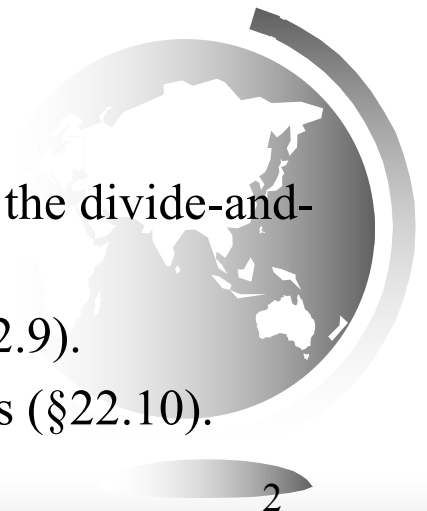


# Chapter 22 Developing Efficient Algorithms



# Objectives

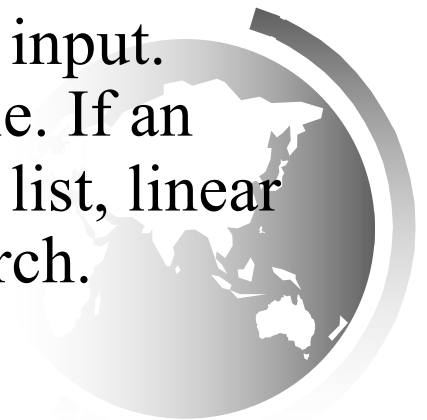
- To estimate algorithm efficiency using the Big notation (§22.2).
- To explain growth rates and why constants and nondominating terms can be ignored in the estimation (§22.2).
- To determine the complexity of various types of algorithms (§22.3).
- To analyze the binary search algorithm (§22.4.1).
- To analyze the selection sort algorithm (§22.4.2).
- To analyze the insertion sort algorithm (§22.4.3).
- To analyze the Tower of Hanoi algorithm (§22.4.4).
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, exponential) (§22.4.5).
- To design efficient algorithms for finding Fibonacci numbers using dynamic programming (§22.5).
- To find the GCD using Euclid's algorithm (§22.6).
- To finding prime numbers using the sieve of Eratosthenes (§22.7).
- To design efficient algorithms for finding the closest pair of points using the divide-and-conquer approach (§22.8).
- To solve the Eight Queens problem using the backtracking approach (§22.9).
- To design efficient algorithms for finding a convex hull for a set of points (§22.10).



# Executing Time

Suppose two algorithms perform the same task such as search (linear search vs. binary search). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.



# Growth Rate

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.



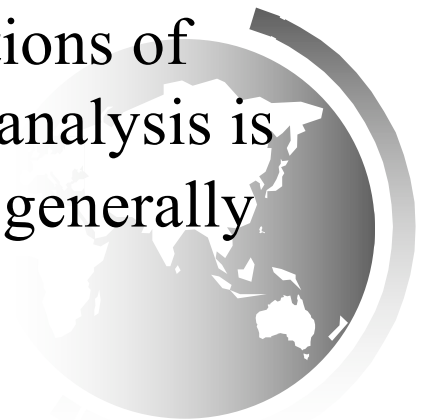
# Big O Notation

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires  $n$  comparisons for an array of size  $n$ . If the key is in the array, it requires  $n/2$  comparisons on average. The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of  $n$ . Computer scientists use the Big  $O$  notation to abbreviate for “order of magnitude.” Using this notation, the complexity of the linear search algorithm is  $O(n)$ , pronounced as “*order of n*.”



# Best, Worst, and Average Cases

For the same input size, an algorithm's execution time may vary, depending on the input. An input that results in the shortest execution time is called the *best-case* input and an input that results in the longest execution time is called the *worst-case* input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.




# Ignoring Multiplicative Constants

The linear search algorithm requires  $n$  comparisons in the worst-case and  $n/2$  comparisons in the average-case. Using the Big  $O$  notation, both cases require  $O(n)$  time. The multiplicative constant ( $1/2$ ) can be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates. The growth rate for  $n/2$  or  $100n$  is the same as  $n$ , i.e.,  $O(n) = O(n/2) = O(100n)$ .


$f(n)$ $n$	$n$	$n/2$	$100n$
100	100	50	10000
200	200	100	20000
	2	2	2

$f(200) / f(100)$



# Ignoring Non-Dominating Terms

Consider the algorithm for finding the maximum number in an array of  $n$  elements. If  $n$  is 2, it takes one comparison to find the maximum number. If  $n$  is 3, it takes two comparisons to find the maximum number. In general, it takes  $n-1$  times of comparisons to find maximum number in a list of  $n$  elements. Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As  $n$  grows larger, the  $n$  part in the expression  $n-1$  dominates the complexity. The Big  $O$  notation allows you to ignore the non-dominating part (e.g.,  $-1$  in the expression  $n-1$ ) and highlight the important part (e.g.,  $n$  in the expression  $n-1$ ). So, the complexity of this algorithm is  $O(n)$ .

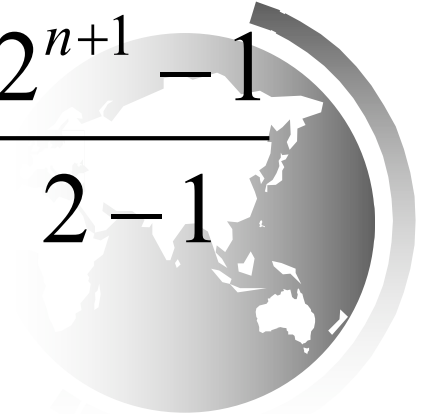




# Useful Mathematic Summations

$$1 + 2 + 3 + \dots + (n - 1) + n = \frac{n(n + 1)}{2}$$

$$a^0 + a^1 + a^2 + a^3 + \dots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1}$$


# Examples: Determining Big-O

- Repetition
- Sequence
- Selection
- Logarithm



# Repetition: Simple Loops

executed  $n$  times

```
{ for (i = 1; i <= n; i++) {  
    k = k + 5;  
}
```

constant time

## Time Complexity

$$T(n) = (\text{a constant } c) * n = cn = \mathbf{O(n)}$$

*Ignore multiplicative constants (e.g., "c").*

PerformanceTest



# Repetition: Nested Loops

executed  $n$  times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= n; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed  $n$  times

constant time

## Time Complexity

$$T(n) = (\text{a constant } c) * n * n = cn^2 = O(n^2)$$

*Ignore multiplicative constants (e.g., “c”).*



# Repetition: Nested Loops

executed  $n$  times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= i; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed  $i$  times

constant time

## Time Complexity

$$T(n) = c + 2c + 3c + 4c + \dots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)$$

