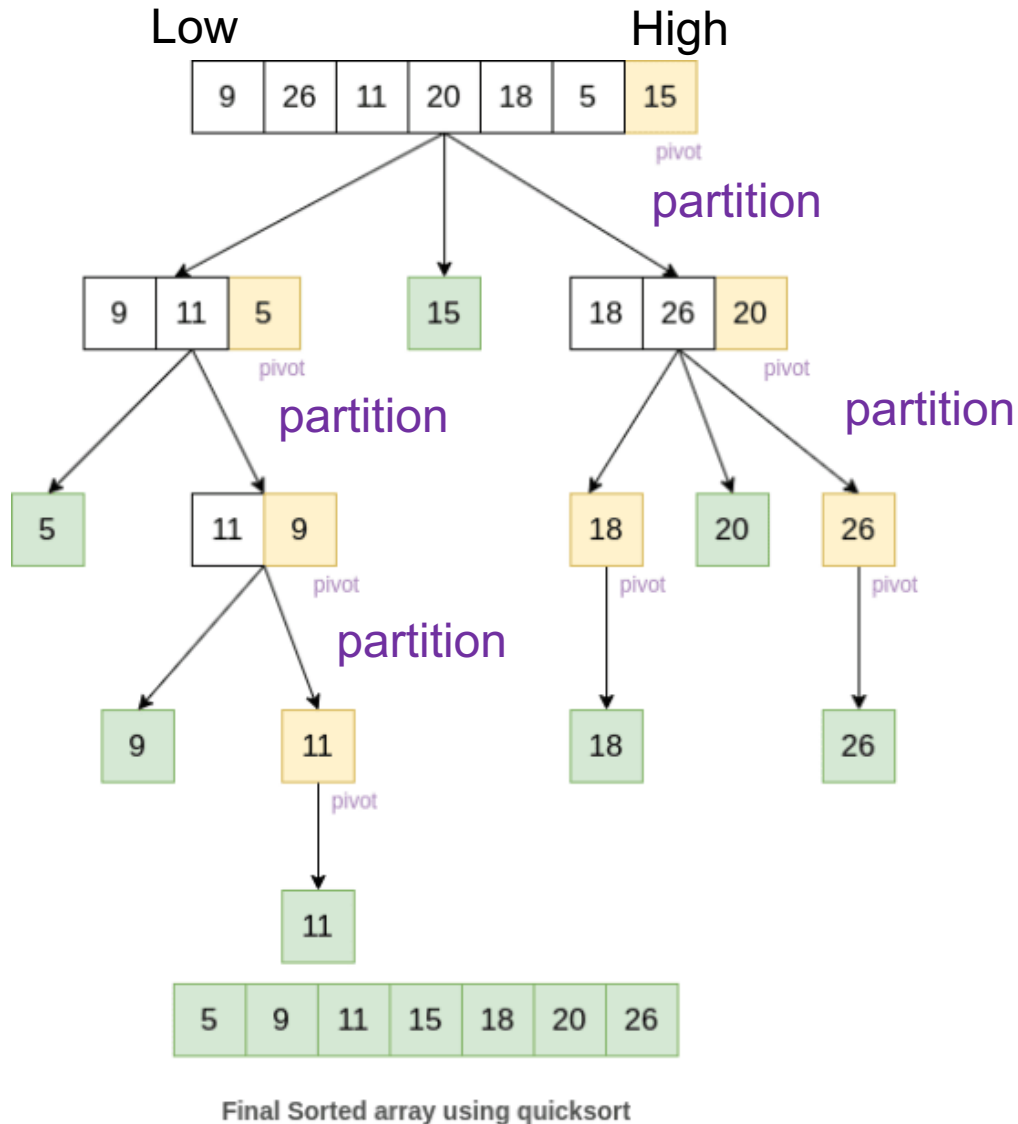
1. Pick an element, called the *pivot*, from the list.
2. Reorder the list so that
 - all elements which are less than the pivot come before the pivot and
 - so that all elements greater than the pivot come after it (equal values can go either way).
 - After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements. The base case of the recursion are lists of size zero or one, which are always sorted.

