Algorithm Analysis

An algorithm is a clearly specified set of simple instructions to be followed to solve a problem.

- What is a problem?
- What is a program, for that matter?
- Topics for analysis:
  - How to estimate the time required for a program
  - How to reduce the running time of a program
  - The consequences of careless use of recursion
  - Very efficient algorithms to compute:
    - \( a^b \)
    - \( \gcd(a, b) \)

Estimation Techniques (Section 2.7)

- "Back of the envelope" or "back of the napkin" calculation
  1. Determine the problem's major parameters
  2. Derive an equation relating the parameters to the problem
  3. Select values for the parameters and apply the equation

- Example:
  - How many bookcases in the library are required to store 1 million pages?
  - Parameters:
    - length of a shelf in a case
    - number of shelves in a case
    - number of pages per unit of length
  - Estimate:
    - Pages per inch
    - feet per shelf
    - shelves per bookcase

Algorithm Efficiency

There are often many algorithms for a given problem. How do we choose the "best?"

- (Conflicting) goals of program design:
  - Algorithm is to be easy to understand, code, debug
  - Algorithm makes efficient use of computer's resources

- How do we measure an efficiency?
  - Empirical comparison (run the programs)
  - Asymptotic algorithm analysis
  - Critical resources:
  - Factors affecting running time:

Examples of Growth Rate

- Example 1:
  ```c
  int largest (int *array, int n) {
    int currlarge = array[0];
    for (int i=1; i<n; i++)
      if (array[i] > currlarge)
        currlarge = array[i];
    return currlarge;
  }
  ```

- Example 2: assignment statement

- Example 3:
  ```c
  sum = 0;
  for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
      sum++;
  ```
Growth Rate Graph

- Certain high growth-rate functions may be more efficient at some locations

Best, Worst, and Average Cases

- Not all inputs of a given size take the same time
- Sequential search for $K$ in an array of $n$ integers:
  - Best case:
  - Worst case:
  - Average case:
- While the average time seems to be the fairest measure, it may be difficult to determine
- When is the worst case time important?

Faster Computer or Algorithm?

- What happens when we buy a computer 10 times faster?

<table>
<thead>
<tr>
<th>$T(n)$</th>
<th>$n$</th>
<th>$n'$</th>
<th>Change</th>
<th>$n'/n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10n$</td>
<td>1,000</td>
<td>10,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$20n$</td>
<td>500</td>
<td>5,000</td>
<td>$n' = 10n$</td>
<td>10</td>
</tr>
<tr>
<td>$5n \log n$</td>
<td>250</td>
<td>1,842</td>
<td>$\sqrt{10n} &lt; n' &lt; 10n$</td>
<td>7.37</td>
</tr>
<tr>
<td>$2n^2$</td>
<td>70</td>
<td>223</td>
<td>$n' = \sqrt{10n}$</td>
<td>3.16</td>
</tr>
<tr>
<td>$2^n$</td>
<td>13</td>
<td>16</td>
<td>$n' = n + 3$</td>
<td>---</td>
</tr>
</tbody>
</table>

$n$: Size of input that can be processed in one hour (10,000 steps)

$n'$: Size of input that can be processed in one hour on the new machine (100,000 steps).

Asymptotic Analysis: $O(f(n))$

- The upper bound defined by “Big-Oh”:
  - Definition: For $T(n)$ a non-negatively valued function, $T(n) = O(f(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$.
  - Usage: “The algorithm is order of $n^2$ in [best, average, worst] case.”
  - Alternatively, “The algorithm is big-Oh of $n^2$…”
  - Meaning: for all data sets big enough (i.e., $n \geq n_0$), the algorithm always executes in at most $cf(n)$ steps [in best, average, or worst case].
- Example: if $T(n) = 3n^2$ then $T(n)$ is ...
  - $T(n) = 3n^2$ is $O(n^2)$, also $O(n^4)$, $O(n^3)$, etc.
  - It is preferable to say that $T(n)$ is $O(n^2)$.
  - (We always wish to get the tightest upper bound.)
**Big-Oh Examples**

- **Example 1:** Finding value $X$ in an array.
  - $T(n) = c_1n/2$.
  - For all values of $n \geq 1$, $c_1n/2 \leq c_1n$.
  - Therefore, by the definition, $T(n)$ is $O(n)$ for $n_0 = 1$ and $c = c_1$.

