COMP2013
Data Structures and Algorithms

Lecture 1. Introduction

Dr. Jieming Shi
COMP2013

Contact Information (COMP2013_20241_B)

- Lecture
  - Jieming SHI
    - jieming.shi@polyu.edu.hk
    - PQ832
- Teaching Assistants

MA Zhiyuan	21123501r@connect.polyu.hk
ZHANG Zhengqiang	22040257r@connect.polyu.hk
WANG Pengfei	22041247r@connect.polyu.hk
YI Qiaosi	23125228r@connect.polyu.hk

- Consultation hours:
  - Wednesday 13:30-15:30

Contact Information (COMP2013_20241_A)(Sat)

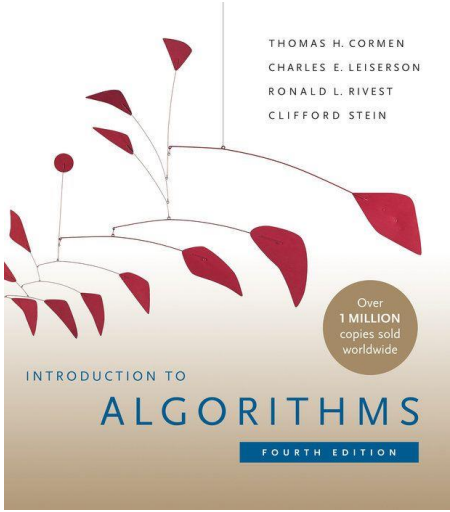
- Lecture
  - Jieming SHI
    - jieming.shi@polyu.edu.hk
    - PQ832
- Teaching Assistants

DING Zhihao	22040455r@connect.polyu.hk
LIU Shuaizheng	22074587r@connect.polyu.hk
LI Huahang	23123034r@connect.polyu.hk
MA Zijing	23132844r@connect.polyu.hk

- Consultation hours:
  - Tuesday 15:30-17:30

Textbook

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.
  Introduction to Algorithms, 4th Edition,
  MIT Press, 2022.
  - The examples in the textbook are clear and excellent for understanding
- You are recommended to read the corresponding textbook chapter after each class



Course materials

- https://learn.polyu.edu.hk/
- Please check *Blackboard* regularly!
  - Announcement
  - Lecture slides
  - Tutorial / lab exercises
  - Assignments
  - Solutions

- Please check your PolyU email account regularly
- Please check your timetable in PolyU eStudent
  - Lecture: 2 hours per week
  - Lab/Tutorial: 1 hour per week (start from Week 2) (check the location carefully)

Tentative schedule

No.	Topic
1	Introduction
2	Analysis of Algorithms
3	Divide-and-Conquer (Assignment 1)
4	Sorting Algorithms
5	Binary Heap (Programming assignment 1)
6	Elementary Data Structures
7	Hashing and Binary Search Tree I

No.	Topic
8	Binary Search Tree II (Quiz)
9	Balanced Binary Search Tree (Assignment 2)
10	Dynamic Programming
11	Greedy Algorithms (Programming assignment 2)
12	Graph Algorithms
13	Revision

- Foundations: 1-3
- Sorting: 4-5
- Data structures: 6-9
- Algorithm design: 3, 10-11

In quiz & the exam, we may ask questions about algorithms/pseudo-codes, but not about programming language (C/C++/Java/Python)

Notes

- **Not** a programming course
  - You have taken a programming course before
- In each lecture
  - Get the main ideas and concepts
  - Try to be active and think about my questions

Try to **understand** and **apply**, not just memorize

Assessments

- Continuous Assessment: 60%
  - 2 written assignments: 15%
  - 2 programming assignments: 20%
  - 1 quiz: 25%
- Exam: 40%

- More information
  - Individual assignments
  - Programming assignments
    - You may choose C/C++/Java/Python
    - We may impose some restrictions on using libraries

Late policy & plagiarism

- Late policy:
  - Your responsibility: start early & submit assignment on time!
  - *x* hours late: $\left(10 \times \left\lceil \frac{x}{12} \right\rceil\right)$% discount on your marks

    e.g., 12 hours late → 10% discount

    24 hours late → 20% discount
- PolyU plagiarism booklet:

  https://www.polyu.edu.hk/ogur/docdrive/Academic_Integrity/Plagiarism_Booklet.pdf
  - Please read "How do I avoid plagiarism?" and "Fast guide to citing and referencing"
- Penalties for plagiarism:
  - 0 marks will be given to plagiarized works
  - Serious cases will be reported to department

Let’s start our journey!