*Ignore non-dominating terms*

*Ignore multiplicative constants*

# Repetition: Nested Loops

executed  $n$  times

```
for (i = 1; i <= n; i++) {  
    for (j = 1; j <= 20; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop executed 20 times

constant time

Time Complexity

$$T(n) = 20 * c * n = O(n)$$

*Ignore multiplicative constants (e.g.,  $20*c$ )*

# Sequence

executed  
*10* times

```
{ for (j = 1; j <= 10; j++) {  
    k = k + 4;  
}
```

executed  
*n* times

```
{ for (i = 1; i <= n; i++) {  
    for (j = 1; j <= 20; j++) {  
        k = k + i + j;  
    }  
}
```

inner loop  
executed  
*20* times

## Time Complexity

$$T(n) = c * 10 + 20 * c * n = O(n)$$



# Selection

$O(n)$

```
if (list.contains(e)) {  
    System.out.println(e);  
}  
else  
    for (Object t: list) {  
        System.out.println(t);  
    }
```

} Let  $n$  be  
list.size().  
Executed  
 $n$  times.

## Time Complexity

$$\begin{aligned} T(n) &= \text{test time} + \text{worst-case (if, else)} \\ &= O(n) + O(n) \\ &= O(n) \end{aligned}$$





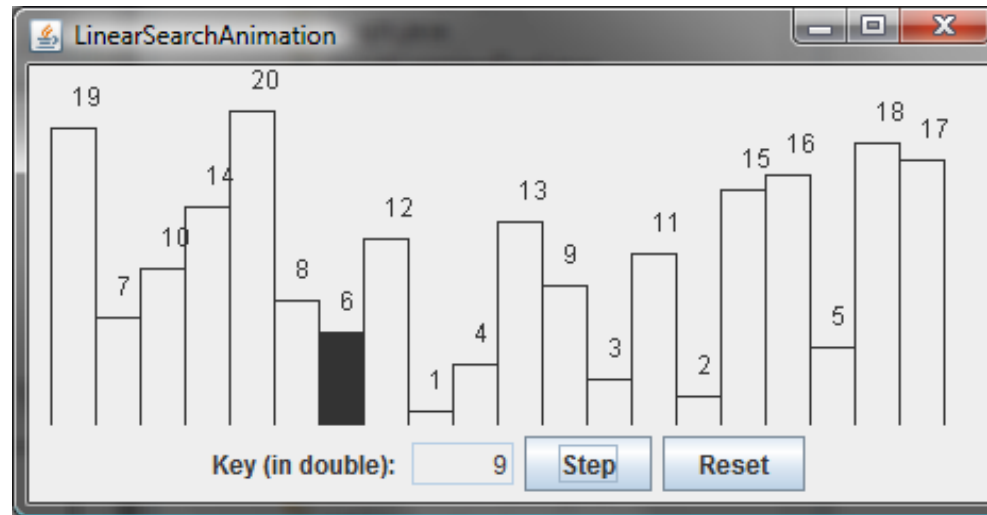
# Constant Time

The Big  $O$  notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take *constant time* with the notation  $O(1)$ . For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.



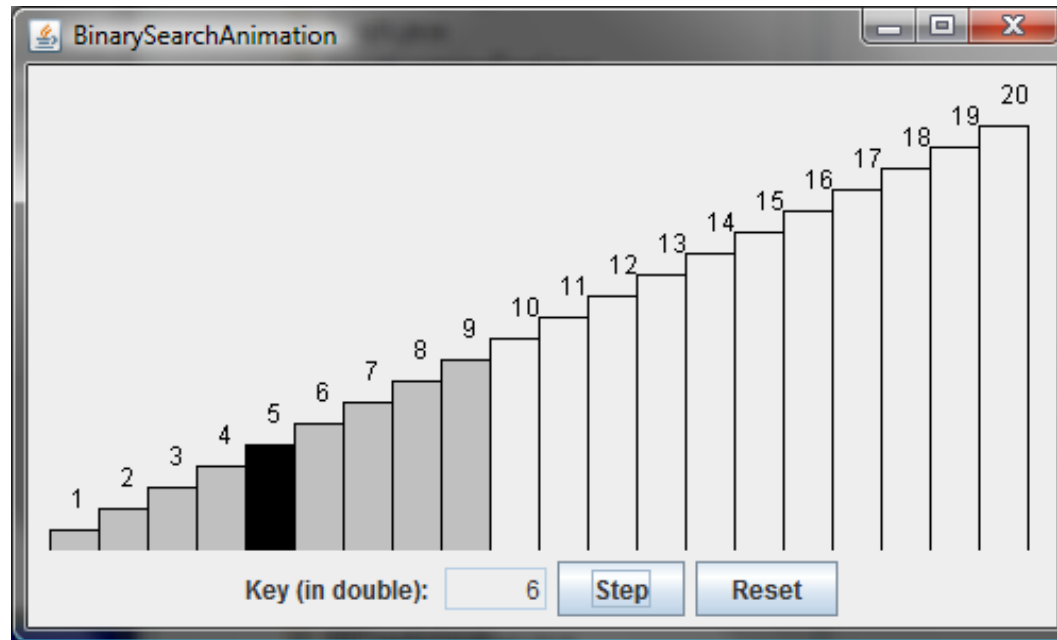
# Linear Search Animation

<https://liveexample.pearsoncmg.com/dsanimation/LinearSearchBook.html>



# Binary Search Animation

<https://liveexample.pearsoncmg.com/dsanimation/BinarySearchBook.html>



# Logarithm: Analyzing Binary Search

$$n = 2^k$$

The binary search algorithm presented in Listing 7.7, `BinarySearch.java`, searches a key in a sorted array. Each iteration in the algorithm contains a fixed number of operations, denoted by  $c$ . Let  $T(n)$  denote the time complexity for a binary search on a list of  $n$  elements. Without loss of generality, assume  $n$  is a power of 2 and  $k = \log n$ . Since binary search eliminates half of the input after two comparisons,

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = \dots = T\left(\frac{n}{2^k}\right) + ck = T(1) + c \log n = 1 + c \log n \\ &= O(\log n) \end{aligned}$$



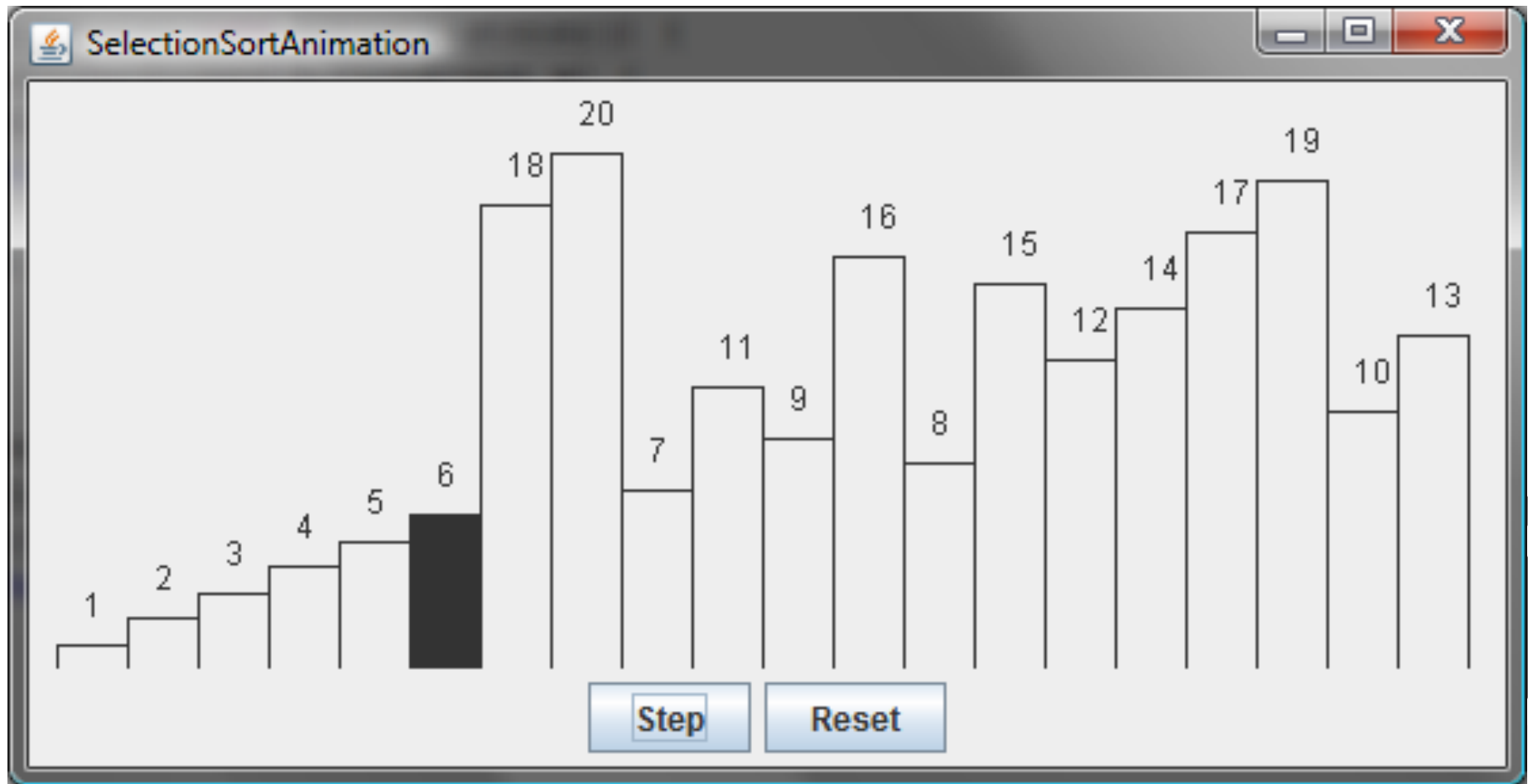
# Logarithmic Time

Ignoring constants and smaller terms, the complexity of the binary search algorithm is  $O(\log n)$ . An algorithm with the  $O(\log n)$  time complexity is called a *logarithmic algorithm*. The base of the log is 2, but the base does not affect a logarithmic growth rate, so it can be omitted. The logarithmic algorithm grows slowly as the problem size increases. If you square the input size, you only double the time for the algorithm.



