Limits of What Computers Can Do
Announcements

• Assignment 6 due Friday at 11AM
  • Cannot be turned in late!
• Regular Office Hours today
• Extended Office Hours this Week
  • Wednesday, Thursday: Noon-5PM
• Graded midterms will be returned tomorrow (Wednesday)
• Please fill out course evaluations!
Limits of Programs

- We've spent a lot of time going over cool stuff computers can do
  - Quickly Sorting, Searching
    - Binary Search, Quicksort
  - Quickly storing and retrieving data
    - Hashing, Binary Search Trees
- An interesting question to consider is what *can't* computers do
Limits of Programs

• There are three I want to consider:
  • What can't a computer *do any faster*?
  • What can't a computer *do fast*?
  • What can't a computer *do at all*?
Limits of Programs

• There are three I want to consider:
  • What can't a computer do any faster?
  • What can't a computer do fast?
  • What can't a computer do at all?
Lower Bounds on Sorting

- Run times of various sorting algorithms:
  - QuickSort: $O(n \log n)$
  - MergeSort: $O(n \log n)$
  - HeapSort: $O(n \log n)$
  - SmoothSort: $O(n \log n)$
  ...

- Notice a pattern?
All of our fast sorting algorithms run in $O(n \log n)$ – what's up with that?
Lower Bounds on Sorting

- I haven't been holding back – we don't have any general-purpose sorting algorithms that are asymptotically faster than $O(n \log n)$.

- In fact, we can prove that we can't do any better (for general purpose algorithms).

- In order to do this we need to find what all our sorting algorithms have in common...
An Initial Idea: Selection Sort
Another Idea: **Insertion Sort**
The Key Insight: **Merge**
Lower Bounds on Sorting

- Observation: All our sorting algorithms involve repeatedly comparing pairs of elements in the array

- One way of measuring the amount of work our sorting algorithms do is by counting how many comparisons are performed
An Initial Idea: Selection Sort

O(n) comparisons per element → O(n^2) runtime!
Merge Sort

O\( (n) \)

O\( (n) \)

O\( (n) \)

O\( (n) \)

O\( (n) \)

O\( (n \log n) \) runtime → O\( (\log n) \) comparisons per node!
Lower Bounds on Sorting

- All our algorithms compare pairs of elements and their runtime is determined by how many comparisons are made.
  - These are all **comparison based** sorting algorithms
- Can we prove that all comparison based sorting algorithms require some minimum number of comparisons?
  - If we can do this, then we can prove a **lower bound** on the runtime of all comparison based sorting algorithms.
Lower Bounds on Sorting

• All our algorithms compare pairs of elements and their runtime is determined by how many comparisons are made.
  • These are all \textit{comparison based} sorting algorithms

• Can we prove that all comparison based sorting algorithms require some minimum number of comparisons?
  • If we can do this, then we can prove a \textit{lower bound} on the runtime of all comparison based sorting algorithms.
Intuition Behind Proof

1 2 3
Intuition Behind Proof

$X_1 \quad X_2 \quad X_3$
Intuition Behind Proof

\[ X_1 < X_2 < X_3 \]

\[ X_1 < X_3 < X_2 \]

\[ X_1 < X_2 < X_3 \]

\[ X_1 < X_3 < X_2 \]

\[ X_1 < X_2 < X_3 \]
Intuition Behind Proof

- Every sorting algorithm needs to be able to sort every possible permutation of \( n \) elements.
- The number of comparisons needed is proportional to the height of the tree.
Intuition Behind Proof
Intuition Behind Proof

- Because any list of elements has $n!$ permutations, we know the tree has $n!$ leaves.
- The height of a balanced binary tree with $L$ leaves is $O(\log L)$
- Therefore, the height of our tree is $O(\log n!)$
- Sterling's Approximation
  - $O(\log n!) = O(n \log n)$
- The height of our tree is $O(n \log n)$
Therefore, **all** comparison based sorting algorithms require $O(n \log n)$ comparisons in the worst case.

This implies the best we can do is $O(n \log n)$ worst case runtime.

(QED)
Other Sorting Algorithms

- Summary: No “comparison-based” sorting algorithms can do better than worse case $O(n \log n)$.

- Should we give up? No!

