Lecture 7
Heaps, Heapsort, Stable sorting, Optimality of Heapsort/Mergesort (revisited)

CS 161 Design and Analysis of Algorithms
Ioannis Panageas
Heapsort

Consider the following version of Selection Sort (sometimes called Max sort):

```python
def maxSort(A, n):
    for k = n - 1 downto 1
        find j such that A[j] == max(A[0], A[1], ..., A[k])
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first $k$ items.

But we can speed this up by using a binary heap.
Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```python
def maxSort(A,n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0],A[1],..., A[k])
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first $k$ items. But we can speed this up by using a binary heap.
Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```python
def maxSort(A, n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0], A[1], ..., A[k])
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first $k$ items.
Heapsort

Consider the following version of Selection Sort (sometimes called Max sort)

```python
def maxSort(A, n):
    for k = n-1 downto 1
        find j such that A[j] == max(A[0], A[1], ..., A[k])
```

A straightforward implementation requires $O(n^2)$ time, because of the time spent repeatedly finding the maximum of the first $k$ items.

But we can speed this up by using a binary heap.
Priority Queues and Heaps

- Priority Queue
- Abstract data type
- Collection of items.
- Each item has an associated key, which corresponds to a priority.
- Supports the following operations:
  - Insert an item with a given key
  - Delete an item
  - Select the item with the most urgent priority in the priority queue.
- Most urgent priority may correspond to the lowest key value or to the highest key value, depending on the application.
Priority Queues and Heaps

- Priority Queue

  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations:
    - Insert an item with a given key
    - Delete an item
    - Select the item with the most urgent priority in the priority queue.
  - Most urgent priority may correspond to the lowest key value or to the highest key value, depending on the application.
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
Priority Queues and Heaps

- **Priority Queue**
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item
Priority Queues and Heaps

- Priority Queue
  - Abstract data type
  - Collection of items.
  - Each item has an associated key, which corresponds to a priority.
  - Supports the following operations
    - Insert an item with a given key
    - Delete an item
    - Select the item with the most urgent priority in the priority queue.
Priority Queues and Heaps

▶ Priority Queue
  ▶ Abstract data type
  ▶ Collection of items.
  ▶ Each item has an associated key, which corresponds to a priority.
  ▶ Supports the following operations
    ▶ Insert an item with a given key
    ▶ Delete an item
    ▶ Select the item with the most urgent priority in the priority queue.
  ▶ Most urgent priority may correspond to the lowest key value or to the highest key value, depending on the application.
Binary Heaps

Specific implementation of priority queue

Items are stored in an array.

The array represents a binary tree in level order (breadth-first order).

Can be max-heap or min-heap

In a max-heap, large key values represent more urgent priorities

In a min-heap, small key values represent more urgent priorities

In this introduction, we will be using a max-heap.

Heap invariant for max-heaps: For any item \( v \) other than the root, \( \text{key}(\text{parent}(v)) \geq \text{key}(v) \)

In a min-heap, the direction of the inequality is reversed.

In our examples, items are integers, key is the integer value
Binary Heaps

- Specific implementation of priority queue

- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities

- In this introduction, we will be using a max-heap.

- Heap invariant for max-heaps: For any item $v$ other than the root, $\text{key}(\text{parent}(v)) \geq \text{key}(v)$.
- In a min-heap, the direction of the inequality is reversed.

- In our examples, items are integers, key is the integer value.
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item \( v \) other than the root,
  \[
  \text{key(parent}(v)) \geq \text{key}(v)
  \]
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be max-heap or min-heap
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- Heap invariant for max-heaps: For any item $v$ other than the root,
  
  \[ \text{key(parent}(v)) \geq \text{key}(v) \]

- In a min-heap, the direction of the inequality is reversed.
Binary Heaps

- Specific implementation of priority queue
- Items are stored in an array.
- The array represents a binary tree in level order (breadth-first order).
- Can be **max-heap** or **min-heap**
  - In a max-heap, large key values represent more urgent priorities
  - In a min-heap, small key values represent more urgent priorities
- In this introduction, we will be using a max-heap.
- **Heap invariant for max-heaps**: For any item \( v \) other than the root,
  \[
  \text{key}(\text{parent}(v)) \geq \text{key}(v)
  \]
- In a min-heap, the direction of the inequality is reversed.
- In our examples, items are integers, key is the integer value
Viewing the array as a binary tree
Viewing the array as a binary tree

<table>
<thead>
<tr>
<th></th>
<th>83</th>
<th>79</th>
<th>27</th>
<th>36</th>
<th>23</th>
<th>18</th>
<th>15</th>
<th>14</th>
<th>31</th>
<th>20</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></td>
</tr>
</tbody>
</table>
Viewing the array as a binary tree

![Binary tree diagram]

- Root is $H[0]$
- Left child of $H[i]$ is $H[2i+1]$ (provided $2i+1 < n$, where $n = H$.size)
- Right child of $H[i]$ is $H[2i+2]$ (provided $2i+2 < n$)
- Parent of $H[i]$ is $H[⌊(i−1)/2⌋]$ (provided $i > 0$)
Viewing the array as a binary tree

- **Root** is $H[0]$
Viewing the array as a binary tree

- **Root** is \( H[0] \)
- **Left child** of \( H[i] \) is \( H[2i + 1] \) (provided \( 2i + 1 < n \), where \( n = H\text{.size} \))
Viewing the array as a binary tree

- **Root** is $H[0]$
- **Left child** of $H[i]$ is $H[2i + 1]$ (provided $2i + 1 < n$, where $n = H$.size)
- **Right child** of $H[i]$ is $H[2i + 2]$ (provided $2i + 2 < n$)
Viewing the array as a binary tree

- **Root** is $H[0]$
- **Left child of** $H[i]$ is $H[2i + 1]$ (provided $2i + 1 < n$, where $n = H$.size)
- **Right child of** $H[i]$ is $H[2i + 2]$ (provided $2i + 2 < n$)
- **Parent of** $H[i]$ is $H[⌊(i − 1)/2⌋]$ (provided $i > 0$)

