Big-O and Sorting

Lecture 15
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 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
  o computing resources
  o language
  o implementation
  o 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
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 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 is 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 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

- 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?

```java
public int addition(int x, int y) {
    return x + y; // 2 operations
}
```
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++){
        //2 ops per iteration
        if (a[i] > max) {
            //2 ops per iteration
            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

Only the largest N expression without constants matters!
6N+3, 4N, 300N are all linear in runtime.
More about this on following slides!
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 elements) * (constant ops/element) = N operations
- **O(N^2)** implies each element is operated on N times
  - (N elements) * (N operations/element) = N^2 operations
- Only consider “asymptotic behavior” i.e., when N >> 1
  - N is tiny when compared to N^2 for 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
  
  o $3N^2$ 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

  o $N/2$ is $O(N)$

  o $4N^2 + 2N$ is $O(N^2)$

● Useful sum for analysis:

  $$1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = \frac{N(N+1)}{2}, \text{ which is } O(N^2)$$

Andries van Dam © 2021 3/16/21
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
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 8.3 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](image)
Graphical Perspective (2/2) – Log Plot

- $f(N)$ on a larger scale →
- For 10 million items ($N = 10^7$)…
  - and $O(\log_{10}N)$ 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!!
Lecture 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(N²)     C) O(1)     D) O(2ᴺ)
Lecture Question (2/3)

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

Consider the `getPreviousTile` method in `HideAndSeek`:

```java
public CS15ColorTile getPreviousTile(){
    return _previousTile;
}
```

A) O(N)  B) O(N^2)  C) O(1)  D) O(2^N)
Lecture 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[0].length; 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
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

```java
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>No. Ops.</th>
<th>Instructions</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>
</tbody>
</table>
| (N-1) + (N-2) + ... + 2 + 1 = N(N-1)/2 | for(int j = 1; j < i; j++) { //2
\[\text{if (a[j-1] > a[j])} \{ \text{//3}\]
\[\text{int temp = a[j-1];} \text{//2}\]
\[\text{a[j-1] = a[j];} \text{//3}\]
\[\text{a[j] = temp;} \text{//2}\]
\[\text{sorted = false;} \text{//1}\]
\[
\}
\}
| N-1      | i--; //1 |

## Worst-case analysis:
- **while** loop iterates N-1 times
- **iteration** i has 2 + 13(i - 1) operations

## Total:

\[
2 + N + 2(N-1) + 13[(N-1) + ... + 2 + 1] = 3N + 13N(N-1)/2 = 13N^2 + ... = O(N^2)
\]
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 ( $a[i-1], a[i-2], \ldots, a[j+1]$ ) to the right, until $a[j] < a[i]$
- Move $a[i]$ into vacated $a[j+1]$
- After iteration $i < a.length$, original $a[0] \ldots a[i]$ are in sorted order, but not necessarily in final position
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;
}
Selection Sort (1/2)

- Find smallest element and put it in $a[0]$
- Find 2$^{\text{nd}}$ smallest element and put it in $a[1]$, etc.
- Less data movement (no bubbling)
Selection Sort (2/2)

What we want to happen:

```java
int n = a.length;
for (int i = 0; i < n; i++) {
    find minimum element a[min] in subsequence a[i...n-1]
    swap a[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 instruction is executed \((N-1) + (N-2) + \ldots + 2 + 1\) 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 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!

<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>(n^2/2)</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>Average</td>
<td>(n^2/2)</td>
<td>(n^2/4)</td>
<td>(n^2/4)</td>
</tr>
<tr>
<td>Worst</td>
<td>(n^2/2)</td>
<td>(n^2/2)</td>
<td>(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>(n^2/4)</td>
<td>(n^2/2)</td>
</tr>
<tr>
<td>Worst</td>
<td>(n)</td>
<td>(n^2/2)</td>
<td>(n^2/2)</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 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, 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)

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;
    }
}

Andries van Dam © 2021 3/16/21
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 (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 phase
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!
}
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)
public ArrayList merge(ArrayList<Integer> A, ArrayList<Integer> B) {
    ArrayList<Integer> result = new ArrayList<>();
    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
Recursive (Top Down) Merge Sort (7/7)

- Each level of the tree performs \textbf{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 $\log_2 N$ merge passes.
- Thus, $O(N \log_2 N)$ — 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!
Iterative (Bottom Up) Merge Sort

- 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!
Comparing Sorting Algorithms

- Bubble Sort – $O(N^2)$
- Insertion Sort – $O(N^2)$
- Merge Sort – $O(N \log_2 N)$

