Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

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 analyze and observe performance, then revise algorithm to improve

- Algorithm analysis and sorting/searching data structures are crucial to computing and will be a central topic in CS0200!
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 in alphanumeric order
    - number of elements in ArrayList to iterate through

- This lecture: a brief introduction to Algorithm Analysis!

- Goal: maximize efficiency and conserve resources

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

Performance of Algorithms

- How fast will recursive Fibonacci(N) run relative to N?
  - \( F(n) = F(n-1) + F(n-2) \)
  - \( F(n-1) = F(n-2) + F(n-3) \), etc.

- How fast will Towers of Hanoi run relative to the number of disks?
  - \( 2^n \) disks

- How fast will N! run relative to N?
  - \( N! \) will take exponentially longer as N increases, even faster than \( 2^n \)
  - not a problem with small N, but we care about large inputs

- One algorithm could take 2 seconds while another could take 1 hour to accomplish the same task.
In analyzing an algorithm, runtime is the total number of times "the principal activity" of all steps in that algorithm is performed:
- varies with input and almost always grows with input size $N$
- measured as a function of $N$ ($N$, $N^2$, $2^N$, etc.)

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 important; best case is interesting, but not insightful

How to determine runtime?
- inspect pseudocode (or actual code if it is small enough) and determine number of elementary operations in all 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

Algorithmic "time" is measured in numbers of elementary operations:
- math (+, -, *, /, max, min, log, sin, cos, abs, ...)
- comparisons (==, >, <=, ...)
- function (method) calls and value return (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".
- only relevant for "asymptotic performance" for large data sets, i.e., as $N$ grows large
- usual choice is a performance test under worst case for sets of billions or even trillions of data, e.g., sorting all the words on the WWW.
Example: Constant Runtime

- 4 operations – 1 addition, 2 array element retrievals, 1 return statement
- How many operations are performed if the input list had 1000 elements? 10,000?
- Runtime is constant

Example: Linear Runtime

- Word 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

- 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

- Used to abstract from implementation by ignoring constants!
- $O(N)$ implies runtime is linearly proportional to number of elements/inputs in the algorithm (constant operations per element)
- $O(N)$ implies each element is operated on $N$ times
- $O(N)$ implies that runtime does not depend on number of inputs
- Only consider "asymptotic behavior" i.e., when $N \gg 1$

Big-O Constants

- Important: Only the largest $N$ expression without constants matters
- We are not concerned about runtime with small numbers of data
- Only consider 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 affect the total runtime
  - $2N$ is $O(N)$
  - $4N^2 + 2N$ is $O(N^2)$
- Useful sum for analysis:
  $$1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = N(N+1)/2$$
  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 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 1 SSN will take 10^{-4} seconds.
  - If we run the algorithm on 10 SSNs, it will take 10^{-3} seconds.
  - If we run it on 100 SSNs, it will take 10^{-2} seconds.
- Note: the actual SS Database is incredibly fast, and the difference in runtime might not be noticeable to an interactive user.
- This changes with large amounts of data, i.e., the actual SS Database.

Social Security Database Example (3/3)

- Say we want to scale this algorithm to every SSN (300+ million).
  - To perform algorithm with O(N) on 300 million people 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) – Linear Plot

- f(N) on a small scale →
Graphical Perspective (2/2) – Log Plot

- \( f(N) \) on a larger scale →
- For 10 million items (\( N = 10^7 \))…
  - \( O(\log N) \) runtime, perform roughly 7 operations
  - \( O(N) \) runtime, perform roughly 10 million operations
  - \( O(N^2) \) 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 getLetter() (or equivalent) method from TicTacToe:

```java
public String getLetter(){
    return this.letter;
}
```

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[][] a){
    int sum = 0;
    for (int i = 0; i < a.length; i++){
        for (int j = 0; j < a[i].length; j++){
            sum = sum + a[j][i];
        }
    }
    return sum;
}
```

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

Outline

- Importance of Algorithm Analysis
- Runtime
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Merge Sort

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
- 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/swap 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

```
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 (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 left. Each time we go through the list, need to switch at least one item fewer than before
- More efficient version: keep track of last largest element inserted so we don’t have to go all the way over to the right

```
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>Case</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>N(N-1)/2</td>
</tr>
<tr>
<td>Average case</td>
<td>N^2</td>
</tr>
<tr>
<td>Best case</td>
<td>N</td>
</tr>
</tbody>
</table>

- **Worst case:**
  - **while** loop iterates N-1 times
  - **for** loop iterates N-1 times
  - Total: N(N-1)/2

- **Average case:**
  - **while** loop iterates N(N-2)/2
  - **for** loop iterates N/2
  - Total: N^2

- **Best case:**
  - **while** loop iterates 0 times
  - **for** loop iterates 0 times
  - Total: N

**Remember!**
Small operations and constants don't majorly affect runtime, so we can ignore them when looking at big-O.

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
Insertion Sort (1/2)

- Like inserting a new card into a partially sorted hand by bubbling to the left in a sorted subarray
  - close to bubble sort but less brute force because we don’t start always from the rightmost entry
- Add one element $a[i]$ at a time
- Find proper position, $j = i-1$, to the left by swapping with neighbors on the left ($a[i-1]$, $a[i-2]$, ..., $a[j+1]$) to the right, until $a[j] < a[i]$
- Move $a[j]$ into vacated $a[j+1]$
- After iteration $1 < i < a.length$, original $a[0]$, ..., $a[i]$ may be sorted but not necessarily in their position, depending on what comes after $a[i]$

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;
}
```
**Insertion Sort - Runtime**

```
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;
}
```

- while loop inside our for loop
- for loop adds N operations
- for loop calls the while loop N times

**O(N^2)** because we have to call on a while loop with ~N operations N different times

Reminder: constants do NOT matter with Big-O!