Quick Sort Algorithms



30 mins

1-based index



QUICKSORT(A, low, high)

```
1  if low < high
2    pivot_index = PARTITION(A, low, high)
3    QUICKSORT(A, low, pivot_index-1)
4    QUICKSORT(A, pivot_index+1, high)
```

PARTITION(A, low, high)

```
1  pivot ← A[high]
2  i ← low
3  tmp ← 0
4  for j ← low+1 to high
5    if Aj < pivot
6      i = i + 1
7      tmp ← Ai
8      Ai ← Aj
9      Aj ← tmp
10 Ai+1 ← pivot
11 A[high] ← Ai+1
12 return i + 1
```


Quick Sort Algorithms

2022 BMI3 Week 1.2 - Session 4 - QuickSort

 编辑试卷

 编程题 1

标号	标题	分数	提交通过率
7-1	Quick Sort	50	0/0(0.00%)



40 mins

General algorithm design techniques 1

Over the last forty years, computer scientists have discovered that many **algorithms share similar ideas**, even though they solve very different problems.

To illustrate the design techniques, we will consider a very simple problem. Suppose your cell phone rings, but you have misplaced it somewhere in your home.



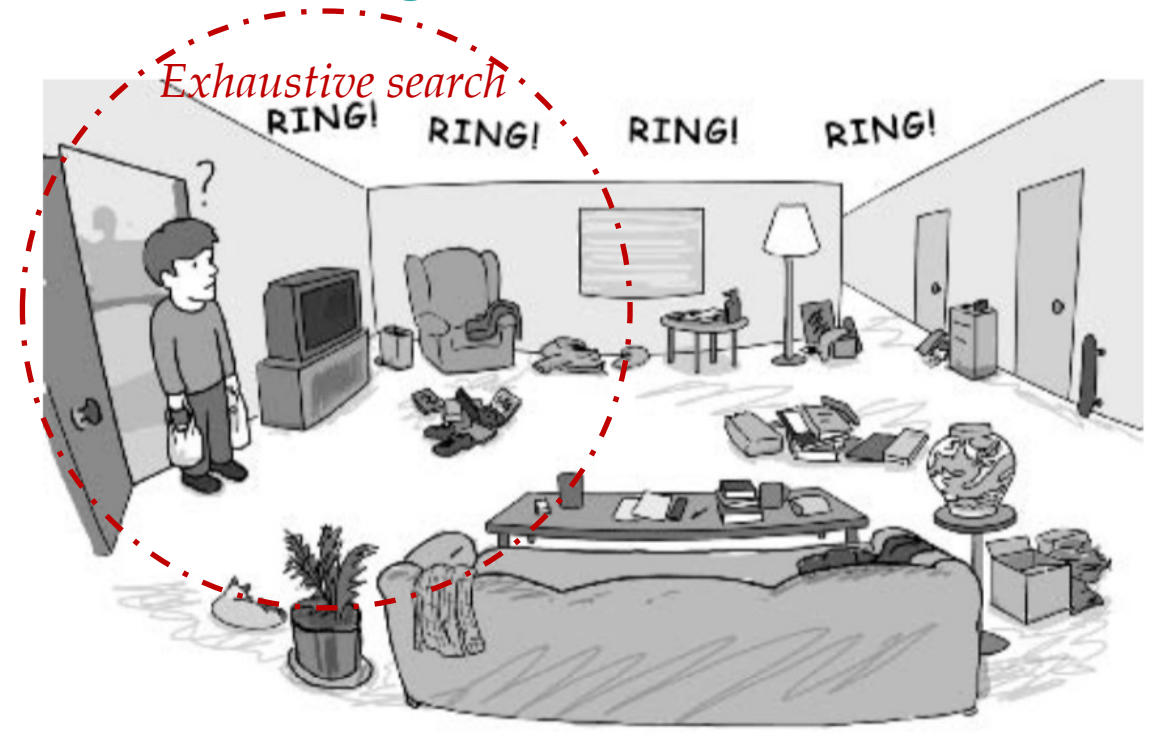
**How do you find
your cell phone?**

General algorithm design techniques 2

Exhaustive Search (or brute force) algorithms
“search every possible corners”

