Algorithmic Analysis and Sorting
Part Two
Announcements

• 3 Handouts on Website
  • Study Skills
  • Practice Midterm
  • Practice Midterm Solutions
• Still working on Reference Sheet...
Big-O Notation

• Ignore *everything* except the dominant growth term, including constant factors.

• Examples:
  
  • $4n + 4 = O(n)$
  
  • $137n + 271 = O(n)$
  
  • $n^2 + 1000n + 100000 = O(n^2)$
  
  • $2^n + n^3 = O(2^n)$
Algorithmic Analysis with Big-O

double average(Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}

O(n)
Types of Analysis

- **Worst-Case Analysis**
  - What's the *worst* possible runtime for the algorithm?
  - Useful for "sleeping well at night."

- **Best-Case Analysis**
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.

- **Average-Case Analysis**
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, CS365, or CS369N for more information!
Worst Case Analysis

```cpp
bool LinearSearch(string& str, char ch) {
    for (int i = 0; i < str.length(); i++)
        if (str[i] == ch)
            return true;
    return false;
}
```

- Assume that “ch” is the worst possible location for this algorithm
  - In this case, “ch” is not in str
  - $O(n)$
Review: Big-O
(Board)
What Can Big-O Tell Us?

- Long-term behavior of a function.
  - If algorithm A is $O(n)$ and algorithm B is $O(n^2)$, **for large inputs** algorithm A will always be faster.
  - If algorithm A is $O(n)$, **for large inputs**, doubling the size of the input roughly doubles the runtime.
    - In other words, Big-O tells us how the running time of an algorithm grows as the size of its input grows

What “large” means on the terms we dropped!
What *Can't* Big-O Tell Us?

- The actual runtime of a function.
  - $10^{100}n = O(n)$
  - $10^{-100}n = O(n)$
- How a function behaves on small inputs.
  - $n^3 = O(n^3)$
  - $10^6 = O(1)$
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**

```
A
B
C
D
E
```
An Initial Idea: **Selection Sort**
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**

1  4  2  7  6
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**

![Bar Chart Illustrating Selection Sort Process]

1  4  2  7  6
An Initial Idea: Selection Sort

1 4 2 7 6
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**

1  2  4  7  6
An Initial Idea: **Selection Sort**

![Sorted Array]

1  2  4  7  6
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**

![Bar chart](image-url)
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**
An Initial Idea: **Selection Sort**

Runtime is $O(n^2)$
Selection Sort
(Pseudocode)
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**

7

4

2

1

6
Another Idea: **Insertion Sort**

![Diagram of Insertion Sort]

- 7
- 4
- 2
- 1
- 6
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**

```
4  2  7  1  6
```
Another Idea: **Insertion Sort**

![Diagram of Insertion Sort](image-url)
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**

```
2
4
7
1
6
```
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**

2 4 1 7 6
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: Insertion Sort
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Another Idea: **Insertion Sort**
Insertion Sort
(Pseudocode)
void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] < v[j + 1]) break;
            swap(v[j], v[j + 1]);
        }
    }
}
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1  2  4  6  7
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1  2  4  6  7
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?

1  2  4  6  7
How Fast is Insertion Sort?

Work done: $O(n)$
How Fast is Insertion Sort?

7

6

4

2

1
How Fast is Insertion Sort?

7 6 4 2 1
How Fast is Insertion Sort?

7   6   4   2   1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

6 7 4 2 1
How Fast is Insertion Sort?

6
7
4
2
1
How Fast is Insertion Sort?

6
7
4
2
1
How Fast is Insertion Sort?

6  7  4  2  1
How Fast is Insertion Sort?

6

4

7

2

1
How Fast is Insertion Sort?

6  4  7  2  1
How Fast is Insertion Sort?

4

6

7

2

1
How Fast is Insertion Sort?

4
6
7
2
1
How Fast is Insertion Sort?

4  6  7  2  1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

4
6
2
7
1
How Fast is Insertion Sort?

