Developing Efficient Algorithms

CPT204 Advanced Object-Oriented Programming

Lecture 8 Developing Efficient Algorithms

Objectives

- To estimate ***algorithm efficiency*** using the Big O notation
- To explain ***growth rates*** and why constants and non-dominating terms can be ignored
- To determine the ***complexity*** of various types of algorithms
- To analyze the ***binary search*** algorithm
- To analyze the ***selection sort*** algorithm
- Polynomial complexity
- To analyze the ***insertion sort*** algorithm
- To analyze the ***Tower of Hanoi*** algorithm
- To describe common growth functions (***constant, logarithmic, linear, log-linear, quadratic, cubic (polynomial), exponential***)
- To design efficient algorithms for finding Fibonacci numbers using ***dynamic programming***
- To find the GCD using Euclid's algorithm
- To finding prime numbers using the sieve of Eratosthenes
- To design efficient algorithms for finding the closest pair of points using the ***divide-and-conquer*** approach
- To solve the Eight Queens problem using the ***backtracking*** approach
- To design efficient algorithms for finding a convex hull for a set of points

Algorithms

- An ***algorithm*** is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation.
- ***Algorithm analysis*** is the process of finding the **computational complexity** of algorithms, that is: the execution time and the storage growth rate, or other resources needed to execute them.
- ***Algorithm design*** is the process of developing an algorithm for solving a problem.

Execution Time?

- Suppose two algorithms perform the same task, such as search (e.g., **linear search** vs. **binary search**) on sorted arrays
  - **Which one is better?**
    - One possible approach to answer this question is to implement these algorithms in Java and run the programs to get the execution times
      - But there are two problems for this approach:
        - First, the execution time is dependent on the **specific input**
          - Consider linear search and binary search of a key in a sorted array:
            - If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search
              - **But** in more cases binary search is much better
        - Second, there are many tasks running concurrently on a computer
          - The execution time of a particular program is dependent on the system load

Measuring Algorithm Efficiency Using Big O Notation!

- It is very difficult to compare algorithms by measuring their execution time
- To overcome this problem, a **"theoretical" approach** was developed to analyze algorithms **independent** of specific inputs and computers (i.e., system load or hardware: CPU speed, memory, multiple cores, etc.)
  - This approach approximates the effect of a change on the size of the input in the execution time, i.e., the ***Growth Rate***
    - How fast an algorithm's execution time increases as the input size increases, so we can compare two algorithms by examining their growth rates on any inputs or computers

Big O Notation

- Consider the linear search for an input array of size **`n`**:

```
public static int linearSearch(int[] list, int key) {
  for (int i = 0; i < list.length; i++)
    if (key == list[i])
      return i;
  return -1;
}
```

Big O Notation

- Linear search for an array of size `n`:
  - The linear search algorithm compares the **`key`** with the **`elements`** in the array **<u>sequentially</u>** until the **`key`** is found or the array is exhausted
    - If the key <u>is not in the array</u>, it requires **`n`** comparisons
    - If the key <u>is in the array</u>, it requires "***on average***" **`n/2`** comparisons
  - The algorithm's execution time is <u>proportional</u> to the size of the array:
    - If you double the size of the array **2*`n`**, you will expect the number of comparisons to double
      - If the key <u>is not in the array</u>, it requires **2*`n`** comparisons
      - If the key <u>is in the array</u>, it requires "***on average***" **2*`n/2=n`** comparisons
- This algorithm **<u>grows</u>** at a ***<u>linear</u>*** rate
  - The growth rate has ***an order of magnitude* growth rate** of **`n`**
  - Computer scientists use the **<u>Big O</u>** 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`**" - (also called "***linear***" time)

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*
  - 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
  - Because 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 inputs of the same size
  - Average-case analysis is ideal, but difficult to perform, because it is hard to determine or estimate the relative probabilities and distributions of various input instances for many problems
  - In most cases the average-case growth matches the worst-case growth rate
- Worst-case analysis is easier to obtain
  - So, the analysis is generally conducted for the worst-case

Ignoring Multiplicative Constants

- The multiplicative constants have no impact on growth rates!
  - The growth rate for **`n/2`** or **`100*n`** 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
growth rate	2	2	2

$f(200) / f(100)$

- For example: the linear search algorithm requires **`n`** comparisons in the worst-case and **`n/2`** comparisons in the average-case
  - Using the **<u>growth rate</u>** Big **`O`** notation, both cases require **`O(n)`** time
    - The multiplicative constant (**`1/2`**) can be omitted

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 sizes**
  - 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 highlights the important part (e.g., **`n`**).
- So, the complexity of this algorithm is **`O(n)`**.

Input size and *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, retrieving an element at a given index in an array takes constant time, because the time does not grow as the size of the array increases

Space complexity

- The Big O notation is usually used to measure the execution time (named ***time complexity***)
- ***Space complexity*** measures the amount of memory space used by an algorithm
  - We can also measure space complexity using the Big-O notation
  - The space complexity for most algorithms presented in our lectures is **`O(n)`** , i.e., they exhibit linear growth rate to the input size
    - For example, the space complexity for linear search is **`O(n)`** (i.e., the space required to store the array in memory grows linearly with the size of the array)

Useful Mathematic Summations

- The following mathematical summations are often useful in algorithm analysis:

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

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

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

Determining Big-O

- We will discuss examples to determine the Big-O value for:
  - Repetition
  - Sequence
  - Selection
  - Logarithm

Repetition: Simple Loops

executed
n times

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

It is a constant time
to execute **`k = k + 5`**

Time Complexity:

T(n) = (a constant c) * n = cn = **O(n)**

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

```java
public class PerformanceTest {
  public static void main(String[] args) {
    getTime(1000000);
    getTime(10000000);
    getTime(100000000);
    getTime(1000000000);
  }
  public static void getTime (long n) {
    long startTime = System.currentTimeMillis();
    long k = 0;
    for (int i = 1; i <= n; i++) {
      k = k + 5;
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Execution time for n = " + n
      + " is " + (endTime - startTime) + " milliseconds");
  }
}
```

```
Execution time for n = 1,000,000 is 6 milliseconds
Execution time for n = 10,000,000 is 61 milliseconds
Execution time for n = 100,000,000 is 610 milliseconds
Execution time for n = 1,000,000,000 is 6048 milliseconds
```

linear time complexity behaviour

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").

This is called

"Quadratic" time

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

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

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

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

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

O(n)

O(n) where **n** is **list.size()**

Time Complexity

T(n) = test time + worst-case (if, else)

= **O(n) + O(n)**

= **O(n)**

In-Class Quiz 1: Big O Notation 1

- What is the primary purpose of Big O notation?
  - To measure actual execution time
  - To estimate worst-case memory usage
  - To describe algorithm growth rate as input size increases
  - To calculate the average execution time on a machine

In-Class Quiz 2: Big O Notation 2

- Which of the following can be ignored in Big O notation?
  - Input size
  - Dominant term
  - Number of recursive calls
  - Constant multipliers and lower-order terms

Logarithmic time

```
result = 1;
for (int i = 1; i <= n; i++)        O(n)
    result *= a;
```

Without loss of generality, assume n = 2^k <=> k = $\log_2$ n .

```
result = a * a * … * a , n times
       = (…((a * a) * (a * a))
            * ((a * a) * (a * a)) …)
```

Therefore, we can improve the algorithm using the following scheme:

```
result = a;
for (int i = 1; i <= k; i++)        k times
    result = result * result;
```

T(n) = k = log n = **O(log n)**

The algorithm takes **O(log n)** (this is called a "logarithmic" time).

Analyzing Binary Search

- Binary search searches for a key in a sorted array:

```
public static int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;
  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }
  return -1 - low;
}
```

Analyzing Binary Search

- 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=`**$\mathbf{log_2 n}$
  - Binary search eliminates half of the input after two comparisons:

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{2^2}\right) + c + c = T\left(\frac{n}{2^k}\right) + kc$$

$$= T(1) + c\log n = 1 + (\log n)c$$

$$= O(\log n)$$

Analyzing Binary Search

- An algorithm with the `O(log n)` time complexity is called a ***logarithmic algorithm*** and it exhibits a logarithmic growth rate
  - 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
  - In the case of binary search, each time you double the array size, at most one more comparison will be required
  - If you square the input size of any logarithmic time algorithm, you only double the time of execution
  - So a logarithmic-time algorithm is very efficient!

Analyzing Selection Sort

```
public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
      // Find the minimum in the list[i..list.length-1]
      double currentMin = list[i];
      int currentMinIndex = i;
      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }
      // Swap list[i] with list[currentMinIndex] if necessary;
      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
}
```

Analyzing Selection Sort

- Selection sort finds the smallest element in the list and swaps it with the first element
  - It then finds the smallest element remaining and swaps it with the first element in the remaining list, and so on until the remaining list contains only one element left to be sorted.
  - The number of comparisons is **`n - 1`** for the first iteration, **`n - 2`** for the second iteration, and so on
  - **`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

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

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

Insertion Sort

`int[] myList = {2,9,5,4,8,1,6};` // Unsorted

The ***insertion sort*** algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: The sorted sublist is {2, 9}. Insert 5 into the sublist.