**Outline**

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

**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 movement)
Selection Sort (2/2)

What we want to happen:

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

Selection Sort - Runtime

- Most executed instructions are in if statement 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 are based on current order 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!

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparisons of data</th>
<th>Movements of data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>(n^2)</td>
<td>(n^2/2)</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
</tbody>
</table>
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, but you can only get information by asking the people in front of you.
  - They also don’t know what row they’re in, so must ask people in front of them.
  - People in first row know that they’re row 1, since there is no row in front (base case).
  - People behind them 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
        int rowAheadNum = this.findRowNumber(myRow.getRowAhead());
        // my row number is one more than the row ahead's number
        return rowAheadNum + 1;
    }
}
```

Recursion (Top Down) Merge Sort (1/7)

- 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? (like auditorium row number problem)
  - One answer is to sort n-1, then just slot the last element into the sorted order (insertion sort).
  - Another answer is 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 (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 number of elements
- Recursively partition and sort each sub-array
- Merge the sorted sub-arrays

Recursion (Top Down) Merge Sort (4/7)

```
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);
    }
    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

Recursion to get down to base case is just halving each subarray: \(O(\log n)\)

- Unwinding the recursion: Each level of the tree performs \(n\) operations to merge and sort the subproblems below
- Each time you merge, you must handle all the elements in the sub-arrays you’re merging, hence \(O(n)\)

- There are \(\log k\) number of merge passes, thus, \(O(\log n) \cdot O(n) = \Theta(n \log n)\)
  - way better than \(O(n^2)\)
  - drop base (2) and say \(O(\log n)\), ignore constants

Learn much more about how to find the runtime of these types of algorithms in CS200!
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 \( \log_2 N \), rounded up to nearest integer. 1000 elements in the list, only 10 iterations!

Iterative merge sort avoids the nested method invocations caused by recursion!
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/run-time 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 and loops to solve most problems
- Subsequent lectures will introduce more data structures beyond arrays and arraylists 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 late deadline tomorrow 11/3 @ 11:59pm
- DoodleJump Code Debriefs Coming Up
  - Keep an eye on your email to see if you were selected
  - If you are selected and miss this debrief you will receive a minus four deduction on your final grade!!!!
- Tetris out Saturday!!!
  - you do NOT want to procrastinate on this assignment
  - the earlier you start, the shorter the lines at debugging hours ☺
  - Please reference the collaboration policy when working on Solo Tetris!!

Socially Responsible Computing

Dark & Addictive Design

Definition

Dark patterns are features of interactive design crafted to trick users into doing things they might not wish to do, but which benefit the business in question.

Term coined by Harry Brignull (UX Specialist) in 2010
Nudging users to give up privacy

More intuitive to tap on the blue box using terms like "personalization" instead of "targeting" or "tracking.

Source: Meta

Manipulating User Psychology

More intuitive to tap the big, dark box if manipulating user psychology.

Source: Florsheim Shoe Company

Addictive patterns in action: TikTok

Variable rewards/ slot machine: Better not knowing what you will get when you swipe.

Infinite scrolling / auto-play feature encourages you to passively engage with the app & makes it hard to stop.

Hick's Law: The more choices a user has, the less likely they are to make one.

Manipulating user psychology restricts user choice by limiting the ways in which you can interact with the app, so you can only scroll.
Infinite scrolling to stimulate “stickiness”

Impact of addictive patterns on the brain

Meta Lawsuit
Regulation

FTC Report Shows Rise in Sophisticated Dark Patterns Designed to Trick and Trap Consumers

Tactics include disguised ads, difficult-to-cancel subscriptions, buried terms, and tricks to obtain data.

Source: FTC.gov (2023)

Issue with defining dark patterns

“All design has a level of persuasion to it. The difference is, if you’re designing to trick people, you’re an asshole.”


67

FTC Regulation

$4.5 million FTC settlement aligns Fortnite owner Epic Games

and digital dark patterns to charge players for unwanted in-
game purchases

Source: FTC.gov (2023)

FTC Sends Nearly Two Million in Refunds to Vintage Consumers Who Were Trapped in Subscriptions By Dark Patterns and Junk Fees

Source: FTC.gov (2023)

68

69
What is Dash?
- MERN stack web application (MongoDB, Express, React, Node.js), hypertext/hypermedia system
- produce, annotate, and consume digital documents

What would I be doing?
- This semester:
  - using Dash and providing feedback
  - becoming familiar with the system, codebase, and technologies used
- Winter break:
  - complete the starter project to join as full member
- Next semester:
  - work on Dash as an independent study, building your very own feature!

Join Dash!
Interest form: