Topic 17

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 feasibility of an algorithm by estimating an *upper bound* on the amount of work performed
  • 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
• Usually expressed as a *proportionality*, rather than an exact function
Time Complexity Measurement

• Essentially based on the number of \textit{basic operations} in an algorithm:
  • Number of arithmetic operations performed
  • Number of comparisons
  • Number of times through a critical loop
  • Number of array elements accessed
  • etc.
• Think of this as the \textit{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:**

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

**Horner’s method:**

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

• What are the *basic operations* here?
  • multiplication, addition, and subtraction
    • We’ll only consider the number of multiplications, since the number of additions and subtractions are the same in each solution
  • We’ll examine the *general form* of a polynomial of degree $n$, and express our result *in terms of* $n$
  • We’ll look at the *worst case* (maximum number of multiplications) to get an *upper bound* on the work
Method of analysis

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)
Analysis of Brute Force Method

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

\( n \) multiplications
\( n-1 \) multiplications
\( n-2 \) multiplications
\( \ldots \)
\( 2 \) multiplications
\( 1 \) multiplication
Number of multiplications needed in the **worst case** is

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

\[ = n \left( n + 1 \right) / 2 \]  \textit{(see below)}

\[ = n^2 / 2 + n / 2 \]

**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:

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

\[ = n (n+1) \]

Therefore, \( T(n) = n (n+1) / 2 \)
Analysis of Horner’s Method

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

- 1 multiplication
- 1 multiplication
- 1 multiplication
- \(n\) times
- 1 multiplication
- 1 multiplication
Analysis of Horner’s Method

Number of multiplications needed in the \textit{worst case} is:

\[ T(n) = n \]
Big-O Notation

• Analysis of Brute Force and Horner’s methods came up with *exact formulae* for the maximum number of multiplications

• In general, though, we want *upper bounds* rather than exact formulae: we will use the Big-O notation introduced earlier …
Big-O : 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-O Analysis

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

Example: Polynomial Evaluation

• What is $f(n)$ for Horner’s method?

  • $T(n) = n$, so choose $f(n) = n$

• 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 $f(n)$ for the Brute Force method?
  
  • Choose the highest order (dominant) term of
    
    $T(n) = \frac{n^2}{2} + \frac{n}{2}$

    with a coefficient of 1, so that
    
    $f(n) = n^2$
Discussion

• Why did we use the dominant term?
  • It determines the basic *shape* of the function

• Why did we use a coefficient of 1?
  • For large $n$, $n^2/2$ is essentially $n^2$
Recall: Shape of Some Typical Functions

- $t(n) = n^3$
- $t(n) = n^2$
- $t(n) = n\log_2 n$
- $t(n) = n$
Big-O Analysis
Example: Polynomial Evaluation

• Is \( f(n) = n^2 \) a good choice? i.e. for large \( n \), how does \( T(n) \) compare to \( f(n) \)?
  • \( T(n)/ f(n) \) approaches \( 1/2 \) for large \( n \)
  • So, \( T(n) \) is approximately \( n^2/2 \) for large \( n \)
• \( n^2/2 \leq T(n) \leq n^2 \), so \( f(n) \) is a close upper bound
Big-O Analysis

Example: Polynomial Evaluation

• So, we say that the number of multiplications in the Brute Force method is $O(n^2)$ ("of the order of $n^2$") and that the time complexity of the Brute Force method is $O(n^2)$

• Think of this as "proportional to $n^2$"
Big-O Analysis: Summary

• We want $f(n)$ to be an easily recognized elementary function

• We want a \textit{tight upper bound} for our choice of $f(n)$
  • $n^3$ is also an upper bound for the Brute Force method, but not a good one!
  • Why not? Look at how $T(n)$ compares to $f(n) = n^3$
    • $T(n) / n^3$ approaches 0 for large $n$, i.e. $T(n)$ is miniscule when compared to $n^3$ for large $n$
Big-0 Example: Polynomial Evaluation Comparison

<table>
<thead>
<tr>
<th>$n$ and $T(n) = n$ (Horner)</th>
<th>$T(n) = n^2/2 + n/2$ (Brute Force)</th>
<th>$f(n) = n^2$ (upper bound for Brute Force)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>210</td>
<td>400</td>
</tr>
<tr>
<td>100</td>
<td>5050</td>
<td>10000</td>
</tr>
<tr>
<td>1000</td>
<td>500500</td>
<td>10000000</td>
</tr>
</tbody>
</table>