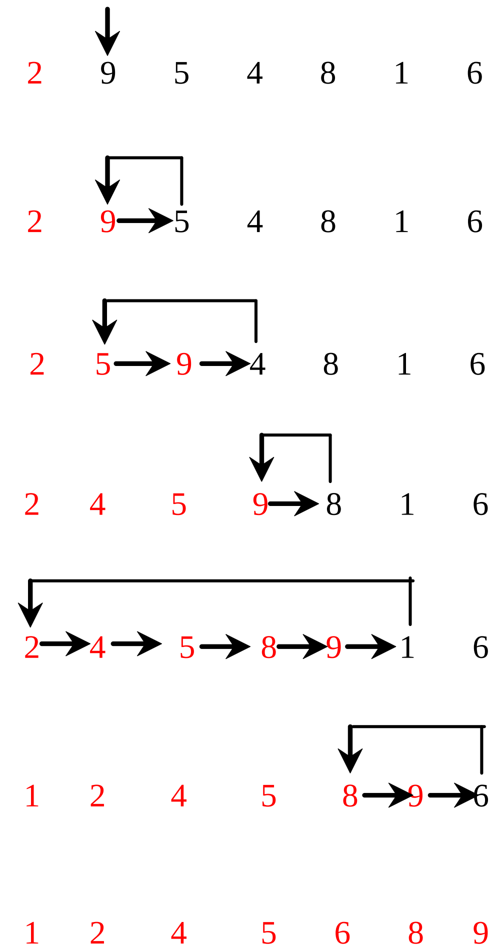
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.

Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.

Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.

Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.



How to Insert 4 in sorted order?

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	5	9	4			

Step 1: Save 4 to a temporary variable currentElement

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	5		9			

Step 2: Move list[2] to list[3]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2		5	9			

Step 3: Move list[1] to list[2]

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
list	2	4	5	9			

Step 4: Assign currentElement to list[1]

InsertionSort - From Idea to Solution

```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
         list[0..i] is sorted
}
```

Expand

```
    int currentElement = list[i];
    int k;
    for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
      list[k + 1] = list[k];
    }
    // Insert the current element into list[k + 1]
    list[k + 1] = currentElement;
```

```java
public static void insertionSort(int[] list) {
  for (int i = 1; i < list.length; i++) {
    int currentElement = list[i];
    int k;
    for (k = i - 1; k >= 0 && list[k] > currentElement; k--)
      list[k + 1] = list[k];
    // Insert the current element into list[k + 1]
    list[k + 1] = currentElement;
  }
}
```

To insert the first element, we need 1 comparison and at most 1 swap

To insert the last element, we need $n-1$ comparisons and at most $n-1$ swaps

c denotes the total number of other operations such as assignments and additional comparisons in each iteration

$$
\begin{aligned}
T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\
&= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\
&= 2\frac{(n - 1)n}{2} + cn - c = n^2 - n + cn - c \\
&= O(n^2)
\end{aligned}
$$

```java
public class InsertionSort {
  public static void main(String[] args) {
    int size = 100000;
    int[] a = new int[size];
    randomInitiate(a);
    long startTime = System.currentTimeMillis();
    insertionSort(a);
    long endTime = System.currentTimeMillis();
    System.out.println((endTime - startTime) + "ms");
  }
  private static void randomInitiate(int[] a) {
    for (int i = 0; i < a.length; i++)
      a[i] = (int) (Math.random() * a.length);
  }
  public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
      int currentElement = list[i];
      int k;
      for (k = i - 1; k >= 0 && list[k] > currentElement; k--)
        list[k + 1] = list[k];
      // Insert the current element into list[k + 1]
      list[k + 1] = currentElement;
    }
  }
}
```

3641ms

In-Class Quiz 3: Time Complexity

- Which of the following algorithms has logarithmic time complexity?
  - Linear search
  - Binary search
  - Insertion sort
  - Selection sort

Polynomial Complexity

- An algorithm is said to be of ***polynomial*** time if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $\mathbf{T(n) = O(n^k)}$ for some positive constant k
- The concept of polynomial time leads to several complexity classes in computational complexity theory:
  - **P** = The complexity class of decision problems that can be solved on a **deterministic** Turing machine in polynomial time.
  - **NP (nondeterministic polynomial time)** = The complexity class of decision problems that can be solved on a **non-deterministic** Turing machine in polynomial time.
    - It means that a given solution can be **verified** in polynomial time.

NP-completeness

- Although a solution to an NP problem **can be verified "quickly" (in polynomial time)**, there is **no known way to find such a solution quickly**.
- ***NP-hard*** ("non-deterministic polynomial acceptable problems") = the class of problems which are at least as hard as the hardest problems in NP.
  - A problem is said to be ***NP-hard*** if everything in NP can be transformed into it in polynomial time.
- A problem is ***NP-complete*** if it is both in NP and NP-hard.
  - P=NP? Unsolved problem

NP-completeness

- **NP-complete Problems:**
  - Boolean satisfiability problem (SAT)
  - Knapsack problem
  - Hamiltonian path problem
  - Traveling salesman problem
  - Graph coloring problem
  - Subgraph isomorphism problem
  - Subset sum problem
  - Clique problem
  - Vertex cover problem
  - Independent set problem
  - Dominating set problem

NP-completeness

- ***Boolean satisfiability problem*** (SAT) (sometimes called ***propositional satisfiability problem*** and abbreviated ***SATISFIABILITY***, ***SAT*** or ***B-SAT***) is the problem of determining if there exists an interpretation (i.e., truth assignment) that satisfies a given Boolean formula (built from boolean variables, operators AND (conjunction, ∧), OR (disjunction, ∨), NOT (negation, ¬), and parentheses).
  - A formula is said to be ***satisfiable*** if it can be made TRUE by assigning appropriate logical values (i.e. TRUE, FALSE) to its propositional variables.

NP-completeness

- ***3-satisfiability problem*** (3-SAT): determine the satisfiability of a formula in conjunctive normal form where each clause is limited to at most three literals

  $(l_1 \lor l_2 \lor l_3) \land (l_4 \lor l_5 \lor x_6) \land \cdots \land (l_{n-2} \lor l_{n-1} \lor l_n)$

  - It is also NP-complete
  - 3-SAT is one of Karp's 21 NP-complete problems

Richard Manning Karp

Stephen Cook

NP-completeness

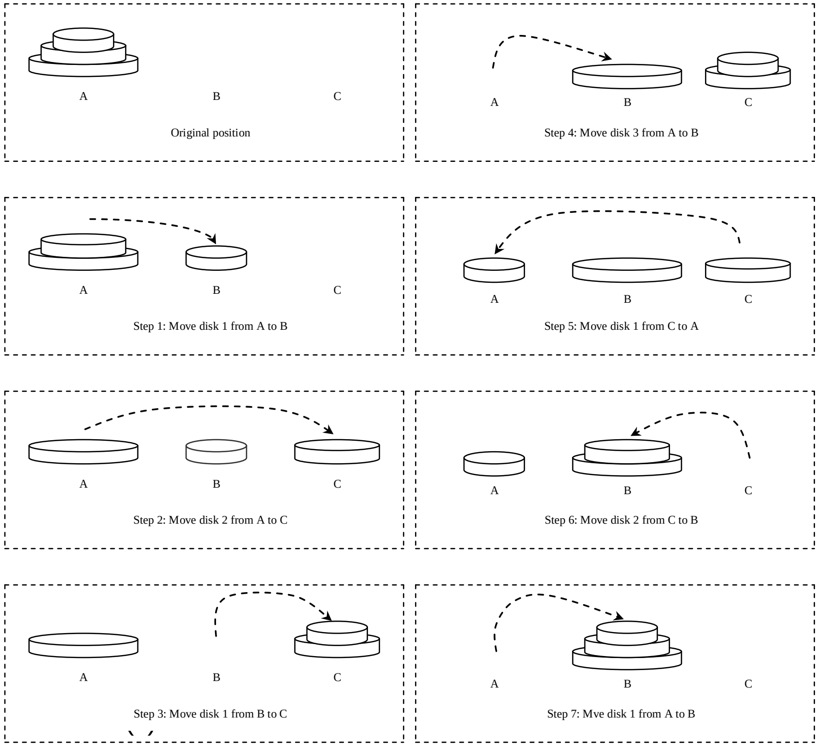
- ***Knapsack <u>decision</u> problem***: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit without exceeding a given weight.
  - The optimization problem is to get the total value as large as possible
- ***Graph coloring problem***: is there an assignment of labels (traditionally called "colors") to elements of a graph subject to certain constraints?
  - Its simplest form, called ***vertex coloring***, asks if there is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color.

NP-completeness

- ***Hamiltonian path problem***: determining whether a Hamiltonian path (i.e., a path in an undirected or directed graph that visits **<u>each vertex exactly once</u>**) or a Hamiltonian cycle exists in a given graph (whether directed or undirected).
  - ***Traveling salesman problem*** (the <u>decision</u> version): Given a list of cities and the distances between each pair of cities, what is the possible route that visits each city exactly once and returns to the origin city?
    - The optimization problem is to find the shortest path:
      - This is an optimization problem, and hence cannot be in NP.

