Analysis of Algorithms
Analysis of Algorithms- Review

• **Efficiency** of an algorithm can be measured in terms of :
  • *Time complexity*: a measure of the amount of time required to execute an algorithm
  • *Space complexity*: amount of memory required

• Which measure is more important?
  • It often depends on the limitations of the technology available at time of analysis (e.g. processor speed vs memory space)
Time Complexity Analysis

- **Objectives** of time complexity analysis:
  - To determine the efficiency of an algorithm by computing an *upper bound* on the amount of work that it performs
  - To compare different algorithms before deciding which one to implement

- Time complexity analysis for an algorithm is *independent* of the programming language and the machine used
Time Complexity Analysis

• Time complexity expresses the relationship between
  • the *size of the input*
  • and the *execution time* for the algorithm
Time Complexity Measurement

• Based on the number of *basic or primitive operations* in an algorithm:
  • Number of arithmetic operations performed
  • Number of comparisons
  • Number of Boolean operations performed
  • Number of array elements accessed
  • etc.

• Think of this as the *work* done
**Example: Polynomial Evaluation**

Consider the polynomial

\[ P(x) = 4x^4 + 7x^3 - 2x^2 + 3x^1 + 6 \]

Suppose that exponentiation is carried out using multiplications. Two ways to evaluate this polynomial are:

**Brute force method:**

\[ P(x) = 4\cdot x\cdot x\cdot x\cdot x + 7\cdot x\cdot x\cdot x - 2\cdot x\cdot x + 3\cdot x + 6 \]

**Horner’s method:**

\[ P(x) = (((4\cdot x + 7) \cdot x - 2) \cdot x + 3) \cdot x + 6 \]
Method of analysis

• What are the *basic operations* here?
  • multiplication, addition, and subtraction

• We will look at the *worst case* (maximum number of operations) to get an *upper bound* on the work and thus of the running time of the algorithm
General form of a polynomial of degree \( n \) is

\[
P(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0
\]

where \( a_n \) is non-zero for all \( n \geq 0 \) (this is the worst case)
An analysis of Brute Force Method

\[ P(x) = a_n \times x \times x \times \ldots \times x \times x + \]
\[ a_{n-1} \times x \times x \times \ldots \times x \times x + \]
\[ a_{n-2} \times x \times x \times \ldots \times x \times x + \]
\[ \vdots + \]
\[ a_2 \times x \times x + \]
\[ a_1 \times x + \]
\[ a_0 \]

- \( n \) multiplications
- \( n-1 \) multiplications
- \( n-2 \) multiplications
- \( \vdots \)
- \( 2 \) multiplications
- \( 1 \) multiplication
- \( n \) total additions
Number of operations needed in the **worst case** is

\[ T(n) = n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 + n \]

\[ = n \frac{(n + 1)}{2} + n \text{ (see below)} \]

\[ = \frac{n^2}{2} + \frac{3n}{2} \]

**Sum of first n natural numbers:**

Write the n terms of the sum in forward and reverse orders:

\[ t(n) = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n \]

\[ t(n) = n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 \]

Add the corresponding terms:

\[ 2*t(n) = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1) \]

\[ = n (n+1) \]

Therefore, \[ t(n) = n \frac{(n+1)}{2} \]
Analysis of Horner’s Method

\[ P(x) = ( \ldots ((( a_n \ast x + a_{n-1}) \ast x + a_{n-2}) \ast x + \ldots + a_2) \ast x + a_1) \ast x + a_0 \] 

1 multiplication
1 multiplication
1 multiplication
1 multiplication
1 multiplication

\( n \) total additions
Analysis of Horner’s Method

Number of operations needed in the worst case is:

\[ T(n) = n + n = 2n \]
Big-Oh Notation

• Analysis of Brute Force and Horner’s methods came up with \textit{exact formulae} for the maximum number of operations.

• In general, though, we want to determine the \textit{running time}, not the number of operations: Thus, we use the Big-Oh notation introduced earlier …
Big-Oh : Formal Definition

- **Time complexity** $T(n)$ of an algorithm is $O(f(n))$ (we say “of the order $f(n)$”) if for some positive constant $c$ and for all but finitely many values of $n$ (i.e. as $n$ gets large)

  $T(n) \leq c \cdot f(n)$

- What does this mean? this gives an **upper bound** on the number of operations, for sufficiently large $n$
Big-Oh Analysis

• We want the complexity function $f(n)$ to be an easily recognized *elementary function* that describes the performance of the algorithm
Big-Oh Analysis

Example: Polynomial Evaluation

• What is the time complexity $f(n)$ for Horner’s method?

  • $T(n) = 2n$, so we say that the number of multiplications in Horner’s method is $O(n)$ ("of the order of $n$") and that the time complexity of Horner’s method is $O(n)$.
Big-O Analysis

Example: Polynomial Evaluation

• What is the complexity $f(n)$ for the Brute Force method?

