Week 1
CS 5006 Algorithms
Algorithm

- Takes input
- Performs some computational step or steps
- Produces an output
- Everything you ever write is an algorithm
- Some are simple
- Some are complex
- Can produce error as output
Correct vs Incorrect

- Correct algorithm
  - Always halts with the correct output
  - Can halt with an error if that is the correct output

- Incorrect algorithm
  - May not halt
  - May halt with the wrong answer
Instance vs Problem

- A problem is a general statement
- It states a logical or mathematical problem
- An instance is a specific input to the problem
- An problem can have a finite or infinite number of instances
Tradeoffs

- 2 out of 3
  - Good
  - Fast
  - Cheap

- CS as a whole is about tradeoffs
- Sacrificing quality or quantity can get you results
- For each problem decide what is important
Operations/Efficiency

- MIPS – Millions of Instructions Per Second
  - Largely an out of date metric
  - Parallelism in CPU's makes this difficult to use
  - Might be useful in demonstrating efficiency of algorithms

- Clock cycle
  - Theoretically single instruction per cycle
  - Clock rate (2.5 Ghz) is number of cycles per second
Sorting Problem

- A classic problem with many algorithms to solve them
- Formally stated:
  - Input: A sequence of $n$ numbers $(a_1, a_2..., a_n)$
  - Output: A permutation $(a'_1, a'_2..., a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq ... \leq a'_n$
Insertion Sort

- Take a number
- Put it in correct slot of currently sorted elements
- Repeat with the next element
- Keep going until all numbers have been inserted
Pseudocode

- A language agnostic way of presenting code
- Talk about algorithms without syntax issues
- Allows us to avoid compiler issues
Insertion Sort Pseudocode

1. for $j=2$ to $A$.length
2. key = $A[j]$
4. $i = j - 1$
5. while $i > 0$ and $A[i] > key$
6. $A[i+1] \text{ and } A[i] > key$
7. $i = i - 1$
8. $A[i+1] = key$
Invariants

- Condition that is always true
- During the execution of a program
- During some portion of a program
- Used to reason about problems
Loop Invariant

- Condition always true at the beginning of a loop
- True prior to first iteration of the loop
- If true before an iteration, remains true before the next one
- At termination the invariant gives a useful property to prove the algorithm is correct
Insertion Sort Loop Invariant

- At the start of each iteration of the for loop, the subarray \( A[1..j-1] \) consists of the elements originally in \( A[1..j-1] \) in sorted order.
Insertion Sort Loop Invariant

1. for j=2 to A.length
2. key=A[j]
4. i=j-1
5. while i>0 and A[i]> key
6. A[i+1] and A[i] > key
7. i = i-1
8. A[i+1] = key

A[1] is sorted and j is 2
Find the right spot for A[j].

j is incremented for the next iteration of the for loop. This allows us to say that A[1..j] is in sorted order at the beginning of each iteration.

When the loop terminates j is at n+1 and the array is sorted that far thus we have a sorted array and the algorithm is correct
Algorithm Analysis

- **Input size**
  - Often number of elements that you are working with
  - The number of items that you are operating on
  - Operating on multiple items, multiple numbers

- **Running Time**
  - Not a specific amount of time
  - Number of operations to solve the problem
  - Also known as the cost of performing an algorithm
## Analysis Insertion Sort

<table>
<thead>
<tr>
<th>Code</th>
<th>Input Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>for j=2 to A.length</td>
<td>$c_1$</td>
<td>$n$</td>
</tr>
<tr>
<td>key = A[j]</td>
<td>$c_2$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>i = j-1</td>
<td>$c_4$</td>
<td>$n-1$</td>
</tr>
<tr>
<td>while i &gt; 0 and A[i] &gt; key</td>
<td>$c_5$</td>
<td>$\sum_{j=2}^{n} t_j$</td>
</tr>
<tr>
<td>A[i+1] = A[i]</td>
<td>$c_6$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>i = i-1</td>
<td>$c_7$</td>
<td>$\sum_{j=2}^{n} (t_j - 1)$</td>
</tr>
<tr>
<td>A[i+1] = key</td>
<td>$c_8$</td>
<td>$n-1$</td>
</tr>
</tbody>
</table>

\[
c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)
\]
Arithmetic Series

\[\sum_{k=1}^{n} = \frac{1}{2}n(n + 1)\]

\[\sum_{k=1}^{n} = \frac{1}{2}n^2 + \frac{1}{2}\]
Break Down This Equation

\[ c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1) \]
Worst and Average Case

- Worst case is the upper bound
  - In the case of insertion sort, this would be reverse order
  - The slowest an algorithm can ever run
  - Can happen often based on the use case

- Average case often close to worst

- Since input is often random, average case is usually just a constant applied to the worst case
Order of Growth

- Don’t care about the actual time an algorithm runs
- Care about its rate of growth with more elements
- We only consider the highest order of our equation
- Constants and lower order variables are ignored
- This is known as Θ notation
- Θ is the worst case scenario
- Insertion sort is Θ (n^2)
## Order of Growth

<table>
<thead>
<tr>
<th>Order</th>
<th>Equation</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(1) )</td>
<td>( y = c )</td>
<td>Constant</td>
</tr>
<tr>
<td>( \Theta(lg\ n) )</td>
<td>( y = \log_2(x) )</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>( \Theta(n) )</td>
<td>( y = x )</td>
<td>Linear</td>
</tr>
<tr>
<td>( \Theta(n\ lg\ n) )</td>
<td>( y = x \log_2(x) )</td>
<td>Linearithmic</td>
</tr>
<tr>
<td>( \Theta(n^2) )</td>
<td>( y = x^2 )</td>
<td>Exponential</td>
</tr>
</tbody>
</table>
Divide and Conquer

