Outline

- How do we analyze an algorithm?
- Definition of Big-O notation
- Overview and analysis of sorting algorithms

Importance of Algorithm Analysis (1/2)

- “Performance” of an algorithm refers to how quickly it executes and how much memory it requires
  - performance matters when data grows in size
  - can observe and analyze performance, then revise algorithm to improve its performance
- Algorithm analysis is so important that all Brown CS students are required to take at least one course covering it

Importance of Algorithm Analysis (2/2)

- Factors that affect performance
  - computing resources
  - language
  - implementation
  - size of data, denoted \( N \)
    - number of elements to be sorted
    - number of elements in ArrayList to iterate through
    - much faster to search through list of CS15 students than list of Brown students
- This lecture: a brief introduction to Algorithm Analysis!
- Goal: to maximize efficiency and conserve resources

Runtime

- Runtime of an algorithm varies with the input and typically grows with input size
- In most of computer science we focus on worst case runtime
  - easier to analyze and important for unforeseen inputs
- Average case is what will happen most often. Best case requires least amount of work and is worst situation you could have
  - average case is also important, best case is interesting but not insightful
- How to determine runtime?
  - inspect pseudocode and determine number of statements executed by algorithm as a function of input size
  - allows us to evaluate approximate speed of an algorithm independent of hardware or software environment
  - memory use may be even more important for smaller devices
Elementary Operations
- Algorithmic "time" is measured in elementary operations
  - math (+, -, *, /, max, min, log, sin, cos, abs, ...)
  - comparisons (==, >, <, <=, ...)
  - function (method) calls and value returns (not counting body of the method)
  - variable assignment
  - variable increment or decrement
  - array allocation
  - creating a new object (careful, object's constructor may have elementary ops too!)
- For purpose of algorithm analysis, assume each of these operations takes same time: "1 operation" = we are only interested in "asymptotic performance" for large data sets (small differences don't matter) – see Slide 9

Example: Linear Runtime
// find max of a set of positive integers
public int maxElement(int[] a) {
    int max = 0; // assignment, 1 op
    for (int i=0; i<a.length; i++) { //3 ops per loop
        if (a[i] > max) { //2 ops per loop
            max = a[i]; //2 ops per loop, sometimes
        }
    }
    return max; //1 op
}

Example: Quadratic Runtime
public void printPossibleSums(int[] a) {
    for (int i=0; i<a.length; i++) { //2 ops per loop
        for (int j=0; j<a.length; j++) { // 4 ops per loop
            System.out.println(a[i] + a[j]); // 4 ops per loop
        }
    }
}

Example: Constant Runtime
public int addition(int x, int y) {
    return x+y; //add and return 2 ops
}

Big-O Notation – OrderOf()
- But how to abstract from implementation…?
- Big O notation
  - \( O(N) \) means an operation is done on each element once
    - \( N \) elements \* constant operations/element = \( N \) operations
  - \( O(N^2) \) means each element is operated on \( N \) times
    - \( N \) elements \* \( N \) operations/element = \( N^2 \) operations
  - Only consider "asymptotic behavior" i.e., when \( N \rightarrow \infty \)
    - \( N \) is tiny when compared to \( N^2 \)

Big-O Constants
- Important: Only the largest \( N \) expression without constants matters.
- We are not concerned about runtime with small numbers of data – we care about running operations on large amounts of inputs
  - \( 3N^2 \) and \( 500N^2 \) are both \( O(N^2) \) – unrealistic if \( N \) is small, of course
  - \( N^2 \) is \( O(N) \)
  - \( 4N^2 + 2N \) is \( O(N^2) \)
- Useful sum that recurs frequently in analysis:
  \[ 1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}, \text{ which is } O(N^2) \]
Social Security Database Example (1/3)

- Hundreds of millions of people in the US have a number associated to them
- If 100,000 people named John Smith each has an individual SSN
- If government wants to look up information they have on John Smith, they use his SSN

Social Security Database Example (2/3)

- Say it takes $10^{-4}$ seconds to perform a constant set of operations on one SSN
  - running an algorithm on 5 social security numbers may take $5 	imes 10^{-4}$ seconds, and running an algorithm on 50 will only take $5 	imes 10^{-3}$ seconds
  - both are incredibly fast, a difference in runtime might not be noticeable by an interactive user
  - this changes with large amounts of data, i.e., the actual SS Database

Social Security Database Example (3/3)

- Say it takes $10^{-4}$ seconds to perform a constant set of operations on one SSN
  - to perform an algorithm with $O(N)$ on 300 million people, it will take 8.3 hours
  - $O(N^2)$ takes 285,000 years
- With large amounts of data, differences between $O(N)$ and $O(N^2)$ are HUGE!