  • Choose the highest order (dominant) term of

    $T(n) = \frac{n^2}{2} + \frac{3n}{2}$

    so

    $T(n)$ is $O(n^2)$
Recall: Shape of Some Typical Functions

\[ t(n) = n^3 \]
\[ t(n) = n^2 \]
\[ t(n) = n \log_2 n \]
Big-Oh Example: Polynomial Evaluation Comparison

<table>
<thead>
<tr>
<th>n</th>
<th>$T(n) = 2n$ (Horner)</th>
<th>$T(n) = \frac{n^2}{2} + \frac{3n}{2}$ (Brute Force)</th>
<th>$f(n) = n^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>65</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td>230</td>
<td>400</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>5150</td>
<td>10000</td>
</tr>
<tr>
<td>1000</td>
<td>2000</td>
<td>501500</td>
<td>1000000</td>
</tr>
</tbody>
</table>

$n$ is the degree of the polynomial.
Big-Oh Example: Polynomial Evaluation

\[ f(n) = n^2 \]

\[ T(n) = n^2/2 + 3n/2 \]

\[ T(n) = n \]

# of op’s

\( n \) (degree of polynomial)
Time Complexity and Input

- Running time can depend on the size of the input (e.g. sorting 5 items vs. 1000 items)
- Running time can also depend on the particular input (e.g. suppose the input is already sorted)
- This leads to several kinds of time complexity analysis:
  - **Worst case** analysis
  - **Average case** analysis
  - **Best case** analysis
Worst, Average, Best Case

- **Worst case analysis**: considers the maximum of the time over all inputs of size $n$
  - Used to find an upper bound on algorithm performance

- **Average case analysis**: considers the average of the time over all inputs of size $n$
  - Determines the average (or expected) performance

- **Best case analysis**: considers the minimum of the time over all inputs of size $n$
Discussion

• What are some difficulties with average case analysis?
  • Hard to determine
  • Depends on distribution of inputs (they might not be evenly distributed)

• So, we usually use worst case analysis (why not best case analysis?)
Example: Linear Search

- **The problem**: search an array $A$ of size $n$ to determine whether it contains some value $key$
- Return *array index* if found, -1 if not found

**Algorithm** linearSearch ($A$, $n$, $key$)

*In*: Array $A$ of size $n$ and value $key$

*Out*: Array index of $key$, if $key$ in $A$; -1 if $key$ not in $A$

```
    k = 0
    while (k < n-1) and (A[k] != key) do
        k = k + 1
    if A[k] = key then return k
    else return -1.
```
• Total amount of work done:
  • **Before loop**: a constant number $c_1$ of operations
  • **Each time through loop**: a constant number $c_2$ of operations (comparisons, the **and** operation, addition, and assignment)
  • **After loop**: a constant number $c_3$ of operations
  • **Worst case**: need to examine all $n$ array locations, so the **while** loop iterates $n$ times
  • So, $T(n) = c_1 + c_2n + c_3$, and the time complexity is $O(n)$
• **Average** case for a **successful** search:
  • Number of **while** loop iterations needed to find the key? 1 or 2 or 3 or 4 … or n
  • Assume that each possibility is equally likely
  • Average number of iterations performed by the **while** loop:
    \[
    \frac{1+2+3+ \ldots + n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}
    \]
  • Average number of operations performed in the average case is \(c_1 + c_3 + c_2(n+1)/2\). The time complexity is therefore \(O(n)\)
Example: Binary Search

- Search a *sorted* array $A$ of size $n$ looking for the value $key$

- *Divide and conquer* approach:
  - Compute the middle index $mid$ of the array
  - If $key$ is found at $mid$, we are done
  - Otherwise repeat the approach on the half of the array that might still contain $key$
Binary Search Algorithm

**Algorithm** binarySearch (A,n,key)

**In:** Array A of size n and value key

**Out:** Array index of key, if key in A; -1 otherwise

```plaintext
first = 0
last = n-1
do {
    mid = (first + last) / 2
    if key < A[mid] then last = mid - 1
    else first = mid + 1
} while (A[mid] != key) and (first <= last)

if A[mid] = key then return mid
else return -1
```
• Number of operations performed before and after the loop is a constant $c_1$, and is independent of $n$

• Number of operations performed during a single execution of the loop is constant, $c_2$

• Time complexity depends on the number of times the loop is executed, so that is what we will analyze
**Worst case**: key is not found in the array

- Each time through the loop, at least half of the remaining locations are rejected:
  - After *first* time through, $\leq n/2$ remain
  - After *second* time through, $\leq n/4$ remain
  - After *third* time through, $\leq n/8$ remain
  - After *$k^{th}$* time through, $\leq n/2^k$ remain
• Suppose in the \textit{worst case} that the maximum number of times through the loop is $k$; we must express $k$ in terms of $n$

• Exit the \texttt{do..while} loop when the number of remaining possible locations is less than 1 (that is, when $\texttt{first > last}$): this means that $n/2^k < 1$ and so $n > 2^k$.