- Two ways we can get around this:
  - Make additional assumptions about the data
  - Use a non-comparison based sorting algorithm
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

  $k = 3$

  4  3  1  5  2  6  9  7  8  12  11  10

Heap
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

$$k = 3$$

3 1 5 2 6 9 7 8 12 11 10
Additional Assumptions

• If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\begin{tabular}{ccccccccc}
1 & 5 & 2 & 6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{tabular}
Additional Assumptions

• If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{cccccccccccc}
1 & 5 & 2 & 6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]
Additional Assumptions

• If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>7</th>
<th>8</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
</table>

Heap

3 4 1
Additional Assumptions

- If we have an unsorted array in which we knew every element was within \( k \) indices of where it should be and ran HeapSort

\[ k = 3 \]

<table>
<thead>
<tr>
<th>5</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>7</th>
<th>8</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
</table>

Heap: 1 4 3
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{cccccccc}
2 & 6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{cccccccc}
2 & 6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

Heap

\[
\begin{array}{cccc}
1 & 4 & 3 & 5 \\
\end{array}
\]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

$$k = 3$$

2 6 9 7 8 12 11 10

1

Heap 4 3 5
Additional Assumptions

- If we have an unsorted array in which we knew every element was within \( k \) indices of where it should be and ran HeapSort

\[
k = 3
\]

\[
\begin{array}{ccccccccc}
6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

Heap

\[
\begin{array}{cccccc}
2 & 3 & 5 & 4 \\
1 \\
\end{array}
\]
Additional Assumptions

• If we have an unsorted array in which we knew every element was within \( k \) indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{cccccccc}
6 & 9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

Heap

\[
\begin{array}{cccc}
1 & 2 \\
3 & 5 & 4 \\
\end{array}
\]
Additional Assumptions

• If we have an unsorted array in which we knew every element was within \( k \) indices of where it should be and ran HeapSort

\[
k = 3
\]

Full Array:
\[
\begin{array}{cccccc}
9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

Heap:
\[
\begin{array}{cccc}
3 & 5 & 4 & 6 \\
\end{array}
\]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{ccccccc}
9 & 7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

Heap

\[
\begin{array}{ccc}
4 & 5 & 6 \\
\end{array}
\]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[
\begin{array}{cccccc}
7 & 8 & 12 & 11 & 10 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 \\
\end{array}
\]

Heap

\[
\begin{array}{cccccc}
4 & 5 & 6 & 9 \\
\end{array}
\]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

$$k = 3$$

1 2 3 4

Heap

5 6 9

7 8 12 11 10
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

$k = 3$
Additional Assumptions

• If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]

\[ \begin{array}{c}
1 & 2 & 3 & 4 & 5 \\
12 & 11 & 10 \\
\end{array} \]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]
Additional Assumptions

- If we have an unsorted array in which we knew every element was within $k$ indices of where it should be and ran HeapSort

\[ k = 3 \]
Heap Sort

- If we know every element is within $k$ indices of its correct location, then we can dequeue whenever the heap has $k + 1$ elements.

- What is the runtime of this algorithm?
  - Each element is added and removed.
  - Both operations are logarithmic in the size of the Heap = $k + 1$.
  - Therefore, add and remove are $O(\log k)$.
  - We have $O(n)$ elements.
  - $O(n \log k)$!!!
Heap Sort

- The smaller we can make $k$, the faster HeapSort will run.
- When $k = n$ it devolves into regular HeapSort with $O(n \log n)$ runtime
Non-Comparison Based Algorithms

• Another way to beat the $O(n \log n)$ bound is to use non-comparison based sorting algorithms:
  • Bucket Sort: Construct a histogram of the elements in the array
Bucket Sort

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
a b c d e f g h i j k l m n o p q r s t u v w x y z

banana
Bucket Sort

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

a b c d e f g h i j k l m n o p q r s t u v w x y z

banana
Bucket Sort

1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

a b c d e f g h i j k l m n o p q r s t u v w x y z
Bucket Sort

110000000001000000000000000

abcdefghijklmnopqrstuvwxyz

banana
Bucket Sort

3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0

a b c d e f g h i j k l m n o p q r s t u v w x y z

banana

↑
### Bucket Sort

| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

- **Bucket Sort** is a sorting algorithm that distributes elements into buckets according to a key.
- In this example, the input is a sequence of 24 elements, and the output is sorted in ascending order.
Bucket Sort

3100000002000000000000

a b c d e f g h i j k l m n o p q r s t u v w x y z

\[ \text{a a a a b} \]
Bucket Sort

3 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0

a b c d e f g h i j k l m n o p q r s t u v w x y z

↑

a a a a b
Bucket Sort
Bucket Sort
Bucket Sort
Bucket Sort
Bucket Sort
Bucket Sort

- Pseudocode:
  - Create an array `histogram` of length $d$ where $d$ is the number of possible values elements can take in the original array.
  - For each element in the array we're sorting, update the `histogram`
  - For each index in the `histogram`, output the corresponding element `histogram[i]` times
- Runtime?
  - $O(d + n)$
- Generally used if $d$ is small (e.g. `char`)
Bucket Sort for `ints`