- Try to think & participate actively in the class

Outline

- **A little about Programming Basics**
- Problems, Problem Instances, and Algorithms
- Insertion Sort Algorithm, Pseudocode
- Analysis of Algorithms
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort
- Analysis of Recursive Algorithms

A glance of designing and analyzing algorithms

*May not finish everything today

A little about programming basics (mainly in C++)

- Building blocks in a C++ program:
  - Constants (e.g., `3`, `"x*x + x = "`),
  - Variables (e.g., `x`) for storing values,
  - Operators (e.g., `+`, `=`, `<<`) for executing operations

stored in memory

→ Expressions (e.g., `x*x+x`)

→ Statements (e.g., `x=3;`)

→ Functions (e.g., `main`)

executed in processor

Programming basics

- The execution starts at the *main* method

Write a program in the file "test.cpp"

```
#include <iostream>
using namespace std;

int main() {
    int x,y;
    x = 3;
    y = x*x+x;
    cout << "x*x+x = " << y << endl;
    return 0;
}
```

Compile a program

```
g++ -o test.exe test.cpp
```

Run a program

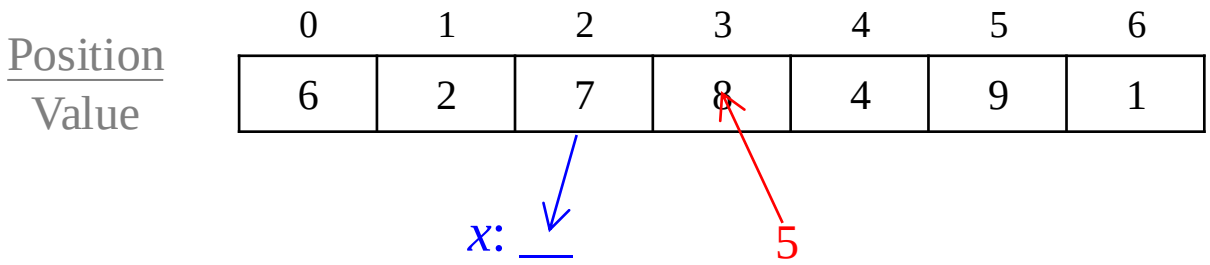
```
test.exe

x*x+x = 12
```

C++ Array

- *Array*: a data structure for storing multiple variables (of the same type)
  - Create an array *A* with *n* integers by

```
int A[n];                    // for small array
int* A = new int[n];         // for large array
```

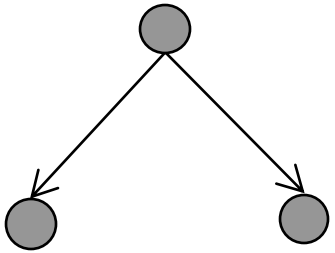
  - *A*[*i*] denotes the variable at position *i* of array *A*
    - In C++, the range of *i* is from 0 to *n* – 1
- Examples
  - `int A[7];` // create an array
  - `x = A[2];` // copy value *A*[2] to variable *x*
  - `A[3] = 5;` // store value 5 to *A*[3]



Control Flow

- **Sequence** (step-by-step)
  - A sequence of statements implies the execution ordering
  - Each statement is a ‘step’, e.g.,
    - `c = a * b;`
    - `cout << "hello" << endl;`
- **Selection** (choose a branch)
  - *if* statement, *switch* statement
  - Example of *if* statement:
    - Evaluate the condition `x>y`
    - Condition is true → execute `v=x;`
    - Condition is false → execute `v=y;`

```
int a,b,c,d;
a = 3;
b = 5;
c = a * b;
d = a + b;
```



```
if (x>y)
    v=x;
else
    v=y;
```

Control Flow

- **Loop**: repeat the execution of the loop body (a sequence of statements) for multiple times
- Example: print “hello” 5 times

```
int i = 5;
while (i>0) {
    cout << "hello" << endl;
    i--;
}
```

