Warm Up

Show $\log(n!) = \Theta(n \log n)$

**Hint:** show $n! \leq n^n$

**Hint 2:** show $n! \geq \left(\frac{n}{2}\right)^\frac{n}{2}$
Warm Up

\[ n! = n \cdot (n - 1) \cdot (n - 2) \cdot \cdots \cdot 2 \cdot 1 \]

\[ n! \leq n^n \]

\[ \Rightarrow \log(n!) \leq \log(n^n) = n \log n \in O(n \log n) \]
\[ n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2} - 1\right) \cdot \ldots \cdot 2 \cdot 1 \]

\[ n! \geq \left(\frac{n}{2}\right)^{n/2} \]

\[ \Rightarrow \log(n!) \geq \frac{n}{2} \log \left(\frac{n}{2}\right) \in \Omega(n \log n) \]
Today’s Keywords

Divide and Conquer
Sorting Algorithms
Quicksort
Decision Tree
Sorting Lower Bounds

CLRS Readings: Chapter 7 and 8
• **HW3** due **Tuesday, October 1, 11pm**
  • Divide and conquer algorithms
  • Written (use LaTeX!) – Submit both zip and pdf!

• **HW0** grades posted on Collab
  • *Regrade office hours:*
    • Thursday 11am-12pm (Rice 210)
    • Thursday 4pm-5pm (Rice 501)
  • Please be prepared to verbally explain your submission if requesting a regrade
Randomized Quicksort

**Divide:** Select a random pivot, and partition about the pivot

![Sorted array]

**Conquer:** Recursively sort left and right sublists

![Sorted array]

**Expected running time:** $O(n \log n)$
Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?

Consider the sorted version of the list

$\Pr[\text{we compare 1 and 12}] = \frac{2}{12}$

Assuming pivot is chosen uniformly at random

Elements only compared if 1 or 12 was chosen as the first pivot since otherwise they are in different sublists.
Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>$j$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Case 1:** Pivot less than $i$

Pr[we compare $i$ and $j$] = Pr[we compare $i$ and $j$ in Quicksort([$p + 1, \ldots, n$])

**Case 2:** Pivot greater than $j$

Pr[we compare $i$ and $j$] = Pr[we compare $i$ and $j$ in Quicksort([1, $\ldots$, $p$])

**Case 3:** Pivot in [$i, i + 1, \ldots, j$]

Pr[we compare $i$ and $j$] = Pr[$i$ or $j$ is selected as pivot] = $\frac{2}{j - i + 1}$
Formal Argument for $n \log n$ Average

Probability of comparing element $i$ with element $j$:

$$\Pr[\text{we compare } i \text{ and } j] = \frac{2}{j - i + 1}$$
Formal Argument for $n \log n$ Average

Probability of comparing element $i$ with element $j$:

$$\Pr[\text{we compare } i \text{ and } j] = \frac{2}{j - i + 1}$$

Expected number of comparisons:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

Substitution: $k = j - i$

$$\frac{1}{k + 1} < \frac{1}{k}$$
Formal Argument for $n \log n$ Average

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}$$

**Substitution:**
\[ k = j - i \]

**Useful fact:**
$$\sum_{k=1}^{n} \frac{1}{k} = \Theta \left( \log n \right)$$

**Intuition (not proof!):**
$$\sum_{k=1}^{n} \frac{1}{k} \approx \int_{1}^{n} \frac{1}{x} \, dx = \ln n$$
Formal Argument for $n \log n$ Average

\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} < 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k}
\]

\[= 2 \sum_{i=1}^{n-1} \Theta(\log n) = \Theta(n \log n)\]

Useful fact: \[\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)\]
Sorting Algorithms

Sorting algorithms we have discussed:

- Mergesort \( O(n \log n) \)
- Quicksort \( O(n \log n) \)

Other sorting algorithms (will discuss):

- Bubble sort \( O(n^2) \)
- Insertion sort \( O(n^2) \)
- Heapsort \( O(n \log n) \)

Can we do better than \( O(n \log n) \)?
Worst Case Lower Bounds

Prove that there is no algorithm which can sort faster than $O(n \log n)$

Non-existence proof!
- Very hard to do

We will show such a lower bound for comparison sorts

Algorithm that only assumes elements can be compared (nothing about representation of the elements)
Comparison sorts use **comparisons** to determine ordering

**Strategy:** Draw tree to illustrate all possible execution paths

How do we measure running time?

- Possible execution path
- Result of comparison
- One comparison

Permutation of original list

- [1,2,3,4,5]
- [2,1,3,4,5]
- [5,2,4,1,3]
- [5,4,3,2,1]
Strategy: Decision Tree

Worst case running time is the longest execution path (measures number of comparisons) – this is the height of the decision tree.