$n$ is the degree of the polynomial. Recall that we are comparing the number of multiplications.
Big-0 Example: Polynomial Evaluation

- $f(n) = n^2$
- $T(n) = n^2/2 + n/2$
- $T(n) = n$

$n$ (degree of polynomial)

# of mult’s

- $T(n) = n$ (linear growth)
- $T(n) = n^2/2 + n/2$ (quadratic growth)
- $f(n) = n^2$ (quadratic growth)

Graph showing the growth of $T(n)$ and $f(n)$ with $n$.
Time Complexity and Input

- Run time can depend on the size of the input only (e.g. sorting 5 items vs. 1000 items)
- Run 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 for large problems (large $n$)

- **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 the array contains the value \textbf{key}

• Return \textbf{array index} if found, \textbf{-1} if not found

Set \( k \) to 0.

While (\( k < n-1 \)) and (\( a[k] \) is not \textbf{key})

Add 1 to \( k \).

If \( a[k] == \textbf{key} \)

Return \( k \).

Else

Return \textbf{-1}. 
• Total amount of work done:
  • *Before loop*: a constant amount of work
  • *Each time through loop*: 2 comparisons and a constant amount of work (the *and* operation and addition)
  • *After loop*: a constant amount of work
  • *So, we consider the number of comparisons only*
• *Worst case*: need to examine all \( n \) array locations
• So, \( T(n) = 2*n \), and time complexity is \( O(n) \)
• Simpler (less formal) analysis:
  • Note that work done before and after the loop is independent of \( n \), and work done during a single execution of loop is independent of \( n \)
  • In worst case, loop will be executed \( n \) times, so amount of work done is proportional to \( n \), and algorithm is \( O(n) \)
• **Average** case for a *successful* search:
  • Number of comparisons necessary to find the key? 1 or 2 or 3 or 4 … or n
  • Assume that each possibility is equally likely
  • Average number of comparisons:
    \[
    \frac{1+2+3+ \ldots +n}{n} = \frac{n\times(n+1)/2}{n} = \frac{n+1}{2}
    \]
  • Average case time complexity is therefore \(O(n)\)
**Example: Binary Search**

- **General case**: 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’re done
  - Otherwise repeat the approach on the half of the array that might still contain $key$
  - etc…
Binary Search Algorithm

Set first to 0.
Set last to n-1.
Do {
    Set mid to (first + last) / 2.
    If key < a[mid], Set last to mid – 1.
    Else Set first to mid + 1.
} While (a[mid] is not key) and (first <= last).
If a[mid] == key Return mid.
Else Return –1.
• Amount of work done before and after the loop is a constant, and is independent of $n$

• Amount of work done during a single execution of the loop is constant

• Time complexity will therefore be proportional to 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, \( \leq \frac{n}{2} \) remain
  - After *second* time through, \( \leq \frac{n}{4} \) remain
  - After *third* time through, \( \leq \frac{n}{8} \) remain
  - After *\( k^{th} \)* time through, \( \leq \frac{n}{2^k} \) remain
• Suppose in the *worst case* that the maximum number of times through the loop is \( k \); we must express \( k \) in terms of \( n \)

• Exit the do..while loop when the number of remaining possible locations is less than 1 (that is, when \( \text{first} > \text{last} \)): this means that \( \frac{n}{2^k} < 1 \)
• Also, \( n/2^{k-1} \geq 1 \); otherwise, looping would have stopped after \( k-1 \) iterations

• Combining the two inequalities, we get
  \[
  n/2^k < 1 \leq n/2^{k-1}
  \]

• Invert and multiply through by \( n \) to get
  \[
  2^k > n \geq 2^{k-1}
  \]
• Next, take base-2 logarithms to get

\[ k > \log_2(n) \geq k-1 \]

which is equivalent to

\[ \log_2(n) < k \leq \log_2(n) + 1 \]

• So, binary search algorithm is \( O(\log_2(n)) \) in terms of the number of array locations examined
Big-O Analysis in General

• To determine the time complexity of an algorithm:
  • Look at the loop structure
  • Identify the basic operation(s)
  • 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,

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

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-O 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 *k*th iteration of outer loop, how many times through inner loop?
- Total number of iterations of inner loop = sum for *k* running from 1 to *n*
- Time complexity of algorithm?
Usefulness of Big-O

• 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>( n = 30 )</th>
<th>( n = 40 )</th>
<th>( n = 50 )</th>
<th>( n = 60 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 sec.</td>
<td>3 hours</td>
<td>130 days</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?