Big-O and Sorting
Lecture 15

Outline
- Importance of Algorithm Analysis
- Runtime
  - TopHat: slide 18, 19, 20
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- TopHat: slide 45

Importance of Algorithm Analysis (1/2)
- **Performance** of algorithm refers to how quickly it executes and how much memory it requires
  - performance matters when amount of data gets large!
  - can observe and analyze performance, then revise algorithm to improve its performance
- Algorithm analysis is so important that it is taught at one point or another in every intro sequence!
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 \texttt{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

Performance of Algorithms

- How fast will \( n! \) run relative to \( n \)?
  - \( n! \) will take exponentially longer as \( n \) increases
  - less difference with small \( n \) but we care about large inputs
  - one algorithm could take 2 seconds and another take 1 hour to accomplish the same task
- How fast will recursive Fibonacci(\( n \)) run relative to \( n \)?
  - proportional to \( 2^n \) – consider how fast this grows with \( n \)
- How fast will \textit{Towers of Hanoi} run relative to the number of disks?
  - proportional to \( 2^n \) for \( n \) disks

Runtime (1/2)

- In analyzing an algorithm, runtime is the total number of times "the principle activity" of all steps in that algorithm are performed
  - varies with 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 typically happen. Best case requires least amount of work and is the best situation you could have
  - average case is also important; best case is interesting, but not insightful
Runtime (2/2)
- 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 than runtime for embedded devices

Elementary Operations
- Algorithmic ‘time’ is measured in numbers of elementary operations
  - math (+, -, /, max, min, log, sin, cos, abs, ...)
  - comparisons ( ==, >, <=, ...)
  - function (method) calls and value returns (body of the method is separate)
  - variable assignment
  - variable increment or decrement
  - array allocation (declaring an array) and array access (retrieving an array from memory)
  - 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, i.e., as N grows large (small differences in performance don’t matter)

Example: Constant Runtime

```java
public int addition(int x, int y) {
    return x + y; // 2 operations
}
```
- Always 2 operations – 1 addition, 1 return statement
- How many operations are performed if this function were to add ten integers? Would it still be constant runtime?
Example: Linear Runtime

```java
//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++) { //2 ops per iteration
        if (a[i] > max) { //2 ops per iteration, sometimes
            max = a[i]; //2 ops per iteration, sometimes
        }
    }
    return max; //1 op
}
```

- Worst case varies proportional to the size of the input list: \(6N + 3\)
- How many operations if the array had 1,000 elements?
- We’ll run the for loop proportionally more times as the input list grows
- Runtime increase is proportional to \(N\), linear

Example: Quadratic Runtime

```java
public void printPossibleSums(int[] a) {
    for (i = 0; i < a.length; i++) { //2 ops per iteration
        for (j = 0; j < a.length; j++) { //2 ops per iteration
            System.out.println(a[i] + a[j]); // 4 ops per iteration
        }
    }
}
```

- Requires about \(8N^2\) operations (it is okay to approximate)
- Number of operations executed grows quadratically!
- If one element added to list, element must be added with every other element in list
- Notice that linear runtime algorithm on previous slide had only one for loop, while this quadratic one has two nested for loops, a typical \(N^2\) pattern

Big-O Notation – OrderOf()

- But how to abstract from implementation…?
- Big O notation
- \(O(N)\) implies runtime is linearly proportional to number of elements/inputs in the algorithm (constant ops per element)
  - \((N\text{ elements}) \times (\text{constant ops/element}) = N \text{ operations}\)
- \(O(N^2)\) implies each element is operated on \(N\) times
  - \((N\text{ elements}) \times (N \text{ operations/element}) = N^2 \text{ operations}\)
- Only consider "asymptotic behavior" i.e., when \(N >> 1\)
- \(O(1)\) implies that runtime does not depend on number of inputs
  - runtime is the same regardless of how large/small input size is
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 $O(N^2)$, because the larger the input, the less the “500” and the “3” will contribute towards the total runtime
  - $N/2$ is $O(N)$
- $4N^2 + 2N$ is $O(N^2)$
- Useful sum for analysis:
  \[
  1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = \frac{N(N+1)}{2},
  \]
  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 are named John Doe, each has an individual SSN
- If the government wants to look up information they have on John Doe, 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 SSNs will take $5 \times 10^{-4}$ seconds, and running an algorithm on 50 will only take $5 \times 10^{-3}$ seconds
  - both are incredibly fast, 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 algorithm with $O(N)$ on 300 million people will take 8.2 hours
  - $O(N^2)$ takes 285,600 years
- With large amounts of data, differences between $O(N)$ and $O(N^2)$ are HUGE!