Possible execution path:

Possible comparison:

Result of comparison:

One comparison:

Permutation of original list:

How many such permutations do we need?
Worst case running time is the longest execution path (measures number of comparisons) – this is the **height** of the decision tree.

\[ \log(n!) \]

\[ \Omega(n \log n) \]

How many such permutations do we need?

Permutation of original list

\( n! \) possible permutations
**Conclusion:** Running time of any comparison sort is $\Omega(n \log n)$
Sorting Algorithms

Sorting algorithms we have discussed:
- Mergesort $O(n \log n)$ Optimal!
- Quicksort $O(n \log n)$ Optimal!

Other sorting algorithms (will discuss):
- Bubble sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heapsort $O(n \log n)$ Optimal!

Can we do better than $O(n \log n)$?
Not with comparison sorts...
Important properties of sorting algorithms:

**Run Time**
- Asymptotic Complexity
- Constants

**In Place**
- Only requires constant additional space

**Adaptive**
- Faster if list is nearly sorted

**Stable**
- Equal elements remain in original order

**Parallelizable**
- Runs faster with many processors

*Relaxed definition:* only need to copy a constant number of elements.
Merge Sort

**Divide:**
- Break $n$-element list into two lists of $n/2$ elements

**Conquer:**
- If $n > 1$: Sort each sublist **recursively**
- If $n = 1$: List is already sorted (**base case**)

**Combine:**
- Merge together sorted sublists into one sorted list

**Run Time?**
$O(n \log n)$
Optimal!

**In Place?**
No

**Adaptive?**
No

**Stable?**
Yes*

**Technically:** depends on how merge is implemented
**Merge Sort**

**Combine:** Merge sorted sublists into one sorted list

We have:
- 2 sorted lists \((L_1, L_2)\)
- 1 output list \((L_{out})\)

While \((L_1 \text{ and } L_2 \text{ not empty})\):

If \(L_1[0] \leq L_2[0]:\)

\[L_{out}.append(L_1.pop())\]

Else:

\[L_{out}.append(L_2.pop())\]

\[L_{out}.append(L_1)\]

\[L_{out}.append(L_2)\]

**Stable:**
If elements are equal, leftmost comes first
Merge Sort

Divide:
• Break \( n \)-element list into two lists of \( \frac{n}{2} \) elements

Conquer:
• If \( n > 1 \): Sort each sublist recursively
• If \( n = 1 \): List is already sorted (base case)

Combine:
• Merge together sorted sublists into one sorted list

Run Time?
\( O(n \log n) \)
Optimal!

In Place?
No

Adaptive?
No

Stable?
Yes*

Parallelizable?
Yes
Merge Sort

**Divide:**
- Break $n$-element list into two lists of $n/2$ elements

**Conquer:**
- If $n > 1$:
  - Sort each sublist recursively
- If $n = 1$:
  - List is already sorted (base case)

**Combine:**
- Merge together sorted sublists into one sorted list

**Parallelizable:**
Allow different processors to sort each sublist
Merge Sort (Sequential)

\[ T(n) = 2T(\frac{n}{2}) + n \]

**Run Time:** \( O(n \log n) \)
Merge Sort (Parallel)

\[ T(n) = T\left(\frac{n}{2}\right) + n \]

Run Time: \( O(\log n) \)
Quicksort

Divide:
- Choose random pivot \( p \), \( \text{Partition}(p) \)

Conquer:
- Recursively sort left and right sublists

Combine:
- Nothing

In Place? Yes*
Adaptive? No
Stable? No
Parallelizable? Yes

Run Time?
\( O(n \log n) \) on expectation
(Better constants than merge sort)

Can sort the list in place, but requires \( \Theta(\log n) \) space on the stack (so not “in place” in a strict sense)
**Bubble Sort**

**Idea:** Iterate through list, swapping adjacent elements if out of order, repeat until sorted.

<table>
<thead>
<tr>
<th>8</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>12</th>
<th>10</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>6</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>
Bubble Sort

**Idea:** Iterate through list, swapping adjacent elements if out of order, repeat until sorted

**In Place?** Yes  **Adaptive?** No

**Run Time?** $O(n^2)$
( Constants worse than insertion sort)

“Compared to straight insertion [...], bubble sorting requires a more complicated program and takes about twice as long!” – Donald Knuth
How Adaptive is Bubble Sort?

Idea: Iterate through list, swapping adjacent elements if out of order, repeat until sorted

Only makes one “pass” if list is already sorted

After one “pass:”

Still requires $n$ passes, thus is $\Omega(n^2)$
**Bubble Sort**

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Idea:** Iterate through list, swapping adjacent elements if out of order, repeat until sorted.

