Developing Efficient Algorithms

CSE260, Computer Science B: Honors
Stony Brook University

http://www.cs.stonybrook.edu/~cse260
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 useful to predict the performance of an algorithm.
  - In the 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.
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 execution time

• 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:
      o 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 these problems, a theoretical approach was developed to analyze algorithms independent of computers (system load) 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
Consider linear search for an array of size $n$:

- The linear search algorithm compares the key with the elements in the array \textit{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 "on average" $n/2$ comparisons.
- The algorithm's execution time is \textit{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 \textit{grows} at a \textit{linear} rate.
  - The growth rate has \textit{an order of magnitude growth rate} of $n$.
  - Computer scientists use the \textit{Big O} notation to abbreviate for “\textit{order of magnitude}”.
- Using this notation, the complexity of the linear search algorithm is $O(n)$, pronounced as “\textit{order of n}”.

\[\text{Big O Notation}\]
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

- 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></th>
<th>$f(n)$</th>
<th>$n$</th>
<th>$n/2$</th>
<th>$100n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n$</td>
<td>$n/2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>50</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>100</td>
<td>20000</td>
<td></td>
</tr>
</tbody>
</table>

- The linear search algorithm requires $n$ comparisons in the worst-case and $n/2$ comparisons in the average-case
  - Using the growth rate 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:

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

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

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

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

Time Complexity:

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

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

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

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

executed \( n \) times
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

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

Time Complexity

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

constant time

inner loop executed \(n\) times

executed \(n\) times

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

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

executed \( n \) times

\text{inner loop executed } i \text{ times}

\text{Time Complexity}

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

\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;
    }
}

constant time

executed $n$ times

inner loop executed 20 times

Time Complexity

$T(n) = 20 \times c \times n = O(n)$

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

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

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

\[\text{executed}\ 10\ \text{times}\]

\[\text{inner loop}\ \text{executed}\ 20\ \text{times}\]

\[\text{executed}\ n\ \text{times}\]

Time Complexity

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

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

Let n be
list.size()
Executed
n times.
O(n)

Time Complexity

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

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

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

result = a;
for (int i = 1; i <= 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 \( k \)th iteration, to insert an element to a array of size \( k \), it may take \( k \) comparisons to find the insertion position, or \( k \) 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,
    \[
    T(n) = 2 + c + 2 \times 2 + c \ldots + 2 \times (n-1) + c = n^2 - n + cn
    \]
    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</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) )</td>
<td>( T(n) = O(n^2) )</td>
<td>Selection sort, insertion sort</td>
</tr>
<tr>
<td>( T(n) = 2T(n-1) + O(1) )</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>( \frac{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

\[
\begin{align*}
\text{fib}(0) &= 0; \\
\text{fib}(1) &= 1; \\
\text{fib}(\text{index}) &= \text{fib}(\text{index} - 1) + \text{fib}(\text{index} - 2); \quad \text{index} \geq 2
\end{align*}
\]

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 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 \text{fib}(\text{index} - 1) + \text{fib}(\text{index} - 2);
Complexity for Recursive Fibonacci Numbers

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

\[
T(n) = T(n-1) + T(n-2) + c
\]
\[
= T(n-2) + T(n-3) + c + T(n-2) + c
\]
\[
\geq 2T(n-2) + 2c
\]
\[
\geq 2(2T(n-4) + 2c) + 2c
\]
\[
\geq 2^2T(n-2-2) + 2^2c + 2c
\]
\[
\geq 2^3T(n-2-2-2) + 2^3c + 2^2c + 2c
\]
\[
\geq 2^{n/2}T(1) + 2^{n/2}c + \ldots + 2^3c + 2^2c + 2c
\]
\[
= 2^{n/2}c + 2^{n/2}c + \ldots + 2^3c + 2^2c + 2c
\]
\[
= O(2^n)
\]

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

```java
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:

```
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
```

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

- The 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 $O(n)$
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;
}
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 \( \gcd(n, m \% n) \).

\[
m = n \times k + r
\]

If \( p \) is a divisor of both \( m \) and \( n \), it must be divisor of \( r \):

\[
m/p = n/p \times k + r/p
\]

\( \epsilon \mathbb{Z} \quad \epsilon \mathbb{Z} \quad \rightarrow \quad \epsilon \mathbb{Z} \)
public static int gcd(int m, int n) {
    if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}

What is the time complexity of this algorithm?
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 \( n \leq m / 2 \), 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 \( \text{gcd}(m, n) \), 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 gcd method is reduced by half after every two iterations.
## GCD algorithms

- **Time Complexity**

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


- We will compare three versions of an algorithm to find all the prime numbers less than some number $n$:
  - Brute-force 😞
  - Check possible divisors up to `Math.sqrt(number)`
  - Check *possible* prime divisors up to `Math.sqrt(number)`
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);
Brute-force Finding Prime Numbers

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

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

```
In fact, there is no need to actually compute \texttt{Math.sqrt(number)} for every \texttt{number}.

- For all the numbers between 36 and 48, inclusively, their \texttt{(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.

```java
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 $\sqrt{2}+\sqrt{3}+\sqrt{4}+...+\sqrt{n}$ steps to find all the prime numbers less than or equal to $n$.
- $\sqrt{2}+\sqrt{3}+\sqrt{4}+...+\sqrt{n} \leq n\sqrt{n}$
- Therefore, the time complexity for this algorithm is $O(n\sqrt{n})$
Finding Prime Numbers

• We can prove that if \( i \) is not prime, there must exist a prime number \( p \) such that \( i = pq \) and \( p \leq q \).
• Let \( \pi(i) \) denote the number of prime numbers less than or equal to \( i \).
  • \( \pi(2) \) is 1, \( \pi(3) \) is 2, \( \pi(6) \) is 3, and \( \pi(20) \) is 8
• It has been proved that \( \pi(i) \) is approximately \( i / \log i \)
  • [http://primes.utm.edu/howmany.html](http://primes.utm.edu/howmany.html)
• For each number \( i \), the algorithm checks whether a prime number less than or equal to \( \sqrt{i} \) is divisible by \( i \).
  • The number of the prime numbers less than or equal to \( \sqrt{i} \) is
    \[
    \frac{\sqrt{i}}{\log \sqrt{i}} = \frac{2\sqrt{i}}{\log i}
    \]
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 $\leq n$
  - use an array named `primes` of $n$ Boolean
    - initially all values are `true`
    - the multiples of 2 are not prime, set $\text{primes}[2*i]$ to `false` for all $2 \leq i \leq n/2$
    - Since the multiples of 3 are not prime, set $\text{primes}[3*i]$ to `false` for all $3 \leq i \leq n/3$
Finding Prime Numbers

```java
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 $\text{primes}[k*i]$ to false
  - This is performed $n/k - k + 1$ times in the for loop
    
    $$\frac{n}{2} - 2 + 1 + \frac{n}{3} - 3 + 1 + \frac{n}{5} - 5 + 1 + \frac{n}{7} - 7 + 1 + \frac{n}{11} - 11 + 1 \ldots$$
    
    $$= O\left(\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} + \ldots\right) < O(n\pi(n))$$
    
    $$= O\left(\frac{\sqrt{n}}{\log n}\right)$$

  - 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 $\text{primes}$ can fit in the memory.
## Comparisons of Prime-Number Algorithms