Graphical Perspective (1/2) – Linear Plot

- $f(N)$ on a small scale →

Graphical Perspective (2/2) – Log Plot

- $f(N)$ on a larger scale →
- For 10 million items ($N = 10^7$)...  
  - and $O(\log(N))$ runtime, perform roughly 7 operations  
  - and $O(N)$ runtime, perform roughly 10 million operations  
  - and $O(N^2)$ runtime, perform roughly 100 trillion operations
- really try to stay sub-quadratic!!
TopHat 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)

TopHat 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)

TopHat 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 (int j = 0; j < dim; j++){
            sum = sum + a[j][i];
        }
    }
    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 your approach.
- How would you sort every SSN in the Social Security Database in increasing order?
- Multiple known algorithms for sorting a list:
  - These algorithms vary in their Big-O 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 sorted sub-array on the right. Each time we go through the list, need to switch at least one item fewer than before.

- 49 2 36 55 4 72 23
  - Before a pass
  - 2 36 49 55 4 72 23
  - Middle of first pass
  - 2 36 49 4 55 23 72
  - After one pass
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 sorted sub-array on the right. Each time we go through the list, need to switch at least one item fewer than before
- N is number of objects in the sequence

```
int i = array.length;
boolean sorted = false;
while ((i > 1) && (!sorted)) {
    sorted = true;
    for(int j = 1; j < i; j++) {
        if (a[j-1] > a[j]) {
            int temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
            sorted = false;
        }
    }
    i--;
}
```

Bubble Sort - Runtime

<table>
<thead>
<tr>
<th>Op.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int i = array.length;</td>
</tr>
<tr>
<td>1</td>
<td>boolean sorted = false;</td>
</tr>
<tr>
<td>N</td>
<td>while ((int i &gt; 1) &amp;&amp; (!sorted)) {</td>
</tr>
<tr>
<td>N-1</td>
<td>sorted = true;</td>
</tr>
<tr>
<td>2N-1</td>
<td>for(int j = 1; j &lt; i; j++) {</td>
</tr>
<tr>
<td>2</td>
<td>if (a[j-1] &gt; a[j]) {</td>
</tr>
<tr>
<td>2</td>
<td>int temp = a[j-1];</td>
</tr>
<tr>
<td>2</td>
<td>a[j-1] = a[j];</td>
</tr>
<tr>
<td>2</td>
<td>a[j] = temp;</td>
</tr>
<tr>
<td>1</td>
<td>sorted = false;</td>
</tr>
<tr>
<td>i-1</td>
<td>}</td>
</tr>
</tbody>
</table>

Worst-case analysis:
- while loop iterates N-1 times
- Iteration i has $2 + 13(i-1)$ operations

Total:
$2 + 2(2N-1) + 13(2N-2) + ... + 2 + 1 = 3N^2 + 9N(N-1)/2 = O(N^2)$

Insertion Sort (1/2)
- Like inserting a new card into a partially sorted hand by bubbling to the left in 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 neighbors on the left ($a[j-1], a[j-2], ... a[j+1]$, move $a[j]$ to the right, until $a[j] < a[i]$
- Move $a[i]$ into vacated $a[j+1]$
- After iteration $i < a.length$, original $a[0], ... a[i]$ are in sorted order, but not necessarily in final position
Insertion Sort (2/2)

```java
for (int i = 1; i < a.length; i++) {
    int toInsert = a[i];
    int j = i - 1;
    while ((j >= 0) && (a[j] > toInsert)) {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = toInsert;
}
```

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)

```
a = [2, 4, 6, 5, 9, 7, 23]
```
Selection Sort (2/2)

What we want to happen:

```java
int n = a.length;
for (int i = 0; i < n; 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;
}
```