```
<table>
<thead>
<tr>
<th>83</th>
<th>79</th>
<th>27</th>
<th>36</th>
<th>23</th>
<th>18</th>
<th>15</th>
<th>14</th>
<th>31</th>
<th>20</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>
</tr>
</tbody>
</table>
```

```
83
   / \   /
  79 27
   / \   /
 36 23 18
     / \ /
    14 31 15
     /
    20
```
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
- **ExtractMax(H)**: Find maximum item and delete it from the heap
- **Insert(H,x)**: Insert the new item x in the heap
- **Delete(H,i)**: Delete the item at location i from the heap
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
- **ExtractMax(H)**: Find maximum item and delete it from the heap
- **Insert(H, x)**: Insert the new item \( x \) in the heap
- **Delete(H, i)**: Delete the item at location \( i \) from the heap
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
- **ExtractMax(H)**: Find maximum item and delete it from the heap
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
- **ExtractMax(H)**: Find maximum item and delete it from the heap
- **Insert(H,x)**: Insert the new item $x$ in the heap
- **Delete(H,i)**: Delete the item at location $i$ from the heap
Heap operations in a max-heap:

- **FindMax(H)**: Find maximum item in the heap
- **ExtractMax(H)**: Find maximum item and delete it from the heap
- **Insert(H,x)**: Insert the new item $x$ in the heap
- **Delete(H,i)**: Delete the item at location $i$ from the heap
FindMax: Find maximum item in the heap

```python
def FindMax(H):
    return H[0]
```
FindMax: Find maximum item in the heap
FindMax: Find maximum item in the heap

Findmax is easy: just report the value at the root.
**FindMax**: Find maximum item in the heap

*Findmax* is easy: just report the value at the root.

```python
def FindMax(H):
    return H[0]
```

![Heap Diagram]

---

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Helper functions

Except for FindMax, the binary heap operations require some data movement. The heap invariant must be preserved after each operation.

We define two helper functions.

$\text{SiftUp}(H, i)$: Move the item at location $i$ up to its correct position by repeatedly swapping the item with its parent, as necessary.

$\text{SiftDown}(H, i)$: Move the item at location $i$ down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.

[GT] calls these “up-heap bubbling” and “down-heap bubbling”
Helper functions

- Except for **FindMax**, the binary heap operations require some data movement.
Helper functions

- Except for \texttt{FindMax}, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
Except for \textbf{FindMax}, the binary heap operations require some data movement.

The heap invariant must be preserved after each operation.

We define two helper functions.

- \textbf{SiftUp}(H,i): Move the item at location \textit{i} up to its correct position by repeatedly swapping the item with its parent, as necessary.
Helper functions

- Except for FindMax, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - SiftUp(H, i): Move the item at location i up to its correct position by repeatedly swapping the item with its parent, as necessary.
  - SiftDown(H, i): Move the item at location i down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.
Helper functions

- Except for \texttt{FindMax}, the binary heap operations require some data movement.
- The heap invariant must be preserved after each operation.
- We define two helper functions.
  - \texttt{SiftUp(H,i)}: Move the item at location $i$ up to its correct position by repeatedly swapping the item with its parent, as necessary.
  - \texttt{SiftDown(H,i)}: Move the item at location $i$ down to its correct position by repeatedly swapping the item with the child having the larger key, as necessary.

[GT] calls these "up-heap bubbling" and "down-heap bubbling"
**SiftUp**: Sift an item up to its correct position

```python
def SiftUp(H, i):
    parent = (i - 1) // 2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H, parent)
```

Analysis: at most 1 comparison at each level, so total time is $O(\log n)$
SiftUp: Sift an item up to its correct position

```python
def SiftUp(H, i):
    parent = (i - 1) // 2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i], H[parent] = H[parent], H[i]
        SiftUp(H, parent)
```

Analysis: at most 1 comparison at each level, so total time is $O(\log n)$
SiftUp: Sift an item up to its correct position

def SiftUp(H, i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H, parent)
SiftUp: Sift an item up to its correct position

```python
def SiftUp(H, i):
    parent = (i-1)/2;
    if (i > 0) and (H[parent].key < H[i].key):
        H[i] ↔ H[parent]
        SiftUp(H, parent)
```

Analysis: at most 1 comparison at each level, so total time is $O(\log n)$
**SiftDown**: Sift an item down to its correct position