# Selection Sort Animation

<https://liveexample.pearsoncmg.com/dsanimation/SelectionSortNew.html>



# Analyzing Selection Sort

The selection sort algorithm presented in Listing 7.8, `SelectionSort.java`, finds the smallest number in the list and places it first. It then finds the smallest number remaining and places it second, and so on until the list contains only a single number. The number of comparisons is  $n-1$  for the first iteration,  $n-2$  for the second iteration, and so on. Let  $T(n)$  denote the complexity for selection sort and  $c$  denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = (n-1) + c + (n-2) + c \dots + 2 + c + 1 + c = \frac{n^2}{2} - \frac{n}{2} + cn$$

Ignoring constants and smaller terms, the complexity of the selection sort algorithm is  $O(n^2)$ .



# Quadratic Time

An algorithm with the  $O(n^2)$  time complexity is called a *quadratic algorithm*. The quadratic algorithm grows quickly as the problem size increases. If you double the input size, the time for the algorithm is quadrupled. Algorithms with a nested loop are often quadratic.





# Analyzing Tower of Hanoi

The Tower of Hanoi problem presented in Listing 18.7, TowerOfHanoi.java, moves  $n$  disks from tower A to tower B with the assistance of tower C recursively as follows:

- Move the first  $n - 1$  disks from A to C with the assistance of tower B.
- Move disk  $n$  from A to B.
- Move  $n - 1$  disks from C to B with the assistance of tower A.

Let  $T(n)$  denote the complexity for the algorithm that moves disks and  $c$  denote the constant time to move one disk, i.e.,  $T(1)$  is  $c$ . So,

$$\begin{aligned} T(n) &= T(n-1) + c + T(n-1) = 2T(n-1) + c \\ &= 2(2(T(n-2) + c) + c) = 2^{n-1}T(1) + c2^{n-2} + \dots + c2 + c = \\ &= c2^{n-1} + c2^{n-2} + \dots + c2 + c = c(2^n - 1) = O(2^n) \end{aligned}$$

# Common Recurrence Relations

Recurrence Relation	Result	Example
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 24)
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort, insertion sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Towers of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm



# Comparing Common Growth Functions

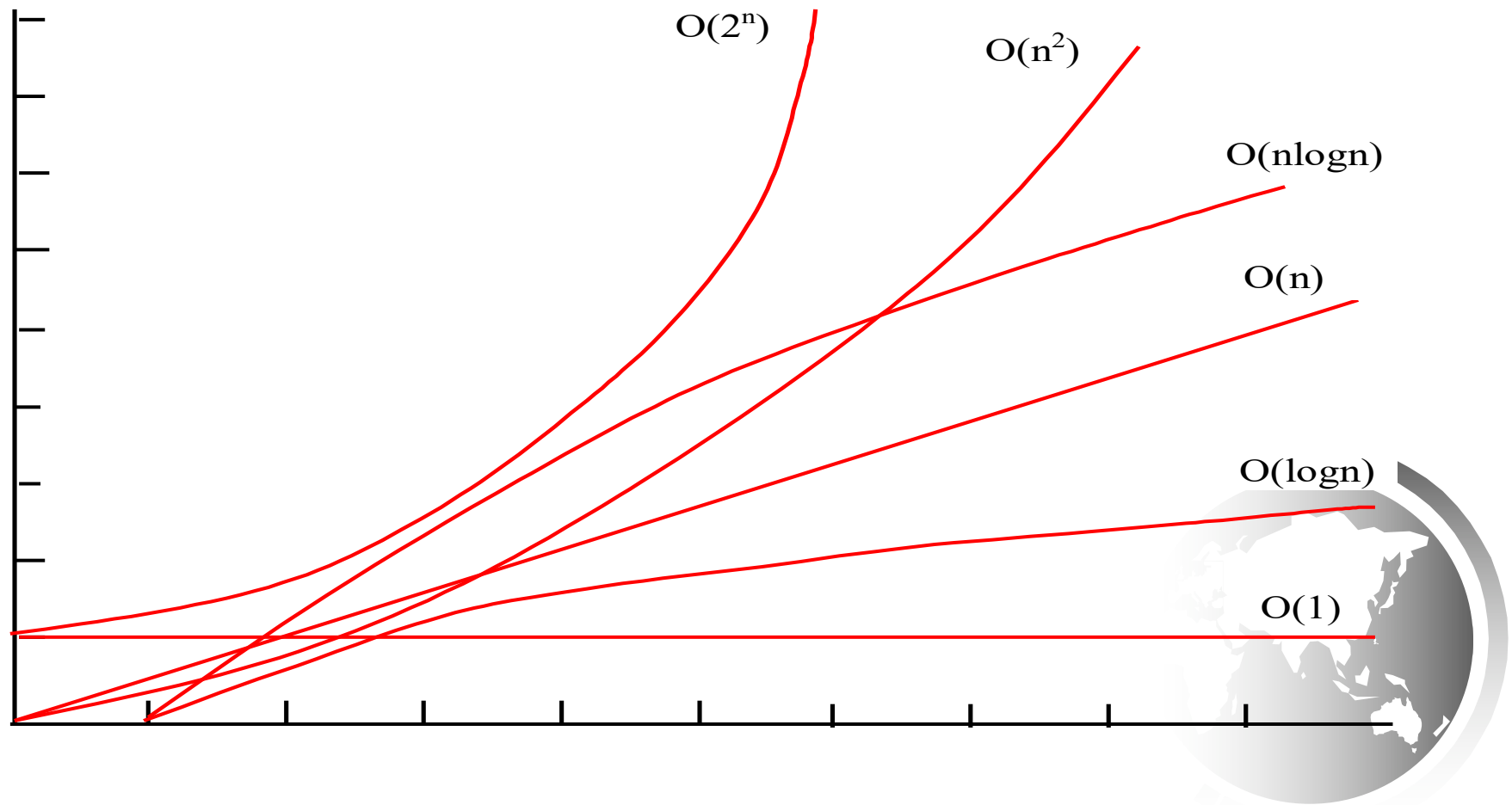
$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time



# Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$



# Case Study: Fibonacci Numbers