- **Example 2:** $T(n) = c_1n^2 + c_2n$ in average case.
  - $c_1n^2 + c_2n \leq c_1n^2 + c_2n^2 \leq (c_1 + c_2)n^2$ for all $n \geq 1$.
  - $T(n) \leq cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

- **Example 3:** $T(n) = c$. We say this is $O(1)$.

**Pitfall**

“The best case is for $n = 1$ since it runs most quickly.”

- Big-Oh refers to a **growth rate** as $n$ grows to infinity.
- The best case is defined as which input of size $n$ is the cheapest among all inputs of size $n$.
- Example: use insert sort to sort these lists
  - list 1: 1
  - list 2: 5, 9, 24, 1, 3, 12, 2, 16
  - list 3: 1, 2, 3, 5, 9, 12, 16, 24

**Asymptotic Analysis: $\Omega(g(n))$**

- The lower bound defined by “Big Omega”:
  - **Definition:** For $T(n)$ a non-negatively valued function, $T(n)$ is $\Omega(g(n))$ if there exist two positive constants $c$ and $n_0$ such that $T(n) \geq cg(n)$ for all $n \geq n_0$.
  - **Usage:** The algorithm is $\Omega(n^2)$ in [best, average, worst] case.
  - **Meaning:** For all datasets big enough (that is, $n \geq n_0$), the algorithm always executes in at least $cg(n)$ steps.

- **Example:** $T(n) = c_1n^2 + c_2n$.
  - $c_1n^2 + c_2n \geq c_1n^2$ for all $n \geq 1$.
  - $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.
  - Therefore, $T(n)$ is $\Omega(n^2)$ by the definition.
  - We want the greatest lower bound.

**Asymptotic Analysis: $\Theta(g(n))$**

- The equality defined by “Big Theta”:
  - When big-Oh and big-Omega meet, we indicate this by using the big-Theta (i.e., $\Theta$) notation.
  - **Definition:** For $T(n)$ a non-negatively valued function, $T(n) = \Theta(g(n))$ if and only if $T(n)$ is $O(g(n))$ and $T(n)$ is $\Omega(g(n))$.
  - **Usage:** “The algorithm is $\Theta(g(n))$.”
  - **Meaning:** $g(n)$ provides the best (tightest) upper and lower bound for $f(n)$.

- **Example:** $T(n) = c_1n^2$.
  - **Big-Oh:**
    - $c_1n^2 \leq c_1n^2$ for all $n \geq 1$
    - Therefore, $T(n)$ is $O(n^2)$
  - **Big-Omega:**
    - $c_1n^2 \geq c_1n^2$ for all $n \geq 1$
    - Therefore, $T(n)$ is $\Omega(n^2)$
  - $T(n)$ is $O(n^2)$ and $\Omega(n^2)$, so $T(n)$ is $\Theta(n^2)$
Asymptotic Analysis: \( o(g(n)) \)

- The strict upper bound defined by “little Oh”
  - When a function is big-Oh but not big-Theta, then it is little-Oh

**Definition:** For \( T(n) \) a non-negatively valued function, \( T(n) = o(g(n)) \) if \( T(n) \) is \( O(g(n)) \) and \( T(n) \neq \Theta(g(n)) \)

**Usage:** “The algorithm is \( o(g(n)) \)…"

- Example: \( T(n) = cn^2 \).
  - \( c_1n^2 \leq cn^2 \) for all \( n \geq 1 \), so \( T(n) \) is \( O(n^3) \)
  - Generally, \( c_1n^2 \not\leq cn^3 \), so \( T(n) \) is not \( \Omega(n^3) \), thus not \( \Theta(n^3) \)
  - Therefore, \( T(n) \) is \( o(n^3) \)

