Algorithm Analysis

• Example: Sequential Search.
  – Given an array of $n$ elements, determine if a given number $val$ is in the array.
  – If so, set $loc$ to be the index of the first occurrence of $val$, and return true.
  – Otherwise, return false.

• The Algorithm:
  ```
  bool SeqSearch(int A[], int n, int val, int &loc) {
    loc = 0;
    while (loc < n && A[loc] != val) ++loc;
    return loc < n;
  }
  ```

Factors Affecting Run Time

• Characteristics of the computer system (e.g., processor speed, amount of memory, file-system type, etc.)
• The way the algorithm is implemented
• The particular instance of data the algorithm is operating on (e.g., amount of data, type of data).

Conclusion?

Given these facts

• What should we use as a measure of how “good” an algorithm is?
• By what should we compare two algorithms with each other?

This is what algorithm analysis is all about.

Simplifying Assumptions

• As stated several times before, the characteristics of the particular computer system that the algorithm will execute on are considered irrelevant.
• Often the implementation of the algorithm is also ignored, although later we’ll see an example where it matters.
• Each instruction, no matter how simple or complex, is considered to take one “unit” of time. Why?
• Some simple measure of the “size” of the data that the algorithm is operating on is made, e.g., the size of an array, the number of nodes in a graph, the dimensions of a matrix.

Example Revisited

Sequential Search

```
SeqSearch(int A[], int n, int val, int &loc) {
    loc := 0;
    while (loc < n && A[loc] != val) ++loc;
    return loc < n;
}
```
Typical Input Data

- We first need to determine what the input is, and how much data is being input.
- We need to determine which of the data affects the running time.
- We usually use $n$ to denote the number of data items to be processed.
- This could be:
  - size of a file
  - size of an array or matrix
  - number of nodes in a tree or graph
  - degree of a polynomial

Abstract Operations

- We talk about abstract operations when we consider operations in a hardware independent fashion.
- Recall that we are interested in rate of growth, not the exact running time. Thus, we can pick operations that will run most often in the code.
- Determine the number of times these operations will be executed as a function of the size of the input data.
- It is crucial that we pick the operations that are executed most often, and that we recognize when an operation can or cannot be performed in a constant amount of time.

Example: Search for Maximum

```c
int max(int a[], int n) {
    int max = INT_MIN;
    for (int i=0; i<n; i++)
        max = MAXIMUM(max, a[i]);
    return max;
}
```

- We focus on the assignment (=) inside the loop and ignore the other instructions.
- for an array of length 1, 1 comparison
- for an array of length 2, 2 comparisons
- for an array of length $n$, $n$ comparisons

Mathematical Analysis

There are three types of analysis that can performed on an algorithm.

- **Best-case analysis**
  Analysis of the performance of the algorithm assuming the “easiest” instance of data input.
  - This is the most useless one. Why?

- **Average-case analysis**
  Analysis of the performance of the algorithm assuming an “average” instance of data input.
  - This may be difficult. Why?

- **Worst-case analysis**
  Analysis of the performance of the algorithm assuming the “worst” instance of data input.
  - This is the most practical. Why?

Example: factorial

```c
factorial(n) {
    if (n==1)
        return 1
    else
        return n * factorial(n-1)
}
```

- We focus on the comparison (==) (this is the abstract operation) and ignore the other instructions.
- For example, if we calculate the number of operations in this function based on the comparison operator, we have:
  - for factorial(1), 1 operation
  - for factorial(2), 2 operations
  - for factorial(n), $n$ operations

Analysis Example: Insertion Sort

```c
void insertion(intType A[], int n) {
    int i, j; intType v;
    for (i=1; i<n; i++) {
        v = A[i];
        j = i-1;
        while (j > 0 && A[j] > v) {
            j--;
        }
        A[j+1] = v;
    }
}
```

- Assume we use the comparisons in the “while” loop as our abstract operation.
  (Is this a good choice?)
- **Worst-case:** When the array $A$ is sorted in descending order, $A[j] > v$ for 1 to $i-1$ for every iteration of the “for” loop. The total number of comparisons is $\sum_{i=2}^{n-1}(i-1) = n(n-1)/2 \approx n^2/2$.
- **Best-case:** When the array $A$ is already sorted in ascending order, the algorithm only executes $n$ comparisons!
Analysis Example: SumOfProducts

```java
double SumOfProducts(double A[], int size)
{
    double V;
    for (int i=1; i<size; i++)
    {
        for (int j=1; j<size; j++)
        {
            V=A[i]*A[j];
        }
    }
    • We will use the assign (V=A[i]*A[j]) as our abstract operation.
      (Is this a good choice?)
    • Since there are no conditionals (if, while) the worst, average, and best case will be the same.
    • Notice that j ranges from 1 to size.
    • Thus, each time the inner loop executes, it uses size operations.
    • The outer loop also executes size times, each time executing the inner loop.
    • Thus, the number of operations is size * size = size^2.
```

