Exercise: In Step 1 on page 37, we used \( n/5 \) groups of length 5.

* Use groups of length 7.

This gives the recurrence

\[
T(n) = n + T\left(\frac{n}{7}\right) + T\left(\frac{5}{7}n\right),
\]

which solves to \( T(n) = O(n) \).

* Use groups of length 3.

This gives the recurrence

\[
T(n) = n + T\left(\frac{n}{3}\right) + T\left(\frac{2}{3}n\right),
\]

which solves to \( T(n) = \Theta(n \log n) \).
Heaps

Assume we have a data structure that

* stores numbers
* supports the operations
  * `insert(x)`: insert the number `x`
  * `extract_max`: return and delete the largest element

Then we can sort any sequence `a_1, a_2, ..., a_n` of numbers:

```
for i = 1, 2, ..., n : insert(a_i)
for i = 1, 2, ..., n : extract_max
```

Data structure that does this: heap
Array $A[1..n]$ is called a heap if for all $i \geq 1$,

$$A[i] \geq A[2i] \quad \text{(if } 2i \leq n\text{)}$$

and

$$A[i] \geq A[2i+1] \quad \text{(if } 2i+1 \leq n\text{)}$$

**Example:**

```
16 14 10 8 7 9 3 2 4 1
1  2  3  4  5  6  7  8  9 10
```

**Visualize this array as a binary tree:**

```
       16
      /   \
    14     10
   /     /   \
  8     7     9
 /     /     / \
2     4     9   3
```


root of the tree: $A[1]$

node with index $i$:

- parent of $i$ has index $\lfloor i/2 \rfloor$
- left child of $i$ has index $2i$
- right child of $i$ has index $2i + 1$

$A[1..n]$ is a heap if for all $i$ with $1 < i \leq n$:

$$A[\text{parent}(i)] \geq A[i]$$

What is the height of a heap $A[1..n]$?

$$= \text{height of the tree in the visualization}$$
Define $h = \text{height}$. 

level 0: 1 node

level 1: 2 nodes

level 2: 4 nodes

level $h-1$: $2^{h-1}$ nodes

level $h$: $\{ \geq 1 \text{ node} \cap \leq 2^h \text{ nodes} \}

1 + 2 + 2^2 + \ldots + 2^{h-1} + 1 \leq n \leq 1 + 2 + 2^2 + \ldots + 2^h

(2^h - 1) + 1 \leq n \leq 2^h - 1 < 2^{h+1}

2^h \leq n < 2^{h+1}

h \leq \log_2 n < h + 1

$$h = \lceil \log_2 n \rceil$$
Outline:

1. maximum (A): return largest element.  
   \( \mathcal{O}(1) \) time

2. increase_key (A, i, x):  
   assumes that \( x \geq A[i] \).  
   this operation increases \( A[i] \) to \( x \) and  
   restores the heap property.  
   \( \mathcal{O}(\log n) \) time

3. insert (A, x): insert element \( x \) and restore the  
   heap property; \( \mathcal{O}(\log n) \) time

4. Algorithm for building a heap: \( \mathcal{O}(n \log n) \) time

5. heapify: \( \mathcal{O}(\log n) \) time

6. Algorithm for building a heap: \( \mathcal{O}(n) \) time

7. heap-sort: \( \mathcal{O}(n \log n) \) time

8. extract_max (A): return and delete the  
   largest element, restore the heap property.  
   \( \mathcal{O}(\log n) \)
From now on:

array $A[1..N]$, integer $n$, $1 \leq n \leq N$

Sequence of $n$ numbers, stored in $A[1..n]$

$A[1..n]$ is a heap

# Heap and Garbage Diagram

- **Heap**
- **Garbage**

1. $\text{maximum}(A) : \text{return } A[1]$
   - time: $O(1)$

2. $\text{increase-key}(A, i, x)$:
   - this operation assumes that $x \geq A[i]$
\textbf{increase this key to 15:}\\
\textit{increase-key} (A, 9, 15),\\
\textit{explain algorithm for this example.}\\

\begin{verbatim}
A[i] = x;
while i > 1 and A[parent(i)] < A[i]:
    swap A[i] and A[parent(i)];
    i = parent(i)
\end{verbatim}

time = \( O(\text{height}) = O(\log n) \)
insert \((A, x)\):

this operation assumes that \(A[1..n]\) is a heap and \(1 \leq n < N\); thus, there is space for the new element \(x\).

explain algorithm insert \((A, 13)\) for example on page 48.

\[
\begin{align*}
n &= n + 1; \\
A[n] &= -\infty; \\
// A[1..n] is a heap \\
\text{increase_key} (A, n, x)
\end{align*}
\]

time = \(O(1) + \text{time for increase_key}\)

= \(O(1) + O(\log n)\)

= \(O(\log n)\)
How to build a heap: given array $A[1..N]$, permute the elements such that $A[1..N]$ is a heap.

\[
\begin{align*}
n &= 1 \\
\text{// } A[1..n] \text{ is a heap} \\
\text{while } n < N : \text{ insert } (A, A[n+1]) \\
\text{// observe that insert increments } n
\end{align*}
\]

\[
\begin{align*}
\text{time } &\sim \log 1 + \log 2 + \log 3 + \ldots + \log N \\
&\leq \log N + \log N + \log N + \ldots + \log N \\
&= N \log N \\
&= \Theta(N \log N)
\end{align*}
\]

By the way:

\[
\begin{align*}
\sum_{n=1}^{N} \log n &\geq \sum_{n=N/2}^{N} \log n \\
&\geq \sum_{n=N/2}^{N} \log \frac{N}{2} \\
&= \frac{N}{2} \log \frac{N}{2} = \frac{1}{2} N \log N - \frac{1}{2} N \\
&\geq \frac{1}{4} N \log N \quad (\text{if } N \geq 4) \\
&= \Omega(N \log N)
\end{align*}
\]
This algorithm for building a heap takes $\Theta(N \log N)$ time.