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>2^{32-6}</th>
<th>2^{32-5}</th>
<th>2^{32-4}</th>
<th>2^{32-3}</th>
<th>2^{32-2}</th>
<th>2^{32-1}</th>
</tr>
</thead>
</table>

| 8 | 112 | 240 | 62 | 987 | 500 |   |   |   |   |   |   |   |   |
Limits of Programs

• There are three I want to consider:
  • What can't a computer do any faster?
  • **What can't a computer do fast?**
  • What can't a computer do at all?
Traveling Salesperson
Traveling Salesperson

Seattle $250

$300

SF

$400

Salt Lake City $350

$550

Austin $400

$600

New York $800

https://www.google.com/maps/vt/data=VLHX1wd2Cgu8wR6jwyh-km8JBWAKezU4.2bUCUBVs3YYr-KB4ccFl-1Q1nWYcyKzmW0Ggf8ar4O0yEuuN9txRnTtIkzlvmH6qy6B4vSoZvopndG7VjMIsoIDayhdkgKbiOykP1wZYm9RcF8-Y6pkecPwDi3xc98B3gNGLchfR7xnPKzCGEmRocrv9OczmELzORvRseZHLyjWOvL0GzUeg0WFJGA4Y
Traveling Salesperson

• Find a minimal cost tour (visits every city and returns to starting city)

• How can we solve this?

• Algorithm 1: Consider all possible permutations of cities and return the cheapest permutation.
  • Worst case $O(n!)$

• Algorithm 2: Dynamic Programming.
  • Technique similar in spirit to memoization except you build up longer and longer paths
  • Worst case $O(2^n)$
Traveling Salesperson

- \( O(n!) \) and \( O(2^n) \) are both exponential runtimes
  - i.e. The runtime of the algorithm grows exponential in the size of the input
- How long it takes to compute depends on constant factors, but if each operation takes 1 millisecond...
### Comparison of Runtimes

(1 operation = 1 microsecond)

<table>
<thead>
<tr>
<th>Size</th>
<th>n</th>
<th>n log n</th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10μs</td>
<td>33μs</td>
<td>100μs</td>
<td>1ms</td>
<td>1ms</td>
<td>1 hour</td>
</tr>
<tr>
<td>20</td>
<td>20μs</td>
<td>86μs</td>
<td>400μs</td>
<td>8ms</td>
<td>17min</td>
<td>8 years</td>
</tr>
<tr>
<td>30</td>
<td>30μs</td>
<td>147μs</td>
<td>900μs</td>
<td>27ms</td>
<td>12 days</td>
<td>2 sixtillion years</td>
</tr>
<tr>
<td>40</td>
<td>40μs</td>
<td>212μs</td>
<td>1.6ms</td>
<td>64ms</td>
<td>34 years</td>
<td>...</td>
</tr>
<tr>
<td>50</td>
<td>50μs</td>
<td>282μs</td>
<td>2.5ms</td>
<td>125ms</td>
<td>3.56e² years</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>60μs</td>
<td>354μs</td>
<td>3.6ms</td>
<td>216ms</td>
<td>3.65e⁷ years</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>70μs</td>
<td>429μs</td>
<td>4.9ms</td>
<td>343ms</td>
<td>3.74e¹⁰ years</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>80μs</td>
<td>506μs</td>
<td>6.4ms</td>
<td>512ms</td>
<td>3.83e¹³ years</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>90μs</td>
<td>584μs</td>
<td>8.1ms</td>
<td>729ms</td>
<td>3.92e¹⁶ years</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100μs</td>
<td>664μs</td>
<td>10ms</td>
<td>1s</td>
<td>40 quintillion years</td>
<td></td>
</tr>
</tbody>
</table>
Traveling Salesperson

• There are many problems in which the best known algorithms run in worst case exponential time...
Sensor Placement
Graph Coloring
Games...

http://kickdes.files.wordpress.com/2011/04/classicbattleship.jpg

http://www.technologyreview.com/blog/arxiv/files/2012/03/SuperMarioBros.jpg

http://www.technologyreview.com/blog/arxiv/files/80466/Pac-Man.png
Complexity Classes

- In Complexity Theory computing problems are put into different **complexity classes**
- **P**: The set of problems that can be solved in polynomial time
  - e.g. sorting, searching an array for a value
- **NP**: The set of problems that can be solved in exponential time
  - e.g. Traveling Salesperson, Graph Coloring
- It has not been proved, but it's assumed that $\mathbf{P} \neq \mathbf{NP}$
Beating Exponential Time

• We have two options to beat exponential time algorithms:
  • Approximation Algorithms
  • Heuristics
Approximation Algorithms

- A **k-Approximation Algorithm** is an algorithm that you can prove gets within a factor $k$ of an optimal solution in the worst case.