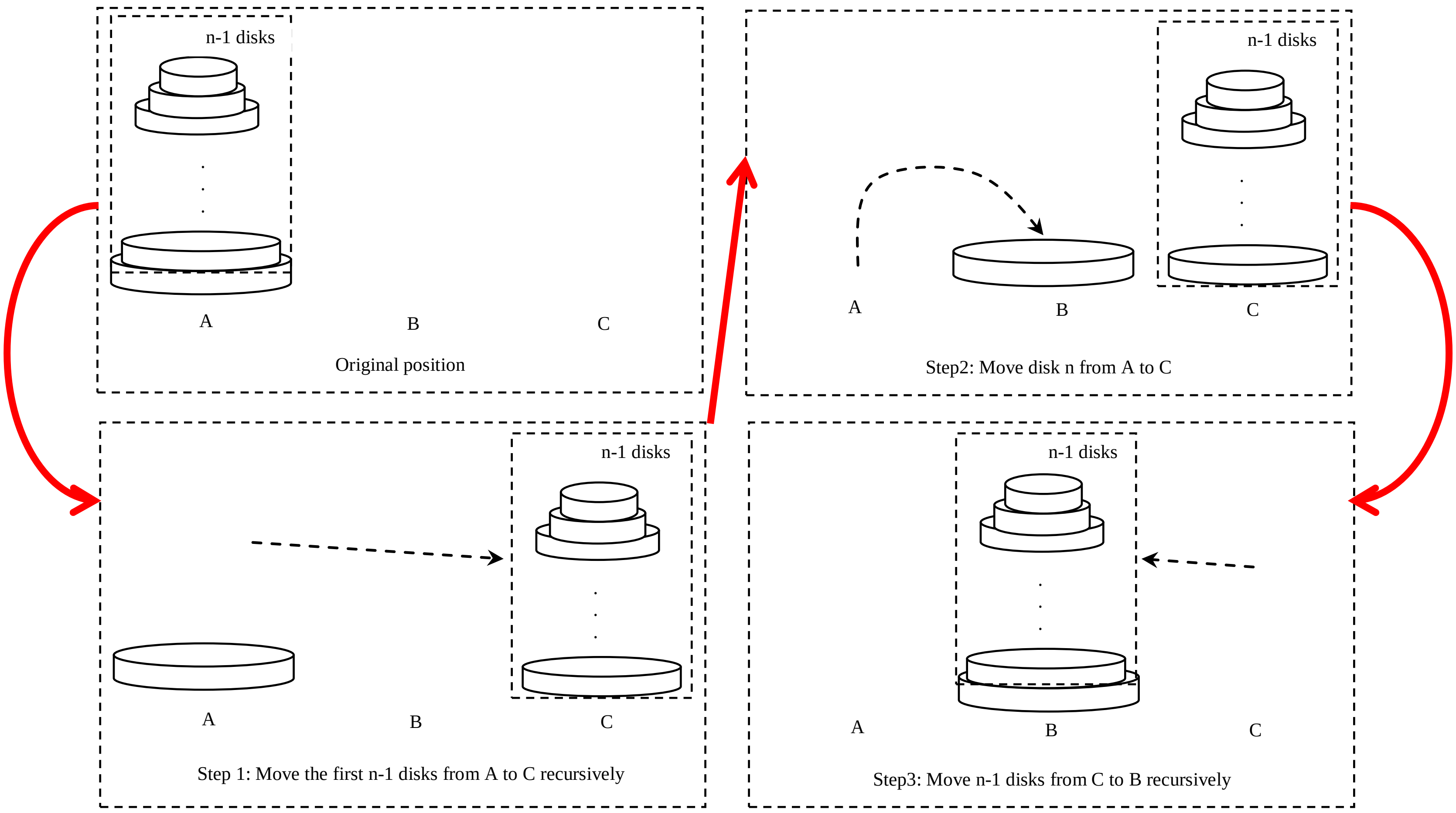
Exponential complexity: Analyzing the Towers of Hanoi

- There are n disks labeled 1, 2, 3,…, n, and three towers labeled A, B, and C.
- No disk can be on top of a smaller disk at any time.
- All the disks are initially placed on tower A.
- Only one disk can be moved at a time, and it must be the top disk on the tower.



The Towers of Hanoi problem can be decomposed into three subproblems:

- Move the first <u>n - 1</u> disks from A to C with the assistance of tower B.
- Move the disk <u>n</u> from A to B.
- Move <u>n - 1</u> disks from C to B with the assistance of tower A.



```java
import java.util.Scanner;
public class TowersOfHanoi {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter number of disks: ");
    int n = input.nextInt(); System.out.println("The moves are:");
    moveDisks(n, 'A', 'B', 'C');
  }
  public static void moveDisks(int n, char fromTower, char toTower,
       char auxTower) {
    if (n == 1) // Stopping condition
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
    else {
      moveDisks(n - 1, fromTower, auxTower, toTower);
      System.out.println("Move disk " + n + " from " +
        fromTower + " to " + toTower);
      moveDisks(n - 1, auxTower, toTower, fromTower);
    }
  }
}
```

Analyzing Towers of Hanoi

- Towers of Hanoi problem recursively moves **n** disks from tower **A** to tower **B** with the assistance of tower **C**:
  - 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**
- The complexity of this algorithm is measured by the number of moves.
  - Let **T(n)** denote the number of moves for the algorithm to move **n** disks from tower **A** to tower **B**:

$$
\begin{aligned}
T(n) &= T(n-1) + 1 + T(n-1) \\
&= 2T(n-1) + 1 \\
&= 2(2T(n-2) + 1) + 1 \\
&= 2(2(2T(n-3) + 1) + 1) + 1 \\
&= 2^{n-1}T(1) + 2^{n-2} + \ldots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = (2^n - 1) = O(2^n)
\end{aligned}
$$

Analyzing Towers of Hanoi

- An algorithm with `O(2ⁿ)` time complexity is called an ***exponential*** algorithm, and it exhibits an exponential growth rate (i.e., as the input size increases, the time for the exponential algorithm grows exponentially; e.g., adding each one extra disk results in double the execution time).
  - Exponential algorithms are not practical for large input sizes:
  - Suppose the disk is moved at a rate of 1 per second.

    To move 32 disks would take: 2^{32}/(365days*24hours*60min*60sec) = 136 years;

    To move 64 disks would take 2^{64}/(365*24*60*60sec) = 585 billion years

Analyzing Recursive Fibonacci Numbers

```
fib(0) = 0;
fib(1) = 1;
fib(index) = fib(index -1) + fib(index -2); 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
```

/** The recursive method for finding the Fibonacci number */

```
public static int fib(int 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);
}
```

```java
import java.util.Scanner;
public class Fibonacci {
   public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the Fibonacci index: ");
        int n = input.nextInt();
        System.out.println("fib(" + n + ") = " + fib(n));
   }

   public static int fib(int 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);
   }
}
```

```
Enter the Fibonacci index: 20
fib(20) = 6765
Enter the Fibonacci index: 30
fib(30) = 832040
```

```java
import java.util.Scanner;
public class Fibonacci2 {
   public static void main(String[] args) {
         Scanner input = new Scanner(System.in);
         System.out.print("Enter the Fibonacci index: ");
         int n = input.nextInt();
         System.out.println("fib(" + n + ") = " + fib(n));
         System.out.println("steps: " + steps);
   }
   static int steps = 0;
   public static int fib(int index) {
         steps++;
         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);
   }
}
```

```
Enter the Fibonacci index: 20
fib(20) = 6765
steps: 21891
Enter the Fibonacci index: 30
fib(30) = 832040
steps: 2692537
```

Complexity for Recursive Fibonacci Numbers

- Let **`T(n)`** denote the complexity for the algorithm that finds **`fib(n)`**

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

and

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

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

This algorithm is not efficient.

- Is there an efficient algorithm for finding a Fibonacci number? Yes, of course. It is a linear sequence.

Non-recursive version of Fibonacci Numbers

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

- The complexity of this new algorithm is ***O*(n)**.

- This is a tremendous improvement over the recursive algorithm.

- Variables **`f0`**, **`f1`**, and **`f2`** store three consecutive Fibonacci numbers in the series:

```
                  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
```

```java
import java.util.Scanner;
public class Fibonacci3 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the Fibonacci index: ");
        int n = input.nextInt();
        System.out.println("fib(" + n + ") = " + fib(n));
        System.out.println("steps: " + steps);
    }
    static int steps = 0;
    public static int fib(int n) {
        if (n == 0)
          return 0;
        else if (n == 1 || n == 2)
                return f2;
        int f0 = 0; // For fib(0)
        int f1 = 1; // For fib(1)
        int f2 = 1; // For fib(2)
        steps = 3;
        for (int i = 3; i <= n; i++) {
                steps++;
                f0 = f1;
                f1 = f2;
                f2 = f0 + f1;
        }
        return f2;
    }
}
```

```
Enter the Fibonacci index: 10
fib(10) = 55
steps: 11
Enter the Fibonacci index: 30
fib(30) = 832040
steps: 31
```

Common Recurrence Relations

- Recurrence relations are a useful tool for analyzing algorithm complexity

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
$T(n) = 2T(n/2) + O(n \log n)$	$T(n) = O(n \log^2 n)$	
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	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

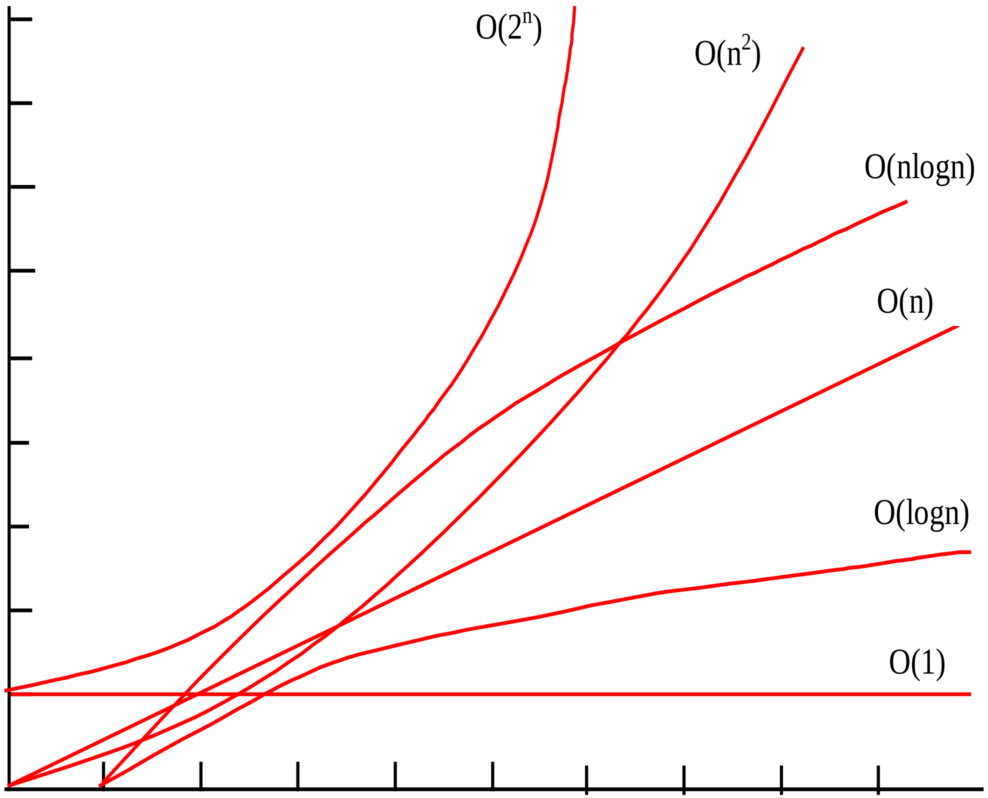
Comparing Common Growth Functions

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

Function	*Name*	$n = 25$	$n = 50$	$f(50)/f(25)$
$O(1)$	Constant time	1	1	1
$O(\log n)$	Logarithmic time	4.64	5.64	1.21
$O(n)$	Linear time	25	50	2
$O(n \log n)$	Log-linear time	116	282	2.43
$O(n^2)$	Quadratic time	625	2,500	4
$O(n^3)$	Cubic time	15,625	125,000	8
$O(2^n)$	Exponential time	3.36×10^7	1.27×10^{15}	3.35×10^7