Example: \( n^2 + n = O(n^3) \)

Proof:
- Here, we have \( f(n) = n^2 + n \), and \( g(n) = n^3 \)
- Notice that if \( n \geq 1 \), \( n \leq n^3 \) is clear.
- Also, notice that if \( n \geq 1 \), \( n^2 \leq n^3 \) is clear.
- **Side Note:** In general, if \( a \leq b \), then \( n^a \leq n^b \) whenever \( n \geq 1 \). This fact is used often in these types of proofs.
- Therefore,
  \[
  n^2 + n \leq n^3 + n^3 = 2n^3
  \]
- We have just shown that
  \[
  n^2 + n \leq 2n^3 \text{ for all } n \geq 1
  \]
- Thus, we have shown that
  \[
  n^2 + n = O(n^3)
  \]
  (by definition of Big-O, with \( n_0 = 1 \), and \( c = 2 \).)

Growth of Functions and Asymptotic Notation
- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interested in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the asymptotic running time.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- **Asymptotic notation** gives us a method for classifying functions according to their rate of growth.

Big-O Notation
- **Definition:** \( f(n) = O(g(n)) \) iff there are two positive constants \( c \) and \( n_0 \) such that
  \[
  |f(n)| \leq cg(n) \text{ for all } n \geq n_0
  \]
- If \( f(n) \) is nonnegative, we can simplify the last condition to
  \[
  0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0
  \]
- We say that “\( f(n) \) is big-O of \( g(n) \).”
- As \( n \) increases, \( f(n) \) grows no faster than \( g(n) \). In other words, \( g(n) \) is an asymptotic upper bound on \( f(n) \).

Example: \( n^3 + 4n^2 = \Omega(n^2) \)

Proof:
- Here, we have \( f(n) = n^3 + 4n^2 \), and \( g(n) = n^2 \)
- It is not too hard to see that if \( n \geq 0 \),
  \[
  n^3 \leq n^3 + 4n^2
  \]
- We have already seen that if \( n \geq 1 \),
  \[
  n^2 \leq n^3 + 4n^2
  \]
- Thus when \( n \geq 1 \),
  \[
  n^2 \leq n^3 + 4n^2 \leq n^3
  \]
- Therefore,
  \[
  1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1
  \]
- Thus, we have shown that \( n^3 + 4n^2 = \Omega(n^2) \)
  (by definition of Big-Omega, with \( n_0 = 1 \), and \( c = 1 \).)
**Big-$\Theta$ notation**

- **Definition:** $f(n) = \Theta(g(n))$ if there are three positive constants $c_1$, $c_2$ and $n_0$ such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.
- If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.
- We say that “$f(n)$ is theta of $g(n)$.”
- As $n$ increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an asymptotically tight bound on $f(n)$.

**Example:** $n^2 + 5n + 7 = \Theta(n^2)$

**Proof:**
- When $n \geq 1$,
  \[ n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2 \]
- When $n \geq 0$,
  \[ n^2 \leq n^2 + 5n + 7 \]
- Thus, when $n \geq 1$
  \[ 1n^2 \leq n^2 + 5n + 7 \leq 13n^2 \]

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-$\Theta$, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$).

**Strategies for Big-O**

- Sometimes the easiest way to prove that $f(n) = O(g(n))$ is to take $c$ to be the sum of the positive coefficients of $f(n)$.
- We can usually ignore the negative coefficients. Why?
- **Example:** To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$, $5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2$, thus $5n^2 + 3n + 20 = O(n^2)$.
- This is not always so easy. How would you show that $(\sqrt{2})^n + \log^2 n + n^4$ is $O(2^n)$? Or that $n^4 = O(n^4 - 13n + 23)$? After we have talked about the relative rates of growth of several functions, this will be easier.
- In general, we simply (or, in some cases, with much effort) find values $c$ and $n_0$ that work. This gets easier with practice.

**Strategies for $\Omega$ and $\Theta$**

- Proving that $f(n) = \Omega(g(n))$ often requires more thought.
  - Quite often, we have to pick $c < 1$.
  - A good strategy is to pick a value of $c$ which you think will work, and determine which value of $n_0$ is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms of $f(n)$ with the positive coefficients. Why?
- The following theorem shows us that proving $f(n) = \Theta(g(n))$ is nothing new:
  - **Theorem:** $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Thus, we just apply the previous two strategies.
- We will present a few more examples using a several different approaches.

**Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$**

**Proof:**
- Notice that if $n \geq 1$,
  \[ \frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n = \frac{7}{2}n^2 \]
- Thus,
  \[ \frac{1}{2}n^2 + 3n = O(n^2) \]
- Also, when $n \geq 0$,
  \[ \frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n \]
- So
  \[ \frac{1}{2}n^2 + 3n = \Omega(n^2) \]
- Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,
  \[ \frac{1}{2}n^2 + 3n = \Theta(n^2) \]
We'll take a closer look at each of these classes.

**Exponential:**
- \( \text{for}(i=0; i<n; i++) \)
- \( \text{return sum;} \)

Thus, it runs in quadratic time.

**Linear:**
- \( \text{int sum=0;} \)
- \( \text{for}(i=0; i<n; i++) \)
- \( \text{sum=sum+A[i];} \)
- \( \text{return sum;} \)

**Summary of the Notation**
- \( f(n) \in O(g(n)) \Rightarrow f \leq g \)
- \( f(n) \in \Omega(g(n)) \Rightarrow f \geq g \)
- \( f(n) \in \Theta(g(n)) \Rightarrow f \geq g \)
- \( f(n) \in O(g(n)) \Rightarrow g \) is an upper bound. So an algorithm that is \( O(n^2) \) might not ever take that much time. It may actually run in \( O(n) \) time.
- Conversely, an \( \Omega \) bound is only a lower bound. So an algorithm that is \( \Omega(n \log n) \) might actually be \( \Theta(n^2) \).
- Unlike the other bounds, a \( \Theta \)-bound is precise. So, if an algorithm is \( \Theta(n^2) \), it runs in quadratic time.

**Common Rates of Growth**

In order for us to compare the efficiency of algorithms, we need to know some common growth rates, and how they compare to one another. This is the goal of the next several slides.

Let \( n \) be the size of input to an algorithm, and \( k \) some constant. The following are common rates of growth.
- Constant: \( \Theta(k) \), for example \( \Theta(1) \)
- Linear: \( \Theta(n) \)
- Logarithmic: \( \Theta(\log_k n) \)
- \( n \log n: \Theta(n \log n) \)
- Quadratic: \( \Theta(n^2) \)
- Polynomial: \( \Theta(n^k) \)
- Exponential: \( \Theta(k^n) \)

We’ll take a closer look at each of these classes.

**Asymptotic Notation**

**Asymptotic Bounds and Algorithms**
- In all of the examples so far, we have assumed we knew the exact running time of the algorithm.
- In general, it may be very difficult to determine the exact running time.
- Thus, we will work our way to a more precise bound on the exact running time.
- **Example:** What is the complexity of the following algorithm?

```java
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      a[i][j] = b[i][j] * x;
```

**Answer:** \( O(n^2) \)
- We will see more examples later.

**Classification of algorithms - \( \Theta(1) \)**
- Operations are performed \( k \) times, where \( k \) is some constant, independent of the size of the input \( n \).
- This is the best one can hope for, and most often unattainable.
- **Examples:**

```java
int Fifth_Element(int A[], int n) {
   return A[5];
}
```

```java
int Partial_Sum(int A[], int n) {
   int sum=0;
   for(int i=0; i<42; i++)
      sum=sum+A[i];
   return sum;
}
```
Classification of algorithms - $\Theta(n)$

- Running time is linear
- As $n$ increases, run time increases in proportion
- Algorithms that attain this look at each of the $n$ inputs at most some constant $k$ times.

Examples:
```c
void sum_first_n(int n) {
    int i, sum=0;
    for (i=1;i<=n;i++)
        sum += i;
}
void m_sum_first_n(int n) {
    int i,k,sum=0;
    for (i=1;i<=n;i++)
    for (k=1;k<7;k++)
        sum += i;
}
```

Classification of algorithms - $\Theta(\log n)$

- A logarithmic function is the inverse of an exponential function, i.e. $b^x = n$ is equivalent to $x = \log_b n$.
- Always increases, but at a slower rate as $n$ increases. (Recall that the derivative of $\log n$ is $\frac{1}{n}$, a decreasing function.)
- Typically found where the algorithm can systematically ignore fractions of the input.