4 6 2 7 1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

4

2

6

7

1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2  4  6  7  1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2  4  6

7  1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2  4  6  1  7
How Fast is Insertion Sort?

2 4 1 6 7
How Fast is Insertion Sort?

2 4 1 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?

Work done: $O(n^2)$
Notes on Insertion Sort

• Insertion sort has runtime $O(n)$ in the best case.

• Insertion sort has runtime $O(n^2)$ in the worst case.

• On average, insertion sort is roughly twice as fast as selection sort.
  
  • In a random array, every element is about halfway away from where it belongs.
  
  • So for the $k$th element, about $(k - 1) / 2$ other elements must be looked at.
# Selection Sort vs Insertion Sort

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
</tr>
</tbody>
</table>
Of insertion sort and selection sort:

1. Which algorithm does more work at the start?  
2. Which algorithm does more work at the end?
Thinking About $O(n^2)$
Thinking About $O(n^2)$

$T(n)$

\[
\begin{array}{cccccccccccc}
14 & 6 & 3 & 9 & 7 & 16 & 2 & 15 & 5 & 10 & 8 & 11 & 1 & 13 & 12 & 4 \\
\end{array}
\]
Thinking About $O(n^2)$

$T(n)$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$
Thinking About $O(n^2)$

$T(n)$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$
Thinking About $O(n^2)$

$$T(n)$$

$T(\frac{1}{2}n) \approx \frac{1}{4}T(n)$

It takes roughly $\frac{1}{2}T(n)$ to sort each half separately!
The Key Insight: **Merge**
The Key Insight: Merge
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**

![Merge Diagram]

1. Merge 2, 4, 7, 8, 10
2. Merge 3, 5, 6, 9
3. Merge 1
The Key Insight: *Merge*
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**

```
  4  7  8  10
  1  2  3
  5  6  9
```
The Key Insight: **Merge**
The Key Insight: **Merge**

- 7, 8, 10
- 5, 6, 9
- 1, 2, 3, 4
The Key Insight: **Merge**

![Diagram showing merging of numbers 1 to 10 into numbers 5 to 9.](image-url)
The Key Insight: Merge
The Key Insight: **Merge**

```
7 8 10

1 2 3 4 5

6 9
```
The Key Insight: **Merge**

```
7  8  10
```

```
1  2  3  4  5  6
```

```
9
```
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: **Merge**
Merge
(Pseudocode)
void merge(Queue<int>& one, Queue<int>& two, Queue<int>& result) {

    while (!one.isEmpty() && !two.isEmpty()) {
        if (one.peek() < two.peek()) {
            result.enqueue(one.dequeue());
        } else {
            result.enqueue(two.dequeue());
        }
    }

    while (!one.isEmpty()) {
        result.enqueue(one.dequeue());
    }

    while (!two.isEmpty()) {
        result.enqueue(two.dequeue());
    }

    return result;
}
“Split Sort”
void splitSort(Vector<int>& v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

    merge(left, right, v);
}
void splitSort(Vector<int> & v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

    merge(left, right, v);
}
void splitSort(Vector<int>& v) {
    Vector<int> left, right;

    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int j = v.size() / 2; j < v.size(); j++)
        right += v[i];

    insertionSort(left);
    insertionSort(right);

    merge(left, right, v);
}
### Performance Comparison

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
</tr>
</tbody>
</table>
## Performance Comparison

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
</tr>
</tbody>
</table>
A Better Idea

- Splitting the input in half and merging halves the work.
- So why not split into four? Or eight?
- **Question**: What happens if we never stop splitting?
MergeSort
(Pseudocode)
High-Level Idea

- A recursive sorting algorithm!

- **Base Case:**
  - An empty or single-element list is already sorted.

- **Recursive step:**
  - Break the list in half and recursively sort each part.
  - Use `merge` to combine them back into a single sorted list.

