Warm up

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

Hint: show \( n! \leq n^n \)

Hint 2: show \( n! \geq \left(\frac{n}{2}\right)^{n/2} \)
\[ \log n! = O(n \log n) \]

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

\[ n! \leq n^n \]
\[ \Rightarrow \log(n!) \leq \log(n^n) \]
\[ \Rightarrow \log(n!) \leq n \log n \]
\[ \Rightarrow \log(n!) = O(n \log n) \]
\[ \log n! = \Omega(n \log n) \]

\[ n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2} - 1\right) \cdot \ldots \cdot 2 \cdot 1 \]

\[ \left(\frac{n}{2}\right)^n = \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2} \cdot 1 \cdot \ldots \cdot 1 \cdot 1 \]

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

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

\[ \Rightarrow \log(n!) \geq \frac{n}{2} \log \frac{n}{2} \]

\[ \Rightarrow \log(n!) = \Omega(n \log n) \]
• Divide and Conquer
• Quicksort
• Decision Tree
• Worst case lower bound
• Sorting
CLRS Readings

- Chapter 7
- Chapter 8
• HW3 due 11pm Tuesday, October 1
  – Divide and conquer
  – Written (use LaTeX!)
  – Submit BOTH a pdf and a zip file (2 separate attachments)

• Regrade Office Hours
  – Thursdays 11am-12pm
  – Thursdays 4pm-5pm
Aside: Divide and Conquer
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem)  # brute force if necessary
        return solution
    subproblems[] = Divide(problem)
    for subproblem in subproblems:
        subsolutions.append(myDCalgo(subproblem))
    solution = Combine(subsolutions)
    return solution
Generic Divide and Conquer Solution

\[ \frac{n}{b^2} \quad \cdots \quad \frac{n}{b^2} \quad \cdots \quad \frac{n}{b^2} \quad \cdots \]
def mergesort(list):
    if list.length < 2:
        return list  # list of size 1 is sorted!
    {listL, listR} = Divide_by_median(list)
    for list in {listL, listR}:
        sortedSubLists.append(mergesort(list))
    solution = merge(sortedL, sortedR)
    return solution
Back to Sorting!
Quicksort

• Idea: pick a pivot element, recursively sort two sublists around that element
• Divide: select an element $p$, $\text{Partition}(p)$
• Conquer: recursively sort left and right sublists
• Combine: Nothing!
Random Pivot

• Using Quickselect to pick median guarantees $\Theta(n \log n)$ run time
  – Approach has very large constants
  – If you really want $\Theta(n \log n)$, better off using MergeSort

• Better approach: Random pivot
  – Very small constant (very fast algorithm)
  – Expected to run in $\Theta(n \log n)$ time
    • Why? Unbalanced partitions are very unlikely
Formal Argument for \( n \log n \) Average

- Remember, run time counts comparisons!
- Quicksort only compares against a pivot
  - Element \( i \) only compared to element \( j \) if one of them was the pivot
Formal Argument for $n \log n$ Average

What is the probability of comparing two given elements?

Consider the sorted version of the list

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

Only compared if 1 or 12 was chosen as the first pivot since otherwise they are in different sublists.
What is the probability of comparing two given elements?

Case 3.1: Pivot contained in $[i + 1, \ldots, j - 1]$

Then $i$ and $j$ are in different sublists and will never be compared

$$
\Pr[\text{we compare } i \text{ and } j] = 0
$$
What is the probability of comparing two given elements?

Case 3.2: Pivot is either $i$ or $j$
Then we will always compare $i$ and $j$

$$\Pr[\text{we compare } i \text{ and } j] = 1$$
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

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

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

Quicksort overall: expected $\Theta(n \log n)$
Expected number of Comparisons

Consider when $i = 1$

Compared if 1 or 2 are chosen as pivot
(these will always be compared)

Sum so far: $\frac{2}{2}$
Expected number of Comparisons

Consider when \( i = 1 \)

Compared if 1 or 3 are chosen as pivot
(but never if 2 is ever chosen)

Sum so far: \( \frac{2}{2} + \frac{2}{3} \)
Expected number of Comparisons

Consider when \( i = 1 \)

Compared if 1 or 4 are chosen as pivot
(but never if 2 or 3 are chosen)

Sum so far: \( \frac{2}{2} + \frac{2}{3} + \frac{2}{4} \)
Expected number of Comparisons

Consider when $i = 1$

Compared if 1 or 12 are chosen as pivot
(but never if 2 -> 11 are chosen)

Overall sum: $\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{n}$
Expected number of Comparisons

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

When \( i = 1 \):

\[ 2 \left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \right) < 2 \sum_{x=1}^{n} \frac{1}{x} = O(\log n) \]

\( n \) terms overall in the outer sum

Quicksort overall: expected \( O(n \log n) \)
• Sorting algorithms we have discussed:
  – Mergesort \( O(n \log n) \)
  – Quicksort \( O(n \log n) \)