```
/** The method for finding the Fibonacci number */  
public static long fib(long index) {  
    if (index == 0) // Base case  
        return 0;  
    else if (index == 1) // Base case  
        return 1;  
    else // Reduction and recursive calls  
        return fib(index - 1) + fib(index - 2);  
}
```

Fibonacci series: 0 1 1 2 3 5 8 13 21 34 55 89...

indices: 0 1 2 3 4 5 6 7 8 9 10 11

$\text{fib}(0) = 0;$

$\text{fib}(1) = 1;$

$\text{fib}(\text{index}) = \text{fib}(\text{index} - 1) + \text{fib}(\text{index} - 2); \text{index} \geq 2$



# Complexity for Recursive Fibonacci Numbers

Since

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c & \text{and} & & T(n) &= T(n-1) + T(n-2) + c \\ &\leq 2T(n-1) + c & & & &= T(n-2) + T(n-3) + c + T(n-2) + c \\ &\leq 2(2T(n-2) + c) + c & & & &\geq 2T(n-2) + 2c \\ &= 2^2 T(n-2) + 2c + c & & & &\geq 2(2T(n-4) + 2c) + 2c \\ &\dots & & & &\geq 2^2 T(n-2-2) + 2^2 c + 2c \\ &\leq 2^{n-1} T(1) + 2^{n-2} c + \dots + 2c + c & & & &\geq 2^3 T(n-2-2-2) + 2^3 c + 2^2 c + 2c \\ &= 2^{n-1} T(1) + (2^{n-2} + \dots + 2 + 1)c & & & &\geq 2^{n/2} T(1) + 2^{n/2} c + \dots + 2^3 c + 2^2 c + 2c \\ &= 2^{n-1} T(1) + (2^{n-1} - 1)c & & & &= 2^{n/2} c + 2^{n/2} c + \dots + 2^3 c + 2^2 c + 2c \\ &= 2^{n-1} c + (2^{n-2} + \dots + 2 + 1)c & & & &= O(2^n) \\ &= O(2^n) \end{aligned}$$

Therefore, the recursive Fibonacci method takes  $O(2^n)$

ComputeFibonacci

# Case Study: Non-recursive version of Fibonacci Numbers

```
public static long fib(long n) {  
    long f0 = 0; // For fib(0)  
    long f1 = 1; // For fib(1)  
    long f2 = 1; // For fib(2)  
  
    if (n == 0)  
        return f0;  
    else if (n == 1)  
        return f1;  
    else if (n == 2)  
        return f2;  
  
    for (int i = 3; i <= n; i++) {  
        f0 = f1;  
        f1 = f2;  
        f2 = f0 + f1;  
    }  
  