- A simple 2-Approximation Algorithm for traveling salesman...
  - Compute a Minimum Spanning Tree of the graph and return a “depth first” path of the tree.
2-Approximation TSP

Seattle → SF: $300

SF → Salt Lake City: $250
Salt Lake City → Austin: $400
Austin → New York: $800

New York → Salt Lake City: $550

Salt Lake City → SF: $350

SF → Seattle: $400

Seattle → $600

$350
2-Approximation TSP

Seattle

$300

SF

Salt Lake City

$250

Austin

$350

$550

New York
WHY????

- Remember we are computing an optimal tour – visit every node at least once and end at the starting node.

- The cost of every optimal tour is going to be less than the cost of a Minimum Spanning Tree

- The cost of our MST is a lower bound of the cost of an optimal tour
2-Approximation TSP

Seattle

$300

SF

$250

Salt Lake City

$350

Austin

$550

New York
2-Approximation TSP

Seattle

SF

Salt Lake City

Austin

New York
2-Approximation TSP
2-Approximation TSP

Seattle

SF

Salt Lake City

Austin

New York
2-Approximation TSP

Seattle → SF
2-Approximation TSP

Seattle → SF → Seattle
2-Approximation TSP

Seattle → SF → Seattle → SLC
2-Approximation TSP

Seattle → SF → Seattle → SLC → Austin
2-Approximation TSP

Seattle → SF → Seattle → SLC → Austin → SLC
2-Approximation TSP

Seattle → SF → Seattle → SLC → Austin → SLC → NY
2-Approximation TSP

Seattle → SF → Seattle → SLC → Austin → SLC
→ NY → SLC
2-Approximation TSP

Seattle → SF → Seattle → SLC → Austin → SLC → NY → SLC → Seattle
2-Approximation TSP

- Because we use every edge twice, the cost of this tour is going to be twice the cost of the MST.

- The cost of the MST is less than or equal to the cost of an optimal tour.

- Therefore, the cost of our tour is less than or equal to twice the cost of an optimal tour.
  
  - Hence, this is a 2-approximation
Approximation Algorithms

- Better approximation algorithms exist for TSP (but they are more difficult to prove)
- Many approximation algorithms exist for different problems in NP
Heuristics

• A different Idea: Construct a heuristic that will give a “good” solution.
  • Even if it performs terribly in the “worst case”, it may perform well in “most” cases.

• Nearest Neighbor Heuristic
  • Iteratively extend path by picking cheapest edge that will get us to an unvisited node
  • Works reasonably well with high probability
  • Has terrible worst case behavior.
    – Okay because worst case is unlikely
Limits of Programs

- There are three I want to consider:
  - What can't a computer do any faster?
  - What can't a computer do fast?
  - What can't a computer do at all?
A Useful Tool

- It would be incredibly useful if Visual Studio and Xcode would detect the following issues before running a program:
  - Infinite Loops/Recursion
  - Memory Leaks
  - Issues dereferencing NULL and uninitialized pointers
  - Automatic grading of assignments
Problem: It is *impossible* to write a program which can, for all input programs, successfully do these tasks!
A Useful Tool

- Example: It is impossible to write a program that, given any program and input, detects if the program will terminate on that input.
  - Called the Halting Problem
  - We say that the Halting Problem is undecideable
- Wait...really?
What about this?

```cpp
int main() {
    while (true) {
        cout << "Counter Example?" << endl;
    }
}
```
int main() {
    for (int i = 0; i < 10; i++)
        cout << "This isn't hard!" << endl;
}

Or this?
Or even this?

```c
int main() {
    return 0;
}
```
Halting Problem

• For many program-input pairs we can easily tell if they terminate.

• We cannot do this for all programs.

• So how can we construct one?
  • It's tricky. Take CS161 to learn more about this.

• We're just going to go over the intuition...
Proof Sketch

The way we prove the Halting Problem is undecideable is through proof by contradiction: we start by assuming that it is decideable then derive a contradiction.

- Common proof technique for proving something cannot exist

Proof Sketch:

- Assume a program $P$ exists that solves the halting problem for all inputs
- Construct a new program $Q$ from $P$
- Show $P$ cannot decide if $Q$ terminates
Proof Intuition

• Constructing $Q$ from $P$ is the heart of the proof.

• It's somewhat confusing, but is similar in spirit to the following contradiction:
  • “The barber of Seville shaves everyone in Seville who doesn't shave himself. Does the barber shave himself?”

• Idea is to run $P$ with input $P$
Halting Problem

• As a corollary, many other useful questions regarding arbitrary programs are also undecideable:
  • Memory Leaks?
  • Dereferencing NULL pointers?
  • Many many more...
How Bad is This?

- As a result of this we run into some issues...
  - Can't prove arbitrary programs are correct - need to test them
  - Tools to detect memory leaks don't catch everything
- Modern tools that detect these types of issues can't detect everything, but can still be useful.
Tomorrow

- Introduction to **Machine Learning**