- Divide the problem into smaller ones
- Solve the simple problems
- Combine the solutions of the simple problems
- Generally leads to $\lg(n)$ or $n \lg(n)$ solutions
Merge Sort

- A divide and conquer algorithm
- Allows us to sort $n \log(n)$ time
- Key is to sort two halves of the array
- Then we merge the array
Merge Function

- **Merge (A, p, q, r)**
  - A – The array to be sorted
  - p – bottom index of first half
  - q – top index of first half
  - q+1 – bottom index of second half
  - r – top index of second half

- First and second halves are guaranteed to be sorted

- $\Theta(n)$ where $n$ is $r-p+1$
Merge Pseudocode

Merge(A, p, q, r)
1 \( n_1 = q \cdot p + 1 \)
2 \( n_2 = r - q \)
3 Let L[1.. n_1+1] and R[1..n_2 +1] be new arrays
4 for i = 1 to n_1
5 \( L[i] = A[p+i-1] \)
6 for j = 1 to n_2
7 \( R[j] = A[q+j] \)
8 \( L[n_1+1] = \infty \)
9 \( R[n_2+1] = \infty \)
10 i = 1
11 j = 1
12 for k = p to r
13 if \( L[i] \leq R[j] \)
14 \( A[k] = L[i] \)
15 \( i = i+1 \)
16 else \( A[k] = R[j] \)
17 \( j = j+1 \)

Loop needs to be invariant in order for this sort to work
Merge Sort Loop Invariant

- At the start of each iteration of the for loop of lines 12-17, the subarray $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to $A$. 

Initialization

- \( k = p \)
- Subarray \( A[p..k-1] \) is empty or non-existent
- The subarray contains 0 elements \((k-p)\)
- \( i \) and \( j \) are 1
- \( L[i] \) is the smallest value in \( L \)
- \( R[j] \) is the smallest value in \( R \)
- They have not yet been copied back to \( A \)
Maintenance

- \( L[i] \leq R[j] \)
  - \( L[i] \) smallest element not copied to \( A \)
  - \( A[p..k-1] \) contains smallest \( k-p \) elements
  - Copying \( L[i] \) into \( A[k] \)
    - \( A[p..k] \) contains \( k-p+1 \) smallest elements
    - update \( k \)
    - update \( i \)
  - Now \( A[p..k] \) contains smallest \( k-p \) elements
  - \( L[i] \) now contains smallest element not copied to \( A \)

- \( L[i] > R[j] \)
  - Above applies but \( R \) instead of \( L \) and \( j \) instead of \( i \)
  - All other steps are the same
Termination

- \( k = r + 1 \)
- \( A[p..k-1] \) same as \( A[p..r] \)
- \( A[p..r] \) now contains \( k - p \) smallest elements
- All but two largest elements have been copied from L and R
- The two largest elements are not part of the original array
Merge Runtime

Merge(A, p, q, r)

\[
\begin{align*}
1 & \quad n_1 = q \cdot p + 1 \\
2 & \quad n_2 = r \cdot q \\
3 & \quad \text{Let } L[1..n_1 + 1] \text{ and } R[1..n_2 + 1] \text{ be new arrays} \\
4 & \quad \text{for } i = 1 \text{ to } n_1 \\
5 & \quad \quad L[i] = A[p+i-1] \\
6 & \quad \text{for } j = 1 \text{ to } n_2 \\
7 & \quad \quad R[j] = A[q+j] \\
8 & \quad L[n_1 + 1] = \infty \\
9 & \quad R[n_2 + 1] = \infty \\
10 & \quad i = 1 \\
11 & \quad j = 1 \\
12 & \quad \text{for } k = p \text{ to } r \\
13 & \quad \quad \text{if } L[i] \leq R[j] \\
14 & \quad \quad \quad A[k] = L[i] \\
15 & \quad \quad \quad i = i + 1 \\
16 & \quad \quad \text{else } A[k] = R[j] \\
17 & \quad \quad \quad j = j + 1
\end{align*}
\]

\( \Theta(n) \)

\( \Theta(n) \)

\( \Theta(1) \)

\( \Theta(1) \)
Merge Sort Pseudocode

Merge-Sort(A, p, r)
1   if p<r
2     q = \left\lceil \frac{(p+r)}{2} \right\rceil
3     Merge-Sort(A, p, q)
4     Merge-Sort(A, q+1, r)
5     Merge(A, p, q, r)
Recurrence

- Describes running time of an algorithm with recursion
- Merge sort is analyzed with a recurrence
Running Time of Merge Sort

- $T(n)$ running time of merge sort
- Problem size small when
  - $n \leq c$, $c$ is 1 in this case
  - Runtime $\Theta(1)$
- Division yields $a$ subproblems
- Each subproblem is $1/b$ size of the original
- $T(n/b)$ is the time to solve one subproblem of size $n/b$
- Takes $aT(n/b)$ to solve $a$ subproblems
- $D(n)$ time to divide the problem, $C(n)$ to combine
Runtime of Merge Sort

\[ \begin{cases} 
\Theta(1) & \text{if } n \leq c \\
aT(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases} \]
Runtime Analysis

<table>
<thead>
<tr>
<th>Action</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide</td>
<td>( D(n) = \Theta(1) )</td>
</tr>
<tr>
<td>Conquer</td>
<td>( a = 2, \ b = 2 \text{ thus } 2T(n/2) )</td>
</tr>
<tr>
<td>Combine</td>
<td>( C(b) = \Theta(n) )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\Theta(1) & \quad \text{if } n = 1 \\
2T(n/2) \Theta(n) & \quad \text{if } n > 1
\end{align*}
\]
How to Calculate $2T(n/2)$