Analysis of Algorithms

- First, there may exist some algorithms for the same problem.
- Then we compare these algorithms: Which one is more efficient?
- We focus on time complexity and space complexity.
- To do so, we need to estimate the growth rate of running time or space usage as a function of the input size n.

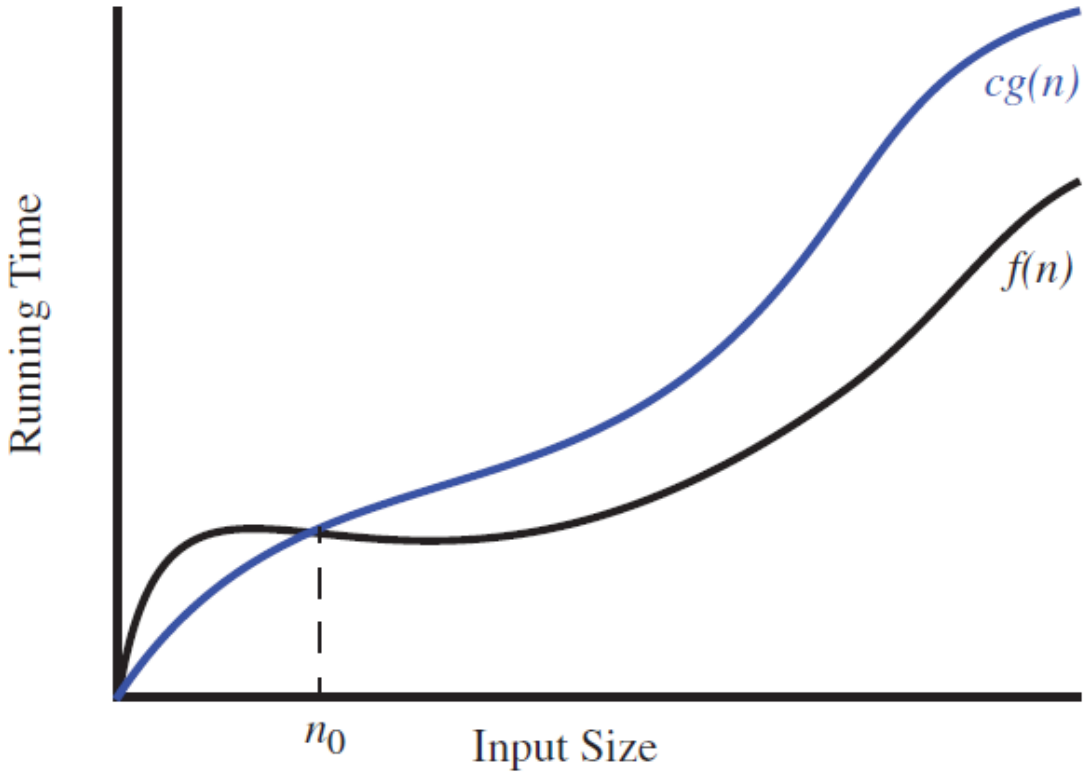
Big-O[1]

- In math, the notation Big-O describes the limiting behavior of a function when the argument approaches infinity, usually in terms of simple functions.
- Now we define $f(n) \in O(g(n))$ as $n \to \infty$ if and only if there is a constant $c > 0$ and a real number n_0 such that

$$|f(n)| \leq c|g(n)| \quad \forall n \geq n_0. \tag{1}$$

- $O(g(n))$ is a set featured by $g(n)$.
- Hence $f(n) \in O(g(n))$ is equivalent to say that $f(n)$ is one instance of $O(g(n))$.

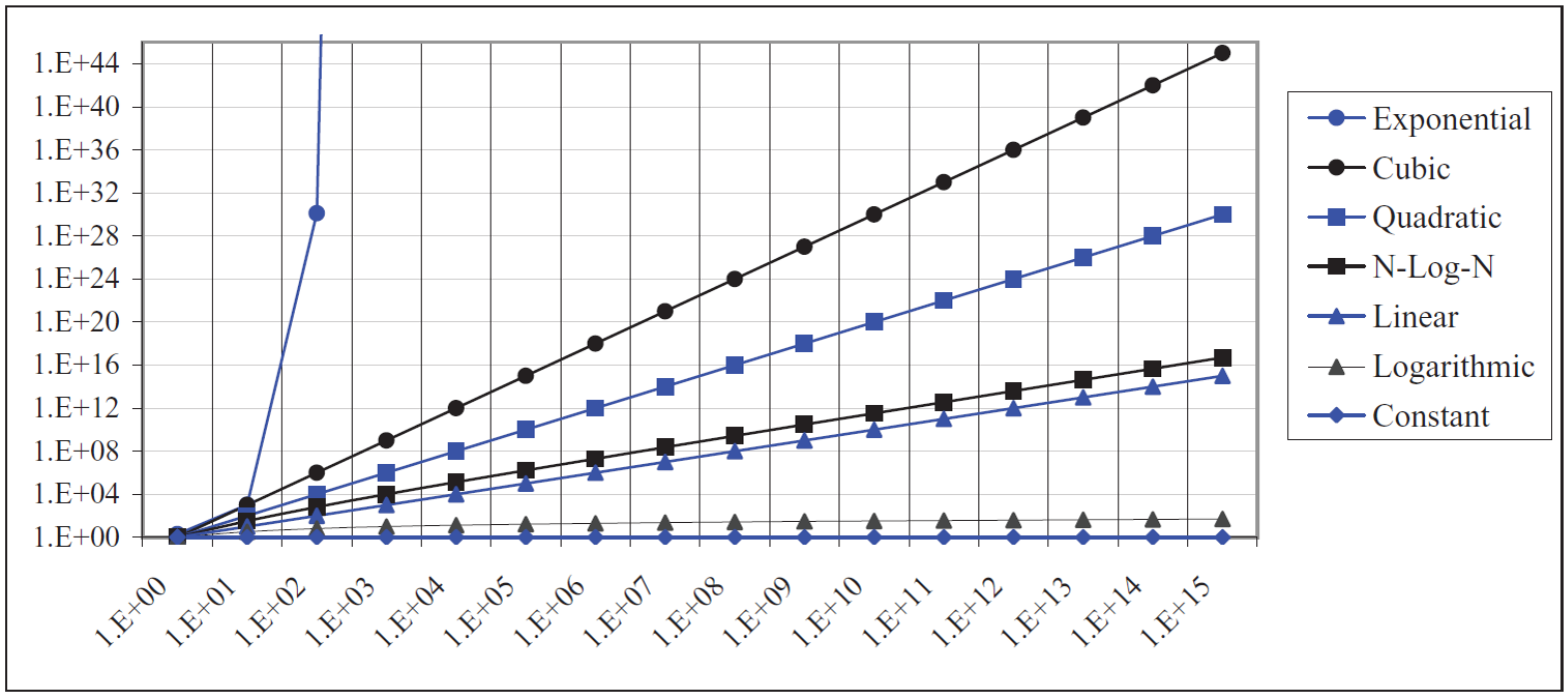
[1] See `https://en.wikipedia.org/wiki/Big_O_notation`.