    return f2;  
}
```

Obviously, the complexity of this new algorithm is  $O(n)$ . This is a tremendous improvement over the recursive algorithm.

ImprovedFibonacci

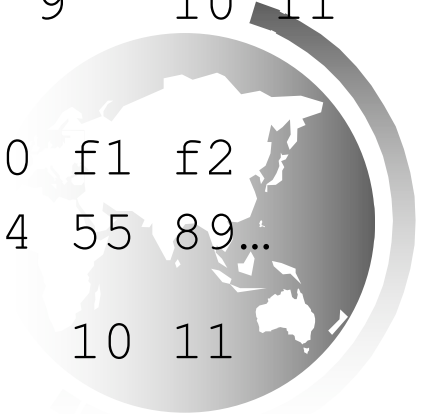


	f0	f1	f2									
Fibonacci series:	0	1	1	2	3	5	8	13	21	34	55	89...
indices:	0	1	2	3	4	5	6	7	8	9	10	11

	f0	f1	f2									
Fibonacci series:	0	1	1	2	3	5	8	13	21	34	55	89...
indices:	0	1	2	3	4	5	6	7	8	9	10	11

	f0	f1	f2									
Fibonacci series:	0	1	1	2	3	5	8	13	21	34	55	89...
indices:	0	1	2	3	4	5	6	7	8	9	10	11

	f0	f1	f2									
Fibonacci series:	0	1	1	2	3	5	8	13	21	34	55	89...
indices:	0	1	2	3	4	5	6	7	8	9	10	11





# Dynamic Programming

The algorithm for computing Fibonacci numbers presented here uses an approach known as *dynamic programming*.

Dynamic programming is to solve subproblems, then combine the solutions of subproblems to obtain an overall solution. This naturally leads to a recursive solution.

However, it would be inefficient to use recursion, because the subproblems overlap. The key idea behind dynamic programming is to solve each subprogram only once and storing the results for subproblems for later use to avoid redundant computing of the subproblems.

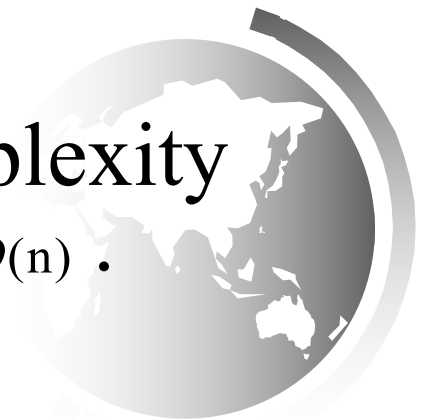


# Case Study: GCD Algorithms

## Version 1

```
public static int gcd(int m, int n) {  
    int gcd = 1;  
    for (int k = 2; k <= m && k <= n; k++) {  
        if (m % k == 0 && n % k == 0)  
            gcd = k;  
    }  
    return gcd;  
}
```

Obviously, the complexity of this algorithm is  $O(n)$ .



# Case Study: GCD Algorithms

## Version 2

```
for (int k = n; k >= 1; k--) {  
    if (m % k == 0 && n % k == 0) {  
        gcd = k;  
        break;  
    }  
}
```

The worst-case time complexity  
of this algorithm is still  $O(n)$ .



# Case Study: GCD Algorithms

## Version 3

```
public static int gcd(int m, int n) {  
    int gcd = 1;  
  
    if (m == n) return m;  
  
    for (int k = n / 2; k >= 1; k--) {  
        if (m % k == 0 && n % k == 0) {  
            gcd = k;  
            break;  
        }  
    }  
  
    return gcd;  
}
```

The worst-case time complexity  
of this algorithm is still  $O(n)$ .



# Euclid's algorithm

Let gcd(m, n) denote the gcd for integers m and n:

♦ If m % n is 0, gcd(m, n) is n.

♦ Otherwise, gcd(m, n) is gcd(n, m % n).

$$m = n * k + r$$

if p is divisible by both m and n, it must be divisible by r


$$m / p = n * k / p + r / p$$



# Euclid's Algorithm Implementation