**Run Time?**

\[ O(n^2) \]

(Constants worse than insertion sort)

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems" – Donald Knuth, *The Art of Computer Programming*
**Insertion Sort**

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

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

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

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

```
| 3 | 5 | 7 | 8 | 9 | 10 | 12 | 2 | 4 | 6 | 1 | 11 |
```
Insertion Sort

Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element

Run Time? \( O(n^2) \)
(but with very small constants; great for short lists)

Fancy quicksort

Code snippet from Java Arrays.sort implementation

Base case is insertion sort!
Insertion Sort

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element

**Run Time?** $O(n^2)$
(but with very small constants; great for short lists)

```
private static final int INSERTION_SORT_THRESHOLD = 47;

// Use insertion sort on tiny arrays
if (length < INSERTION_SORT_THRESHOLD) {
    // Traditional (without sentinel) insertion sort,
    // optimized for server VM, is used in case of
    // the leftmost part.
    for (int i = left, j = i; i < right; j = ++i) {
        int ai = a[i - 1];
        while (ai < a[j]) {
            a[j + 1] = a[j];
            a[j] = ai;
            // (j--) or if (j > 0) ai = a[j];
        }
    }
```

Code snippet from Java `Arrays.sort` implementation

Base case is insertion sort!
Insertion Sort

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

**In Place?** Yes  **Adaptive?** Yes

**Run Time?** $O(n^2)$
(but with very small constants; great for short lists)
**Insertion Sort is Adaptive**

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

![Sorted Prefix](image)

Only one comparison needed per element!  
**Runtime:** $O(n)$
## Insertion Sort

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

**Run Time:** $O(n^2)$ (but with very small constants; great for short lists)
Insertion Sort is Stable

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element

Sorted Prefix

<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>10'</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>1</th>
<th>11</th>
</tr>
</thead>
</table>

Sorted Prefix

<table>
<thead>
<tr>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>10'</th>
<th>12</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>1</th>
<th>11</th>
</tr>
</thead>
</table>

Observation: The “second” 10’ will stay to the right
**Insertion Sort**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

*Online:* Can sort a list as it is received (e.g., streamed from the network); we do not require the entire list to begin sorting.

**Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.

**Online:** Yes
Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build a sorted list (form right-to-left)

Max heap property:
Each node is larger than its children
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling \texttt{Heapify}(root)

\begin{center}
\begin{tabular}{cccccccccc}
9 & 6 & 8 & 7 & 5 & 2 & 4 & 1 & 3 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{tabular}
\end{center}

\textbf{Max heap property:}
Each node is larger than its children
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling `Heapify(root)`

Max heap property:
Each node is larger than its children
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling **Heapify**(root)

Max heap property:
Each node is larger than its children

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling `Heapify(root)`

Max heap property: Each node is larger than its children

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling **Heapify**(root)

Max heap property:
Each node is larger than its children

**Heapify**(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling **Heapify**(root)

Max heap property:
Each node is larger than its children

**Heapify**(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree.
Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling **Heapify**(root)

Max heap property:
Each node is larger than its children

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Running time: \( O(\log n) \)
Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

Running time:

- Constructing heap by calling Heapify on each node in tree (bottom up): $O(n \log n)$
- Extracting maximum element to sort list: $O(n \log n)$
Heap Sort

**In Place?** Yes

**Idea:** Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

**Run Time?** $O(n \log n)$ (constants worse than quicksort)

When removing an element from the heap, move it to the (now unoccupied) end of the list

Constructing heap is also in-place (just requires calling Heapify)
**In-Place Heap Sort**

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list

```
      10
[ 9, 6, 8, 7, 5, 2, 4, 1, 3]
```

**Max heap property:**
Each node is larger than its children
**In-Place Heap Sort**

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list

Max heap property:
Each node is larger than its children
**In-Place Heap Sort**

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

**Max heap property:** Each node is larger than its children.
**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

**Max heap property:**
Each node is larger than its children.
**In-Place Heap Sort**

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max heap property: Each node is larger than its children.
In-Place Heap Sort

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

Max heap property:
Each node is larger than its children.
**In-Place Heap Sort**

**Idea:** When removing an element from the heap, move it to the (now unoccupied) end of the list.

```
7  4  6  3  1  5  2  8  9  10
```

**Max heap property:** Each node is larger than its children.
### Heap Sort

**Idea:** Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Run Time?**

$O(n \log n)$

(constants worse than quicksort)
Sorting Algorithms

Sorting algorithms we have discussed:

- Mergesort $O(n \log n)$
- Quicksort $O(n \log n)$

Other sorting algorithms (will discuss):

- Bubble sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heapsort $O(n \log n)$

Can we do better than $O(n \log n)$?