```python
def SiftDown(H, i):
    n = H.size // number of item in heap
    left = 2i + 1; right = 2i + 2
    if (right < n) and (H[right].key > H[left].key):
        largerChild = right
    else:
        largerChild = left
    if (largerChild < n) and (H[i].key < H[largerChild].key):
        H[i], H[largerChild] = H[largerChild], H[i]
        SiftDown(H, largerChild)
```

Analysis: at most 2 comparisons at each level, so total time is $O(\log n)$
SiftDown: Sift an item down to its correct position
SiftDown: Sift an item down to its correct position

```python
def SiftDown(H, i):
    n = H.size // number of item in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key):
        largerChild = right
    else:
        largerChild = left
    if (largerChild < n) and (H[i].key < H[largerChild].key):
        H[i], H[largerChild] = H[largerChild], H[i]
        SiftDown(H, largerChild)
```

Analysis: at most 2 comparisons at each level, so total time is $O(\log n)$. 

```
14 31 20
36 23 18 15
79 27
83
```
SiftDown: Sift an item down to its correct position

def SiftDown(H,i):
    n = H.size // number of item in heap
    left = 2i+1; right = 2i+2
    if (right < n) and (H[right].key > H[left].key):
        largerChild = right
    else:
        largerChild = left
    if (largerchild < n) and (H[i].key < H[largerChild].key):
        H[i] ↔ H[largerchild]
        SiftDown(H,largerchild)

Analysis: at most 2 comparisons at each level, so total time is $O(\log n)$
Insert: Insert the new item $x$

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1       # index of last position
    H[k] = x             # insert x in last position
    SiftUp(H, k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$
Insert: Insert the new item $x$

```
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1       # index of last position
    H[k] = x             # insert $x$ in last position
    SiftUp(H, k)
```

Analysis: Siftup time dominates, so total time is $O(\log n)$.
Insert: Insert the new item $x$

def Insert(H, x):
    H.size = H.size + 1 // increment number of items
    k = H.size - 1 // index of last position
    H[k] = x // insert $x$ in last position
    SiftUp(H, k)
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1  # index of last position
    H[k] = x  # insert x in last position
    SiftUp(H, k)

Analysis: Siftup time dominates, so total time is $O(\log n)$
Insert: Insert the new item $x$

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1  # index of last position
    H[k] = x  # insert x in last position
    SiftUp(H, k)
```

**Analysis:** Siftup time dominates, so total time is $O(\log n)$

**Insert(H, 81)**
**Insert: Insert the new item $x$**

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1       # index of last position
    H[k] = x             # insert x in last position
    SiftUp(H, k)
```

**Analysis:** Siftup time dominates, so total time is $O(\log n)$

**Insert** $(H, 81)$

![Insertion Diagram]
**Insert: Insert the new item $x$**

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1  # index of last position
    H[k] = x  # insert $x$ in last position
    SiftUp(H, k)
```

**Analysis:** Siftup time dominates, so total time is $O(\log n)$

**Insert($H, 81$)**

![Binary tree before and after insertion](image-url)
Insert: Insert the new item $x$

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1  # index of last position
    H[k] = x  # insert x in last position
    SiftUp(H, k)
```

**Analysis:** Siftup time dominates, so total time is $O(\log n)$

---

**Insert(H, 81)**

![Binary Tree Diagram]

1. Insert 81 into the binary tree.
2. The tree structure changes to accommodate the new element.

---

![Binary Tree Diagram]

1. The updated binary tree after inserting 81.
**Insert: Insert the new item x**

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1  # index of last position
    H[k] = x  # insert x in last position
    SiftUp(H, k)
```

**Analysis:** Siftup time dominates, so total time is $O(\log n)$

---

Insert(H, 81)
Insert: Insert the new item $x$

```python
def Insert(H, x):
    H.size = H.size + 1  # increment number of items
    k = H.size - 1       # index of last position
    H[k] = x             # insert x in last position
    SiftUp(H, k)
```

**Analysis**: Siftup time dominates, so total time is $O(\log n)$

**Insert(H, 81)**

![Binary Search Tree](image)
Delete: Delete the item at location $i$
Delete: Delete the item at location $i$
Delete: Delete the item at location $i$

```python
def Delete(H, i):
    k = H.size - 1  # index of last position
    H[i] = H[k]     # overwrite item being deleted with
element in last position
    H.size = H.size - 1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

Analysis: Siftup/siftdown time dominates, so total time is $O(\log n)$

```
14 31 20
36 23 18 15
79 27
83
```
Delete: Delete the item at location \( i \)

```python
def Delete(H, i):
    k = H.size-1  # index of last position
    H[i] = H[k]  # overwrite item being deleted with
                 # element in last position
    H.size = H.size-1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is \( O(\log n) \)
Delete: Delete the item at location $i$

```python
def Delete(H,i):
    k = H.size-1  # index of last position
    H[i] = H[k]   # overwrite item being deleted with element in last position
    H.size = H.size-1  # decrement number of item
    SiftUp(H,i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H,i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is $O(\log n)$

**Delete(H,3)**
Delete: Delete the item at location \( i \)