- *while*-loop
  - Evaluate <cond> **before** the loop body; the loop stops when <cond> is false
- *for*-loop: counter-based
  - Execute <init> once before the loop starts
  - Evaluate <cond> **before** the loop body; the loop stops when <cond> is false
  - Execute <update> **after** the loop body

```
for (int i=0; i<5; i++) {
    cout << "hello" << endl;
}
```

C++ programming

- How to use the C++ standard template library (STL)?
  - Data structures: https://en.cppreference.com/w/cpp/container
    `stack, queue, list, priority_queue, ……`
  - Algorithms: https://en.cppreference.com/w/cpp/algorithm
    `sort,  find, binary_search, ……`

We focus on designing solutions to solve **problems**

- **Algorithms + data structures**

- You should have already known programming
  - Including but not limited to previous slides

Outline

- A little about Programming Basics
- **Problems, Problem Instances, and Algorithms**
- Insertion Sort Algorithm, Pseudocode
- Analysis of Algorithms
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort
- Analysis of Recursive Algorithms

Problem and instances

- What is a problem? What is a problem instance?

- Sorting Problem
  - **Input**: A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$
    - Problem size: *n*
  - **Output**: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

  - A problem **instance**: ⟨ 31, 41, 59, 26, 41, 58 ⟩
    - Output: 26, 31, 41, 41, 58, 59

Algorithm

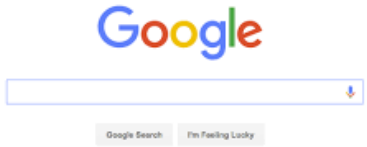
- **Input**: takes some value or a set of values.
- **Output**: produces some value, or set of values in a finite amount of time.
- An algorithm is *a sequence of computational steps* that transform the input into the output.

A correct algorithm

- For every problem instance
  - Halts-finishes its computing in finite time
  - Solves the given computational problem

- An incorrect algorithm
  - Not halt at all on some input instances, or
  - Halt with incorrect answer

Problems and Applications of Algorithms



Algorithms are used everywhere!

Outline

- A little about Programming Basics
- Problems, Problem Instances, and Algorithms
- **Insertion Sort Algorithm, Pseudocode**
- Analysis of Algorithms
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort
- Analysis of Recursive Algorithms

Sorting Problem

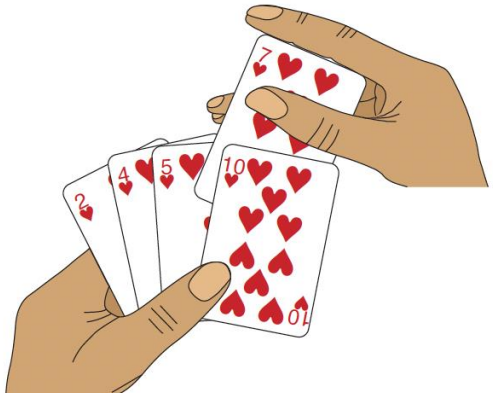
- **Input**: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - Problem size: n
- **Output**: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- A problem **instance**: ⟨ 31, 41, 59, 26, 41, 58 ⟩
  - Output: 26, 31, 41, 41, 58, 59

How?
Any idea?