Graphical Perspective (1/2)

- $f(N)$ on a small scale

Graphical Perspective (2/2)

- If $N$ on a larger scale
  - If I have 10 million items ($N = 10^7$)
    - $O(\log N)$ runtime, I perform roughly 7 operations
    - $O(N)$ runtime, I perform roughly 10 million operations
    - $O(N^2)$ runtime, I perform roughly 100 trillion operations

Clicker Question (1/3)

What is the big-O runtime of this algorithm?

```java
public int sumArray(int[] array){
    int sum = 0;
    for (int i=0; i<array.length; i++){
        sum = sum + array[i];
    }
    return sum;
}
```

A) $O(N)$ B) $O(N^2)$ C) $O(1)$ D) $O(2^n)$
Clicker Question (2/3)

What is the big-O runtime of this algorithm?

Consider the getColor method in LiteBrite:

```java
public javafx.scene.paint.Color getColor(){
    return _currentColor;
}
```

A) O(N)  B) O(N^2)  C) O(1)  D) O(2^N)

---

Clicker Question (3/3)

What is the big-O runtime of this algorithm?

```java
public int sumSquareArray(int dim, int[][] a){
    int sum = 0;
    for (int i=0; i<dim; i++){
        for (j=0; j<dim; j++){
            sum = sum + a[i][j];
        }
    }
    return sum;
}
```

A) O(N)  B) O(N^2)  C) O(1)  D) O(2^N)

---

Sorting

- We use runtime analysis to help choose the best algorithm to solve a problem.
- Two common problems: sorting and searching through a list of objects.
- This lecture we will analyze different sorting algorithms to find out which is fastest.

Sorting – Social Security Numbers

- Consider an example where run-time influences approach.
- How would you sort every SSN in the Social Security Database in increasing order?
- There are multiple known algorithms for sorting a list. These algorithms vary in their runtime.

Bubble Sort (1/2)

- Iterate through sequence, comparing each element to its right neighbor.
- Exchange adjacent elements if necessary; largest element bubbles to the right.
- End up with a sorted sub-array on the right. Each time we go through the list, need to switch one fewer item.

Bubble Sort (2/2)

- Iterate through sequence, comparing each element to its right neighbor.
- Exchange adjacent elements if necessary; largest element bubbles to the right.
- End up with a sorted sub-array on the right. Each time we go through the list, need to switch one fewer item.
- N is the number of objects in sequence.
Bubble Sort - Runtime

Worst-case analysis (sorted in inverse order):
- the while-loop is iterated N-1 times
- iteration i has 2 + 6(i-1) operations

Total:
2 + N + 2(N-1) + 6[(N-1) + ... + 2 + 1]
= 3N^2 + ... = O(N^2)

Insertion Sort (1/2)
- Like inserting a new card into a partially sorted hand by
  bubbling to the left into a sorted subarray. Less brute-force than bubble sort
- Add one element a[i] at a time
- Find proper position, j+1, to the left by shifting to the
  right a[i-1], a[i-2], ..., a[j+1] left neighbors, until a[j] < a[i]
- Move a[i] into vacated a[j+1]
- After iteration i, the original a[0]...a[i] are in
  sorted order, but not necessarily in final position

Insertion Sort (2/2)
```c
for (int i = 1; i < n; i++) {
    int toInsert = a[i];
    int j = i - 1;
    while ((j >= 0) && (a[j] > toInsert)) {
        move a[j] forward;
        j--;
    }
    move toInsert to a[j+1];
}
```

Selection Sort (1/2)
- Find smallest element and put it in a[0]
- Find 2nd smallest element and put it in a[1], etc
- Less data movement (no bubbling)

Selection Sort (2/2)
```c
for (int i = 0; i < n-1; i++) {
    for (int j = i+1; j < n; j++) {
        if (a[j] < a[min]) {
            min = j;
        }
    }
    temp = a[min];
    a[min] = a[i];
    a[i] = temp;
}
```
Selection Sort - Runtime

- Most executed instructions are those in inner for loop
- Each such instruction is executed \((N - 1) + (N - 2) + ... + 2 + 1\) times
- Time Complexity: \(O(N^2)\)

```java
for (int i = 0; i < n - 1; i++) {
    int min = i;
    for (int j = i + 1; j < n; j++) {
        if (a[j] < a[min]) {
            min = j;
        }
    }
    temp = a[min];
    a[min] = a[i];
    a[i] = temp;
}
```

Comparison of Basic Sorting Algorithms

- Differences in Best and Worst case performance result from the state (ordering) of the input before sorting
- Selection Sort wins on data movement
- For small data, even the worst sort – Bubble – is fine!