```python
def Delete(H, i):
    k = H.size - 1  # index of last position
    H[i] = H[k]  # overwrite item being deleted with
                  # element in last position
    H.size = H.size - 1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is \( O(\log n) \)

Delete(\( H, 3 \))
Delete: Delete the item at location $i$

```python
def Delete(H, i):
    k = H.size - 1  # index of last position
    H[i] = H[k]  # overwrite item being deleted with element in last position
    H.size = H.size - 1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is $O(\log n)$
Delete: Delete the item at location \( i \)

```python
def Delete(H, i):
    k = H.size - 1  # index of last position
    H[i] = H[k]  # overwrite item being deleted with element in last position
    H.size = H.size - 1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is \( O(\log n) \)

Delete(H, 3)
Delete: Delete the item at location \( i \)

```python
def Delete(H, i):
    k = H.size - 1  # index of last position
    H[i] = H[k]  # overwrite item being deleted with element in last position
    H.size = H.size - 1  # decrement number of item
    SiftUp(H, i)  # either SiftUp or SiftDown will do nothing
    SiftDown(H, i)
```

**Analysis:** Siftup/siftdown time dominates, so total time is \( O(\log n) \)

Delete(H, 3)
ExtractMax: Find maximum item and delete it
ExtractMax: Find maximum item and delete it

```python
def ExtractMax(H):
    x = H[0]
    Delete(H, 0)
    return x
```

Analysis: Delete time dominates, so total time is $O(\log n)$.
ExtractMax: Find maximum item and delete it

```python
def ExtractMax(H):
    x = H[0]
    Delete(H, 0)
    return x
```

Analysis: Delete time dominates, so total time is $O(\log n)$. 

```
83
 /    \
79    27
 / \
36 23 18 15
 / \
14 31 20
```
ExtractMax: Find maximum item and delete it

```python
def ExtractMax(H):
    x = H[0]
    Delete(H,0)
    return x
```

**Analysis:** Delete time dominates, so total time is $O(\log n)$
Constructing a heap

How do we efficiently construct a brand-new heap storing $n$ given items? If we insert the items one at a time, time spent on the $k$th insertion is $O(\log k)$. So total time is $O(n - 1 \sum_{k=1}^{n} \log k) = O(n \log n)$.

There is a better way that only requires $O(n)$ time.
Constructing a heap

How do we efficiently construct a brand-new heap storing $n$ given item?
Constructing a heap

How do we efficiently construct a brand-new heap storing $n$ given item?

If we insert the items one at a time, time spent on $k$th insertion is $O(\log k)$.
Constructing a heap

How do we efficiently construct a brand-new heap storing \( n \) given item?

If we insert the items one at a time, time spent on \( k \)th insertion is \( O(\log k) \).

So total time is

\[
O \left( \sum_{k=1}^{n-1} \log k \right) = O(n \log n)
\]
Constructing a heap

How do we efficiently construct a brand-new heap storing $n$ given item?

If we insert the items one at a time, time spent on $k$th insertion is $O(\log k)$.

So total time is

$$O\left(\sum_{k=1}^{n-1} \log k\right) = O(n \log n)$$

There is a better way that only requires $O(n)$ time...
Constructing a heap in $O(n)$ time

1. Put the data in $H$, in arbitrary order. (So $H$ stores the correct data, but does not satisfy the heap invariant.)
2. Run the following Heapify function.

```python
def heapify(H, n):
    for i = n-1 down to 0:
        SiftDown(H, i)
```

The code given above can be improved: We can start at $i = \lfloor (n-2)/2 \rfloor$ (or equivalently, $i = \lfloor n/2 \rfloor - 1$), rather than $i = n - 1$. 
Constructing a heap in $O(n)$ time

1. Put the data in $H$, in arbitrary order. (So $H$ stores the correct data, but does not satisfy the heap invariant.)
Constructing a heap in $O(n)$ time

1. Put the data in $H$, in arbitrary order. (So $H$ stores the correct data, but does not satisfy the heap invariant.)
2. Run the following $\text{Heapify}$ function.

```python
def heapify(H, n):
    for i = n-1 down to 0:
        SiftDown(H, i)
```

The code given above can be improved: We can start at $i = \lfloor (n-2)/2 \rfloor$ (or equivalently, $i = \lfloor n/2 \rfloor - 1$), rather than $i = n - 1$. 
Constructing a heap in $O(n)$ time

1. Put the data in $H$, in arbitrary order. (So $H$ stores the correct data, but does not satisfy the heap invariant.)

2. Run the following `Heapify` function.

```python
def heapify(H, n)
    for i = n-1 down to 0:
        SiftDown(H, i)
```

The code given above can be improved: We can start at $i = \lfloor (n-2)/2 \rfloor$ (or equivalently, $i = \lfloor n/2 \rfloor - 1$), rather than $i = n - 1$.  

