Objectives

- To estimate algorithm efficiency using the Big O notation.
- To explain growth rates and why constants and non-dominating terms can be ignored in the estimation.
- To determine the complexity of various types of algorithms.
- To analyze the binary search algorithm.
- To analyze the selection sort algorithm.
- To analyze the insertion sort algorithm.
- To analyze the Tower of Hanoi algorithm.
- To describe common growth functions (constant, logarithmic, log-linear, quadratic, cubic, 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

- *Algorithm design* is to **develop** a mathematical process for solving a problem.
- *Algorithm analysis* is to **predict** the performance of an algorithm.
- In previous lectures, we introduced classic data structures (lists, stacks, queues, priority queues, sets, and maps) and applied them to solve problems.
- We will use a variety of examples to introduce common **algorithmic techniques** (dynamic programming, divide-and-conquer, and backtracking) for developing efficient algorithms.
Executing 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 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 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.
Measuring Algorithm Efficiency Using Big O Notation

- 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 inputs.
- This approach approximates the effect of a change on the size of the input: *Growth Rate*
  - In this way, we can see how fast an algorithm’s execution time increases as the input size increases, so we can compare two algorithms by examining their growth rates.
Big O Notation

- Consider linear search for an array of size $n$:
  - 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.
    - 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
  - 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 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.
- 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 $100 \times n$ is the same as $n$, i.e., $O(n) = O(n/2) = O(100n)$

<table>
<thead>
<tr>
<th>$f(n)$</th>
<th>$n$</th>
<th>$n/2$</th>
<th>$100n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>100</td>
<td>50</td>
<td>10000</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The growth rate is $2$ for all cases, i.e., $f(200) / f(100) = 2$. 

(c) Paul Fodor (CS Stony Brook) & Pearson
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

• 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 $O(n)$. 
Useful Mathematic Summations

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

\[1 + 2 + 3 + \ldots + (n - 1) + n = \frac{n(n + 1)}{2}\]

\[a^0 + a^1 + a^2 + a^3 + \ldots + a^{(n-1)} + a^n = \frac{a^{n+1} - 1}{a - 1}\]

\[2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{(n-1)} + 2^n = \frac{2^{n+1} - 1}{2 - 1}\]
Determining Big-O

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

executed \( n \) times

\[
\text{for } (i = 1; i <= n; i++) \{
    k = k + 5;
\}
\]

It is a constant time to execute \( k = k + 5 \)

Time Complexity:

\[
T(n) = (a \text{ constant } c) \times n = cn = O(n)
\]

Ignore multiplicative constants (e.g., “c”).
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 = 1000000 is 6 milliseconds
Execution time for n = 10000000 is 61 milliseconds
Execution time for n = 100000000 is 610 milliseconds
Execution time for n = 1000000000 is 6048 milliseconds

linear time complexity
Repetition: Nested Loops

\[
T(n) = (a \text{ constant } c) \times n \times n = cn^2 = O(n^2)
\]

Ignore multiplicative constants (e.g., “c”).
Repetition: Nested Loops

\[
T(n) = c + 2c + 3c + 4c + \ldots + nc = cn(n+1)/2 = (c/2)n^2 + (c/2)n = O(n^2)
\]

\[
\text{for } (i = 1; i <= n; i++) \\
\quad \text{for } (j = 1; j <= i; j++) \\
\quad \quad k = k + i + j;
\]

executed \(n\) times

inner loop executed \(i\) times

constant time

\text{Ignore non-dominating terms}

\text{Ignore multiplicative constants}
Repetition: Nested Loops

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

Time Complexity

\[ T(n) = 20 \times c \times n = O(n) \]

Ignore multiplicative constants (e.g., \(20 \times c\))
Sequence

\[
T(n) = c \cdot 10 + 20 \cdot c \cdot n = O(n)
\]

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

executed \(n\) times

inner loop executed \(20\) times

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

executed \(10\) times

Time Complexity
Selection

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

Time Complexity

\[ T(n) = \text{test time} + \text{worst-case (if, else)} \]
\[ = O(n) + O(n) \]
\[ = O(n) \]
Logarithmic time

\[
result = 1; \\
O(n) \quad \text{for} \ (\text{int} \ i = 1; \ i \leq n; \ i++) \\
result \ *= a;
\]

Without loss of generality, assume \( n = 2^k \).
We can improve the algorithm using the following scheme:

\[
result = a; \\
\text{for} \ (\text{int} \ i = 1; \ i \leq k; \ i++) \\
result = result \* result;
\]

\[
T(n) = k = \log n = O(\log n)
\]
The algorithm takes \( O(\log n) \) time.
Analyzing Binary Search