Comparing Common Growth Functions

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



Algorithm Design

- Typical steps in the development of algorithms:
  1. Problem definition
  2. Development of a model
  3. Specification of the algorithm
  4. Designing an algorithm
  5. Checking the correctness of the algorithm
  6. Analysis of algorithm
  7. Implementation of algorithm
  8. Program testing
  9. Documentation preparation

Algorithm Techniques

- Techniques for designing and implementing algorithm designs are called ***algorithm design patterns***
  - ***Brute-force*** or ***exhaustive search***: the naive method of trying every possible solution to see which is best.
  - ***Divide and conquer***: repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily.
    - An example of divide and conquer is merge sorting: divide the data into 2 halves and sort them, then the conquer phase of merging the segments

Algorithm Techniques

- ***Dynamic programming:*** when the same subproblems are used to solve many different problem instances, dynamic programming avoids recomputing solutions that have already been computed.
  - The main difference between dynamic programming and divide and conquer is that subproblems are more independent in divide and conquer, whereas subproblems overlap in dynamic programming.
  - The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls.
  - Non-recursive Fibonacci is an example of dynamic programming

Algorithm Techniques

- ***Greedy algorithms*** follow the problem-solving heuristic of making the **<u>locally optimal choice</u>** at each stage.
  - Example: a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is to follow the heuristic: "*At each step of the journey, visit the nearest unvisited city.*"
    - This heuristic does not intend to find a best solution, but it terminates in a reasonable number of steps.
      - Finding an optimal solution to such a complex problem typically requires unreasonably many steps.
- ***Backtracking***: multiple solutions are built incrementally and abandoned when it is determined that they cannot lead to a valid full solution.

Dynamic Programming

- The non-recursive algorithm for computing Fibonacci numbers is an example of ***dynamic programming***

```
public static long[] f;
public static long fib(long index) {
  if (index == 0)
    return 0;
  if (index == 1) {
    f[1]=1;
    return 1;
  }
  if(f[index]!=0)
    return f[index];
  else  // Reduction and recursive calls
    f[index] = fib(index - 1) + f[index - 2];
  return f[index];
}
```

```java
import java.util.Scanner;
public class Fibonacci4 {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the Fibonacci index: ");
        int n = input.nextInt();
        f = new long[n+1];
        System.out.println("fib(" + n + ") = " + fib(n));
    }
    public static long[] f;
    public static long fib(int index) {
        if (index == 0)
            return 0;
        if (index == 1) {
            f[1] = 1;
            return 1;
        }
        if (f[index] != 0)
            return f[index];
        else // Reduction and recursive calls
            f[index] = fib(index - 1) + fib(index - 2);
        return f[index];
    }
}
```

```
Enter the Fibonacci index: 10
fib(10) = 55
```

In-Class Quiz 4: Fibonacci

- How does the performance of recursive Fibonacci compare to its Dynamic Programming version?
    - They have the same efficiency
    - Recursive version is faster
    - Dynamic programming avoids redundant computation and is faster
    - Recursive version uses less memory and is better

Dynamic Programming

- The Fibonacci algorithm solves subproblems, then combines the solutions of subproblems to obtain an overall solution
  - This naturally leads to original recursive solution
  - However, it is inefficient to use just recursion, because the **subproblems overlap**
- Recognize Dynamic programming:
  - The solution of subproblems are used in many places
  - The key idea behind dynamic programming is to **solve each subprogram only once** and **store the results** for subproblems for later use to avoid redundant computing of the subproblems

Analyzing 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;
}
```

The complexity of this algorithm is $O(\mathrm{n})$

```java
import java.util.Scanner;

public class GCD {

   public static void main(String[] args) {
         Scanner input = new Scanner(System.in);
         System.out.print("Enter the numbers: ");
         int n1 = input.nextInt();
         int n2 = input.nextInt();
         System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
   }

   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;
   }

}
```

```
Enter the numbers: 5 40
gcd(5,40) = 5
```

Analyzing 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 $O(n)$

```java
import java.util.Scanner;

public class GCD2 {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the numbers: ");
        int n1 = input.nextInt();
        int n2 = input.nextInt();
        System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
    }

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

}
```

```
Enter the numbers: 5 40
gcd(5,40) = 5
```

Analyzing 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 O(n)

Euclid's algorithm

- A more efficient algorithm for finding the GCD was discovered by Euclid around 300 b.c

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).

If you divide m by n: $m = n*k + r$

if p is a divisor of both m and n, it must be

divisor of r: $m/p = n/p *k + r/p$

$\in Z \quad \in Z \quad -> \in Z$

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).

```java
import java.util.Scanner;

public class GCD4 {

    public static void main(String[] args) {
          Scanner input = new Scanner(System.in);
          System.out.print("Enter the numbers: ");
          int n1 = input.nextInt();
          int n2 = input.nextInt();
          System.out.println("gcd(" + n1 + "," + n2 + ") = " + gcd(n1, n2));
    }

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

}
```

```
Enter the numbers: 5 40
gcd(5,40) = 5
```

Euclid's algorithm

- Time Complexity **Proof:**
  - In the best case when `m % n` is 0, the algorithm takes just one step to find the GCD.
  - The worst-case time complexity is `O(log n)`:
    - Assuming `m >= n`, we can show that `m % n < m / 2`, as follows:
      - If `n <= m / 2`, then `m % n < m / 2` since the remainder of `m` divided by `n` is always less than `n`.
      - If `n > m / 2`, then `m % n = m – n < m / 2`.
      - Therefore, `m % n < m / 2`.
    - Euclid's algorithm recursively invokes the gcd method: it first calls `gcd(m, n)`, then calls `gcd(n, m % n)`, and `gcd(m % n, n % (m % n))`, and so on.
    - Since `m % n < m / 2` and `n % (m % n) < n / 2`, the argument passed to the `gcd` method is reduced by half after every two iterations.

GCD algorithms

- Time Complexity

Comparisons of GCD Algorithms

Description	*Complexity*
Brute-force, checking all possible divisors	$O(n)$
Checking half of all possible divisors	$O(n)$
Euclid's algorithm	$O(\log n)$

Efficient Algorithms for Finding Prime Numbers

- An integer greater than 1 is *prime* if its only positive divisors are 1 and itself.
- **A $150,000 award awaits the first individual or group who discovers a prime number with at least 100,000,000 decimal digits:**

  http://w2.eff.org/awards/coop-prime-rules.php

  - We will compare three versions of an algorithm to find all the prime **`number`**s less than some number **`n`**:
    - Brute-force ☹
    - Check possible divisors up to **`Math.sqrt(number)`**
    - Check ***possible*** prime divisors up to **`Math.sqrt(number)`**

Brute-force Finding Prime Numbers

```
Scanner input = new Scanner(System.in);
System.out.print("Find all prime numbers <= n, enter n: ");
int n = input.nextInt();
final int NUMBER_PER_LINE = 10; // Display 10 per line
int count = 0; // Count the number of prime numbers
int number = 2; // A number to be tested for primeness
System.out.println("The prime numbers are:");
// Repeatedly find prime numbers
while (number <= n) {
  // Assume the number is prime
  boolean isPrime = true; // Is the current number prime?
  // ClosestPair if number is prime
  for (int divisor = 2; divisor <= (int)(Math.sqrt(number));
      divisor++) {
    if (number % divisor == 0) { // If true, number is not prime
      isPrime = false; // Set isPrime to false
      break; // Exit the for loop
    }
  }
  // Print the prime number and increase the count
  if (isPrime) {
    count++; // Increase the count
    if (count % NUMBER_PER_LINE == 0) {
      // Print the number and advance to the new line
      System.out.printf("%7d\n", number);
    }
    else
      System.out.printf("%7d", number);
  }
  // Check if the next number is prime
  number++;
}
System.out.println("\n" + count + " prime(s) less than or equal to " + n);
```

```java
import java.util.Scanner;
public class Primes {
   public static void main(String[] args) {
         Scanner input = new Scanner(System.in);
         System.out.print("Find all prime numbers <= n, enter n: ");
         int n = input.nextInt();
         final int NUMBER_PER_LINE = 10; // Display 10 per line
         int count = 0; // Count the number of prime numbers
         int number = 2; // A number to be tested for primeness
         System.out.println("The prime numbers are:");
         // Repeatedly find prime numbers
         while (number <= n) {
                  // Assume the number is prime
                  boolean isPrime = true; // Is the current number prime?
                  // ClosestPair if number is prime
                  for (int divisor = 2; divisor <= (int) (Math.sqrt(number)); divisor++) {
                           if (number % divisor == 0) { // If true, number is not prime
                                    isPrime = false; // Set isPrime to false
                                    break; // Exit the for loop
                           }
                  }
                  // Print the prime number and increase the count
                  if (isPrime) {
                           count++; // Increase the count
                           if (count % NUMBER_PER_LINE == 0) {
                                    // Print the number and advance to the new line
                                    System.out.printf("%7d\n", number);
                           } else
                                    System.out.printf("%7d", number);
                  }
                  // Check if the next number is prime
                  number++;
         }
         System.out.println("\n" + count + " prime(s) less than or equal to " + n);
   }

}
```

```
Find all prime numbers <= n, enter n: 20
The prime numbers are:
      2      3      5      7     11     13     17     19
8 prime(s) less than or equal to 20
```

Brute-force Finding Prime Numbers

- Brute force algorithm improvements:
  - The program is not efficient if you have to compute **`Math.sqrt(number)`** for every iteration of the **`for`** loop.
  - A good compiler should evaluate **`Math.sqrt(number)`** only once for the entire **`for`** loop

```
int squareRoot =
    (int)(Math.sqrt(number));
for (int divisor = 2;
    divisor <= squareRoot; divisor++){
```

Finding Prime Numbers

- In fact, there is no need to actually compute **`Math.sqrt(number)`** for every **`number`**
  - For all the numbers between 36 and 48, inclusively, their **`(int)(Math.sqrt(number))`** is 6.
  - We only need to look for the perfect squares such as 4, 9, 16, 25, 36, 49, and so on.

```
int squareRoot = 1;
// Repeatedly find prime numbers
while (number <= n) {
     // Assume the number is prime
     boolean isPrime = true; // Is the current number prime?
     if (squareRoot * squareRoot < number) squareRoot++;
     // Test if number is prime
     for (int divisor = 2; divisor <= squareRoot; divisor++) {
      if (number % divisor == 0) { // If true, number is not prime
           isPrime = false; // Set isPrime to false
           break; // Exit the for loop
      }
     }
```

Finding Prime Numbers

- Since it takes $\sqrt{i}$ steps in the for loop to check whether number **`i`** is prime, the algorithm takes $\sqrt{2}+\sqrt{3}+\sqrt{4}+\ldots+\sqrt{n}$ steps to find all the prime numbers less than or equal to **`n`**.
- $\sqrt{2}+\sqrt{3}+\sqrt{4}+\ldots+\sqrt{n} <= n\sqrt{n}$
- Therefore, the time complexity for this algorithm is $O(n\sqrt{n})$

Actually, it is better than $n\sqrt{n}$

- We can prove that if **`i`** is not prime, there must exist a prime number **`p`** such that **`i = pq`** and **`p <= q`**.
- Let **`π(i)`** denote the number of prime numbers less than or equal to **`i`**.
  - **`π(2)`** is 1, **`π(3)`** is 2, **`π(6)`** is 3, and **`π(20)`** is 8
- It has been proved that **`π(i)`** is approximately **`i/log i`**
  - http://primes.utm.edu/howmany.html
  - The number of the prime numbers less than or equal to $\sqrt{i}$ is

$$\frac{\sqrt{i}}{\log\sqrt{i}} = \frac{2\sqrt{i}}{\log i}$$

  - Moreover, prime numbers are relatively uniformly distributed

Finding Prime Numbers

- Thus, the complexity for finding all prime numbers up to **n** is:

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \ldots + \frac{2\sqrt{n}}{\log n}$$

Since $\frac{\sqrt{i}}{\log i} < \frac{\sqrt{n}}{\log n}$ for $i < n$ and $n \geq 16$,

$$\frac{2\sqrt{2}}{\log 2} + \frac{2\sqrt{3}}{\log 3} + \frac{2\sqrt{4}}{\log 4} + \frac{2\sqrt{5}}{\log 5} + \frac{2\sqrt{6}}{\log 6} + \frac{2\sqrt{7}}{\log 7} + \frac{2\sqrt{8}}{\log 8} + \ldots + \frac{2\sqrt{n}}{\log n} < \frac{2n\sqrt{n}}{\log n}$$

Therefore, the complexity of this algorithm is $O\left(\frac{n\sqrt{n}}{\log n}\right)$.

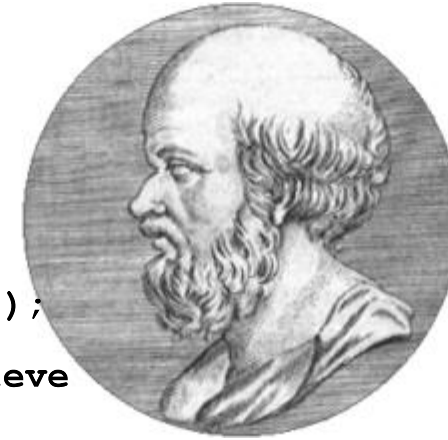
Finding Prime Numbers

- Sieve of Eratosthenes (276–194 b.c.) for finding all prime numbers ≤n
  - use an array named **`primes`** of **`n`** Boolean
    - initially all values are **`true`**
    - the multiples of 2 are not prime, set **`primes[2*i]`** to **`false`** for all **`2≤i≤n/2`**
    - Since the multiples of 3 are not prime, set **`primes[3*i]`** to **`false`** for all **`3≤i≤n/3`**

primes array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
initial	×	×	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$k = 2$	×	×	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
$k = 3$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F
$k = 5$	×	×	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F	F	F

Finding Prime Numbers



```
import java.util.Scanner;
public class SieveOfEratosthenes {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Find all prime numbers <= n, enter n: ");
    int n = input.nextInt();
    boolean[] primes = new boolean[n + 1]; // Prime number sieve
    for (int i = 0; i < primes.length; i++) {
      primes[i] = true;
    }
    for (int k = 2; k <= n / k; k++) {
      if (primes[k]) {
        for (int i = k; i <= n / k; i++) {
          primes[k * i] = false; // k * i is not prime
        }
      }
    }
    final int NUMBER_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers found so far
    for (int i = 2; i < primes.length; i++) {
      if (primes[i]) {
        count++;
        if (count % 10 == 0)
          System.out.printf("%7d\n", i);
        else
          System.out.printf("%7d", i);
      }
    }
    System.out.println("\n" + count + " prime(s) less than or equal to " + n);
  }
}
```

- 2, 3, 5, 7, 11, 13, 17, 19, and 23 are prime numbers

Finding Prime Numbers

- For each prime number **`k`**, the algorithm sets **`primes[k*i]`** to **`false`**
  - This is performed **`n/k - k + 1`** times in the for loop (i = k; i<=n/ k; i++) primes[k * i] = false; so:

$$\frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \ldots$$

$$< O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \ \ldots\right) < O(n\pi(n))$$

$$= O\left(n\frac{n}{\log n}\right)$$

The number of items in the series is $\pi(n)$.

- This upper bound is very loose: The actual time complexity is much better
  - The Sieve of Eratosthenes algorithm is good for a small **`n`** such that the array **`primes`** can fit in the memory.

Divide-and-Conquer

- The ***divide-and-conquer*** approach <u>divides the problem into subproblems</u>, <u>solves the subproblems</u>, then <u>combines the solutions of subproblems to obtain the solution for the entire problem</u>.
  - 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.

Finding the Closest Pair of Points Using Divide-and-Conquer

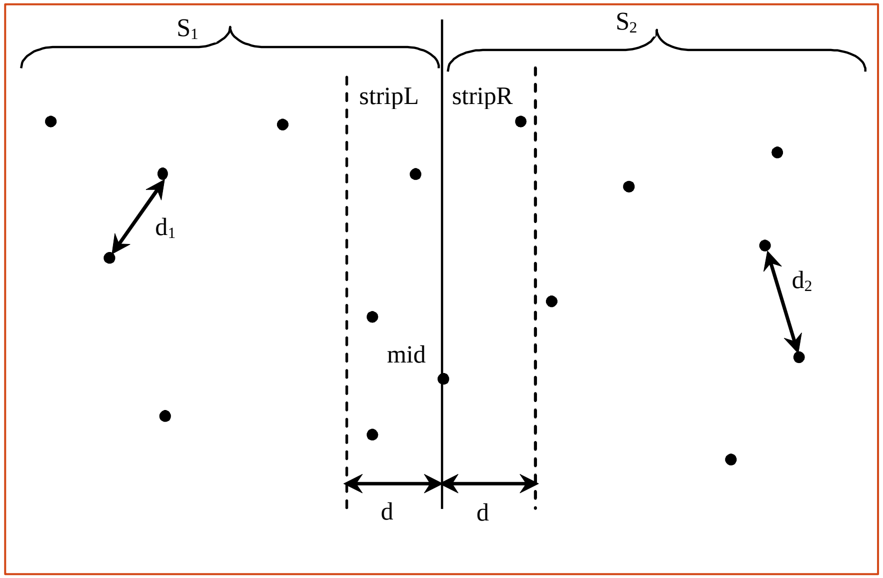
- Given a set of points, the ***closest-pair problem*** is to find the two points that are nearest to each other
  - A brute-force algorithm for finding the closest pair of points that computes the distances between all pairs of points and finds the one with the minimum distance takes $O(n^2)$
- ***Divide-and-conquer*** divides the problem into subproblems, solves the subproblems, then combines the solutions of the subproblems to obtain the solution for the entire problem

Finding the Closest Pair of Points Using Divide-and-Conquer

- Step 1: Sort the points in increasing order of **x**−coordinates.
  - For the points with the same **x**-coordinates, sort on **y**-coordinates.
  - This results in a sorted list **S** of points.
- Step 2: Divide **S** into two subsets, **S1** and **S2**, of equal size using the midpoint **mid** in the sorted list.
  - Let the midpoint **mid** be in **S1**.
  - Recursively find the closest pair in **S1** and **S2**.
  - Let **d1** and **d2** denote the distance of the closest pairs in the two subsets, respectively.
    - Compute **d = min(d1, d2)**
- Step 3: Find the closest pair between a point in **S1** and a point in **S2** with **y**-coordinate in the range [middle point **x**-coordinate – **d**, middle point **x**-coordinate + **d**] and denote their distance as **d3**
- The closest pair is the one with the distance **min(d1,d2,d3)**

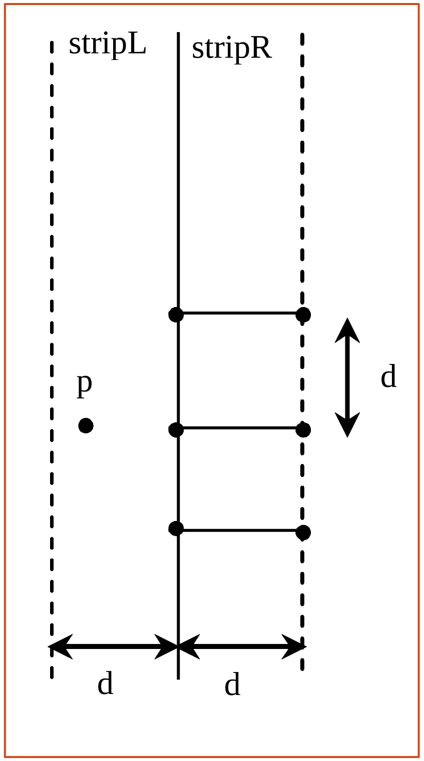
Analysis

- Step 1: can be done in **`O(n log n)`**
  - it is just sorting
- Step 3: can be done in **`O(n)`**
  - Let **`d = min(d1, d2)`**
  - For a point in **`S1`** and a point in **`S2`** to form the closest pair in **`S`**, the left point MUST be in **`stripL`** and the right point in **`stripR`** (sorted by Y-coordinate)



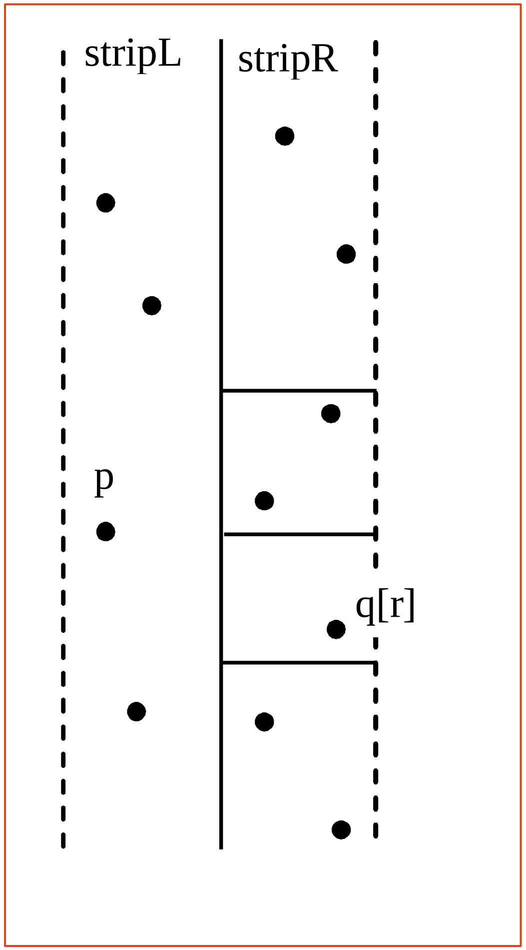
Algorithm for Obtaining stripL and stripR