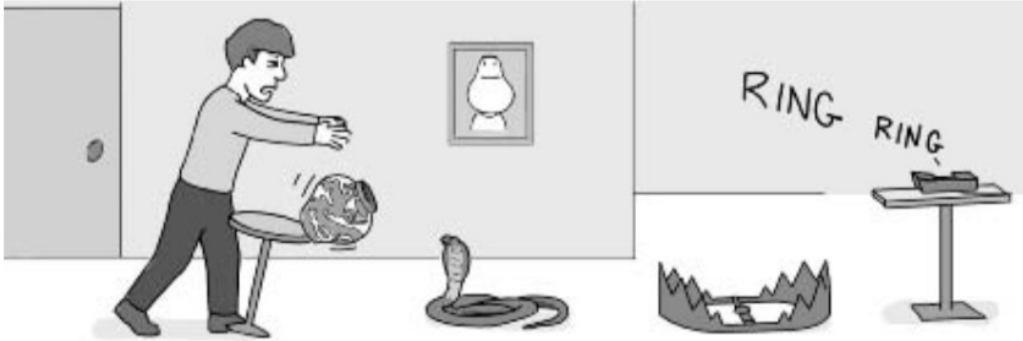


Branch-and-Bound Algorithms (or pruning)
“omit a large number of alternatives”

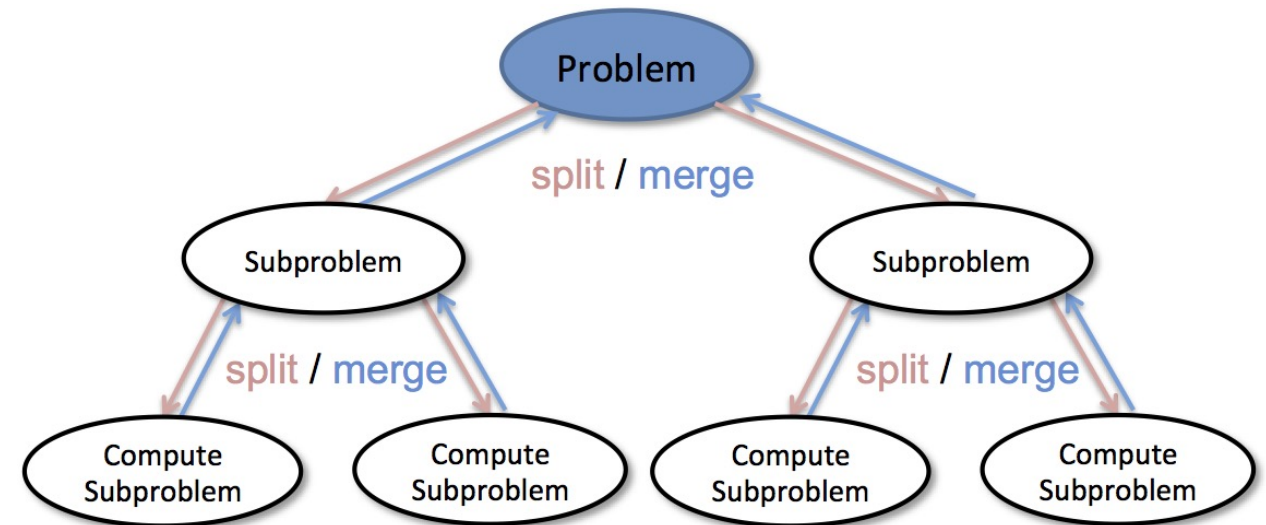


General algorithm design techniques 3

Greedy algorithms
“most attractive”