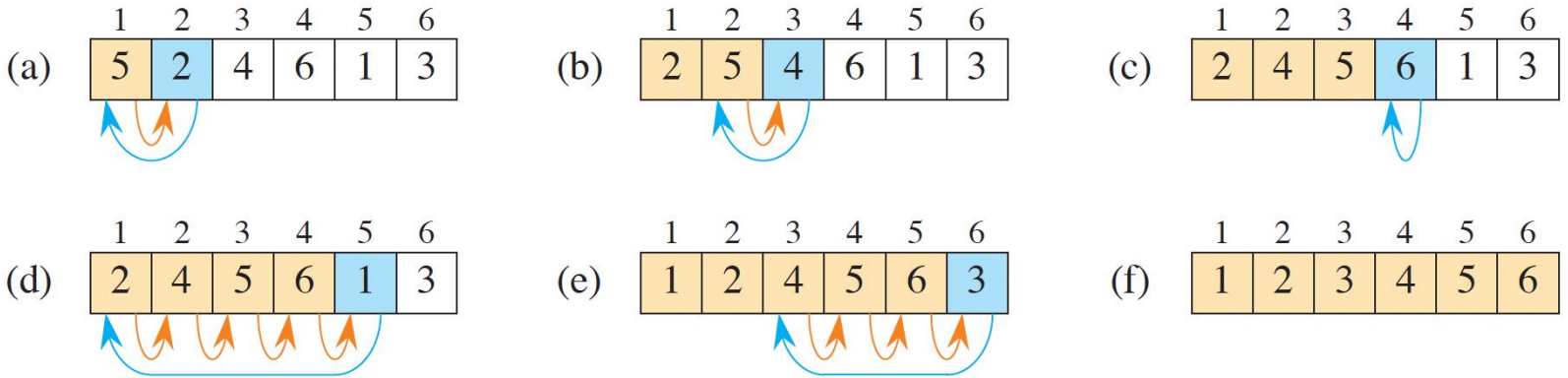
Insertion Sort

- Playing cards
  - Start with an *empty left hand* and a pile of *unsorted cards* on the table
  - Use right hand to get the first card from the pile and **insert** it into the **correct** position in your *left hand*
    - How to insert correctly?
    - Starting at the right and moving left until seeing a card in your left hand less than or equal to the card in right hand, to insert it to the right of this card.
  - At all times, the cards in left hand are sorted.



Insertion Sort

- Perform sorting in an array *A* in place
  - Yellow numbers are the sorted cards in left hand
  - Blue number is the card just got from the table (holding in right hand)
  - White numbers are the unsorted cards on the table



Pseudocode and real code

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

```cpp
// C++ program for implementation of Insertion Sort
#include <iostream>
using namespace std;

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

* In pseudocode we use 1-origin index

Pseudocode

- In pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm.
- Similar in many aspects to programming languages, e.g., C++, C, Java, python, but not real code.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Pseudocode

- Indentation indicates block structure.
  - for, while loops
  - if-else statements
- // for comments
- Local variables

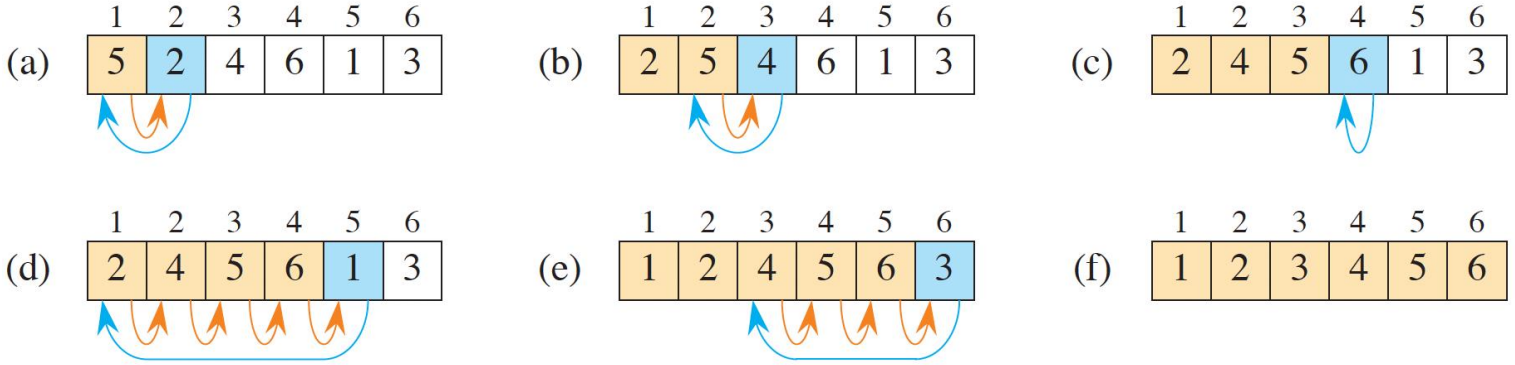
```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Pseudocode

- *A* array with *n* values to sort
- *A[i]* value at *i*-th position
- *A*[*i*:*j*] subarray from *i*-th position to *j*-th position ($i \leq j$)
  - a contiguous portion of the array *A[i],…,A[j]*
- 1-origin indexing v.s. 0-origin indexing