```
for each point p in pointsOrderedOnY
  if (p is in S1 and mid.x - p.x <= d)
    append p to stripL;
  else if (p is in S2 and p.x - mid.x <= d)
    append p to stripR;
```



Algorithm for Finding the Closest Pair in Step 3

```
d = min(d1, d2);
r = 0; // r is the index of a point in stripR
for (each point p in stripL) {
  // Skip the points in stripR below p.y - d
  while (r < stripR.length && q[r].y <= p.y - d)
    r++;

  let r1 = r;
  while (r1 < stripR.length && |q[r1].y - p.y| <= d) {
    // Check if (p, q[r1]) is a possible closest pair
    if (distance(p, q[r1]) < d) {
      d = distance(p, q[r1]);
      (p, q[r1]) is now the current closest pair;
    }

    r1 = r1 + 1;
  }
}
```



Finding the Closest Pair of Points Using Divide-and-Conquer

$$T(n) = \underbrace{2T(n/2)}_{\text{Step 2}} + \underbrace{O(n)}_{\text{Step 3}} = O(n \log n)$$

```
import java.util.*;
public class ClosestPair {
  // Each row in points represents a point
  private double[][] points;
  Point p1, p2;
  public static void main(String[] args) {
    double[][] points = new double[500][2];
    for (int i = 0; i < points.length; i++) {
      points[i][0] = Math.random() * 100;
      points[i][1] = Math.random() * 100;
    }
    ClosestPair closestPair = new ClosestPair(points);
    System.out.println("shortest distance is " +
      closestPair.getMinimumDistance());
    System.out.print("(" + closestPair.p1.x + ", " +
      closestPair.p1.y + ") to ");
    System.out.println("(" + closestPair.p2.x + ", " +
      closestPair.p2.y + ")");
  }

  public ClosestPair(double[][] points) {
    this.points = points;
  }

  public double getMinimumDistance() {
    Point[] pointsOrderedOnX = new Point[points.length];
    for (int i = 0; i < pointsOrderedOnX.length; i++)
      pointsOrderedOnX[i] = new Point(points[i][0], points[i][1]);
    Arrays.sort(pointsOrderedOnX);
    // Locate the identical points if exists
    if (checkIdentical(pointsOrderedOnX))
      return 0; // The distance between the identical points is 0
    Point[] pointsOrderedOnY = pointsOrderedOnX.clone();
    Arrays.sort(pointsOrderedOnY);
    return distance(pointsOrderedOnX, 0,
        pointsOrderedOnX.length - 1, pointsOrderedOnY);
```

```
public boolean checkIdentical(Point[] pointsOrderedOnX) {
  for (int i = 0; i < pointsOrderedOnX.length - 1; i++) {
    if (pointsOrderedOnX[i].compareTo(pointsOrderedOnX[i + 1]) == 0) {
      p1 = pointsOrderedOnX[i];
      p2 = pointsOrderedOnX[i + 1];
      return true;
    }
  }
  return false;
}
/** Compute the distance between two points p1 and p2 */
public static double distance(Point p1, Point p2) {
  return distance(p1.x, p1.y, p2.x, p2.y);
}
/** Compute the distance between two points (x1, y1) and (x2, y2) */
public static double distance(
    double x1, double y1, double x2, double y2) {
  return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}
static class Point implements Comparable<Point> {
  double x;
  double y;
  Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public int compareTo(Point p2) {
    if (this.x < p2.x)
      return -1;
    else if (this.x == p2.x) {
      // Secondary order on y-coordinates
      if (this.y < p2.y)
        return -1;
      else if (this.y == p2.y)
        return 0;
      else
        return 1;
    } else return 1; }  }
```

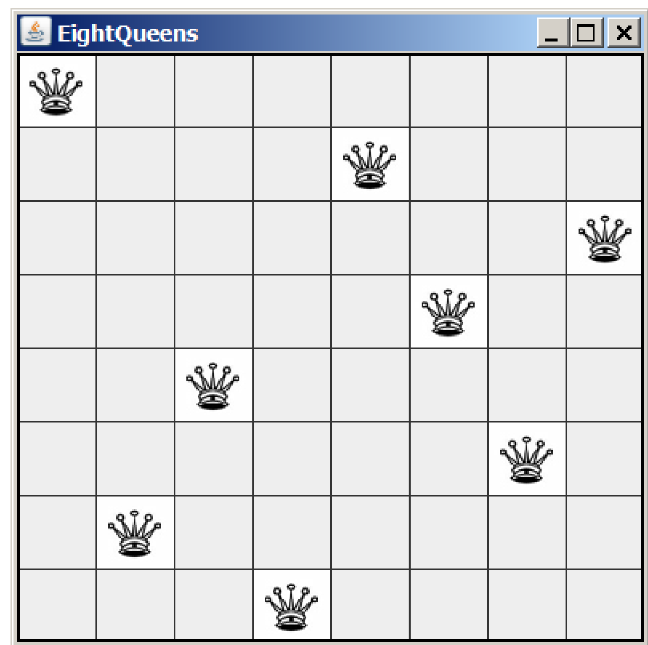
```java
public double distance(
    Point[] pointsOrderedOnX, int low, int high,
    Point[] pointsOrderedOnY) {
  if (low >= high) // Zero or one point in the set
    return Double.MAX_VALUE;
  else if (low + 1 == high) {
    p1 = pointsOrderedOnX[low];
    p2 = pointsOrderedOnX[high];
    return distance(pointsOrderedOnX[low], pointsOrderedOnX[high]);
  }
  int mid = (low + high) / 2;
  Point[] pointsOrderedOnYL = new Point[mid - low + 1];
  Point[] pointsOrderedOnYR = new Point[high - mid];
  int j1 = 0; int j2 = 0;
  for (int i = 0; i < pointsOrderedOnY.length; i++) {
    if (pointsOrderedOnY[i].compareTo(pointsOrderedOnX[mid]) <= 0)
      pointsOrderedOnYL[j1++] = pointsOrderedOnY[i];
    else
      pointsOrderedOnYR[j2++] = pointsOrderedOnY[i];
  }
  // Recursively find the distance of the closest pair in the left
  // half and the right half
  double d1 = distance(pointsOrderedOnX, low, mid, pointsOrderedOnYL);
  double d2 = distance(pointsOrderedOnX, mid + 1, high, pointsOrderedOnYR);
  double d = Math.min(d1, d2);