```
public static int gcd(int m, int n) {  
    if (m % n == 0)  
        return n;  
    else  
        return gcd(n, m % n);  
}
```

The time complexity of this algorithm is  $O(\log n)$ .  
See the text for the proof.



# Finding Prime Numbers

Compare three versions:

- Brute-force `PrimeNumber`  
`PrimeNumbers`
- Check possible divisors up to `Math.sqrt(n)`
- Check possible prime divisors up to `Math.sqrt(n)` `EfficientPrimeNumbers`  
`SieveOfEratosthenes`



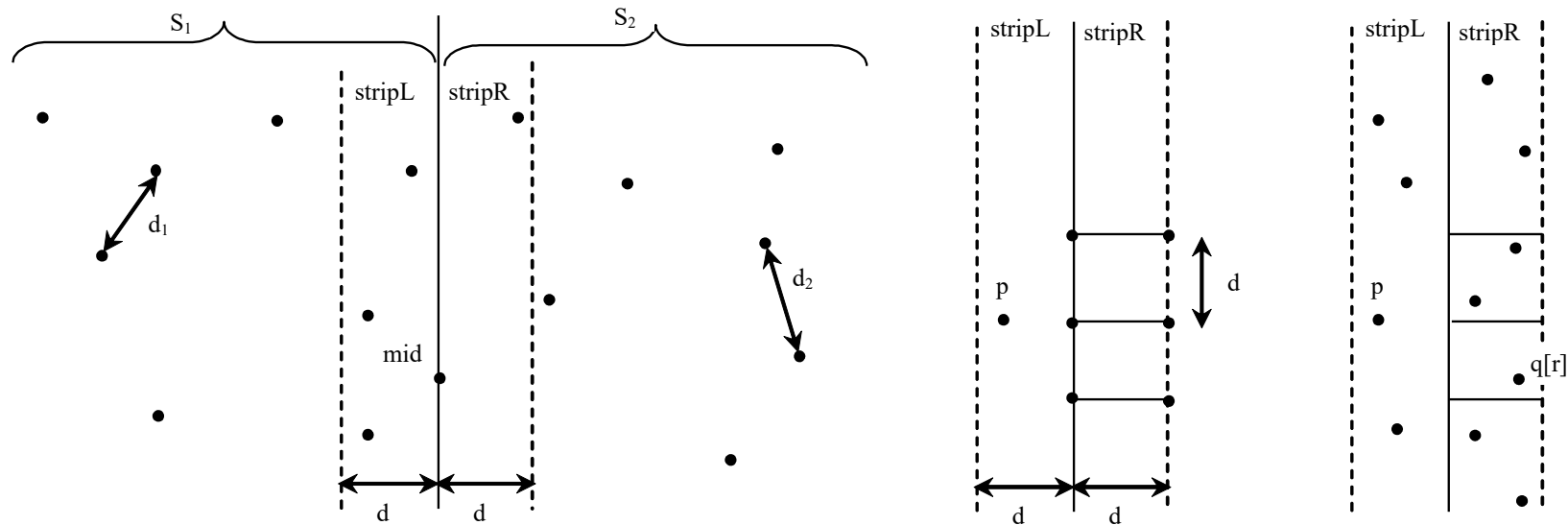
# Divide-and-Conquer

The *divide-and-conquer* approach divides the problem into subproblems, solves the subproblems, then combines the solutions of subproblems to obtain the solution for the entire problem. Unlike the dynamic programming approach, the subproblems in the divide-and-conquer approach don't overlap. A subproblem is like the original problem with a smaller size, so you can apply recursion to solve the problem. In fact, all the recursive problems follow the divide-and-conquer approach.





# Case Study: Closest Pair of Points

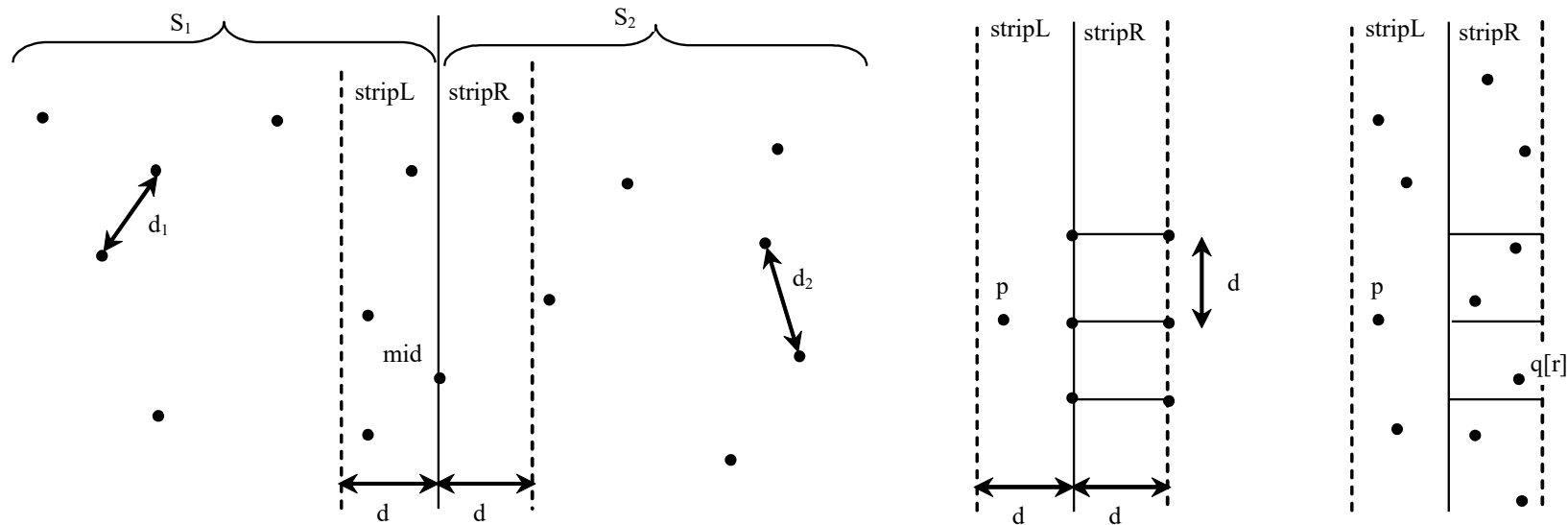


$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$



<http://www.cs.armstrong.edu/liang/animation/web/ClosestPair.html>

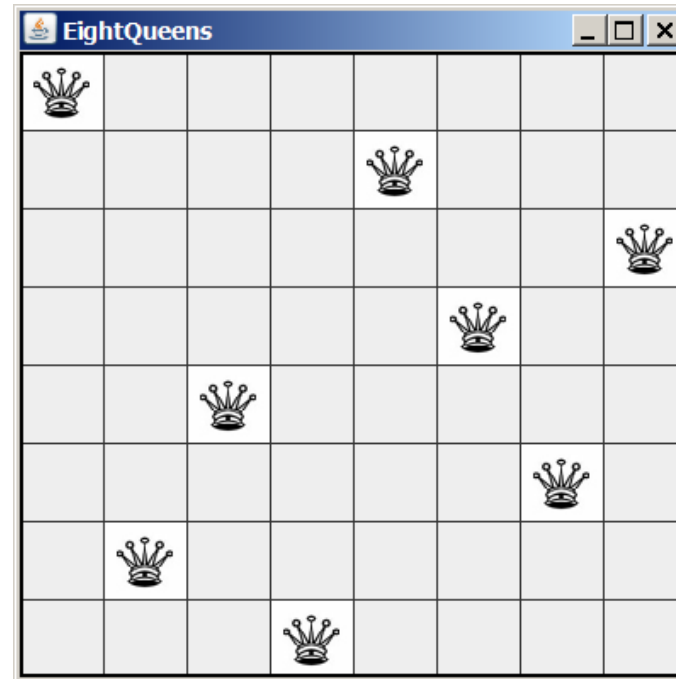
# Case Study: Closest Pair of Points



$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

# Eight Queens

queens[0]	0
queens[1]	4
queens[2]	7
queens[3]	5
queens[4]	2
queens[5]	6
queens[6]	1
queens[7]	3



 <https://liveexample.pearsoncmg.com/html/dsanimation/EightQueenseBook.html>

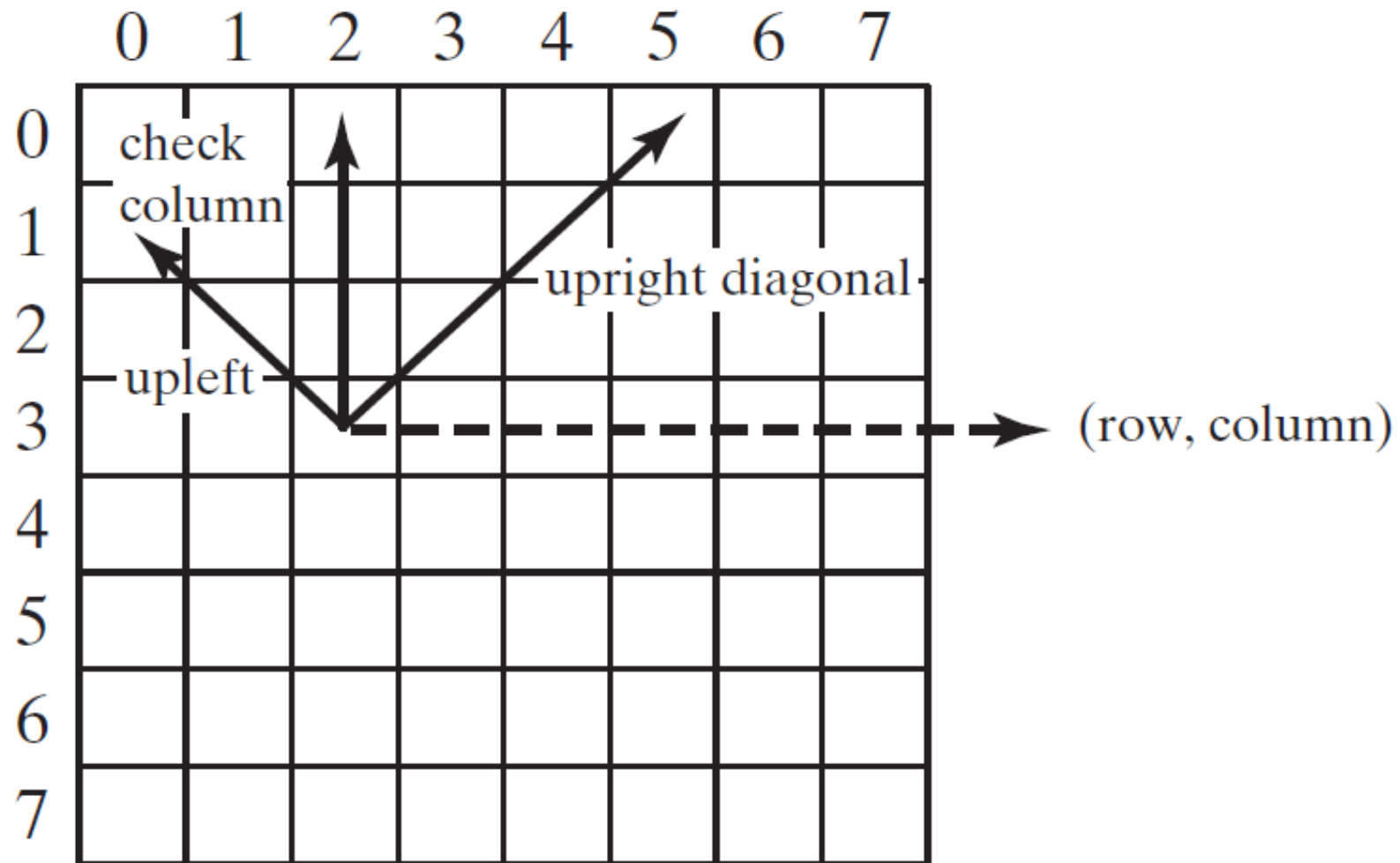
EightQueens