- This is used for the asymptotic upper bound of complexity of the algorithm.
- In layman's term, Big-O describes the worst case of this algorithm.

- For example, $8n^2 - 3n + 4 \in O(n^2)$.
  - For large n, you could ignore the last two terms.
  - It is easy to find a constant $c > 0$ so that $cn > 8n^2$, say $c = 9$.
- Also, $8n^2 - 3n + 4 \in O(n^3)$ but we seldom say this. (Why?)
- However, $8n^2 - 3n + 4 \notin O(n)$. (Why?)
- What is this analysis related to the program?
- Any insight?

Common Simple Functions[2]



constant	*logarithm*	*linear*	*n-log-n*	*quadratic*	*cubic*	*exponential*
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

[2]See Table 4.1 and Figure 4.2 in Goodrich and etc, p. 161.

Some Interesting Facts

- We often make a trade-off between time and space.
  - Unlike time, we can reuse memory.
  - Users are sensitive to time.
- Playing game well is hard.[3]
- Earn money? Try to solve one of Millennium Prize Problems.[4]

[3] See `https://en.wikipedia.org/wiki/Game_complexity`.

[4] See `https://en.wikipedia.org/wiki/P_versus_NP_problem`.

```
1 class Lecture5 {
2
3                     "Arrays"
4
5 }
```

Arrays

An array stores a large collection of data which is of the same type.

```
1 ...
2         // assume the size variable exists above
3         T[] A = new T[size];
4         // this creates an array of T type, referenced by A
5 ...
```

- **T** can be any data type.
- This statement comprises two parts:
  - Declaring a reference
  - Creating an array

Variable Declaration for Arrays

- In the left-hand side, it is a declaration for an array variable, which does not allocate real space for the array.
- In reality, this variable occupies only a certain space for the reference to an array.[5]
- If a reference variable does not refer to an array, the value of the variable is null.[6]
- In this case, you cannot assign elements to this array variable unless the array object has already been created.

[5] Recall the stack and the heap in the memory layout.

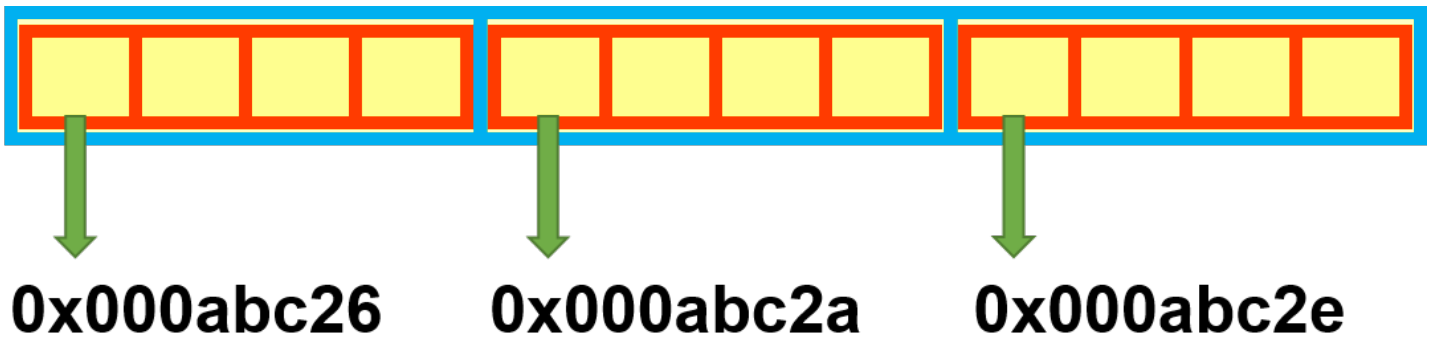
[6] Moreover, this holds for any reference variable. For example, the **Scanner** type.

Creating A Real Array

- All arrays of Java are objects.
- As seen before, the new operator returns the memory address of that object.
  - Recall that the type of reference variables must be compatible to that of the array object.
- The variable *size* must be a positive integer for the number of elements.
- Note that the size of an array cannot be changed after the array is created.[7]

[7] Alternatively, you may try the class **ArrayList**, which is more useful in practice.

Array in Memory




```
1        int[] A = new int[3];
```

- The array is allocated contiguously in the memory.
- All arrays are zero-based indexing.[8] (Why?)
- So we have A[0], A[1], and A[2].

[8] Same in C, C++, python, Javascript, and more.

Array Initializer

The elements of arrays are initialized once created.