Live Demo: http://cs.brown.edu/courses/cs015/demos/Rainbow.jar
Lecture 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 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 on-time deadline tomorrow 3/17 @ 11:59 PM
  ○ Late deadline 3/19 @ 11:59 PM

● Arrays lab 3/16 and 3/17
  ○ Be sure to complete pre-lab video/quiz before your lab slot
IT in the News

ft. Socially Responsible Computing!
Non-Fungible Tokens (NFTs) and the Environment

- **NFTs** = tokens of “authenticity” of digital art, usually bought & sold with cryptocurrency
  - NFTs ≠ copyright, trademarks, exclusive ownership
- Recently, an NFT sold for $69M 😲
  - good news for support of digital arts!
- But NFTs use massive amounts of computing power → enormous environmental impact

“Given the urgency of the current climate crisis, we are already faced with the immense challenge of having to change our existing habits….Is it really wise to adopt new systems and habits that are as ecologically devastating as this?”

– Memo Akten, computational artist

---

1 single-edition NFT

The ecological cost of tracking sales & bids on the blockchain:

- Energy usage: 340 kWh
- Emissions: 211 KgCO₂

Equivalent to:

- An EU resident’s electricity consumption for 1 month
- Flying for 2 hours
- Driving 1 thousand Km (petrol)
- Boiling a kettle 4.5 thousand times
- Using a laptop for 3 years
- Using a computer for 10 months

source: Memo Akten
Ethics in AI

- AI ethics: one of the oldest fields of ethical inquiry vis-à-vis computation
- Fear of AI is not new…
  - follows folk myths of artificial beings (e.g., Golems, Frankenstein’s monster) and “superintelligent” entities, long-held fears of worker displacement
- …but concerns grow more credible with recent AI advances & increasing awareness of AI’s flaws
  - e.g., concern over bots & fake communication, AI bias and misinformation, real concern of worker displacement in coming decade(s)

*recall: ML is a subset of AI!
The accelerating pace of change...

Agricultural Revolution 8,000 years
Industrial Revolution 120 years
Light-bulb 90 years
Moon landing 22 years
World Wide Web 9 years
Human genome sequenced

...and exponential growth in computing power...
Computer technology, shown here climbing dramatically by powers of 10, is now progressing more each hour than it did in its entire first 90 years

COMPUTER RANKINGS
By calculations per second per $1,000

Analytical engine
Never fully built, Charles Babbage’s invention was designed to solve computational and logical problems

Colossus
The electronic computer, with 1,500 vacuum tubes, helped the British crack German codes during WW II

UNIVAC I
The first commercially marketed computer, used to tabulate the U.S. Census, occupied 943 cu. ft.

Apple II
At a price of $1,298, the compact machine was one of the first massively popular personal computers

UNIVAC

Power Mac G4
The first personal computer to deliver more than 1 billion floating-point operations per second

Electromechanical
- Relays
- Vacuum tubes
- Transistors
- Integrated circuits

Andries van Dam © 2021 3/16/21

MRNA vaccines!
Al as existential threat?

- “Singularity”: postulated point at which AI surpasses human intelligence (Verner Vinge, 1993)
  - for some (e.g., Ray Kurzweil, 2006), singularity includes “merging” of human and computational intelligence

- Some predict singularity will arrive within 1-2 decades, is a serious threat to human society
  - e.g., Elon Musk, Bill Gates, Stephen Hawking\(^1\) all express fear of inevitable computational “superintelligence;” Musk encourages AI regulation

- Others believe the “singularity” is not as serious of a concern as mass job replacement due to automation, reliance on biased algorithms, environmental impact, etc.

\(^1\)Stephen Hawking (1942-2018) was a cosmologist whose work revolutionized our understanding of black holes.
AI and labor

- AI follows long history of worker displacement by machines
  - in past iterations (e.g., industrial revolution), jobs destroyed were replaced by other jobs created
  - but will this happen for AI?

- “Unskilled” labor at most immediate risk of replacement
  - e.g., truckers, drivers, warehouse/delivery workers, etc.

- But longer-term concerns of replacement of “skilled” or “intellectual” labor, “knowledge work”
  - e.g., accounting, legal professions, middle management, …

- What does this mean for the planet? (recall: environmental impact)

- What does this mean for (capitalist) society? For the future of work? What can we do about it?
  - stay tuned for lectures on UBI and labor in tech!

“Everything that can be automated will be automated.”
– Shoshana Zuboff (Zuboff’s first law)