<table>
<thead>
<tr>
<th>Description</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-force, checking all possible divisors</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Checking divisors up to $\sqrt{n}$</td>
<td>$O(n\sqrt{n})$</td>
</tr>
<tr>
<td>Checking prime divisors up to $\sqrt{n}$</td>
<td>$O\left(\frac{n\sqrt{n}}{\log n}\right)$</td>
</tr>
<tr>
<td>Sieve of Eratosthenes</td>
<td>$O\left(\frac{n\sqrt{n}}{\log n}\right)$</td>
</tr>
</tbody>
</table>
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.
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, $S_1$ and $S_2$, of equal size using the midpoint in the sorted list.
  - Let the midpoint be in $S_1$.
  - Recursively find the closest pair in $S_1$ and $S_2$.
  - Let $d_1$ and $d_2$ denote the distance of the closest pairs in the two subsets, respectively.
- Step 3: Find the closest pair between a point in $S_1$ and a point in $S_2$ and denote their distance as $d_3$
- The closest pair is the one with the distance $\min(d_1, d_2, d_3)$
Finding the Closest Pair of Points Using Divide-and-Conquer

- Step 1: can be done in $O(n \log n)$
- Step 3: can be done in $O(n)$
- Let $d = \min(d_1, d_2)$
- For a point in $S_1$ and a point in $S_2$ to form the closest pair in $S$, the left point must be in $\text{stripL}$ and the right point in $\text{stripR}$
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

\[ T(n) = 2T(n/2) + O(n) = O(n \log n) \]
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);
    }
}

(c) Paul Fodor (CS Stony Brook) & Pearson
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;
    }
}
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 six 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);
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 recursion
- Assign \( j \) to \( \text{queens}[i] \) to denote that a queen is placed in row \( i \) and column \( j \)
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 $j = 0, 1, \ldots, 7$, 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.
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
    }
}

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

- A convex hull has many applications in pattern recognition, image processing, game programming, etc.
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 point $h_0$.
- Add $h_0$ to list $H$. $H$ is initially empty. $H$ will hold all points in the convex hull after the algorithm is finished.
- Let $t_0$ be $h_0$.

Step 2: Let $t_1$ be any point not yet seen. For every point $p$ in $S$, if $p$ is on the right side (by computing the angle $t_1-t_0-p$) of the direct line from $t_0$ to $t_1$, then let $t_1$ be $p$.

After Step 2, no points lie on the right side of the direct line from $t_0$ to $t_1$. 

(c) Paul Fodor (CS Stony Brook) & Pearson
Geometry: point position

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

\[
(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \begin{cases} 
> 0 & \text{p}_2 \text{ is on the left side of the line} \\
= 0 & \text{p}_2 \text{ is on the same line} \\
< 0 & \text{p}_2 \text{ is on the right side of the line} 
\end{cases}
\]
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 line 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.

Therefore, the algorithm takes $O(hn)$ time.

In the worst-case, $h$ is $n$. 
Graham’s Algorithm

Step 1: Given a set of points $S$, select the rightmost lowest point and name it $p_0$ in the set $S$.

Step 2: 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 $p_0, p_1, p_2, \ldots, p_{n-1}$.
**Graham’s Algorithm**

**Step 3:** Push $p_0$, $p_1$, and $p_2$ into stack $H$.

**Step 4:**

$i = 3$;

while $(i < n)$ {

Let $t_1$ and $t_2$ be the top first and second element in stack $H$;

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

Push $p_i$ to $H$;

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

} else

Pop the top element off stack $H$.

}

**Step 5:** The points in $H$ form a convex hull.
Graham’s Algorithm

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 $H$.

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 $H$. 
Graham’s Algorithm

- Finding the rightmost lowest point in Step 1 can be done in $O(n)$ time.
- The angles can be computed using trigonometry functions.
- However, you can sort the points without actually computing their angles.
  - Observe that $p_2$ would make a greater angle than $p_1$ if and only if $p_2$ lies on the left side of the line from $p_0$ to $p_1$.
- Whether a point is on the left side of a line can be determined in $O(1)$ time.
- Sorting in Step 2 can be done in $O(n \log n)$.
- Step 4 can be done in $O(n)$ time.
- Therefore, the algorithm takes $O(n \log n)$ 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.
- As shown in the preceding example, both algorithms have the same complexity, but the second is obviously better practically.