- By default, every element is assigned as follows:
  - 0 for all numeric primitive data types
  - \u0000 for char type
  - false for boolean type
- An array can also be initialized by enumerating all the elements without using the new operator.
- For example,

```
1       int[] A = {1, 2, 3};
```

Processing Arrays

When processing array elements, we often use for loops.

- Recall that arrays are objects.
- They have an attribute called **length** which records the size of the arrays.
  - For example, use A.length to get the size of A.
- Since the size of the array is known, it is natural to use a for loop to manipulate with the array.

Many Examples

Initialization of arrays by a Scanner object

```
1 ...
2         // let x be an integer array with a certain size
3         for (int i = 0; i < A.length; ++i) {
4             A[i] = input.nextInt();
5         }
6 ...
```

Initialization of arrays by random numbers

```
1 ...
2         for (int i = 0; i < A.length; ++i) {
3             A[i] = (int) (Math.random() * 10);
4         }
5 ...
```

Display of array elements

```
1 ...
2         for (int i = 0; i < A.length; ++i) {
3             System.out.printf("%3d", A[i]);
4         }
5 ...
```

Sum of array elements

```
1 ...
2         int sum = 0;
3         for (int i = 0; i < A.length; ++i) {
4             sum += A[i];
5         }
6 ...
```

Extreme values in the array

```
1 ...
2         int max = A[0];
3         int min = A[0];
4         for (int i = 1; i < A.length; ++i) {
5             if (max < A[i]) max = A[i];
6             if (min > A[i]) min = A[i];
7         }
8 ...
```

- How about the location of the extreme values?
- Can you find the 2nd max of A?
- Can you keep the first k max of A?

Shuffling over array elements

```
1  ...
2          for (int i = 0; i < A.length; ++i) {
3              // choose j randomly
4              int j = (int) (Math.random() * A.length);
5              // swap
6              int tmp = A[i];
7              A[i] = A[j];
8              A[j] = tmp;
9          }
10 ...
```

- How to swap values of two variables without *tmp*?
- However, this naive algorithm is biased.[9]

[9] See https://blog.codinghorror.com/the-danger-of-naivete/.

Exercise

Write a program which picks first 5 cards at random from a deck of 52 cards.

- 4 suits: Spade, Heart, Diamond, Club.
- 13 ranks: $3, \ldots, 10$, J, Q, K, A, 2.
- Label 52 cards by $0, 1, \cdots, 51$.
- Shuffle the numbers.
- Deal the first 5 cards.

```
...
        String[] suits = {"Spade", "Heart", "Diamond", "Club"};
        String[] ranks = {"3", "4", "5", "6" , "7",
                          "8", "9", "10", "J", "Q", "K",
                          "A", "2"};

        int size = 52;
        int[] deck = new int[size];
        for (int i = 0; i < deck.length; i++)
            deck[i] = i;

        // shuffle over deck; correct version
        for (int i = 0; i < size - 1; i++) {
            int j = (int) (Math.random() * (size - i)) + i;
            int z = deck[i];
            deck[i] = deck[j];
            deck[j] = z;
        }

        for (int i = 0; i < 5; i++) {
            String suit = suits[deck[i] / 13];
            String rank = ranks[deck[i] % 13];
            System.out.printf("%-3s%8s\n", rank, suit);
        }
...
```

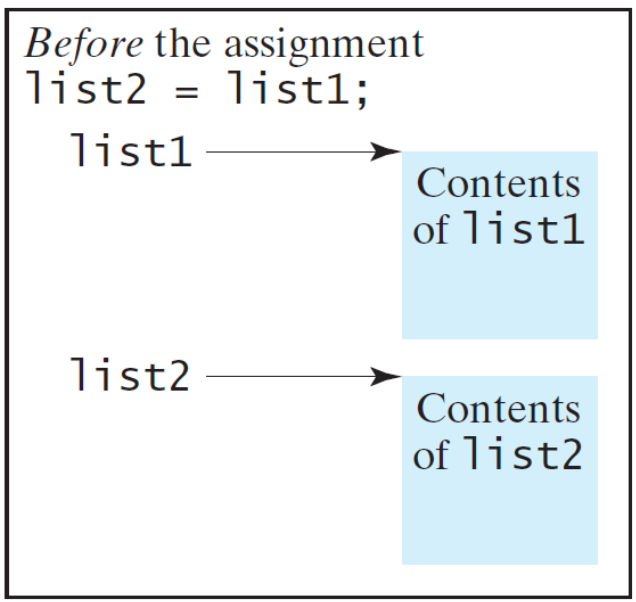
Cloning Arrays

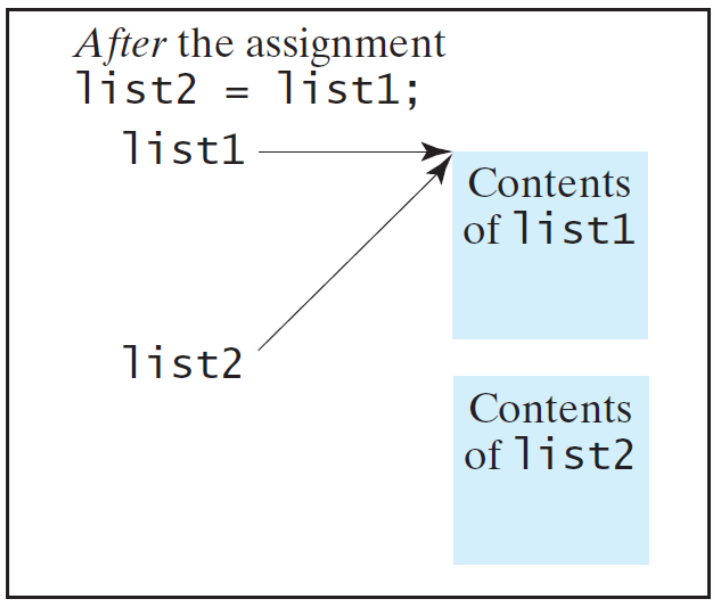
- In practice, one might duplicate an array for some reason.
- One could attempt to use the assignment statement (=), for example,

```
1 ...
2         T[] A = {...}; // assume A is an array
3         T[] B = A; // shallow copy; you don't have a new array
4 ...
```

- However, this is impossible to make two distinct arrays.
- Recall that the array variables are simply references to the arrays in the heap.

- Moreover, all the reference variables share this property!
- For example,





- Use a loop to copy individual elements one by one.

```
...
        // Assume A is an array to be copied.

        // deep copy
        int[] B = new int[A.length];
        for (int i = 0; i < A.length; ++i) {
            B[i] = A[i];
        }
...
```

- Alternatively, you may use the *arraycopy* method in the **System** class.

```
...
        System.arraycopy(A, 0, B, 0, A.length);
...
```

for-each Loops[10]

- A for-each loop is designed to iterate over a collection of objects, such as arrays and other data structures, in strictly sequential fashion, from start to finish.
- For example,

```
...
        T[] A = {...}; // assume some T-type array

        for (T element: A) {
            // body
        }
...
```

- Note that the type **T** should be compatible to the element type of *A*.

[10] Beginning with JDK5.

Example

```
...
        // assume A is an array of integers
        int s = 0;
        for (int i = 0; i < A.length; ++i)
            s += A[i];
...
```

- Not only is the syntax streamlined, but it also prevents boundary errors.

```
...
        int s = 0;
        for (int item: A)
            s += item;
...
```

Short Introduction to Data Structures[11]

- A data structure is a particular way of organizing data in a program so that it can be used efficiently.
- Another basic data structure, called linked list, is an alternative to arrays.
- The choice among data structures depends on applications.
- You will see plenty of data structures in the future:
  - arrays, linked lists;
  - priority queues (queues, stacks, and more);
  - trees, graphs;
  - hash table;
  - and many many.

[11] See `http://bigocheatsheet.com/`.

Common Operations on Data

- The **Arrays** class contains useful methods for common array operations such as sorting and searching.
- For example,

```
1 import java.util.Arrays;
2
3 ...
4         int[] A = {5, 2, 8};
5         Arrays.sort(A); // sort the whole array
6
7         char[] B = {'A', 'r', 't', 'h', 'u', 'r'};
8         Arrays.sort(B, 1, 4); // sort the array partially
9 ...
```

Example: Selection Sort

```
1  ...
2          // selection sort: O(n ^ 2) time
3          for (int i = 0; i < A.length; i++) {
4              int k = i; // the position of min starting from i
5              for (int j = i + 1; j < A.length; j++) {
6                  if (A[k] > A[j])
7                      k = j;
8              }
9              // swap(A[i], A[k])
10             int tmp = A[k];
11             A[k] = A[i];
12             A[i] = tmp;
13         }
14 ...
```

- Time complexity: $O(n^2)$
- You can find more sorting algorithms.[12]

[12] See `http://visualgo.net/`.

Example: Searching Problem

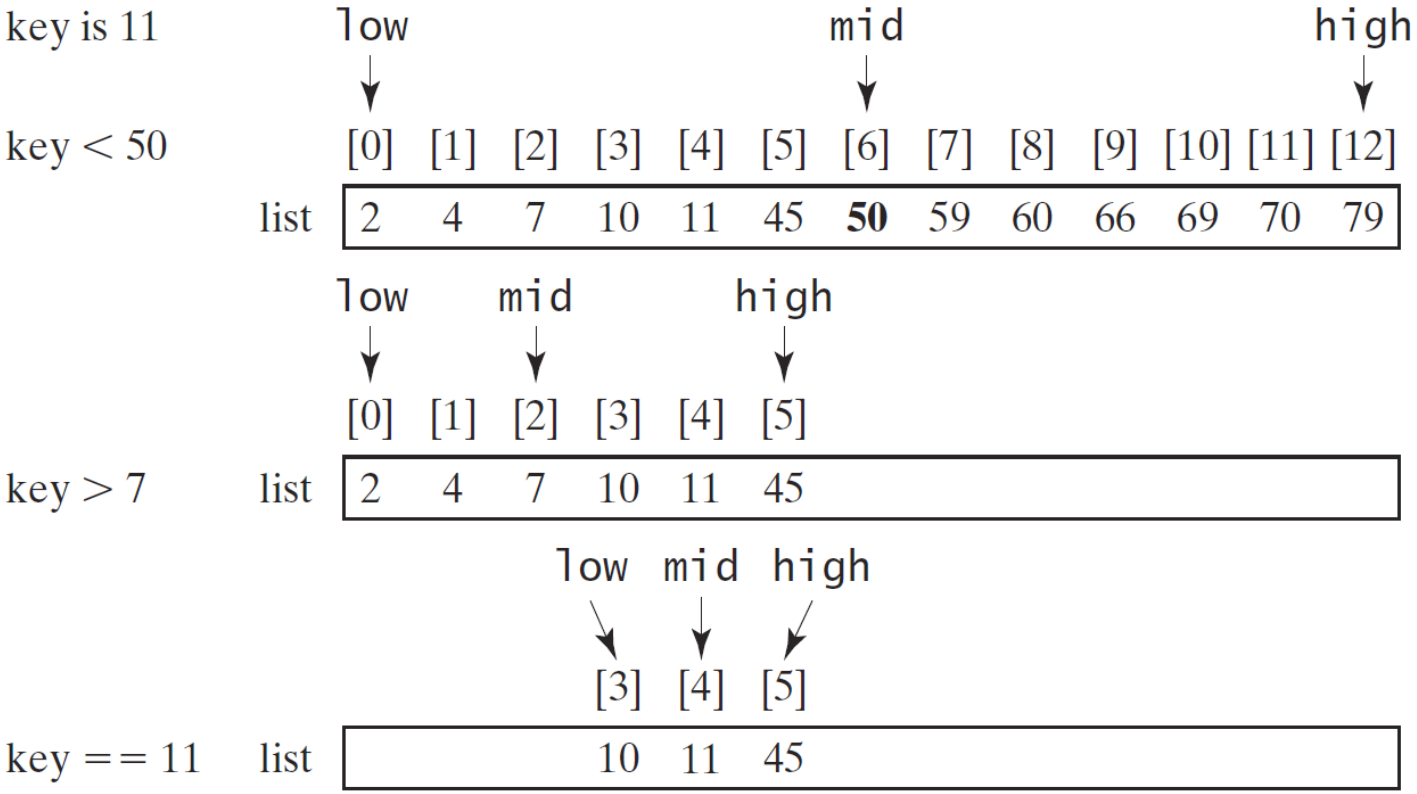
Write a program which searches for the index associated with the key.

- For convenience, assume that there is no duplicate key.
- The linear search approach compares the key with each element in the array sequentially.

```
1 ...
2         // assume A is an array
3         for (int i = 0; i < A.length; i++) {
4             if (A[i] == key) System.out.println(i);
5         }
6 ...
```

- Time complexity: $O(n)$.

Alternative: Binary Search (Revisited)



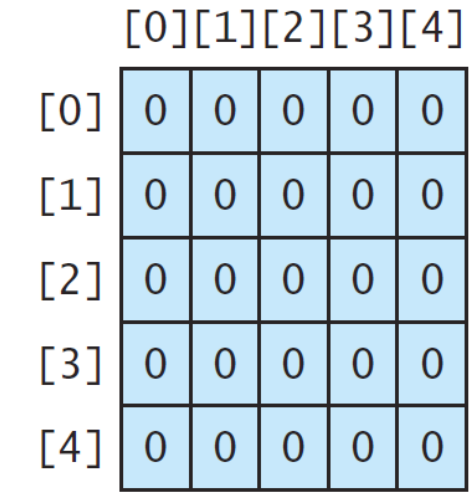
- Time complexity: $O(\log n)$
- Overall time complexity (sorting + searching): still $O(\log n)$?

```
1  ...
2          int index = -1; // why?
3          int high = A.length - 1, low = 0, mid;
4          while (high > low) {
5              mid = (high + low) / 2;
6              if (A[mid] == key) {
7                  index = mid;
8                  break;
9              } else if (A[mid] > key)
10                 high = mid - 1;
11             else
12                 low = mid + 1;
13         }
14
15         if (index > -1)
16             System.out.printf("%d: %d\n", key, index);
17         else
18             System.out.printf("%d: does not exist\n", key);
19 ...
```

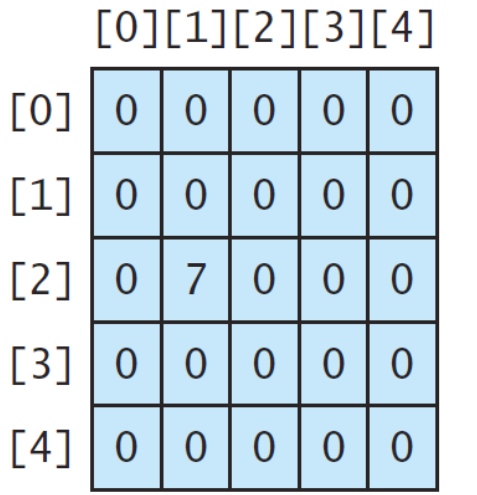
Beyond 1-Dimensional Arrays

- 2D or high-dimensional arrays are widely used.
  - For example, a colorful image is represented by three 2D arrays (R, G, B).
- We can create a 2D **T**-type array with 4 rows and 3 columns as follows:

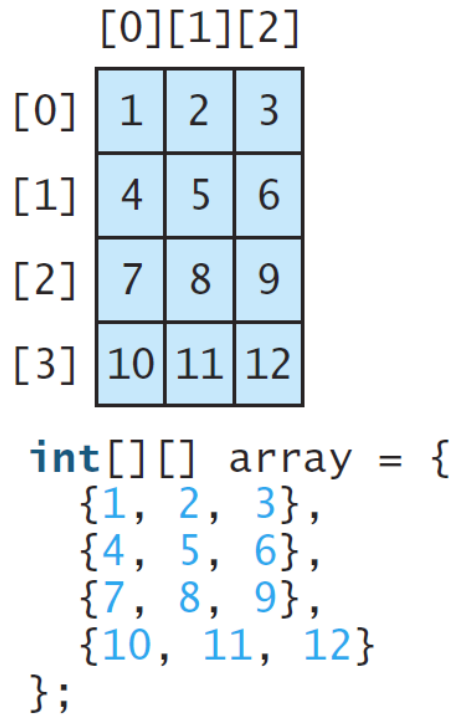
```
1 ...
2         int rowSize = 4; // row size
3         int colSize = 3; // column size
4         T[][] M = new T[rowSize][colSize];
5 ...
```



(a)



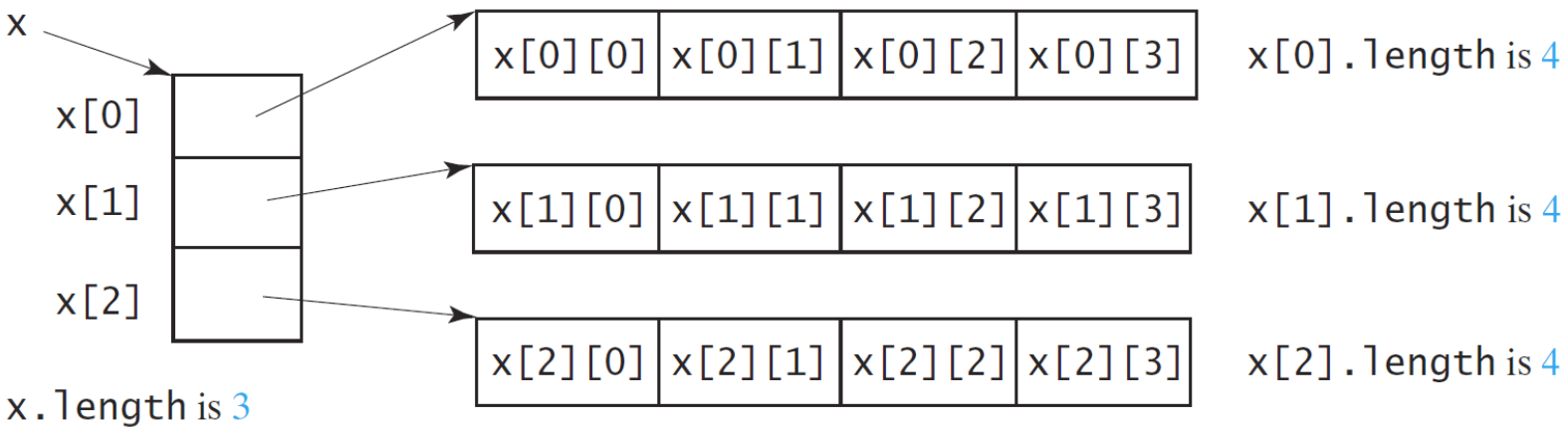
(b)



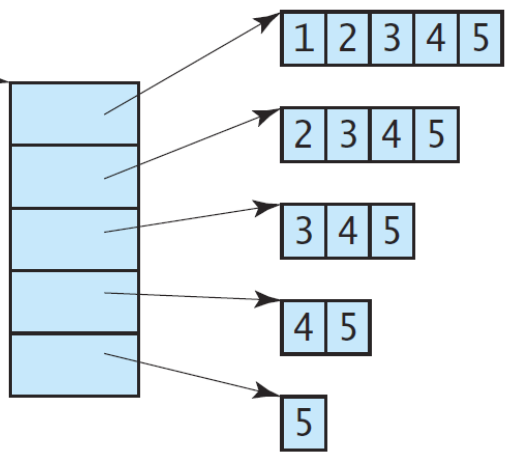
(c)

- Case (c) shows that we can create a 2D array by enumeration.

Reality




```
int[][] triangleArray = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```



Example[13]

```
...
        int[][] A = {{1, 2, 3}, {4, 5}, {6}};

        // conventional for loop
        for (int i = 0; i < A.length; i++) {
            for (int j = 0; j < A[i].length; j++)
                System.out.printf("%2d", A[i][j]);
            System.out.println();
        }

        // for-each loop
        for (int[] row: A) {
            for (int item: row)
                System.out.printf("%2d", item);
            System.out.println();
        }
...
```

[13] Thanks to a lively discussion on January 31, 2016.

Exercise: Matrix Multiplication

Write a program which determines $C = A \times B$ for the input matrices $A_{m \times n}$ and $B_{n \times q}$ for $m, n, q \in \mathbb{N}$.

- You may use the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

  where a_{ik}, $i = 1, 2, \ldots, m$ is a shorthand for A and b_{kj}, $j = 1, 2, \ldots, q$ for B.
- Time complexity: $O(n^3)$ (Why?)

```
1 class Lecture6 {
2
3                     "Methods"
4
5 }
6
7 // keywords:
8 return
```

Methods[15]

- Methods can be used to define reusable code, and organize and simplify code.
- The idea of function originates from math, that is,

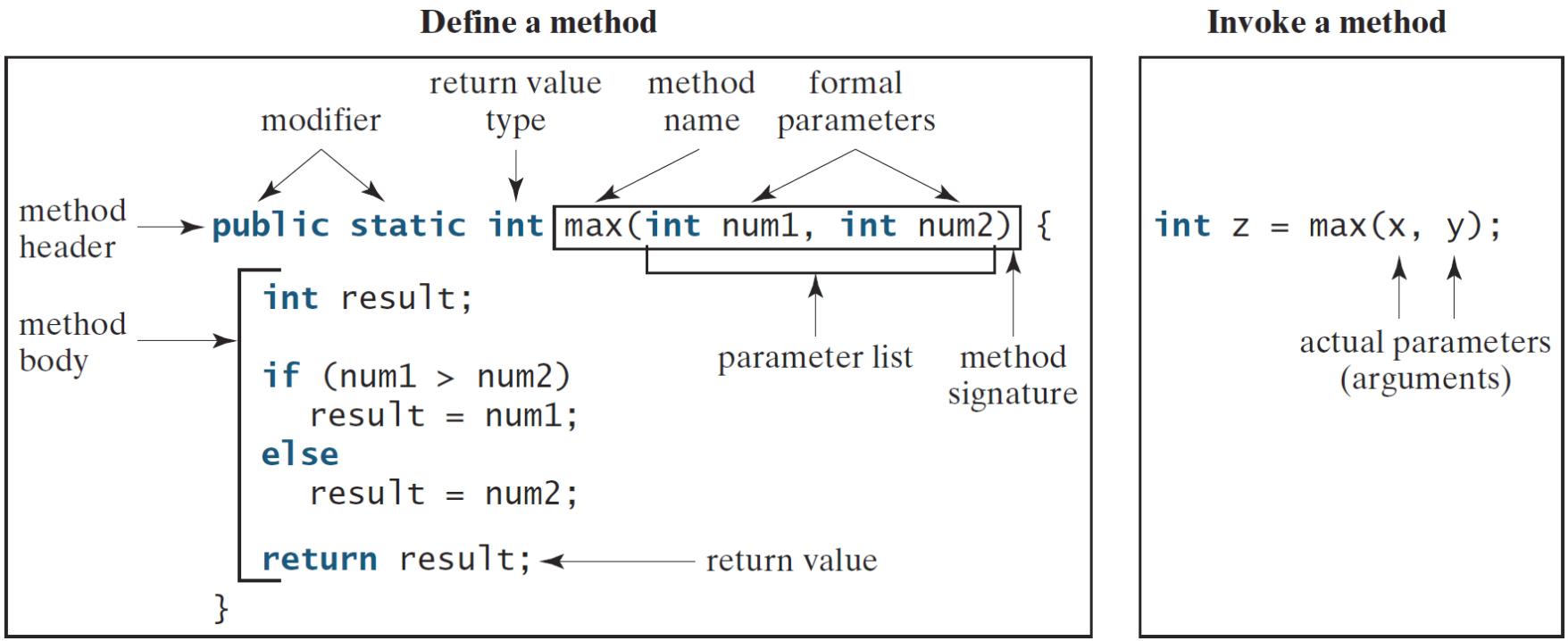
$$y = f(x),$$

  where x is the input parameter[14] and y is the function value.
- In computer science, each input parameter should be declared with a specific type, and a function should be assigned with a return type.

[14] Recall the multivariate functions. The input can be a vector, say the position vector (x, y, z).

[15] Aka procedures and functions.

Example: max


Define a method
Invoke a method
return value type
modifier
method name
formal parameters
method header
method body
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
parameter list
method signature
return value
int z = max(x, y);
actual parameters (arguments)

```
1 ...
2     modifiers returnType methodName(listOfParameters) {
3         // method body
4     }
5 ...
```

- *modifiers* could be static and public (for now).
- *returnType* could be primitive types and reference types.
  - If the method does not return any value, then the return type is declared void.
- *listOfParameters* is used to indicate the method inputs, each separated by commas.
  - Note that a method could have no input.[16]
- The method name and the parameter list together are called the method signature.[17]

[16] For example, **Math.random**().

[17] Method overloading depends signatures. We will see it soon.

More Observations

- There are alternatives to the method **max**():

```
1 ...
2     public static int max(int num1, int num2) {
3         if (num1 > num2) {
4             return num1;
5         } else {
6             return num2;
7         }
8     }
9 ...
```

```
1 ...
2     public static int max(int num1, int num2) {
3         return num1 > num2 ? num1 : num2;
4     }
5 ...
```

"All roads lead to Rome."
– Anonymous

"但如你根本並無招式，敵人如何來破你的招式？"
– 風清揚，笑傲江湖。第十回。傳劍

The return Statement

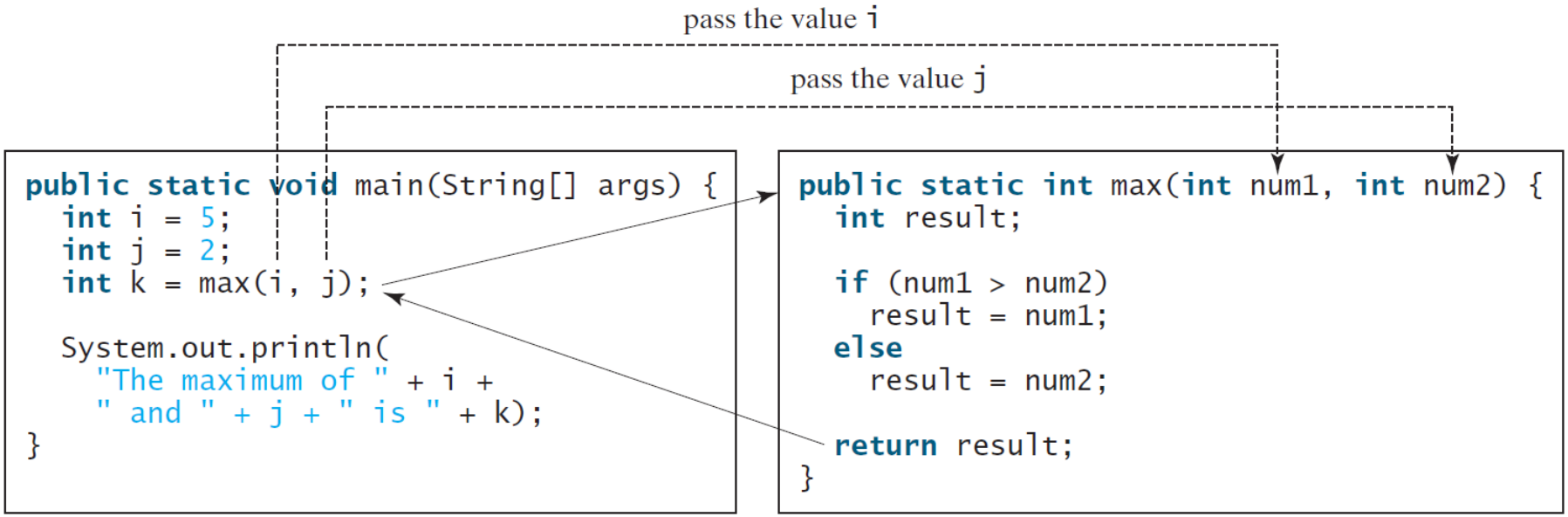
- The return statement is the end point of the method.
- A callee is a method invoked by a caller.
- The callee returns to the caller if the callee
  - completes all the statements (w/o a return statement, say **main**());
  - reaches a return statement;
  - throws an exception (introduced later).
- As you can see, the return statement is not necessarily at the bottom of the method.[18]
- Once one defines the return type (except void), the method should guarantee to return a value or an object of that type.

[18] Thanks to a lively discussion on November 22, 2015.

Bad Exampls

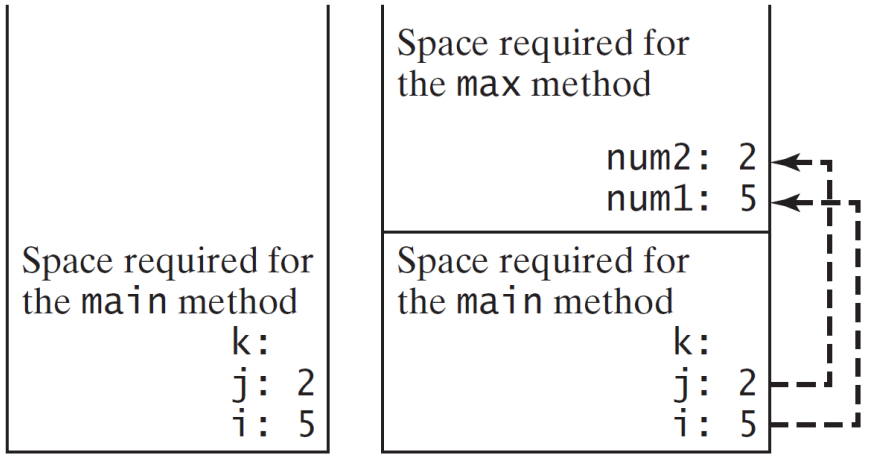
```
1  ...
2      public static int foo1() {
3          while (true);
4          return 0; // unreachable code
5      }
6
7      public static int foo2(int x) {
8          if (x > 0) {
9              return x;
10         }
11         // what if x < 0?
12     }
13 ...
```

Method Invocation



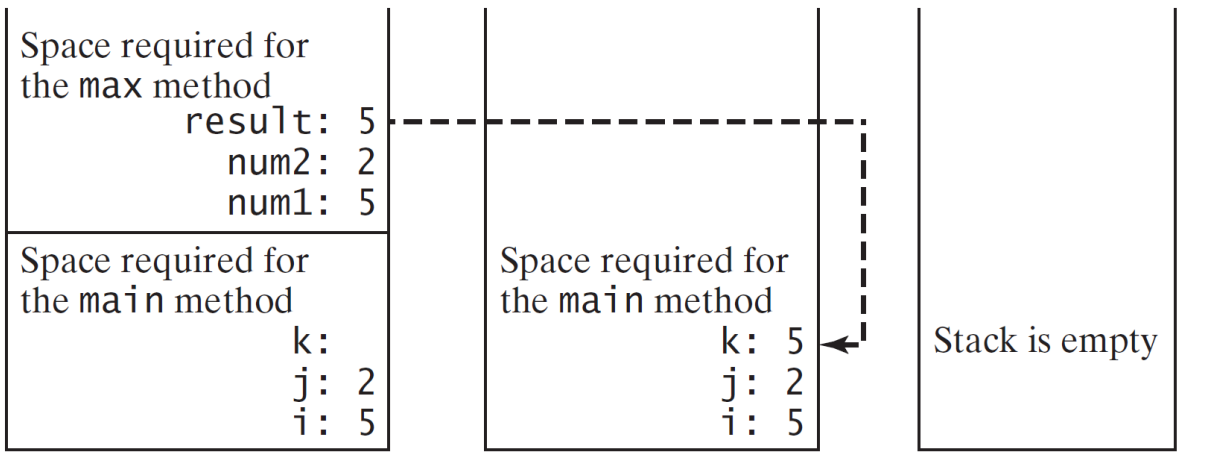
- Note that the input parameters are sort of variables declared within the method as placeholders.
- When calling the method, it's the obligation of callers to provide arguments in order, number, and compatible type, as defined in the method signature.

- In Java, method invocation uses pass-by-value.
- When the callee is invoked, the program control (pc) is transferred from the caller to the callee.
- For each method invocation, JVM pushes a frame which stores necessary information in the call stack.
- The caller resumes its work once the callee finishes its routine.



(a) The `main` method is invoked.

(b) The `max` method is invoked.



(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

Variable Scope

- A variable scope refers to the region where a variable can be referenced.
- A pair of balanced curly braces defines the variable scope.
- In general, variables can be declared in class level, method level, or loop level.
- We cannot duplicate the variables whose names are identical in the same level.

Example

```
1  public class ScopeDemo {
2
3      public static int x = 10; // class level
4
5      public static void main(String[] args) {
6
7          System.out.println(x); // output 10
8
9          int x = 100; // method level, aka local variable
10         x++;
11         System.out.println(x); // output 101
12         addOne();
13         System.out.println(x); // output ?
14     }
15
16     public static void addOne() {
17         x = x + 1;
18         System.out.println(x); // output ?
19     }
20 }
```

A Math Toolbox: **Math** Class

- The **Math** class provides basic mathematical functions and 2 global constants **Math.PI**[19] and **Math.E**[20].
- All methods are public and static.
  - For example, max, min, round, ceil, floor, abs, pow, exp, sqrt, cbrt, log, log10, sin, cos, asin, acos, and random.
- Full document for **Math** class can be found here.
- You are expected to read the document!

[19] The constant π is a mathematical constant, the ratio of a circle's circumference to its diameter, commonly approximated as 3.141593.

[20] The constant e is the base of the natural logarithm. It is approximately equal to 2.71828.

Method Overloading

- Methods with the same name can coexist and be identified by the method signatures.

```
1 ...
2         public static int max(int x, int y) { ... }
3         // different numbers of inputs
4         public static int max(int x, int y, int z) { ... }
5          // different types
6         public static double max(double x, double y) { ... }
7 ...
```