- This algorithm is called `mergesort`.
void mergesort(Vector<int>& v) {
    /* Base case: 0- or 1-element lists are already sorted. */
    if (v.size() <= 1) return;

    /* Split v into two subvectors. */
    Vector<int> left, right;
    for (int i = 0; i < v.size() / 2; i++)
        left += v[i];
    for (int i = v.size() / 2; i < v.size(); i++)
        right += v[i];

    /* Recursively sort these arrays. */
    mergesort(left);
    mergesort(right);

    /* Combine them together. */
    merge(left, right, v);
}
What is the complexity of mergesort?
void mergesort(Vector<int>& v) {
  /* Base case: 0- or 1-element lists are already sorted. */
  if (v.size() <= 1) return;

  /* Split v into two subvectors. */
  Vector<int> left, right;
  for (int i = 0; i < v.size() / 2; i++)
    left += v[i];
  for (int i = v.size() / 2; i < v.size(); i++)
    right += v[i];

  /* Recursively sort these arrays. */
  mergesort(left);
  mergesort(right);

  /* Combine them together. */
  merge(left, right, v);
}
A Graphical Intuition

\[ O(n) \]

\[ O(n) \]

\[ O(n) \]
A Graphical Intuition

How many levels are there?
Slicing and Dicing

- After zero recursive calls: $n$
- After one recursive call: $n / 2$
- After two recursive calls: $n / 4$
- After three recursive calls: $n / 8$
- \[ \ldots \]
- After $k$ recursive calls: $n / 2^k$
Cutting in Half

- After $k$ recursive calls, there are $n / 2^k$ elements left.
- Mergesort stops recursing when there are zero or one elements left.
- Solving for the number of levels:
  \[
  n / 2^k = 1
  \]
  \[
  n = 2^k
  \]
  \[
  \log_2 n = k
  \]
- So mergesort recurses $\log_2 n$ levels deep.
A Graphical Intuition

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(n)$

$O(n \log n)$
## Mergesort Times

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
</tr>
</tbody>
</table>
# Mergesort Times

<table>
<thead>
<tr>
<th>Size</th>
<th>Selection Sort</th>
<th>Insertion Sort</th>
<th>“Split Sort”</th>
<th>Mergesort</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.304</td>
<td>0.160</td>
<td>0.161</td>
<td>0.006</td>
</tr>
<tr>
<td>20000</td>
<td>1.218</td>
<td>0.630</td>
<td>0.387</td>
<td>0.010</td>
</tr>
<tr>
<td>30000</td>
<td>2.790</td>
<td>1.427</td>
<td>0.726</td>
<td>0.017</td>
</tr>
<tr>
<td>40000</td>
<td>4.646</td>
<td>2.520</td>
<td>1.285</td>
<td>0.021</td>
</tr>
<tr>
<td>50000</td>
<td>7.395</td>
<td>4.181</td>
<td>2.719</td>
<td>0.028</td>
</tr>
<tr>
<td>60000</td>
<td>10.584</td>
<td>5.635</td>
<td>2.897</td>
<td>0.035</td>
</tr>
<tr>
<td>70000</td>
<td>14.149</td>
<td>8.143</td>
<td>3.939</td>
<td>0.041</td>
</tr>
<tr>
<td>80000</td>
<td>18.674</td>
<td>10.333</td>
<td>5.079</td>
<td>0.042</td>
</tr>
<tr>
<td>90000</td>
<td>23.165</td>
<td>12.832</td>
<td>6.375</td>
<td>0.048</td>
</tr>
</tbody>
</table>
Can we do Better?

• Mergesort is $O(n \log n)$.

• This is asymptotically better than $O(n^2)$

• Can we do better?
  
  • In general, **no**: comparison-based sorts cannot have a worst-case runtime better than $O(n \log n)$.

• **In the worst case, we can only get faster by a constant factor!**
Growth Rates

- $O(n)$
- $O(n \log n)$
- $O(n^2)$