Heapify example

13 23 18 94 42 12 37 81 52 56
Heapify example, continued

13 23 18 94 42 12 37 81 52 56

```
0 1 2 3 4 5 6 7 8 9
13 23 18 94 56 12 37 81 52 42
```

```
13
  / \  \
23 23
  / \ /\ \
94 94 94
  / \ /\ /\ \
56 56 56 56
 / \ /\ /\ /\ \
81 81 81 81 81
```

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Heapify example, continued

13 23 18 94 42 12 37 81 52 56
Heapify example, continued

13 23 18 94 42 12 37 81 52 56
Heapify example, continued
Heapify example, continued

13 23 18 94 42 12 37 81 52 56
Analysis of heap construction algorithm using Heapify
Analysis of heap construction algorithm using Heapify

Algorithm heapify(H,n);
    for i = n-1 down to 0:
        SiftDown(H,i)
Analysis of heap construction algorithm using Heapify

Algorithm heapify(H,n);
    for i = n-1 down to 0:
        SiftDown(H,i)

- **Correctness:** After SiftDown(H,i) is executed, subtree rooted at node $i$ satisfies heap invariant. (Can show by induction).
Analysis of heap construction algorithm using Heapify

Algorithm heapify(H,n);
    for i = n-1 down to 0:
        SiftDown(H,i)

- **Correctness:** After SiftDown(H,i) is executed, subtree rooted at node i satisfies heap invariant. (Can show by induction).
- **Running time:** Heapify runs in $O(n)$ time. We will prove this on the next slide.
Proof that Heapify runs in $O(n)$ time

Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^d + 1$).

If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.

There are at most $2^j$ nodes at level $j$.

So total number of comparisons is no more than:

$$d \sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2d \sum_{j=0}^{d} j2^j = 2d(2^d + 1 - 1) - 2d \cdot 2^{d+1} = 2d \cdot 2^d - 2d - 2d \cdot 2^d = 2d \cdot 2^d - 4d < 4 \cdot 2^d \leq 4n = O(n)$$

So heap can be constructed using $O(n)$ comparisons.
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H,i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$. 
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, `SiftDown(H,i)` needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

$$\sum_{j=0}^{d} 2(d - j)2^j$$
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

$$\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j$$

So heap can be constructed using $O(n)$ comparisons.
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

\[
\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j
\]

\[
= 2d(2^{d+1} - 1) - 2 \left[ (d - 1)2^{d+1} + 2 \right]
\]
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

$$
\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j
$$

$$
= 2d(2^{d+1} - 1) - 2 \left[(d - 1)2^{d+1} + 2\right]
$$

$$
= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4
$$

$\text{So heap can be constructed using } O(n) \text{ comparisons.}$
Proof that Heapify runs in $O(n)$ time

> Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
> If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
> There are at most $2^j$ nodes at level $j$.
> So total number of comparisons is no more than:

\[
\begin{align*}
\sum_{j=0}^{d} 2(d - j)2^j &= 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j \\
&= 2d(2^{d+1} - 1) - 2 \left[ (d - 1)2^{d+1} + 2 \right] \\
&= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4 \\
&= 4 \cdot 2^d - 2d - 4
\end{align*}
\]
Proof that Heapify runs in $O(n)$ time

▶ Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
▶ If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
▶ There are at most $2^j$ nodes at level $j$.
▶ So total number of comparisons is no more than:

$\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j$

$= 2d(2^{d+1} - 1) - 2 \left[(d - 1)2^{d+1} + 2\right]$

$= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4$

$= 4 \cdot 2^d - 2d - 4$

$< 4 \cdot 2^d$
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H, i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

$$
\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j \\
= 2d(2^{d+1} - 1) - 2 \left[ (d - 1)2^{d+1} + 2 \right] \\
= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4 \\
= 4 \cdot 2^d - 2d - 4 \\
< 4 \cdot 2^d \leq 4n
$$

So heap can be constructed using $O(n)$ comparisons.
Proof that Heapify runs in $O(n)$ time

- Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
- If node $i$ is at level $j$, $\text{SiftDown}(H,i)$ needs $\leq 2(d - j)$ comparisons.
- There are at most $2^j$ nodes at level $j$.
- So total number of comparisons is no more than:

$$
\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j
$$

$$
= 2d(2^{d+1} - 1) - 2 \left[ (d - 1)2^{d+1} + 2 \right]
$$

$$
= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4
$$

$$
= 4 \cdot 2^d - 2d - 4
$$

$$
< 4 \cdot 2^d \leq 4n = O(n)
$$
Proof that Heapify runs in $O(n)$ time

> Suppose the tree has $n$ nodes and $d$ levels (so $2^d \leq n < 2^{d+1}$).
> If node $i$ is at level $j$, $\text{SiftDown}(H,i)$ needs $\leq 2(d - j)$ comparisons.
> There are at most $2^j$ nodes at level $j$.