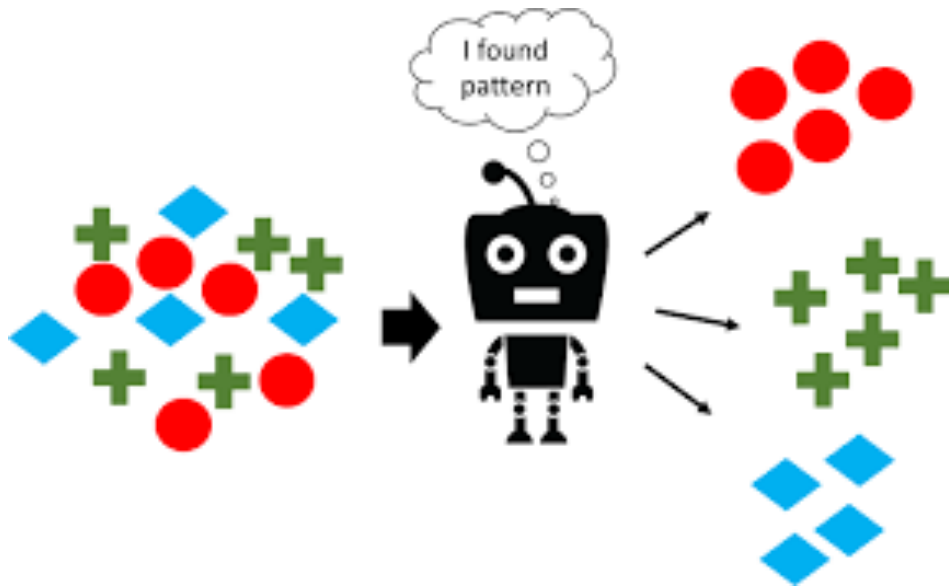


Divide-and-Conquer Algorithms



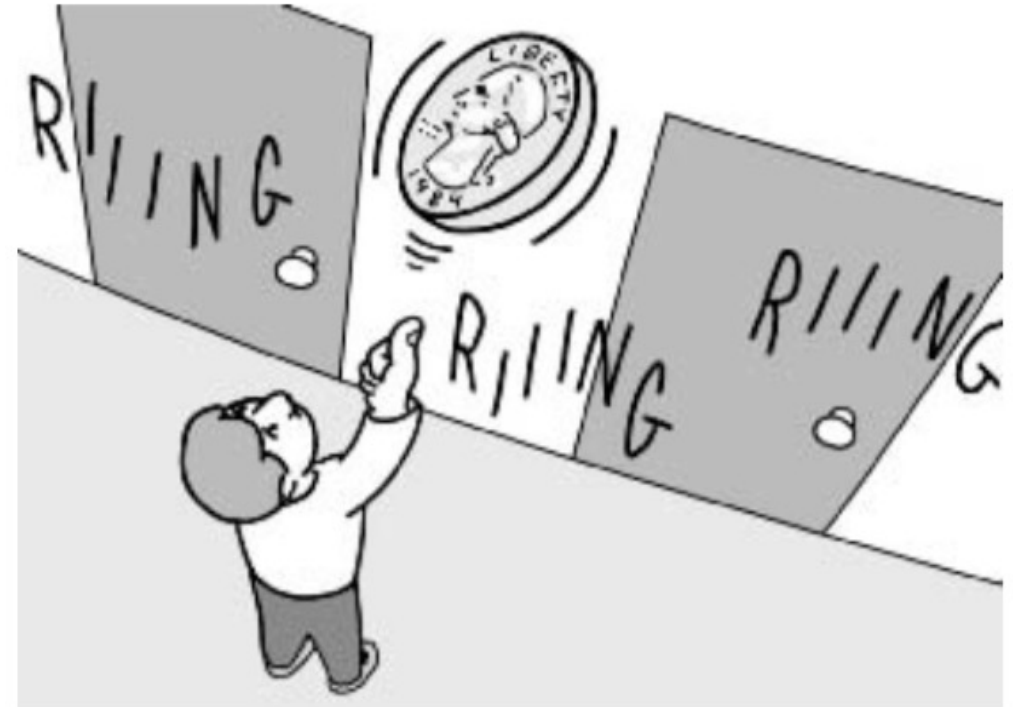
General algorithm design techniques 4

Machine Learning Algorithms



Based on previous collected data, the phone was left in the bathroom 80% of the time, in the bedroom 15% of the time, and in the kitchen 5% of the time