Selection Sort - Runtime

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

Comparison of Basic Sorting Algorithms

- Differences in **Best** and **Worst** case performance result from state (ordering) of input before sorting
- Selection Sort wins on data movement
- For small data, even the worst sort – Bubble (based on comparisons and movements) – is fine!
Recap: Recursion (1/2)

- Recursion is a way of solving problems by breaking them down into smaller sub-problems, and using results of sub-problems to find the answer

- Example: You want to determine what row number you're sitting in (in Salomon DECI), but you can only get information by asking the people in front of you
  - they also don't know what row they're in, and must ask people in front of them
  - people in first row know that they're row 1, since there is no row in front
  - they tell people behind them, who know that they're 1 behind row 1, so they are row 2, etc.
  - this "unwinds" the recursion

Recap: Recursion (2/2)

```java
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());
        return rowAheadNum + 1;
    }
}
```
Let's say you don't know how to sort n elements, but you have a friend who can sort any number less than n. How can you use the results to do your work? (similar to row problem)
- one answer is to sort n-1, then just slot the last element into the sorted order (insertion sort)
- another answer is for you to pick the smallest single entry, then give remaining elements to your friend to sort and add your element to the beginning of her results (selection sort)
- What if your friend can only sort things of size n/2 or smaller? She can sort the two pieces... can we quickly make a sorted list from what's left? (merge sort)

Recursion (Top Down) Merge Sort (1/7)
- Partition sequence into two sub-sequences of N/2 elements
- Recursively partition and sort each sub-array
- Merge the sorted sub-arrays

Recursion (Top Down) Merge Sort (2/7)
- Partition sequence into two sub-sequences of N/2 elements
- Recursively partition and sort each sub-array
- Merge the sorted sub-arrays

Recursion (Top Down) Merge Sort (3/7)
- Partition sequence into two sub-sequences of N/2 elements
- Recursively partition and sort each sub-array
- Merge the sorted sub-arrays

Figure: Merge sort divide and conquer
Recursion (Top Down) Merge Sort (4/7)

```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!
}
```

Recursion (Top Down) Merge Sort (5/7)

```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!
}
```

Recursion (Top Down) Merge Sort (6/7)

```java
public ArrayList<Integer> 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;
}
```

- Add elements from the two sequences in increasing order.
- If there are elements left that you haven’t added, add the remaining elements to your result.
Each level of the tree performs \( n \) operations to merge and sort the subproblems below it.

Each time you merge, you have to handle all the elements of the sub-arrays you’re merging, hence \( O(N) \). Recursion adds to runtime, but not much.

There are \( \log N \) merge passes.

Thus, \( O(N \log N) \) — way better than \( O(N^2) \).

- Can also drop \( \log \) base \( 2 \) and say \( O(N \log N) \), since we can ignore constants.

Learn much more about how to find the runtime of these types of algorithms in CS16!

Merge sort can also be implemented iteratively... non-recursive!

Loop through array of size \( N \), sorting 2 items each. Loop through the array again, combining the 2 sorted items into sorted item of size 4. Repeat, until there is a single item of size \( N \! \). 

Number of iterations is \( \log N \), rounded up to nearest integer. 1000 elements in the list, only 10 iterations!!

Iterative merge sort avoids the nested method invocations caused by recursion!

Comparing Sorting Algorithms

- Bubble Sort: \( O(N^2) \)
- Insertion Sort: \( O(N^2) \)
- Merge Sort: \( \mathcal{O}(N \log N) \)

Live Demo:
http://cs.brown.edu/courses/cs015/demos/Rainbow.jar
TopHat Question
Which sorting algorithm that we have looked at is the fastest (in terms of worst case runtime)?
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 runtime is what we generally focus on
  - know the difference between constant, linear, and quadratic run-time
  - calculate/define runtime in terms of Big O Notation
- Sorting!
  - runtime analysis is very significant for sorting algorithms
  - types of simple sorting algorithms - bubble, insertion, selection, merge sort
  - fancier sorts perform even better, but tough to analyze, e.g., QuickSort
  - 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

- DoodleJump Help Session **TONIGHT**
  - **time:** 8:00-9:30 PM
  - **location:** MacMillan 117
- Help Slides released after Help Session
- Section Slides released after last Section today
- HTA office hours
  - **Tuesday 4:30-6:30pm**