- Binary search searches for 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 \( k = \log_2 n \)
- 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

• 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.

\[
T(n) = (n - 1) + c + (n - 2) + c + \ldots + 2 + c + 1 + c
\]
\[
= \frac{(n - 1)(n - 1 + 1)}{2} + c(n - 1) = \frac{n^2}{2} - \frac{n}{2} + cn - c
\]
\[
= O(n^2)
\]
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 Insertion Sort

- The insertion sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted partial array until the whole array is sorted.
- At the kth iteration, to insert an element to a array of size k, it may take k comparisons to find the insertion position, and k moves to insert the element.
- Let $T(n)$ denote the complexity for insertion sort and $c$ denote the total number of other operations such as assignments and additional comparisons in each iteration. So,

$$T(n) = 2 + c + 2 \times 2 + c ... + 2 \times (n - 1) + c = n^2 - n + cn$$

Ignoring constants and smaller terms, the complexity of the insertion sort algorithm is $O(n^2)$. 
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 $B$ with $T(1) = 1$

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

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

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

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

$$= 2^{n-1}T(1) + 2^{n-2} + \ldots + 2 + 1$$

$$= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 = (2^n - 1) = O(2^n)$$
Analyzing Towers of Hanoi

• An algorithm with $O(2^n)$ time complexity is called an exponential algorithm, and it exhibits an exponential growth rate: as the input size increases, the time for the exponential algorithm grows exponentially.

• Exponential algorithms are not practical for large input size.

  • Suppose the disk is moved at a rate of 1 per second. It would take $2^{32}/(365*24*60*60) = 136$ years to move 32 disks and $2^{64}/(365*24*60*60) = 585$ billion years to move 64 disks.
Common Recurrence Relations

- Recurrence relations are a useful tool for analyzing algorithm complexity.

<table>
<thead>
<tr>
<th>Recurrence Relation</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n/2) + O(1)$</td>
<td>$T(n) = O(\log n)$</td>
<td>Binary search, Euclid’s GCD</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(1)$</td>
<td>$T(n) = O(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(1)$</td>
<td>$T(n) = O(n)$</td>
<td></td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n)$</td>
<td>$T(n) = O(n \log n)$</td>
<td>Merge sort (Chapter 24)</td>
</tr>
<tr>
<td>$T(n) = 2T(n/2) + O(n \log n)$</td>
<td>$T(n) = O(n \log^2 n)$</td>
<td></td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n \log n)$</td>
<td>$T(n) = O(n)$</td>
<td></td>
</tr>
<tr>
<td>$T(n) = T(n-1) + O(n^2)$</td>
<td>$T(n) = O(2^n)$</td>
<td>Towers of Hanoi</td>
</tr>
<tr>
<td>$T(n) = T(n-1) + T(n-2) + O(1)$</td>
<td>$T(n) = O(2^n)$</td>
<td>Recursive Fibonacci algorithm</td>
</tr>
</tbody>
</table>
Comparing Common Growth Functions

\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \]

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
<th>( n = 25 )</th>
<th>( n = 50 )</th>
<th>( f(50)/f(25) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>Constant time</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( O(\log n) )</td>
<td>Logarithmic time</td>
<td>4.64</td>
<td>5.64</td>
<td>1.21</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>Linear time</td>
<td>25</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>( O(n \log n) )</td>
<td>Log-linear time</td>
<td>116</td>
<td>282</td>
<td>2.43</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>Quadratic time</td>
<td>625</td>
<td>2,500</td>
<td>4</td>
</tr>
<tr>
<td>( O(n^3) )</td>
<td>Cubic time</td>
<td>15,625</td>
<td>125,000</td>
<td>8</td>
</tr>
<tr>
<td>( O(2^n) )</td>
<td>Exponential time</td>
<td>( 3.36 \times 10^7 )</td>
<td>( 1.27 \times 10^{15} )</td>
<td>( 3.35 \times 10^7 )</td>
</tr>
</tbody>
</table>
Comparing Common Growth Functions

\[ O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) \]
Analysing Fibonacci Numbers

\[
\begin{align*}
\text{fib}(0) &= 0; \\
\text{fib}(1) &= 1; \\
\text{fib(} \text{index} \text{)} &= \text{fib(} \text{index} - 1 \text{)} + \text{fib(} \text{index} - 2 \text{)}; \text{ index } \geq 2 \\
\text{Fibonacci series: } 0 & 1 1 2 3 5 8 13 21 34 55 89... \\
\text{indices: } 0 & 1 2 3 4 5 6 7 8 9 10 11 \\
\end{align*}
\]