# Backtracking

There are many possible candidates? How do you find a solution? The backtracking approach is to search for a candidate incrementally and abandons it as soon as it determines that the candidate cannot possibly be a valid solution, and explores a new candidate.



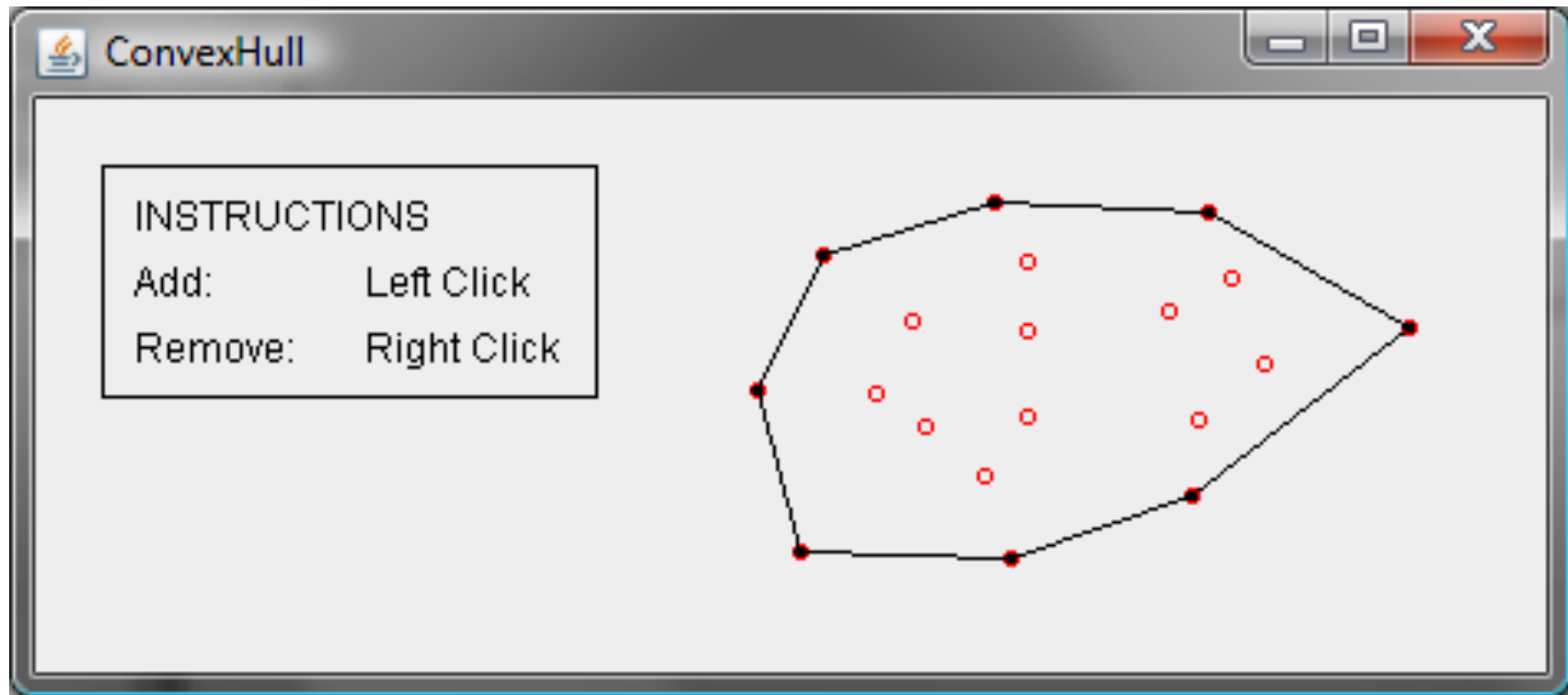
# Eight Queens



# Convex Hull Animation



<https://liveexample.pearsoncmg.com/dsanimation/ConvexHull.html>



# Convex Hull

Given a set of points, a convex hull is a smallest convex polygon that encloses all these points, as shown in Figure a. A polygon is convex if every line connecting two vertices is inside the polygon. For example, the vertices  $v_0$ ,  $v_1$ ,  $v_2$ ,  $v_3$ ,  $v_4$ , and  $v_5$  in Figure a form a convex polygon, but not in Figure b, because the line that connects  $v_3$  and  $v_1$  is not inside the polygon.

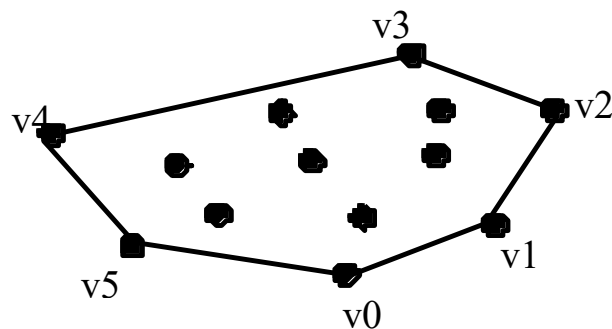


Figure a

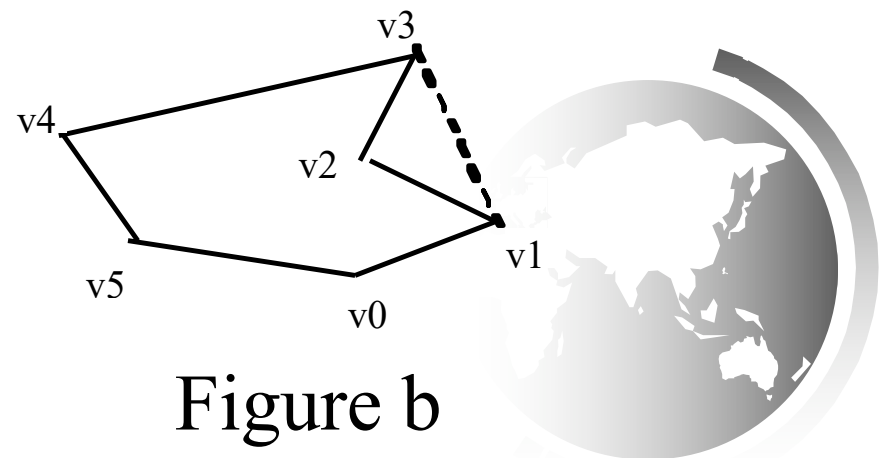
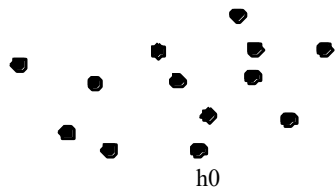
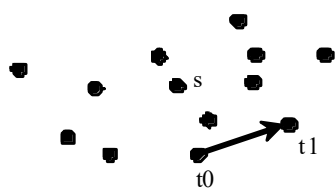


Figure b

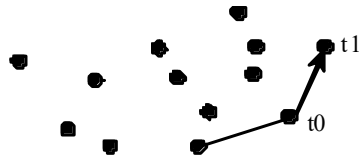
# Gift-Wrapping



Step 1: Given a set of points  $S$ , let the points in  $S$  be labeled  $s_0, s_1, \dots, s_k$ . Select the rightmost lowest point  $h_0$  in the set  $S$ . Let  $t_0$  be  $h_0$ .

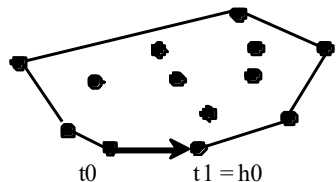


(Step 2: Find the rightmost point  $t_1$ ): Let  $t_1$  be  $s_0$ . For every point  $p$  in  $S$ , if  $p$  is on the right side of the direct line from  $t_0$  to  $t_1$ , then let  $t_1$  be  $p$ .



Step 3: If  $t_1$  is  $h_0$ , done.

Step 4: Let  $t_0$  be  $t_1$ , go to Step 2.



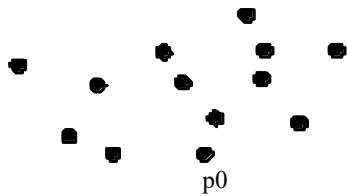


# Gift-Wrapping Algorithm Time

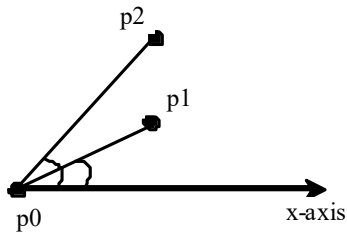