<table>
<thead>
<tr>
<th></th>
<th>Selection</th>
<th>Insertion</th>
<th>Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>(n/2)</td>
<td>(n/2)</td>
<td>(n)</td>
</tr>
<tr>
<td>Average</td>
<td>(n/2)</td>
<td>(n/4)</td>
<td>(n/4)</td>
</tr>
<tr>
<td>Worst</td>
<td>(n/2)</td>
<td>(n/2)</td>
<td>(n/2)</td>
</tr>
</tbody>
</table>

Merge Sort

Recap: Recursion (1/2)

- Recursion is a way of solving problems by breaking them down into smaller sub-problems, and using the results of the sub-problems to find the answer
- Example: You want to determine what row number you’re sitting in (in Salomon), but you can only get information from asking the people in front of you
  - they also don’t know what row they’re in, and ask the people in front of them
  - people in the front know that they’re row 1, since there is no row in front
  - they tell the people behind them, who know that they’re 1 behind row 1, so row 2

```
public int findRowNumber(Row myRow) {
    if (myRow.getRowAhead() == null) { // base case
        return 1;
    } else { // recursive case - ask the row in front
        int rowAheadNum = this.findRowNumber(myRow.getRowAhead());
        // my row number is one more than the row ahead’s number
        return rowAheadNum + 1;
    }
}
```

Recap: Recursion (2/2)

- Partition sequence into two sub-sequences of \(N/2\) Elements.
- Recursively partition and sort each sub-arrays.
- Merge the sorted sub-arrays.
Recursive (Top Down) Merge Sort (2/6)

- Partition sequence into two sub-sequences of N/2 Elements.
- Recursively partition and sort each sub-arrays.
- Merge the sorted sub-arrays.

Recursive (Top Down) Merge Sort (3/6)

```java
public class Sorts {
    public ArrayList<Integer> mergeSort(ArrayList<Integer> list) {
        if (list.size() == 1) {
            return list;
        }
        int middle = list.size() / 2;
        ArrayList<Integer> left = this.mergeSort(list.subList(0, middle));
        ArrayList<Integer> right = this.mergeSort(list.subList(middle, list.size()));
        return this.merge(left, right);
    }
}
```

Recursive (Top Down) Merge Sort (4/6)

```java
public class Sorts {
    public ArrayList<Integer> mergeSort(ArrayList<Integer> list) {
        if (list.size() == 1) {
            return list;
        }
        int middle = list.size() / 2;
        ArrayList<Integer> left = this.mergeSort(list.subList(0, middle));
        ArrayList<Integer> right = this.mergeSort(list.subList(middle, list.size()));
        return this.merge(left, right);
    }

    // code for merge() coming next!
}
```

Recursive (Top Down) Merge Sort (5/6)

```java
public ArrayList merge(ArrayList<Integer> A, ArrayList<Integer> B) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    int aIndex = 0;
    int bIndex = 0;
    while (aIndex < A.size() && bIndex < B.size()) {
        if (A.get(aIndex) <= B.get(bIndex)) {
            result.add(A.get(aIndex));
            aIndex++;
        } else {
            result.add(B.get(bIndex));
            bIndex++;
        }
    }
    if (aIndex < A.size()) {
        result.addAll(A.subList(aIndex, A.size()));
    }
    if (bIndex < B.size()) {
        result.addAll(B.subList(bIndex, B.size()));
    }
    return result;
}
```

Recursive (Top Down) Merge Sort (6/6)

- Each part of the tree performs \( n \) operations to merge the two subproblems below it.
- Because we divide each sequence by two, the algorithm makes \( \log N \) merges.
- \( O(N \log N) \) which is way better than \( O(N^2) \).
- We can also drop the log base (2) and say \( O(N \log N) \), like how we can remove constants.
- You will learn much more about how to find the runtime of these types of algorithms if you take CS 16!

Iterative (Bottom Up) Merge Sort

- Merge sort can also be implemented iteratively...non-recursive!
- Begin by looping through the array of size \( N \), sorting 2 items each. Loop through the array again, combining the 2 sorted items into a sorted item of size 4. Repeat... until there is a single item if size \( N \! \)
- Number of iterations is \( \log N \), rounded up to nearest integer. 1000 elements in the list, only 10 iterations!!!
Comparing Algorithms Side by Side

Comparing Algorithms Side by Side

Insertion Sort – \(O(N^2)\)

Bubble Sort – \(O(N^2)\)

Merge Sort – \(O(N \log N)\)

Clicker Question

Which sorting algorithm is the fastest?
A. Bubble Sort
B. Insertion Sort
C. Merge Sort
D. Selection Sort

That's It!

- Runtime is a very important part of Algorithm analysis!
  - worst case run-time is what we generally focus on
  - know the difference between constant, linear, and quadratic run-time
  - calculate/ define run-time in terms of Big-O Notation

- Sorting!
  - runtime analysis is very significant for Sorting Algorithms!
  - types of sorting algorithms - bubble, insertion, selection, merge sort
  - different algorithms have different performances and time complexities.

What’s next?

- You have now seen how different approaches to solving problems can dramatically affect speed of algorithms
  - this lecture utilized arrays to solve most problems

- Subsequent lectures will introduce more data structures beyond arrays that can be used to handle collections of data

- We can use our newfound knowledge of algorithm analysis to strategically choose different data structures to further speed up algorithms!

Announcements

- Optional Q&A Review sessions in Macmillan 115 tonight at 7:30 and Sunday at noon - focused on recursion

- DoodleJump due dates:
  - early: Tuesday 11/1, 11:59pm
  - on-time: Thursday 11/3, 11:59pm
  - late: Saturday 11/5, 10:00pm

- Start early, start today, start yesterday!