/** 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);
}
Complexity for Recursive Fibonacci Numbers

- Let $T(n)$ denote the complexity for the algorithm that finds $\text{fib}(n)$

\[
T(n) = T(n-1) + T(n-2) + c
\]

\[
= T(n-2) + T(n-3) + c + T(n-2) + c
\]

\[
\geq 2T(n-2) + 2c
\]

\[
\geq 2(2T(n-4) + 2c) + 2c
\]

\[
\geq 2^2 T(n-2 - 2) + 2^2 c + 2c
\]

\[
\geq 2^{n/2} T(1) + 2^{n/2} c + \ldots + 2^3 c + 2^2 c + 2c
\]

\[
\leq 2^{n-1} T(1) + 2^{n-2} c + \ldots + 2c + c
\]

\[
= 2^{n-1} T(1) + (2^{n-2} + \ldots + 2 + 1)c
\]

\[
= 2^{n-1} c + (2^{n-2} + \ldots + 2 + 1)c
\]

\[
= O(2^n)
\]

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

This algorithm is not efficient.

- Is there an efficient algorithm for finding a Fibonacci number?
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;
}

- The complexity of this new algorithm is $O(n)$.
- This is a tremendous improvement over the recursive algorithm.
• Variables $f_0$, $f_1$, and $f_2$ store three consecutive Fibonacci numbers in the series:

\[
\begin{array}{llllllllllll}
\text{f0} & \text{f1} & \text{f2} \\
\text{Fibonacci series:} & 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 \\
\text{indices:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11
\end{array}
\]
Dynamic Programming

- The non-recursive algorithm for computing Fibonacci numbers uses an approach known as *dynamic programming*.
- Dynamic programming solves subproblems, then combines 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 **store the results** for subproblems for later use to avoid redundant computing of the subproblems.
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 $O(n)$
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 $O(n)$
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 $\text{gcd}(m, n)$ denote the gcd for integers $m$ and $n$:

● If $m \% n$ is 0, $\text{gcd}(m, n)$ is $n$.

● Otherwise, $\text{gcd}(m, n)$ is $\text{gcd}(n, m \% n)$.

$$m = n*k + r$$

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

```java
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 $O(\log n)$. 
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 \( m \geq n \), we can show that \( m \% n < m / 2 \), as follows:
      - If \( n \leq m / 2 \), \( m \% n < m / 2 \), since the remainder of \( m \) divided by \( n \) is always less than \( n \).
      - If \( n > m / 2 \), \( m \% n = m - n < m / 2 \). Therefore, \( m \% n < m / 2 \).
    - Euclid’s algorithm recursively invokes the \( \text{gcd} \) method: it first calls \( \text{gcd}(m, n) \), then calls \( \text{gcd}(n, m \% n) \), and \( \text{gcd}(m \% n, n \% (m \% n)) \), and so on.
    - Since \( m \% n < m / 2 \) and \( n \% (m \% n) < n / 2 \), the argument passed to the \( \text{gcd} \) method is reduced by half after every two iterations.
GCD algorithms

- Time Complexity

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>Brute-force, checking all possible divisors</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Checking half of all possible divisors</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Euclid’s algorithm</td>
</tr>
</tbody>
</table>
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
  

- Compare three versions
  - Brute-force
  - Check possible divisors up to $\text{Math.sqrt(n)}$
  - Check possible prime divisors up to $\text{Math.sqrt(n)}$
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);
Finding Prime Numbers

- Brute force algorithm:
  - The program is not efficient if you have to compute `Math.sqrt(number)` for every iteration of the for loop.
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

\[ T(n) = 2T(n/2) + O(n) = O(n \log n) \]
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 |
Eight Queens

(check column)

(upright diagonal)

(uleft)

(row, column)
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.

\[\text{Figure a}\] \hspace{2cm} \text{Figure b}\]
Gift-Wrapping

Step 1: Given a set of points $S$, let the points in $S$ be labeled $s_0, s_1, \ldots, s_k$. 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.
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 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 $p_0$, $p_1$, $p_2$, ..., $p_{n-1}$.

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 $p_2$, 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.

$O(n \log n)$
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 have the same complexity, but the second is obviously better practically.