CSC 375-Turner, Page 13

Simplifying Rules:

- Certain rules allow us to simplify the analysis
  1. If \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \), then \( f(n) = O(h(n)) \)
  2. If \( f(n) = O(kg(n)) \) for any constant \( k > 0 \) then \( f(n) = O(g(n)) \)
  3. If \( T_1(n) = O(f(n)) \) and \( T_2(n) = O(g(n)) \), then
     (a) \( T_1(n) + T_2(n) = \max(O(f(n)), O(g(n))) \)
     (b) \( T_1(n) \times T_2(n) = O(f(n) \times g(n)) \)
  4. If \( T(n) \) is a polynomial of degree \( k \), then \( T(n) = \Theta(n^k) \)
  5. \( \log^n n = O(n) \) for any constant \( k \).

- Points of style:
  - Do not include constants or low-order terms in a Big-Oh (or \( \Omega \) or \( \Theta \))
  - It is bad to say \( f(n) \leq O(g(n)) \) (it’s implied)
  - It is incorrect to say \( f(n) \geq O(g(n)) \) (it makes no sense)
  - L’Hôpital’s rule can be applied, if necessary, to find relative growth rates

CSC 375-Turner, Page 14

Typical Growth rates

- There is a terminology for certain growth rate functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>Constant</td>
</tr>
<tr>
<td>( \log n )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( \log^2 n )</td>
<td>Log-squared</td>
</tr>
<tr>
<td>( n )</td>
<td>Linear</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

CSC 375-Turner, Page 15

Model

- Model of computation:
  - Normal computer
  - Sequential instruction execution
  - All simple instructions take one time unit
  - Fixed-size integers
  - No fancy operations that take one time unit
  - Infinite memory

- Weaknesses in the model:
  - Not all operations really take "one time unit"
  - Infinite memory allows us to ignore system realities

CSC 375-Turner, Page 16
What Behavior to Analyze

- Two resources can generally be analyzed:
  - Time (we usually focus on this)
  - Space

- For a given input size, three running time functions can be defined:
  - $T_{\text{best}}(n)$: best case running time
  - $T_{\text{worst}}(n)$: worst case running time
  - $T_{\text{avg}}(n)$: average case running time
  - Of course, $T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n)$

- Normally, worst and average case are of more interest:
  - Worst case represents a performance guarantee
  - Average case represents typical behavior, but it can be difficult to find
  - Best case can be useful for particular programming problems

Rules for Program Analysis

- Shortcuts used to analyze code:
  - For loops: running time is at most the running time of the statements inside times the number of iterations
  - Example:
    ```java
    for (i = 0; i < n; i++) {  // n iterations *
      j = j + i;          // (1 statement
      k = k * i;          // + 1 statement
    }
    ```

- Nested loops: analyze inside-out; total running time is the running time of the statements multiplied by the product of the sizes of all loops
  - Example:
    ```java
    for (i = 0; i < n; i++)  // n iterations *
      for (j = 0; j < n; j++)  // n iterations *
        k++;                // 1 statement
    ```

Rules for Program Analysis (cont.)

- Shortcuts (cont.)
  - Consecutive statements: they are added; thus take the maximum
    - Example:
      ```java
      for (i = 0; i < n; i++)  // n iterations
        a[i] = 0;           // * one statement
      ```

- If/Then/Else clause: running time of the test plus the higher complexity clause
  - Example:
    ```java
    if (x < 5) {
      for (i = 0; i < n*n; i++)
        j = j + i;
    } else
      j = 0;
    ```
Maxim um Subsequence Sum

The maximum subsequence sum (MSS) problem has many possible algorithms, whose complexity varies greatly.

- Given (possibly negative) numbers $A_1, A_2, \ldots, A_m$, find the maximum value of $\sum_{k=1}^m A_k$.