So total number of comparisons is no more than:

\[
\sum_{j=0}^{d} 2(d - j)2^j = 2d \sum_{j=0}^{d} 2^j - 2 \sum_{j=0}^{d} j2^j
\]

\[
= 2d(2^{d+1} - 1) - 2 \left[(d - 1)2^{d+1} + 2\right]
\]

\[
= 2d2^{d+1} - 2d - 2d2^{d+1} + 2 \cdot 2^{d+1} - 4
\]

\[
= 4 \cdot 2^d - 2d - 4
\]

\[
< 4 \cdot 2^d \leq 4n = O(n)
\]

So heap can be constructed using $O(n)$ comparisons.
Heapsort: version based on Max Sort

```python
def heapsort(A, n):
    heapify(A, n) // form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
```
Heapsort: version based on Max Sort

```python
def heapsort(A, n):
    heapify(A, n)  # form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
```

CompSci 161—Spring 2022—© M. B. Dillencourt—University of California, Irvine
def heapsort(A, n):
    heapify(A, n) // form max heap using array A
    for k = n-1 down to 1:
        A[k] = ExtractMax(A)
Heapsort example

Sort: 13 23 18 94 42 12 37 81 52 56

Heapify:
Heapsort example, continued
Heapsort example, continued
Heapsort example, continued

Exercise: Finish this example.
Analysis of Heapsort

Storage: $O(1)$ extra space (in place)

Time:

- Heapify: $O(n)$
- All calls to ExtractMax:
  $n - 1 \sum_{k=1}^{n} O(log(k + 1)) = O(n \log n)$

Hence total time is $O(n \log n)$. 
Analysis of Heapsort

- **Storage:** $O(1)$ extra space (in place)
Analysis of Heapsort

- **Storage:** $O(1)$ extra space (in place)
- **Time:**

  

  Heapify: $O(n)$

  All calls to ExtractMax:

  $n - 1 \sum_{k=1}^{n} O(\log(k+1)) = O(n \log n)$

  Hence total time is $O(n \log n)$. 

CompSci 161—Spring 2022—© M. B. Dillencourt—University of California, Irvine
Analysis of Heapsort

- **Storage:** $O(1)$ extra space (in place)
- **Time:**
  - **Heapify:** $O(n)$

\[
\sum_{k=1}^{n-1} O(\log(k+1)) = O(n \log n)
\]

Hence total time is $O(n \log n)$. 
Analysis of Heapsort

- **Storage:** $O(1)$ extra space (in place)
- **Time:**
  - **Heapify:** $O(n)$
  - All calls to **ExtractMax:**

\[
\sum_{k=1}^{n-1} O(\log(k + 1)) = O(n \log n)
\]
Analysis of Heapsort

- **Storage:** $O(1)$ extra space (in place)
- **Time:**
  - **Heapify:** $O(n)$
  - All calls to **ExtractMax**:
    \[
    \sum_{k=1}^{n-1} O(\log(k + 1)) = O(n \log n)
    \]
  - Hence total time is $O(n \log n)$. 
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```python
def heapsort(A, n):
    heapify(A, n)  # Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
- If we stop after computing the first $k$ entries, total work is $O(n + k \log n)$
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```
def heapsort(A, n):
    heapify(A, n)  # Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

Same analysis as previous version:

- $O(n \log n)$ time,
- $O(1)$ extra space

If we stop after computing the first $k$ entries, total work is $O(n + k \log n)$. 
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```python
def heapsort(A, n):
    heapify(A, n)  # Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
- If we stop after computing the first $k$ entries, total work is $O(n + k \log n)$
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```python
def heapsort(A, n):
    heapify(A, n) // Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
Heapsort: Alternate version

- Uses a min-heap (instead of a max-heap)
- Output items in sorted order rather than storing them back in the array

```python
def heapsort(A, n):
    heapify(A, n)  # Form min heap
    for k = 1 to n:
        x = ExtractMin(A)
        output(x)
```

- Same analysis as previous version: $O(n \log n)$ time, $O(1)$ extra space
- If we stop after computing the first $k$ entries, total work is $O(n + k \log n)$
## Comparison-based sorts: Summary/Comparison

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst-case Time Requirement</th>
<th>Storage Requirement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>In-place</td>
<td>Good if input is almost sorted.</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(n^2)$</td>
<td>$O(\log n)$ extra</td>
<td>$O(n \log n)$ expected time.</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ extra</td>
<td>For merge</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>In-place</td>
<td>Can output $k$ smallest in sorted order in $O(n + k \log n)$ time.</td>
</tr>
</tbody>
</table>
Comparison-based sorts: Summary/Comparison

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst-case Time</th>
<th>Storage Requirement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
<td>In-place</td>
<td>Good if input is almost sorted.</td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(n^2)$</td>
<td>$O(\log n)$ extra for stack</td>
<td>$O(n \log n)$ expected time.</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$ extra for merge</td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>In-place</td>
<td>Can output $k$ smallest in sorted order in $O(n + k \log n)$ time.</td>
</tr>
</tbody>
</table>
Stable sorting

A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.

\[\begin{array}{c|c|c|c}
3 & 2 & 1 & 2 \\
\hline
1 & 2 & 2 & 3 \\
\end{array}\]

Stable

\[\begin{array}{c|c|c|c}
3 & 2 & 1 & 2 \\
\hline
1 & 2 & 2 & 3 \\
\end{array}\]

Not Stable

<table>
<thead>
<tr>
<th>Sort</th>
<th>Stable (without special care)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick</td>
<td>No</td>
</tr>
<tr>
<td>Merge</td>
<td>Yes (as described here)</td>
</tr>
<tr>
<td>Heap</td>
<td>No</td>
</tr>
</tbody>
</table>

CompSci 161—Spring 2022—© M. B. Dillencourt—University of California, Irvine
Stable sorting

