Big-O and Sorting

Lecture 15
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 the data grows in size!
  - can observe and analyze the performance, then revise the 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
  - implementation (like machine, language)
  - 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 the **worst case runtime**
  - easier to analyze and important for unforeseen inputs.
- **Average case** is what will happen most often. **Best case** requires the least amount of work and is the best situation you could have.
  - average case is also important, best case is interesting but not insightful.
- **How to determine runtime?**
  - inspect **pseudocode** and determine the number of statements executed by the algorithm as a function of input size.
  - allows us to evaluate approximate speed of an algorithm independent of the 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 the 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 where small differences don’t matter – see S9
Example: Constant Runtime

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

- **always** 2 operations performed
- How many operations performed if this function were to add ten integers? Would it still be constant runtime?

1 – Note that this is *technically* 3 operations, because the computer first saves x in a ‘register’ then adds y to the value stored in this register, then returns this sum. This is beyond the scope of CS15 - Take CS33 to find out more! – 2 op, 3 op are both constant runtime!
Example: Linear Runtime

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

- How many operations if the list had 1,000 elements?
- Worst case varies proportional to the size of the input list: \(5n + 2\)
- We’ll be in the for loop longer as the input list grows
- The runtime increase is proportional to \(N\), linear

Only the largest \(N\) expression without constants matters!
5n+2, 4n, 300n are all linear in runtime. More about this on following slides!
Example: Quadratic Runtime

```java
class Example {
    public void printPossibleSums(int[] a) {
        int len = a.length; //1 op; storing avoids .length in exit check
        for (int i = 0; i < len; i++) {
            //2 op per loop
            for (int j = 0; j < len; j++) {
                //2 op per loop
                System.out.println(a[i] + a[j]); // 4 ops per loop
            }
        }
    }
}
```

- Requires about $8n^2 + 1$ 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
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 * 1 operation/element = N operations
- **O(N^2)** means each element is operated on N times
  - N elements * N operations/element = N^2 operations
  - memory use may be even more important for smaller devices
- Only consider “**asymptotic behavior**” i.e., when N>>1 (N is much greater than 1)
  - N is unimportant 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:
  \[
  1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = \frac{N(N+1)}{2}, \text{ which is } O(N^2)
  \]
Social Security Database Example 1/3

- Hundreds of millions of people in US, each 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 $5 \times 10^{-4}$ seconds, and running an algorithm on 50 will only take $5 \times 10^{-3}$ 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 **linear** graph paper
Graphical Perspective 2/2

- $f(N)$ on log-log graph paper
- **Base** for logs often depends on the data structures we use in our algorithms
Quiz! 1/3

● What is the big-O runtime of this algorithm?

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

● Answer: \( O(N) \)
Quiz! 2/3

● What is the big-O runtime of this algorithm?

● Consider the getColor method in LiteBrite

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

● Answer: O(1)
Quiz! 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;
}
```

- Answer: O(N²)
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

```java
i = N;
sorted = false;
while((i > 1) && (!sorted)) {
    sorted = true;
    for(int j=1; j<i; j++) {
        if (a[j-1] > a[j]) {
            temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
            sorted = false;
        }
    }
    sorted = false;
    i--;
}
```
Bubble Sort - Runtime

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

Worst-case analysis (sorted in inverse order):

- the **while**-loop is iterated N-1 times
- iteration i has $2 + 6(i - 1)$ operations

**Total:**

$$2+N+2(N-1)+6[(N-1)+...+2+1]=3N+6N(N-1)/2 = 3N^2+... = O(N^2)$$
Insertion Sort 1/2

- Like inserting a new card into a partially sorted hand by bubbling to the left into a sorted subarray; little 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 $i<n$, the original $a[0] \ldots a[i]$ are in sorted order, but not necessarily in final position
Insertion Sort 2/2

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];
}
Insertion Sort - Runtime

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

What we want to happen:

```java
for (int i = 0; i < n; i++) {
    find minimum element \(a[\text{min}]\) in subsequence \(a[i...n-1]\)
    swap \(a[\text{min}]\) and \(a[i]\)
}
```

```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;
}
```
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 – if fine!

<table>
<thead>
<tr>
<th></th>
<th>Selection</th>
<th>Insertion</th>
<th>Bubble</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparisons</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>$\frac{n^2}{2}$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Average</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{4}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td>Worst</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td><strong>Movements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>$n$</td>
<td>$\frac{n^2}{4}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
<tr>
<td>Worst</td>
<td>$n$</td>
<td>$\frac{n^2}{2}$</td>
<td>$\frac{n^2}{2}$</td>
</tr>
</tbody>
</table>
Merge Sort
Recursive (Top Down) Merge Sort 1/6

- **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 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 {
    // other code here

    public void mergeSort(ItemSequence listSequence, int first, int last) {
        if (first < last) {
            int middle = (first + last) / 2;
            mergeSort(listSequence, first, middle);  // recursively mergeSort sub-sequences
            mergeSort(listSequence, middle+1, last);
            merge(listSequence, first, middle, last);  // merges the two sub-sequences
        }
    }
}
```

- `listSequence` is the sequence to sort.
- `first` and `last` are smallest and largest indices of sequence.
public class Sorts {
    // other code here
    public void mergeSort(ItemSequence listSequence, int first, int last) {
        if (first < last) {
            int middle = (first + last) / 2;
            mergeSort(listSequence, first, middle);
            mergeSort(listSequence, middle+1, last);
            merge(listSequence, first, middle, last);
        }
    }
}
Recursive (Top Down) Merge Sort 5/6

public ItemSequence merge(ItemSequence A, ItemSequence B){
    result = new ItemSequence()
    int aIndex = 0;
    int bIndex = 0;
    while (aIndex < A.length and bIndex < B.length){
        if (A[aIndex] <= B[bIndex]){
            add A[aIndex] to result;
            aIndex++;                
        }
        else {
            add B[bIndex] to result;
            bIndex++;
        }
    }
    if (aIndex < A.length){
        result = result + A[aIndex...end];
    }
    if (bIndex < B.length){
        result = result + B[bIndex...end];
    }
    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 \( \log_2 N \) merges.
- \( O(N \log_2 N) \) which is way better than \( O(N^2) \)
- You will learn much more about how to find the runtime of these types of algorithms in CS16!
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 $\log_2 N$, rounded up to nearest integer. 1000 elements in the list, *only 10* iterations!!!
Comparing Algorithms Side by Side

Bubble Sort – $O(N^2)$

Insertion Sort – $O(N^2)$

Merge Sort – $(N\log_2 N)$
That’s It!

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

● Sorting!
  o Runtime Analysis is very significant for Sorting Algorithms!
  o Types of Sorting Algorithms - Bubble, Insertion, Selection, Merge Sort
  o 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 **now** (4pm) in Barus & Holley 166

• Optional Q&A Review session in CIT 201 on Saturday 1pm-2:30pm (tentative)

• Hours lines have been short – good time to ask conceptual questions

• Cartoon due dates:
  
  early: Wed 11/4  
  on-time: Friday 11/6  
  late: Sunday 11/8