- Example
  - Input is -1, 11, -4, 13, -5, -2
  - Answer is 20 ($A_2$ through $A_4$)

- Run times for several algorithms solving this problem:

Cubic algorithm:
```cpp
int maxSubSum1(const vector<int> &a) {
    /* 1*/    int maxSum = 0;
    /* 2*/    for (int i = 0; i < a.size(); i++)
    /* 3*/        for (int j = i; j < a.size(); j++)
        { 
            /* 4*/            int thisSum = 0;
            /* 5*/            for (int k = i; k <= j; k++)
                /* 6*/                thisSum += a[k];
            /* 7*/            if (thisSum > maxSum)
                /* 8*/                maxSum = thisSum;
            } 
    /* 9*/    return maxSum;
} 
```
### Analysis of MSS Problem

**Quadratic algorithm:**

```cpp
int maxSubSum2(const vector<int> & a) {
    int maxSum = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i; j < a.size(); j++)
            thisSum += a[j];
    if (thisSum > maxSum)
        maxSum = thisSum;
    thisSum = 0;
}
```

### Analysis of MSS Problem

**Linear algorithm:**

```cpp
int maxSubSum4(const vector<int> & a) {
    int maxSum = 0, thisSum = 0;
    for (int j = 0; j < a.size(); j++)
        thisSum += a[j];
    if (thisSum > maxSum)
        maxSum = thisSum;
    else if (thisSum < 0)
        thisSum = 0;
}
```

### Analysis of MSS Problem

**O(n log n) algorithm:**

```cpp
int maxSumRec(const vector<int> & a, int left, int right) {
    if (left == right) // Base case
        return a[left];
    else
        return 0;
}
```

### Analysis of MSS Problem

**Logarithms in Running Time**

- **General rules:**
  - If it takes $O(1)$ time to cut problem size by a fraction, then the algorithm is $O(\log n)$
  - If it takes $O(1)$ time to cut problem size by a constant amount, then the algorithm is $O(n)$

- **Remember:** simply reading the input is $\Omega(n)$

- **Binary search:** find an element in a sorted array (or vector)

  ```cpp
  int binary(int X, int* array, int left, int right) {
      int l = left-1;
      int r = right+1;
      while (l+1 != r) {
          int i = (l + r) / 2;
          if (X < array[i]) r = i;
          if (X == array[i]) return i;
          if (X > array[i]) l = i;
      }
      return UNSUCCESSFUL;
  }
  ```

Analysis: how many elements can be examined in the worst case?
Logarithms in Running Time

- **Euclid's algorithm**: find the greatest common divisor (gcd)
  
  ```c
  long gcd(long m, long n)
  {       
/* 1*/ while(n != 0)       
/* 2*/   {              
/* 3*/       long rem = m % n;              
/* 4*/       m = n;                  
/* 5*/       n = rem;                
/* 6*/   }           
/* 7*/   return m;                     
  }
  
  - **Efficient exponentiation**: compute \( x^n \)
  
  ```

CSC 375-Turner, Page 29

Analyzing Problems

- Upper bound: upper bound of best known algorithm.
- Lower bound: lower bound for every possible algorithm.
- Example: sorting.
  - Cost of I/O: \( \Omega(n) \)
  - Bubble or insertion sort: \( O(n^2) \)
  - A better sort (Quicksort, Mergesort, heapsort, etc.): \( O(n \log n) \)

CSC 375-Turner, Page 30

Multiple Parameters

Compute the rank ordering for all \( C \) pixel values in a picture of \( P \) pixels.

```c
for (i = 0; i < C; i++)
    count[i] = 0; // initialize counts
for (i = 0; i < P; i++) // examine each pixel to
    count[value[i]]++; // find freq of occurrence
sort(count); // Sort the array

- If we use \( P \) as a measure, then the time is \( \Theta(P \log P) \)
- What is a more accurate measure? Why?
```

CSC 375-Turner, Page 31

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

- Algorithms: analyze time
- Data structures: analyze space
- Space/Time Tradeoff Principle:
  One can often achieve a reduction in time if one is willing to sacrifice space, or vice versa.

- Examples:
  - Encoding or packing information
  - Table lookup

CSC 375-Turner, Page 32