A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.
Stable sorting

A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.

\[ [3 \ 2 \ 1 \ 2] \rightarrow [1 \ 2 \ 2 \ 3] : \text{Stable} \]
Stable sorting

A sort is **stable** if keys having the same value appear in the same order in the output array as they do in the input array.

\[
\begin{align*}
[3 \ 2 \ 1 \ 2] & \rightarrow [1 \ 2 \ 2 \ 3] : \text{Stable} \\
[3 \ 2 \ 1 \ 2] & \rightarrow [1 \ 2 \ 2 \ 3] : \text{Not Stable}
\end{align*}
\]
Stable sorting

A sort is stable if keys having the same value appear in the same order in the output array as they do in the input array.

\[
\begin{align*}
[3 \ 2 \ 1 \ 2] & \rightarrow [1 \ 2 \ 2 \ 3] : \text{Stable} \\
[3 \ 2 \ 1 \ 2] & \rightarrow [1 \ 2 \ 2 \ 3] : \text{Not Stable}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Sort</th>
<th>Stable (without special care)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick-Sort</td>
<td>No</td>
</tr>
<tr>
<td>Merge-Sort</td>
<td>Yes (as described here)</td>
</tr>
<tr>
<td>Heap-Sort</td>
<td>No</td>
</tr>
</tbody>
</table>
Lower bound on comparison-based sorting

Based on Decision Tree model.

Any algorithm that sorts a list or array of size \( n \) using comparisons can be modeled as a decision tree:

- Each internal node is labeled \( i:j \), representing a comparison between \( L[i] \) and \( L[j] \).
- The left (respectively, right) of a node labeled \( i:j \) describes what happens if \( L[i] < L[j] \) (respectively, \( L[i] > L[j] \)).
- Each leaf node is a permutation of 0, ..., \( n-1 \).

Example: Decision tree for sorting 3 items

```
0 1 2
0 2 1
1 2 0
2 1 0

1 : 2
2 0 1
1 0 2
1 : 2
0 : 2
0 : 2
0 : 1
```

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Lower bound on comparison-based sorting

- Based on Decision Tree model.
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.

**Example:** Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size \( n \) using comparisons can be modeled as a decision tree:
  - Each internal node is labeled \( i : j \), representing a comparison between \( L[i] \) and \( L[j] \).
  - The left (respectively, right) of a node labeled \( i : j \) describes for what happens if \( L[i] < L[j] \) (respectively, \( L[i] > L[j] \)).

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i:j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i:j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n-1$.

Example: Decision tree for sorting 3 items

```
0 1 2
0 2 1
1 2 0
```
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i:j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i:j$ describes what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n-1$.

**Example:** Decision tree for sorting 3 items
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items

```
0 1 2
0 2 1
1 2 0
1 0 2
1 : 2
0 : 2
0 : 1
```

CompSci 161—Spring 2022—©M. B. Dillencourt—University of California, Irvine
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items

```
0 1 2 0 2 1 1 2 0 2 1 0
1 : 2 2 0 1 1 0 2 1 : 2
0 : 2 0 : 2
0 : 1
```
Lower bound on comparison-based sorting

- Based on Decision Tree model.
- Any algorithm that sorts a list or array of size $n$ using comparisons can be modeled as a decision tree:
  - Each internal node is labeled $i : j$, representing a comparison between $L[i]$ and $L[j]$.
  - The left (respectively, right) of a node labeled $i : j$ describes for what happens if $L[i] < L[j]$ (respectively, $L[i] > L[j]$).
  - Each leaf node is a permutation of $0, \ldots, n - 1$.

Example: Decision tree for sorting 3 items
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.

2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.

3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.

4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.

The previous statement and Fact #4 imply an asymptotic bound:

Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\Omega(n \log n)$ comparisons in the worst case.
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size \( n \) can be modeled by a decision tree with at least \( n! \) leaf nodes.
2. Since the decision tree is a binary tree with \( n! \) leaves, the depth is at least \( \lceil \lg n! \rceil \).
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. \( \lg n! = \Omega(n \log n) \) (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size \( n \) can be modeled by a decision tree with at least \( n! \) leaf nodes.
2. Since the decision tree is a binary tree with \( n! \) leaves, the depth is at least \( \lceil \lg n! \rceil \).
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. \( \lg n! = \Omega(n \log n) \) (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size \( n \) must perform at least \( \lceil \lg n! \rceil \) comparisons in the worst case.
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size $n$ can be modeled by a decision tree with at least $n!$ leaf nodes.
2. Since the decision tree is a binary tree with $n!$ leaves, the depth is at least $\lceil \lg n! \rceil$.
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. $\lg n! = \Omega(n \log n)$ (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

\textit{Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\lceil \lg n! \rceil$ comparisons in the worst case.}

The previous statement and Fact #4 imply an asymptotic bound:
Lower bound on comparison-based sorting (continued)

1. Any comparison-based algorithm for sorting a list of size \( n \) can be modeled by a decision tree with at least \( n! \) leaf nodes.
2. Since the decision tree is a binary tree with \( n! \) leaves, the depth is at least \( \lceil \lg n! \rceil \).
3. The worst-case number of comparisons for the algorithm is the depth of the decision tree.
4. \( \lg n! = \Omega(n \log n) \) (proof on next slide)

Fact #2 and Fact #3 imply an exact bound:

Any comparison-based algorithm for sorting a list of size \( n \) must perform at least \( \lceil \lg n! \rceil \) comparisons in the worst case.