Finding the rightmost lowest point in Step 1 can be done in  $O(n)$  time. Whether a point is on the left side of a line, right side, or on the line can be decided in  $O(1)$  time (see Exercise 3.32). Thus, it takes  $O(n)$  time to find a new point  $t_1$  in Step 2. Step 2 is repeated  $h$  times, where  $h$  is the size of the convex hull. Therefore, the algorithm takes  $O(hn)$  time. In the worst case,  $h$  is  $n$ .



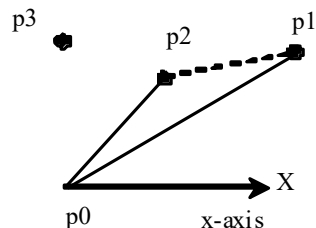
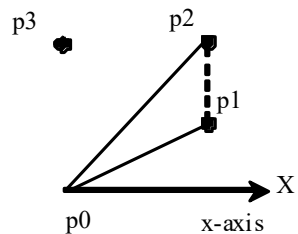
# Graham's Algorithm



Given a set of points  $S$ , select the rightmost lowest point and name it  $p_0$  in the set  $S$ . As shown in Figure 22.10a,  $p_0$  is such a point.



Sort the points in  $S$  angularly along the x-axis with  $p_0$  as the center. If there is a tie and two points have the same angle, discard the one that is closest to  $p_0$ . The points in  $S$  are now sorted as  $p_0, p_1, p_2, \dots, p_{n-1}$ .



The convex hull is discovered incrementally. Initially,  $p_0, p_1$ , and  $p_2$  form a convex hull. Consider  $p_3$ .  $p_3$  is outside of the current convex hull since points are sorted in increasing order of their angles. If  $p_3$  is strictly on the left side of the line from  $p_1$  to  $p_2$ , push  $p_3$  into  $H$ . Now  $p_0, p_1, p_2$ , and  $p_3$  form a convex hull. If  $p_3$  is on the right side of the line from  $p_1$  to  $p_2$  (see Figure 22.10d), pop  $p_2$  out of  $H$  and push  $p_3$  into  $H$ . Now  $p_0, p_1$ , and  $p_3$  form a convex hull and  $p_2$  is inside of this convex hull.

# Graham's Algorithm Time

$O(n \log n)$



# Practical Considerations

The big O notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient. As shown in the preceding example, both algorithms in Listings 5.6 and 22.2 have the same complexity, but the one in Listing 22.2 is obviously better practically.