  // stripL: the points in pointsOrderedOnYL within the strip d
  int count = 0;
  for (int i = 0; i < pointsOrderedOnYL.length; i++)
    if (pointsOrderedOnYL[i].x >= pointsOrderedOnX[mid].x - d)
      count++;
  Point[] stripL = new Point[count];
  count = 0;
  for (int i = 0; i < pointsOrderedOnYL.length; i++)
    if (pointsOrderedOnYL[i].x >= pointsOrderedOnX[mid].x - d)
      stripL[count++] = pointsOrderedOnYL[i];
```

```
    // stripR: the points in pointsOrderedOnYR within the strip d
    count = 0;
    for (int i = 0; i < pointsOrderedOnYR.length; i++)
      if (pointsOrderedOnYR[i].x <= pointsOrderedOnX[mid].x + d)
        count++;
    Point[] stripR = new Point[count];
    count = 0;
    for (int i = 0; i < pointsOrderedOnYR.length; i++)
      if (pointsOrderedOnYR[i].x <= pointsOrderedOnX[mid].x + d)
        stripR[count++] = pointsOrderedOnYR[i];

    // Find the closest pair for a point in stripL and
    // a point in stripR
    double d3 = d;
    int j = 0;
    for (int i = 0; i < stripL.length; i++) {
      while (j < stripR.length && stripL[i].y > stripR[j].y + d)
        j++;
      // Compare a point in stripL with points in stripR
      int k = j; // Start from r1 up in stripR
      while(k < stripR.length && stripR[k].y <= stripL[i].y + d) {
        if (d3 > distance(stripL[i], stripR[k])) {
          d3 = distance(stripL[i], stripR[k]);
          p1 = stripL[i];
          p2 = stripR[k];
        }
        k++;
      }
    }

    return Math.min(d, d3);
  }
}
```

Backtracking: Eight Queens

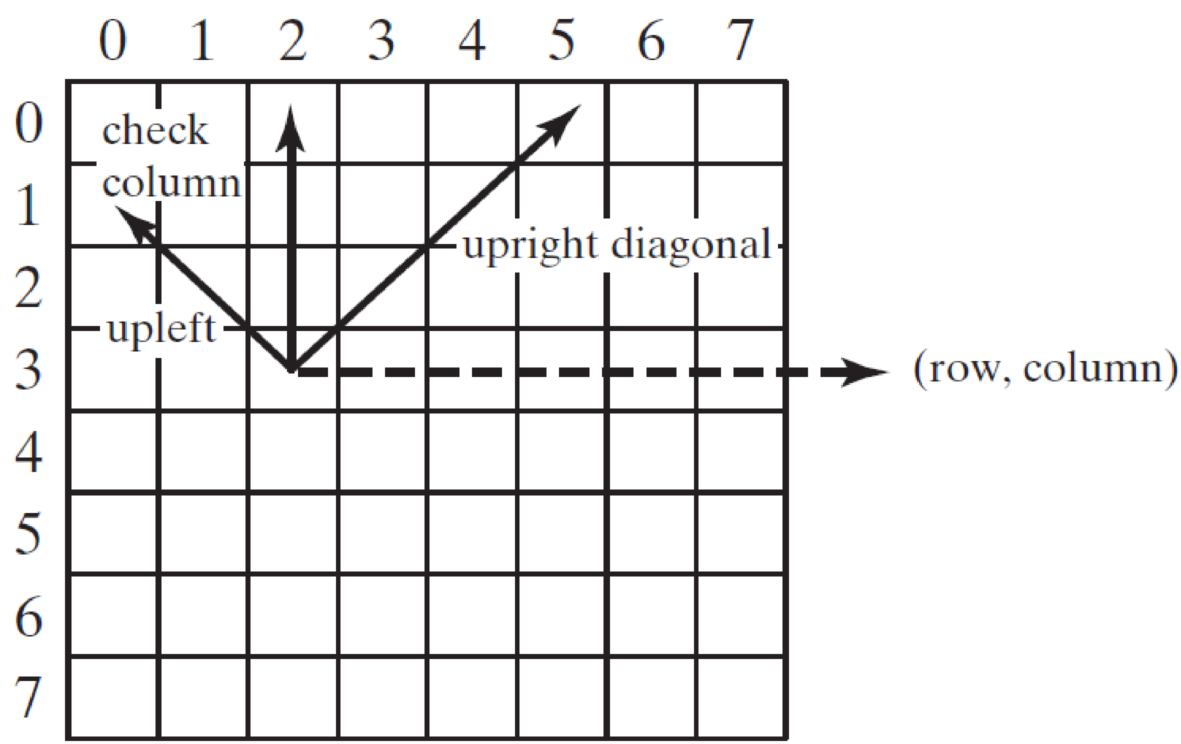
- The Eight Queens problem is to find a solution to place a queen in each row on a chessboard such that no two queens can attack each other.
  - The problem can be solved using <u>**recursion**</u>
  - Assign **`j`** to **`queens[i]`** to denote that a queen is placed in row **`i`** and column **`j`**

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



Eight Queens

- The search starts from the first row with **`k = 0`**, where **`k`** is the index of the current row being considered. The algorithm checks whether a queen can be possibly placed in the **`j`**th column in the row for **`j = 0, 1, ..., 7`**, in this order



Eight Queens

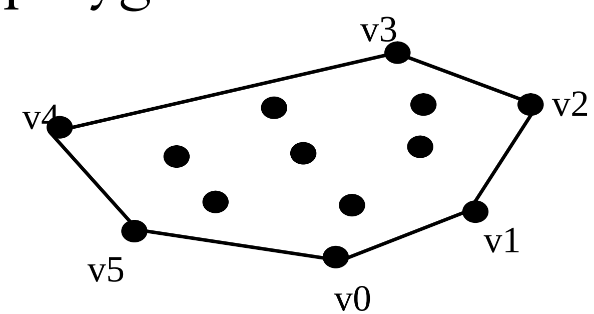
- If successful, it continues to search for a placement for a queen in the next row.
  - If the current row is the last row, a solution is found
- If not successful, it backtracks to the previous row and continues to search for a new placement in the next column in the previous row.
- If the algorithm backtracks to the first row and cannot find a new placement for a queen in this row, no solution can be found.
- This algorithm is called ***backtracking***.

```java
public class NQueens {
   static final int SIZE = 8;
   private int[] queens = new int[SIZE];
   public static void main(String[] args) {
         NQueens nq = new NQueens();
         nq.search();
         print(nq.queens);
   }
   private static void print(int[] queens) {
         System.out.print("[");
         for(int q:queens)
                  System.out.print(q + " ");
         System.out.println("]");
   }
   public NQueens() {
   }

   /** Search for a solution */
   private boolean search() {
     // k - 1 indicates the number of queens placed so far
     // We are looking for a position in the kth row to place a queen
     int k = 0;
     while (k >= 0 && k < SIZE) {
       // Find a position to place a queen in the kth row
       int j = findPosition(k);
       if (j < 0) {
         queens[k] = -1;
         k--; // back track to the previous row
       } else {
         queens[k] = j;
         k++;
       }
     }
     if (k == -1)
       return false; // No solution
     else
       return true; // A solution is found
   }
```

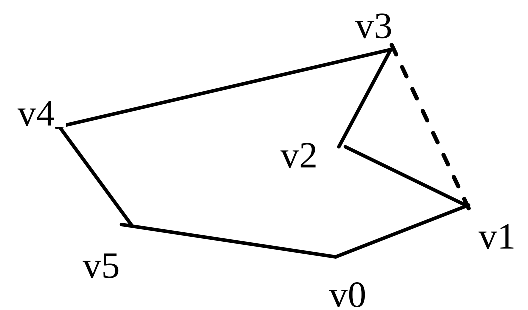
```
  public int findPosition(int k) {
    int start = queens[k] + 1; // Search for a new placement
    for (int j = start; j < SIZE; j++) {
      if (isValid(k, j))
        return j; // (k, j) is the place to put the queen now
    }
    return -1;
  }

  /** Return true if a queen can be placed at (row, column) */
  public boolean isValid(int row, int column) {
    for (int i = 1; i <= row; i++)
      if (queens[row - i] == column // Check column
        || queens[row - i] == column - i // Check upleft diagonal
        || queens[row - i] == column + i) // Check upright diagonal
        return false; // There is a conflict
    return true; // No conflict
  }