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Insertion Sort Pseudocode




```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

How fast is this algorithm?
How powerful CPU does it require?
How much memory does it require?
…

Outline

- A little about Programming Basics
- Problems, Problem Instances, and Algorithms
- Insertion Sort Algorithm, Pseudocode
- **Analysis of Algorithms**
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort
- Analysis of Recursive Algorithms

How to measure the running time of an algorithm?

- Actually run the algorithm
  - Implementation first
  - What testing environment is used?
  - Exhaust all input instances?
- Analyze the complexity of an algorithm
  - Without implementing it
  - Common Assumptions:
    - A basic instruction has constant time
    - A memory access has constant time
    - No concurrent executions
    - …

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Analyze an algorithm

- Input problem size
  - Number of items n to be sorted (Different problems vary)
- Running time (The number of instructions and data accesses executed)
  - Analyze the running time per line
  - How many times each line is executed
  - Identify the most significant terms in a formula
  - Dependent on problem size

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

t_i is the number of times the while loop test is executed for that value of i (t_i in different instances are different)

Analyze an algorithm

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i - 1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Best case vs **worst case**

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$
- A best-case input instance A_1={1,2,3,4,5}
- A worst-case input instance A_2 ={5,4,3,2,1}

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i - 1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Best case

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$
- A best-case input instance A_1={1,2,3,4,5}
- Best case: $T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
  - Linear function of *n*

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i - 1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Worst case running time

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$
- A worst-case input instance $A_2 = \{5,4,3,2,1\}$
- Worst case: $T(n) = \frac{c_5 + c_6 + c_7}{2}\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$
  - *Quadratic function of n*

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i - 1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i - 1)$
7 $j = j - 1$	c_7	$\sum_{i=2}^{n}(t_i - 1)$
8 $A[j+1] = key$	c_8	$n-1$

Worst case running time

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$
- A worst-case input instance A_2 ={5,4,3,2,1}
- Worst case: $T(n) = \frac{c_5 + c_6 + c_7}{2}\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$