Examples:
```c
int binarysearch(int a[], int n, int val) {
    int l=1, r=n, m;
    while (r-l>1) {
        m = (l+r)/2;
        if (a[m]==val) return m;
        if (a[m]>val) r=m-1;
        else l=m+1;
    }
    return -1;
}
```

Comparison of growth rates

<table>
<thead>
<tr>
<th>$\log n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0.6931</td>
<td>2</td>
<td>1.39</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>1.099</td>
<td>3</td>
<td>3.30</td>
<td>9</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>1.386</td>
<td>4</td>
<td>5.55</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>1.609</td>
<td>5</td>
<td>8.05</td>
<td>25</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>1.792</td>
<td>6</td>
<td>10.75</td>
<td>36</td>
<td>216</td>
<td>64</td>
</tr>
<tr>
<td>1.946</td>
<td>7</td>
<td>13.62</td>
<td>49</td>
<td>343</td>
<td>128</td>
</tr>
<tr>
<td>2.079</td>
<td>8</td>
<td>16.64</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>2.197</td>
<td>9</td>
<td>19.78</td>
<td>81</td>
<td>729</td>
<td>512</td>
</tr>
<tr>
<td>2.303</td>
<td>10</td>
<td>23.03</td>
<td>100</td>
<td>1000</td>
<td>1024</td>
</tr>
<tr>
<td>2.398</td>
<td>11</td>
<td>26.38</td>
<td>121</td>
<td>1331</td>
<td>2048</td>
</tr>
<tr>
<td>2.465</td>
<td>12</td>
<td>29.82</td>
<td>144</td>
<td>1728</td>
<td>4096</td>
</tr>
<tr>
<td>2.565</td>
<td>13</td>
<td>33.34</td>
<td>169</td>
<td>2197</td>
<td>8192</td>
</tr>
<tr>
<td>2.639</td>
<td>14</td>
<td>36.95</td>
<td>196</td>
<td>2744</td>
<td>16384</td>
</tr>
<tr>
<td>2.708</td>
<td>15</td>
<td>40.62</td>
<td>225</td>
<td>3375</td>
<td>32768</td>
</tr>
<tr>
<td>2.773</td>
<td>16</td>
<td>44.36</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>2.833</td>
<td>17</td>
<td>48.16</td>
<td>289</td>
<td>4913</td>
<td>131072</td>
</tr>
<tr>
<td>2.890</td>
<td>18</td>
<td>52.03</td>
<td>324</td>
<td>5832</td>
<td>262144</td>
</tr>
</tbody>
</table>

Examples:
```
```
More growth rates

<table>
<thead>
<tr>
<th>n</th>
<th>100n</th>
<th>n^2</th>
<th>11n^2</th>
<th>n^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>4</td>
<td>44</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>9</td>
<td>99</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>16</td>
<td>176</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>25</td>
<td>275</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
<td>36</td>
<td>396</td>
<td>216</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>700</td>
<td>49</td>
<td>539</td>
<td>343</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>800</td>
<td>64</td>
<td>704</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>81</td>
<td>891</td>
<td>729</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>100</td>
<td>1000</td>
<td>1000</td>
<td>1024</td>
</tr>
<tr>
<td>11</td>
<td>1100</td>
<td>121</td>
<td>1331</td>
<td>1331</td>
<td>2048</td>
</tr>
<tr>
<td>12</td>
<td>1200</td>
<td>144</td>
<td>1584</td>
<td>1728</td>
<td>4096</td>
</tr>
<tr>
<td>13</td>
<td>1300</td>
<td>169</td>
<td>1859</td>
<td>2197</td>
<td>8192</td>
</tr>
<tr>
<td>14</td>
<td>1400</td>
<td>196</td>
<td>2156</td>
<td>2744</td>
<td>16384</td>
</tr>
<tr>
<td>15</td>
<td>1500</td>
<td>225</td>
<td>2475</td>
<td>3375</td>
<td>32768</td>
</tr>
<tr>
<td>16</td>
<td>1600</td>
<td>256</td>
<td>2816</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>17</td>
<td>1700</td>
<td>289</td>
<td>3179</td>
<td>4913</td>
<td>131072</td>
</tr>
<tr>
<td>18</td>
<td>1800</td>
<td>324</td>
<td>3564</td>
<td>5832</td>
<td>262144</td>
</tr>
<tr>
<td>19</td>
<td>1900</td>
<td>361</td>
<td>3971</td>
<td>6859</td>
<td>524288</td>
</tr>
</tbody>
</table>
Why Constants and Non-Leading Terms Don’t Matter

- Only the leading term is important.
- Constants don’t make a significant difference.
- The following inequalities hold asymptotically:
  \[ c < \log n < \log^2 n < \sqrt{n} < n < n \log n \]

\[ n < n \log n < n^{1.1} < n^2 < n^3 < n^4 < 2^n \]

- In other words, an algorithm that is \( \Theta(n \log(n)) \) is more efficient than an algorithm that is \( \Theta(n^3) \).