Today’s class:

• Introduction to algorithm analysis
• Growth of functions
Introduction

• What is an algorithm?
  – A sequence of computation steps that takes a set of values as input and produces a set of values as output
  – A method for solving a specific computational problem
  – Many different algorithms may exist for the same problem and they may different characteristics in terms of resources consumed
    • Time
    • Memory
    • Bandwidth
    • Area
Algorithm Analysis

- How do you predict the resources that an algorithm uses?
- For the most part, we are concerned with time - the overall computational speed of the algorithm
- For numerical applications, memory can be also be a concern
Algorithm Analysis

- Use a high-level “pseudocode” representation of the algorithm
- To get an estimate of time, we need to count the number of operations needed to complete the algorithm
- The number of operations will be a function of $n$ where $n$ is the size of the input
- Allows us to characterize the performance of an algorithm independent of the implementation and runtime platform
Algorithm Analysis

• What is the input size?
• Depends on the problem
• Examples
  – Size of an array to be sorted
  – Number of bits in two numbers being multiplied
  – Number of edges and vertices in a graph
Algorithm Analysis

- What operations do we count?

- Primitive operations
  - Doing a comparison
  - Assigning a value
  - Evaluating an arithmetic expression
  - Calling a subroutine

- In general, count a line of pseudocode as an operation

- Assume that each operation takes constant time
  - Constant time means that the operation time is not dependent on the input size
Sorting algorithms

- Problem: Sort an array of numbers in ascending order

- Several algorithms
  - Insertion sort
  - Merge sort
  - Quick sort
Insertion sort

- Start at the beginning of the array and move towards the end of the array
- Insert each array element that you encounter back into the array in ordered fashion
- Requires comparisons with each element of the sorted part of the array
Insertion sort

6 5 9 7 8 3 4
6 5 9 7 8 3 4
5 6 9 7 8 3 4
5 6 9 7 8 3 4
5 6 7 9 8 3 4
5 6 7 8 9 3 4
Insertion sort

for j=2 to n
    k = A[j]
    i = j-1
while i > 0 and A[i] > k
    A[i+1] = A[i]
    i = i-1
A[i+1] = k

Cost #times

\[
c_1 \quad n \\
c_2 \quad n-1 \\
c_3 \quad n-1 \\
c_4 \quad \sum_{j=2}^{n} t_j \\
c_5 \quad \sum_{j=2}^{n} (t_j - 1) \\
c_6 \quad \sum_{j=2}^{n} (t_j - 1) \\
c_7 \quad n-1
\]

\[
T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)
\]
Insertion sort

\[ T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1) \]

- **Best case, already sorted**

  \[ t_j = 1 \]

  \[ T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_7 (n-1) = an + b \Rightarrow a = c_1 + c_2 + c_3 + c_4 + c_7, b = c_2 + c_3 + c_4 + c_7 \]

- **Worst case, reverse sort**

  \[ t_j = j \]

  \[ T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2} + c_7 (n-1) = an^2 + bn + c \]
Algorithm analysis

- Worst-case running time is more meaningful since it gives an upper bound on the running time.
- Does the equation give us an accurate estimate of the running time?
- What are the constants?
- Do we care?
Algorithm Analysis

• The actual precise running time is not that important especially since we haven’t said anything about the implementation or runtime platform.

• What’s more interesting to analyze is the order of growth with respect to the input size.

• When looking at the order of growth, we don’t care about the constants.

• An algorithm is considered more efficient than another if it has a lower worst-case order of growth.
Order of growth

- Use $O(g(n))$ notation to represent order of growth

- $O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n > n_0 \}$

- $O(g(n))$ is the upper bound on the worst case running time of the function $f(n)$
Order of growth

- Use the highest degree term in a polynomial $f(n)$
- If $f(n)$ is a sum, use the larger term
- No constants in $g(n)$
Order of growth

• Linear order of growth
  \[ f(n) = an + b \Rightarrow O(n) = f(n) \]

• Quadratic order of growth
  \[ f(n) = an^2 + bn + c \Rightarrow O(n^2) = f(n) \]

• Insertion sort is an \( O(n^2) \) algorithm
Merge sort

- Recursive “divide-and-conquer” approach

```java
merge_sort(A, p, r)
    if p < r
        q = (p+r)/2
        merge_sort(A, p, q)
        merge_sort(A, q+1, r)
        merge(A, p, q, r)
```
Merge sort

6 5 9 7 8 3 4 1

5 6 7 9 8 3 1 4

5 6 7 9 1 3 4 8

1 3 4 5 6 7 8 9
Merge sort

• Assume that the run time of \texttt{merge\_sort} is \(T(n)\)

• The \texttt{merge} routine is \(O(n)\)

• Then, we can write the following recurrence

\[
T(n) = \begin{cases} 
O(1) & \text{if } n = 1 \\
2T\left(\frac{n}{2}\right) + O(n) & \text{if } n > 1
\end{cases}
\]
Merge sort

\[ 6 \quad 5 \quad 9 \quad 7 \quad 8 \quad 3 \quad 4 \quad 1 \]
\[ 5 \quad 6 \quad 7 \quad 9 \quad 8 \quad 3 \quad 1 \quad 4 \]
\[ 5 \quad 6 \quad 7 \quad 9 \quad 1 \quad 3 \quad 4 \quad 8 \]
\[ 1 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \]
\[ 8 \cdot n/8 \]
\[ 4 \cdot n/4 \]
\[ 2 \cdot n/2 \]
\[ n \]
\[ n \log_2 n \]
Merge sort

- Merge sort has an asymptotic upper bound on the running time of $O(n \log n)$

- Since $n \log n < n^2$ as $n$ gets large, we can say that merge sort is more efficient than insertion sort.
Algorithm analysis

• Order of growth analysis allows us to compare different algorithms for a problem

• $1 < \log n < n < n \log n < n^2 < 2^n < n!$
Algorithm Analysis

![Graph showing different time complexities](image-url)
Algorithm Analysis

• Why does order of growth matter?
• With the wrong algorithm, a supercomputer can be slower than an old PC
• Eg. Highly optimized insertion sort on a 10 BIPS supercomputer may take $2n^2$ instructions.
• Poorly optimized merge sort on a 100 MIPS PC takes $50n \log n$ instructions.
• For $n=10^7$, the supercomputer will take roughly 5 1/2 hours, while the PC will take less than 17 minutes.
Next Lecture

• Error Analysis

• Read Chapter 3