  - *Quadratic function of n*
- Usually concentrate on finding the **worst-case running time**
  - Analyze running time on *any* input
  - Bad cases or close-to-bad cases are fairly common

Order of growth

- $T(n) = \frac{c_5+c_6+c_7}{2}\boldsymbol{n^2} + \left(c_1 + c_2 + c_4 + \frac{c_5-c_6-c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$
  - Only focus on the leading term of a formula (*rate/order of growth*)
    - The lower-order terms are relatively insignificant for large values of n
- Insertion sort: $\Theta(n^2)$
  - Informally, means "roughly proportional to n^2 when n is large"

Order of growth

Function	n=16	n=1024	n=1000000
$\lg n_{(\text{base } 2)}$	4	10	~20
$\sqrt{n}$	4	32	1000
n	16	1024	1000000
$n \lg n$	64	10240	~20000000
n^2	256	1048576	10^{12}
n^3	4096	$\sim 10^9$	10^{18}
2^n	65536	… too big to show	…
$n!$	$\sim 2 * 10^{13}$	…	…

Outline

- A little about Programming Basics
- Problems, Problem Instances, and Algorithms
- Insertion Sort Algorithm, Pseudocode
- Analysis of Algorithms
- **Methods to Design Algorithms**
  - Divide and Conquer
  - Merge sort
- Analysis of Recursive Algorithms

Methods to Design Algorithms

- Incremental approach
  - Build a solution into a larger solution
    - E.g., insertion sort: we have a sorted subarray *A[1:i–1]*, then append an item to obtain a sorted subarray *A[1:i]*
- Divide and conquer approach
  - Reduce the problem into smaller subproblems to solve it
  - E.g., merge sort
- (More methods in subsequent lectures)

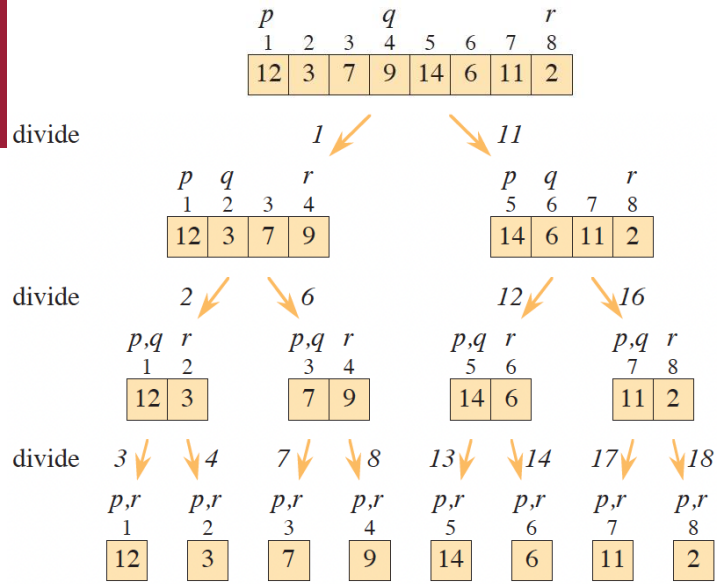
Divide and Conquer

- Divide
  - Break a problem into 1 or more subproblems (smaller instances of the same problem)
- Conquer
  - Solve the subproblems recursively
    - May recurse several levels
- Combine
  - Combine subproblem solutions to form a solution to the original problem

- Many algorithms are recursive in structure

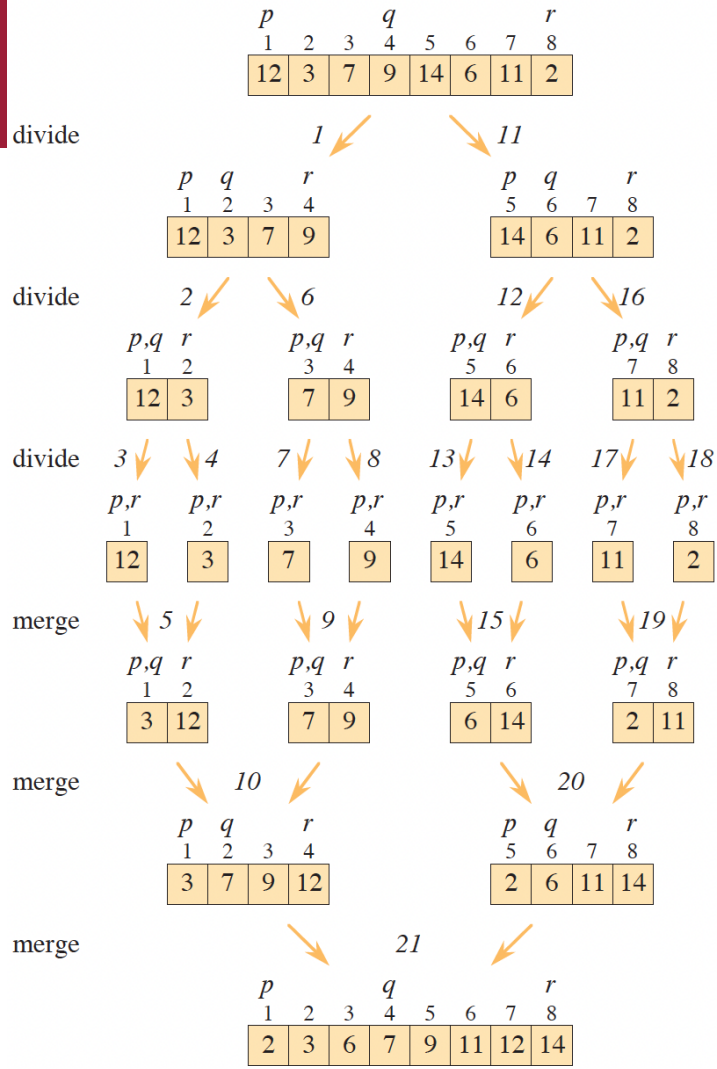
Merge Sort

- Divide
  - (Sub)array $A[p:r]$ to be sorted into two adjacent subarrays, each of half the size, $A[p:q]$ and $A[q:r]$, where q is the midpoint of $A[p:r]$
- Conquer
  - Sort subarrays $A[p:q]$ and $A[q+1:r]$ recursively via merge sort
- Combine
  - The sorted subarrays $A'[p:q]$ and $A'[q+1:r]$ back to get $A'[p:r]$



Merge Sort

- **Base case of merge sort**: a subarray with only 1 element
  - It is always sorted
  - Base case: a simple case that does not recursively call the function itself
- Which operations perform the sorting?
  - The key operation of merge sort occurs in the *combine* step
  - Different algorithms vary



Summary

- A little about Programming Basics
- Problems, Problem Instances, and Algorithms
- Insertion Sort Algorithm, Pseudocode
- Analysis of Algorithms
- Methods to Design Algorithms
  - Divide and Conquer
  - Merge sort