The previous statement and Fact #4 imply an asymptotic bound:

Any comparison-based algorithm for sorting a list of size \( n \) must perform at least \( \Omega(n \log n) \) comparisons in the worst case.
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$n! \geq \lceil n/2 \rceil \lceil n/2 \rceil \geq (n/2)^{n/2}$

Take log base 2 of both sides:

$\lg n! \geq (n/2)^{\lg n} = (n/2)(\lg n - 1) = \Omega(n \lg n)$
Proof that $\lg n! = \Omega(n \log n)$:

$n! = n \cdot (n-1) \cdot (n-3) \cdots 2 \cdot 1$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$n! \geq \lceil n/2 \rceil \cdot \lceil n/2 \rceil \geq (n/2)^n$

Take log of both sides:

$\lg n! \geq (n/2)^n \lg (n/2) = (n/2)^n (\lg n - 1)$

$= \Omega(n \lg n)$
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil \frac{n}{2} \rceil$. 

\[ \lg n! \geq \left( \frac{n}{2} \right) \left( \lg n - 1 \right) = \Omega(n \log n) \]
Lower bound on comparison-based sorting (continued)

Proof that \( \lg n! = \Omega(n \log n) \):

\[
n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1
\]

The first \( \lceil n/2 \rceil \) terms in the product are all \( \geq \lceil n/2 \rceil \).

This implies:
Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$

The first $\left\lceil n/2 \right\rceil$ terms in the product are all $\geq \left\lceil n/2 \right\rceil$.

This implies:

$$n! \geq \left\lceil n/2 \right\rceil^{\left\lceil n/2 \right\rceil}$$
Proof that \( \log n! = \Omega(n \log n) \):

\[
n! = n \cdot (n - 1) \cdot (n - 3) \cdot \ldots \cdot 2 \cdot 1
\]

The first \( \lceil n/2 \rceil \) terms in the product are all \( \geq \lceil n/2 \rceil \).

This implies:

\[
n! \geq \left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{2} \right\rceil \geq \left( \frac{n}{2} \right)^n
\]
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{2} \right\rceil \geq \left( \frac{n}{2} \right)^n$$

Take $\log_2$ of both sides:
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$$n! \geq \left\lfloor \frac{n}{2} \right\rfloor \left\lfloor \frac{n}{2} \right\rfloor \geq \left( \frac{n}{2} \right)^n$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left( \frac{n}{2} \right) \lg \left( \frac{n}{2} \right)$$
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$

The first $\lfloor n/2 \rfloor$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$$n! \geq \left\lfloor \frac{n}{2} \right\rfloor \left\lceil \frac{n}{2} \right\rceil \geq \left( \frac{n}{2} \right)^n$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left( \frac{n}{2} \right) \lg \left( \frac{n}{2} \right) = \left( \frac{n}{2} \right) (\lg n - 1)$$
Lower bound on comparison-based sorting (continued)

Proof that $\lg n! = \Omega(n \log n)$:

$$n! = n \cdot (n - 1) \cdot (n - 3) \cdots 2 \cdot 1$$

The first $\lceil n/2 \rceil$ terms in the product are all $\geq \lceil n/2 \rceil$.

This implies:

$$n! \geq \left\lceil \frac{n}{2} \right\rceil^{\lceil n/2 \rceil} \geq \left( \frac{n}{2} \right)^{\frac{n}{2}}$$

Take $\log_2$ of both sides:

$$\lg n! \geq \left( \frac{n}{2} \right) \lg \left( \frac{n}{2} \right) = \left( \frac{n}{2} \right) (\lg n - 1) = \Omega(n \log n)$$
Asymptotic optimality of MergeSort and HeapSort

We have just shown:
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\Omega(n \log n)$ comparisons in the worst case.
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

*Any comparison-based algorithm for sorting a list of size* \( n \) *must perform at least* \( \Omega(n \log n) \) *comparisons in the worst case.*

Earlier we showed:

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.
2. The lower bound is asymptotically tight (i.e., cannot be improved asymptotically).
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

*Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\Omega(n \log n)$ comparisons in the worst case.*

Earlier we showed:

*The worst-case running time of MergeSort and HeapSort on an input of size $n$ is $O(n \log n)$.*
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size \( n \) must perform at least \( \Omega(n \log n) \) comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size \( n \) is \( O(n \log n) \).

Conclusions:
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size $n$ is $O(n \log n)$.

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.
Asymptotic optimality of MergeSort and HeapSort

We have just shown:

Any comparison-based algorithm for sorting a list of size $n$ must perform at least $\Omega(n \log n)$ comparisons in the worst case.

Earlier we showed:

The worst-case running time of MergeSort and HeapSort on an input of size $n$ is $O(n \log n)$.

Conclusions:

1. MergeSort and HeapSort are asymptotically optimal.
2. The lower bound is asymptotically tight (i.e., cannot be improved asymptotically)