• Other sorting algorithms (will discuss):
  – Bubblesort \( O(n^2) \)
  – Insertionsort \( O(n^2) \)
  – 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 $O(n \log n)$
  – Every algorithm, in the worst case, must have a certain lower bound
• Non-existence proof!
  – Very hard to do
• Sorting algorithms use comparisons to figure out the order of input elements
• Draw tree to illustrate all possible execution paths

Strategy: Decision Tree
Strategy: Decision Tree

- Worst case run time is the longest execution path
- i.e., “height” of the decision tree

Possible execution path

One comparison

Result of comparison

$\log(n!)$

$\Theta(n \log n)$

Possible permutations

$n!$ Possible permutations

Permutation of sorted list
Strategy: Decision Tree

- Conclusion: Worst Case Optimal run time of sorting is $\Theta(n \log n)$
  - There is no (comparison-based) sorting algorithm with run time $o(n \log n)$
Sorting, so far

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

• Other sorting algorithms (will discuss):
  – Bubblesort $O(n^2)$
  – Insertionsort $O(n^2)$
  – Heapsort $O(n \log n)$  Optimal!
Important properties of sorting algorithms:

- **Run Time**
  - Asymptotic Complexity
  - Constants

- **In Place (or In-Situ)**
  - Done with only constant additional space

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

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

- **Parallelizable**
  - Runs faster with multiple computers
**Mergesort**

- **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?**

$\Theta(n \log n)$

Optimal!

---

**In Place?**

No

**Adaptive?**

No

**Stable?**

Yes! (usually)
• **Combine:** Merge sorted sublists into one sorted list

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

While \((L_1\) and \(L_2\) not empty):

\[
\begin{align*}
\text{If } L_1[0] & \leq L_2[0]: \\
L_{out}.append(L_1.pop()) \\
\text{Else:} \\
L_{out}.append(L_2.pop())
\end{align*}
\]

Stable:
If elements are equal, leftmost comes first
Mergesort

• **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?**
\( \Theta(n \log n) \)
Optimal!

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Yes!</td>
<td>Yes! (usually)</td>
</tr>
</tbody>
</table>
Mergesort

• **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 machines to work on each sublist
Mergesort (Sequential)

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

- \( n \) total / level
- \( \log_2 n \) levels of recursion

Run Time: \( \Theta(n \log n) \)
Mergesort (Parallel)

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

Run Time: \( \Theta(n) \)
Quicksort

- Idea: pick a partition element, recursively sort two sublists around that element
- Divide: select an element \( p \), \( \text{Partition}(p) \)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

Run Time?
\[ \Theta(n \log n) \]
(almost always)
Better constants than Mergesort

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>kinda</td>
<td>No!</td>
<td>No</td>
<td>Yes!</td>
</tr>
<tr>
<td>Uses stack for recursive calls</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Bubble Sort**

- **Idea:** March 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>
</tbody>
</table>
### Bubble Sort

- **Run Time?**
  - $\Theta(n^2)$

- **Constants worse than Insertion Sort**

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

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

“Compared to straight insertion [...], bubble sorting requires a more complicated program and takes about twice as long!”

—Donald Knuth
Bubble Sort is “almost” Adaptive

- **Idea:** March through list, swapping adjacent elements if out of order

<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>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>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Only makes one “pass”

<table>
<thead>
<tr>
<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>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<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>11</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

After one “pass”

<table>
<thead>
<tr>
<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>1</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<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>11</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>

Requires $n$ passes, thus is $O(n^2)$
Bubble Sort

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

<table>
<thead>
<tr>
<th>In Place?</th>
<th>Adaptive?</th>
<th>Stable?</th>
<th>Parallelizable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes!</td>
<td>Kinda</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Not really</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"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*.

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

Sorted Prefix
Insertion Sort

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

In Place?  Yes!
Adaptive?  Yes

Run Time?
Θ(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.

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

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

### Characteristics

<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>

### Run Time

$$\Theta(n^2)$$

(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

The “second” 10 will stay to the right
### Insertion Sort

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

<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>Yes</td>
</tr>
</tbody>
</table>

- **Run Time:** $\Theta(n^2)$ (but with very small constants).
  - Great for short lists!

- **Can sort a list as it is received, i.e., don’t need the entire list to begin sorting.**

- **“All things considered, it’s actually a pretty good sorting algorithm!”** – Nate Brunelle
Heap Sort

- **Idea**: Build a Heap, repeatedly extract max element from the heap to build sorted list 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: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

• Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

- Remove the Max element (i.e. the root) from the Heap: replace with last element, call Heapify(root)

Max Heap Property: Each node is larger than its children

Heapify(node): if node satisfies heap property, done. Else swap with largest child and recurse on that subtree
Heap Sort

- **Idea**: Build a Heap, repeatedly extract max element from the heap to build sorted list Right-to-Left

**In Place?** Yes!

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

**Run Time?** $\Theta(n \log n)$

Constants worse than Quick Sort
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.

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.
## Heap Sort

### Idea:
Build a Heap, repeatedly extract max element from the heap to build sorted list 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?
$\Theta(n \log n)$

Constants worse than Quick Sort.