Randomized algorithms



General algorithm design techniques 5

Dynamic programming

SUMINTEGERS(2)=1
+2=3 **record this**,
SUMINTEGERS(3)=
SUMINTEGERS(2)+3
=3+3=6

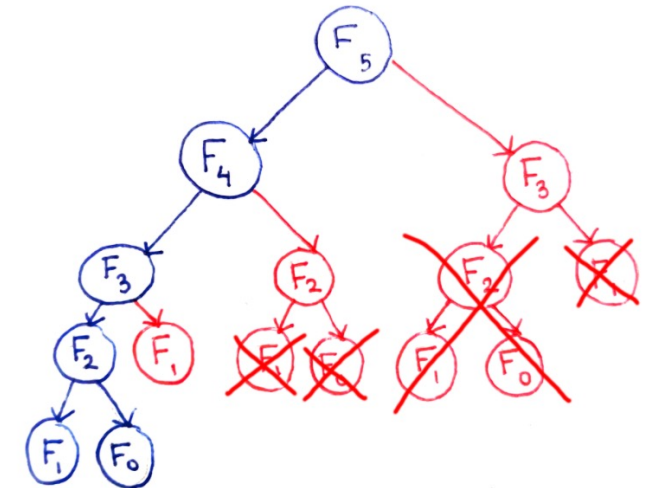
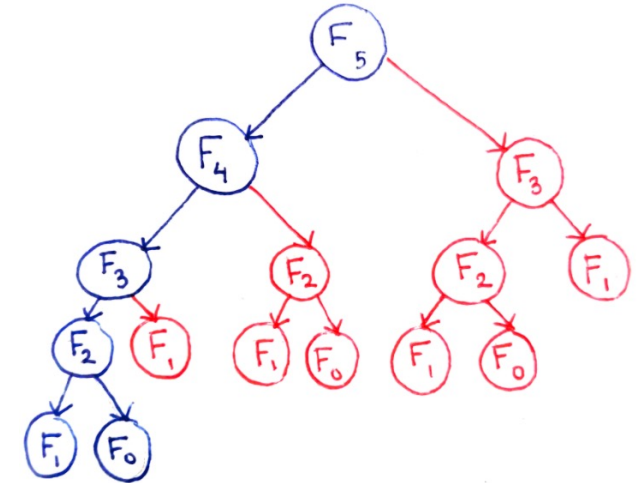
RECURSIVEFIBONACCI(n)

```
1  if  $n=1$  or  $n=2$ 
2    return 1
3  else
4     $a \leftarrow \text{RECURSIVEFIBONACCI}(n-1)$ 
5     $b \leftarrow \text{RECURSIVEFIBONACCI}(n-2)$ 
6    return  $a+b$ 
```

FIBONACCI(n)

```
1   $F_1 \leftarrow 1$ 
2   $F_2 \leftarrow 1$ 
3  for  $i \leftarrow 3$  to  $n$ 
4     $F_i \leftarrow F_{i-1} + F_{i-2}$ 
5  return  $F_n$ 
```

This is dynamic programming!



Summary

- Describe the principles of algorithm complexity
 - Big O, Big Omega, Big Theta notations
- Formulate different sorting algorithms
 - Bubble sort, insertion sort, quick sort ...
- Understand and compare different algorithm design techniques
 - Exhaustive search, greedy algorithm, branch-and-bound, divide-and-conquer, machine learning, randomized algorithm