Taking base-2 logarithms we get, $k < \log_2 n$.

Therefore, the total number of operations performed by the algorithm is at most $c_1 + c_2 \log_2 n$ and so the time complexity is $O(\log_2 n)$ or just $O(\log n)$. 
Big-Oh Analysis in General

• To determine the time complexity of an algorithm:
  • Identify the basic operation(s)
  • Carefully analyze the most expensive parts of the algorithm: loops and calls
  • Express the number of operations as $f_1(n) + f_2(n) + \ldots$
  • Identify the dominant term $f_i$
  • Then the time complexity is $O(f_i)$
• **Examples of dominant terms:**
  - $n$ dominates $\log_2(n)$
  - $n \log_2(n)$ dominates $n$
  - $n^2$ dominates $n \log_2(n)$
  - $n^m$ dominates $n^k$ when $m > k$
  - $a^n$ dominates $n^m$ for any $a > 1$ and $m \geq 0$

• That is, for sufficiently large $n$,

\[
\log_2(n) < n < n \log_2(n) < n^2 < \ldots < n^m < a^n
\]

for $a > 1$ and $m > 2$
Recall: Shape of Some Typical Functions

- $f(n) = n^3$
- $f(n) = n^2$
- $f(n) = n \log_2 n$
- $f(n) = n$
Examples of Big-Oh Analysis

• *Independent nested loops:*
  ```java
  int x = 0;
  for (int i = 1; i <= n/2; i++){
    for (int j = 1; j <= n*n; j++){
      x = x + i + j;
    }
  }
  ```

  • Number of iterations of inner loop is independent of the number of iterations of the outer loop (*i.e.* the value of *i*)
  • How many times through outer loop?
  • How many times through inner loop?
  • Time complexity of algorithm?
• **Dependent nested loops:**

```java
int x = 0;
for (int i = 1; i <= n; i++){
    for (int j = 1; j <= 3*i; j++){
        x = x + j;
    }
}
```

• Number of iterations of inner loop **depends on** the value of `i` in the outer loop

• On `ith` iteration of outer loop, how many times through inner loop?

• Total number of iterations of inner loop = sum for `i` running from `1` to `n`

• Time complexity of algorithm?
Usefulness of Big-Oh

• We can *compare algorithms* for efficiency, for example:
  • *Linear search* vs *binary search*
  • Different sort algorithms
  • Iterative vs recursive solutions
    (recall Fibonacci sequence!)

• We can *estimate actual run times* if we know the time complexity of the algorithm(s) we are analyzing
Estimating Run Times

- Assuming a million operations per second on a computer, here are some typical complexity functions and their associated runtimes:

<table>
<thead>
<tr>
<th>f(n)</th>
<th>n = 10^3</th>
<th>n = 10^5</th>
<th>n = 10^6</th>
</tr>
</thead>
<tbody>
<tr>
<td>\log_2(n)</td>
<td>10^{-5} sec.</td>
<td>1.7*10^{-5} sec.</td>
<td>2*10^{-5} sec.</td>
</tr>
<tr>
<td>n</td>
<td>10^{-3} sec.</td>
<td>0.1 sec.</td>
<td>1 sec.</td>
</tr>
<tr>
<td>n \log_2(n)</td>
<td>0.01 sec.</td>
<td>1.7 sec.</td>
<td>20 sec.</td>
</tr>
<tr>
<td>n^2</td>
<td>1 sec.</td>
<td>3 hours</td>
<td>12 days</td>
</tr>
<tr>
<td>n^3</td>
<td>17 mins.</td>
<td>32 years</td>
<td>317 centuries</td>
</tr>
<tr>
<td>2^n</td>
<td>10^{285} cent.</td>
<td>10^{10000} years</td>
<td>10^{100000} years</td>
</tr>
</tbody>
</table>
Discussion

• Suppose we want to perform a sort that is $O(n^2)$. What happens if the number of items to be sorted is 100000?

• Compare this to a sort that is $O(n \log_2(n))$. Now what can we expect?

• Is an $O(n^3)$ algorithm practical for large $n$?

• What about an $O(2^n)$ algorithm, even for small $n$? e.g. for a Pentium, runtimes are:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>11 sec.</td>
</tr>
<tr>
<td>40</td>
<td>3 hours</td>
</tr>
<tr>
<td>50</td>
<td>130 days</td>
</tr>
<tr>
<td>60</td>
<td>365 years</td>
</tr>
</tbody>
</table>
Intractable Problems

• A problem is said to be *intractable* if solving it by computer is impractical
• Algorithms with time complexity $O(2^n)$ take too long to solve even for moderate values of $n$
  • What are some examples we have seen?