}
```

Convex Hull

- Given a set of points, a *convex hull* is a smallest convex polygon that encloses all these points.
- A polygon is *convex* if every line connecting two vertices is inside the polygon.



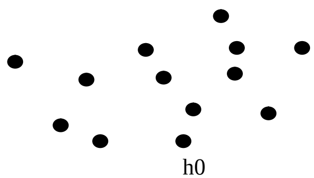
Convex



Not convex

- A convex hull has many applications in pattern recognition, image processing, game programming, etc.

The Gift-Wrapping Algorithm

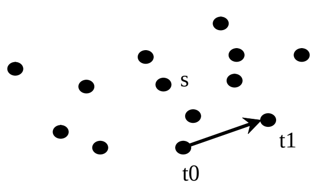


- Set a list $\mathbf{H}$ initially empty. $\mathbf{H}$ will hold all points in the convex hull after the algorithm is finished.

Step 1: Given the set of points $\mathbf{S}$, let the points in $\mathbf{S}$ be labeled $\mathbf{s}_0$, $\mathbf{s}_1$, …, $\mathbf{s}_k$, select the **rightmost** **lowest** point $\mathbf{h}_0$.

- Add $\mathbf{h}_0$ to list $\mathbf{H}$.
- Let $\mathbf{t}_0$ be $\mathbf{h}_0$.

Step 2: Let $\mathbf{t}_1$ be any point not yet seen. For every point $\mathbf{p}$ in $\mathbf{S}$, if $\mathbf{p}$ is on the right side of the direct line from $\mathbf{t}_0$ to $\mathbf{t}_1$, then let $\mathbf{t}_1$ be $\mathbf{p}$

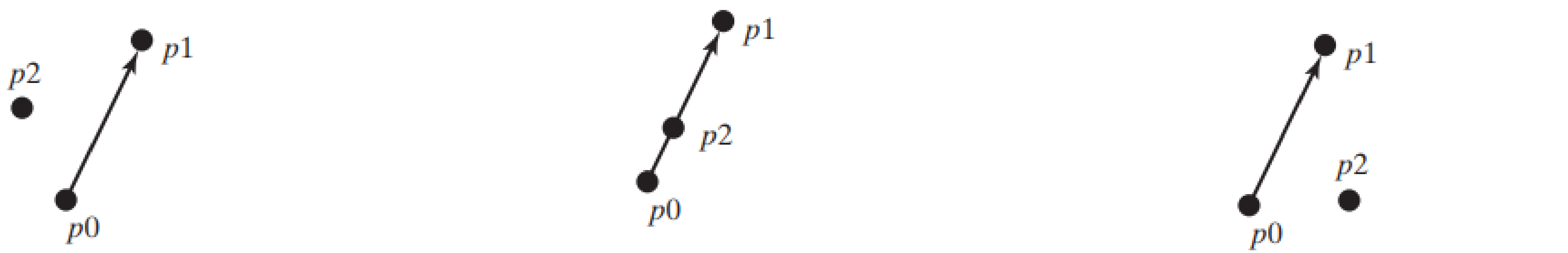


- After Step 2, no points lie on the right side of the direct line from $\mathbf{t}_0$ to $\mathbf{t}_1$

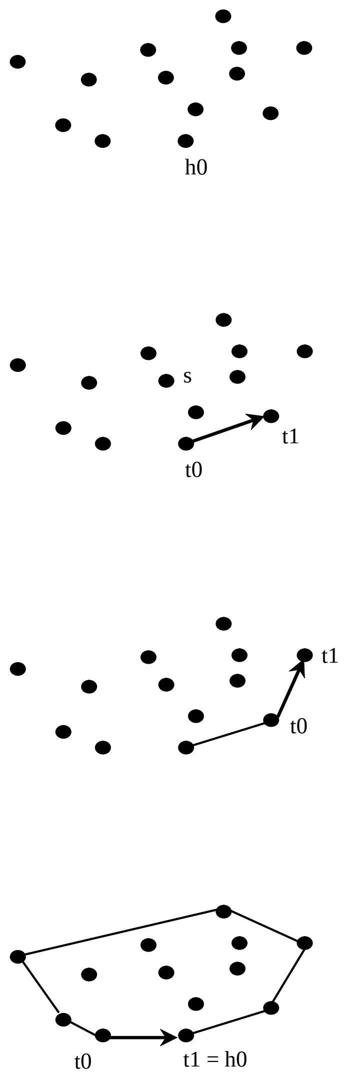
Geometry: point position relative to a directed segment

- Given a directed line from point $\mathbf{p_0(x_0,\ y_0)}$ to $\mathbf{p_1(x_1,\ y_1)}$, you can use the following condition to decide whether a point $\mathbf{p_2(x_2,\ y_2)}$ is on the left of the line, on the right, or on the same line:

$$(x1 - x0)*(y2 - y0) - (x2 - x0)*(y1 - y0) \begin{cases} >0 \text{ p2 is on the left side of the line} \\ =0 \text{ p2 is on the same line} \\ <0 \text{ p2 is on the right side of the line} \end{cases}$$



Gift-Wrapping

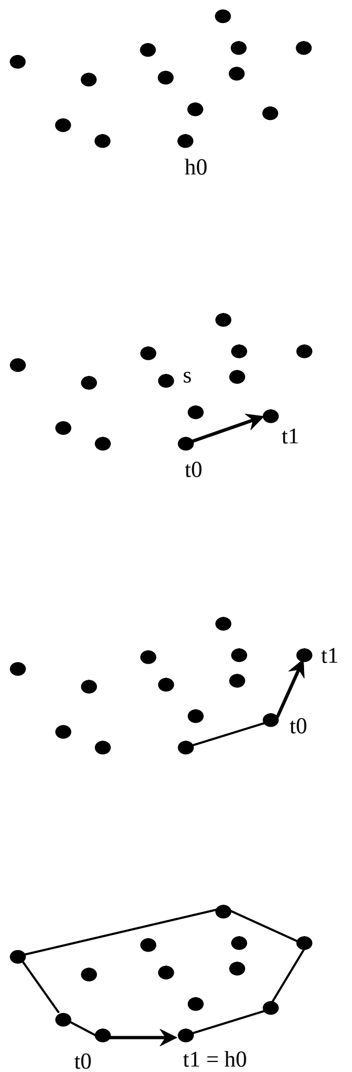


Step 3: If t_1 is h_0, done.

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

- The convex hull is expanded incrementally.
- The correctness is supported by the fact that no points lie on the right side of the direct line from t_0 to t_1 after Step 2.
  - This ensures that every segment with two points in S falls **inside** the polygon
- 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 determined in **O(1)** time

Gift-Wrapping



Step 2 is repeated **`h`** times, where **`h`** is the size of the convex hull.

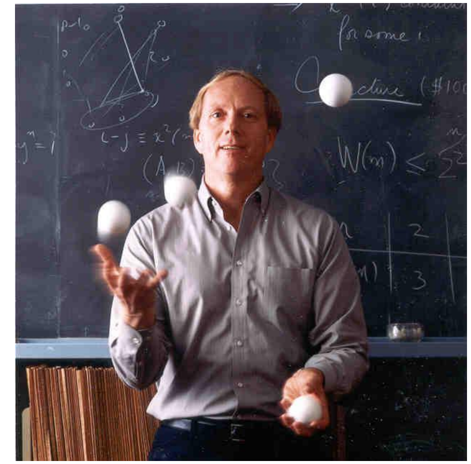
In Step 2, we iterate over every point in **`S`**.

Therefore, the algorithm takes **`O(hn)`** time.

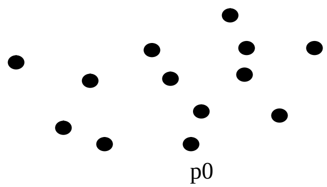
In the worst-case, **`h`** is **`n`**.

So, the complexity of this algorithm is **`O(n^2)`**

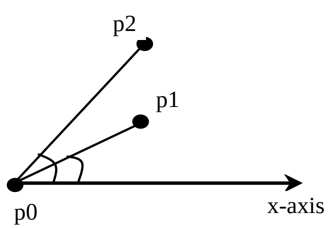
Graham scan



Ronald Graham



<u>Step 1:</u> Select the **<u>rightmost lowest</u>** point and name it $\mathbf{p_0}$ in the set $\mathbf{S}$.

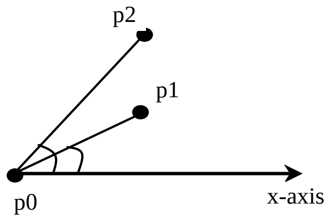


<u>Step 2:</u> Sort the points in $\mathbf{S}$ angularly along the x-axis with $\mathbf{p_0}$ as the center.

If there is a tie and two points have the same angle, discard the one that is closest to $\mathbf{p_0}$.

The points in $\mathbf{S}$ are now sorted as $\mathbf{p_0, p_1, p_2, ..., p_{n-1}}$.

Graham scan



<u>Step 3:</u> Push $\mathbf{p_0}$, $\mathbf{p_1}$, and $\mathbf{p_2}$ into stack **H**.

<u>Step 4:</u>

i = 3;

while (i < n) {

Let $\mathbf{t_1}$ and $\mathbf{t_2}$ be the top first and second element in stack **H**;

if ($\mathbf{p_i}$ is on the left side of the direct line from $\mathbf{t_2}$ to $\mathbf{t_1}$) {

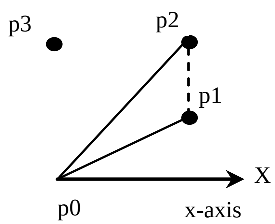
Push $\mathbf{p_i}$ to **H**;

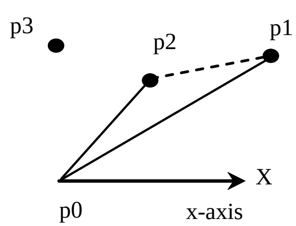
i++; // Consider the next point in **S**.

} else

Pop the top element off stack **H**.

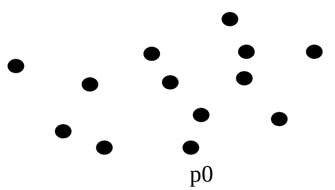
}



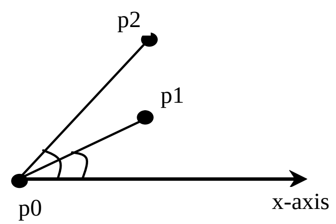


<u>Step 5:</u> The points in **H** form a convex hull.

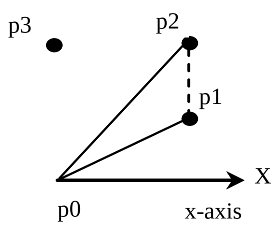
Graham scan Complexity

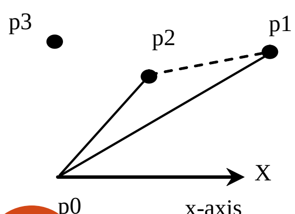


- Finding the rightmost lowest point in Step 1 can be done in ***O*(n)** time.



- The angles can be computed using trigonometry functions, however, we can sort the points/compare the angles without actually computing their angles:
  - Observe that a point **`p2`** would make a greater angle than a point **`p1`** if and only if **`p2`** lies on the left side of the line from **`p0`** to **`p1`**.
- Sorting in Step 2 can be done in ***O*(n log n)**.





- Step 3 is done in ***O*(1).**
- Step 4 is done in ***O*(n)** time.
- Therefore, the algorithm takes ***O*(n logn)** time.

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 (e.g., if an algorithm takes 100*n, while another takes n/2, then the second one should be used).