By the way:

\[
\sum_{n=1}^{N} \ln n = \sum_{n=2}^{N} \ln n
= \text{total area of the rectangles}
\geq \int_{1}^{N} \ln x \, dx = x \ln x - x \Big|_{x=1}^{x=N}
= N \ln N - N + 1
= \Omega(N \ln N)
= \Omega(N \log N)
\]
\[ \sum_{n=1}^{N} \ln n = \sum_{n=2}^{N} \ln n \]

= total area of the rectangles

\[ \int_{2}^{N+1} \ln x \, dx = x \ln x - x \bigg|_{x=2}^{x=N+1} \]

= \( (N+1) \ln(N+1) - (N+1) - 2 \ln 2 + 2 \)

= \( O(N \ln N) \)

= \( O(N \log N) \)
\[\text{heapify}(A, i)\]:

this operation assumes:

\[1 \leq i \leq n,\]

subtree rooted at \(\text{left}(i)\) is a heap,
subtree rooted at \(\text{right}(i)\) is a heap.

at termination: subtree rooted at \(i\) is a heap.

explain \text{heapify}(A, 2)

current position: 2
\[ l = \text{left}(i); \]
\[ r = \text{right}(i); \]
\[ \text{if } l \leq n \text{ and } A[l] > A[i] \]
\[ \text{then } \text{max} = l \]
\[ \text{else } \text{max} = i \]
\[ \text{endif}; \]
\[ \text{if } r \leq n \text{ and } A[r] > A[\text{max}] \]
\[ \text{then } \text{max} = r \]
\[ \text{endif}; \]
\[ \text{if } \text{max} \neq i \]
\[ \text{then swap } A[i] \text{ and } A[\text{max}]; \]
\[ \text{heapify}(A, \text{max}) \]
\[ \text{endif} \]
time = O(1) + time for recursive call at a node one level lower in the tree
\[ \therefore \text{time} = O(\text{height}) = O(\log n). \]

More precisely: time = O(\text{height of node } i)
6) build-heap (A):
   input: array A[1..N],
   output: heap A[1..N] containing the same elements.

   \[ n = N; \]
   for \( i = \lfloor N/2 \rfloor \) downto 1 : heapify (A, i)

A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7], N = 10

In general: all A[i], \( \lfloor N/2 \rfloor + 1 \leq i \leq N \), are leaves.

\( i \) starts at \( \lfloor N/10 \rfloor = 5 \).

why: because A[6], ..., A[10] are leaves and the subtree rooted at a leaf is a heap.
running time: \( \lfloor N/2 \rfloor \) calls to heapify, each call takes \( O(\log N) \) time.

\[ \text{total time} = O(N \log N). \]

Note: this is an upper bound.

Is there a better upper bound: yes.

Recall: heapify \((A, i)\) takes time \( O(\text{height of node } i) \)

To simplify: assume the bottom level of the tree is full: \( N = 1 + 2 + 2 + \ldots + 2^h = 2^{h+1} - 1 \), where \( h = \text{height of the tree} \).

<table>
<thead>
<tr>
<th>height</th>
<th>number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h )</td>
<td>1</td>
</tr>
<tr>
<td>( h-1 )</td>
<td>2</td>
</tr>
<tr>
<td>( h-2 )</td>
<td>2^2</td>
</tr>
<tr>
<td>( h-3 )</td>
<td>2^3</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>1</td>
<td>2^1</td>
</tr>
<tr>
<td>0</td>
<td>2^0</td>
</tr>
</tbody>
</table>
Running time of build-heap:

\[ 1 \cdot h + 2(h-1) + \frac{2^2}{2}(h-2) + \frac{2^3}{2}(h-3) + \ldots + \]

\[ \frac{h-2}{2} \cdot 2 + \frac{h-1}{2} \cdot 1 \]

\[ = \sum_{i=1}^{h} \frac{h-i}{2} \cdot i \]

\[ = 2 \sum_{i=1}^{h} i \cdot \left(\frac{1}{2}\right)^i \]

// \ h = \lfloor \log N \rfloor \leq \log N

// \ \therefore \ 2 \leq 2^{\log N} = N

\leq N \cdot \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i \]

\text{constant}

\[ = O(N) \]
By the way: for $-1 < x < 1$:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

differentiate:

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}$$

multiply by $x$:

$$\sum_{i=1}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

$x = \frac{1}{2}$:

$$\sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2.$$
heap-sort:

output: array $A[1..N]$ containing the same numbers in sorted order.

\[ \text{build-heap}(A); \]
\[ n = N; \]

while $n \geq 2$

do // $A[1..n]$ is a heap, $A[n+1..N]$ contains the $N-n$ largest elements in sorted order

\[ \text{Swap } A[1] \text{ and } A[n]; \]
\[ n = n-1; \]
\[ \text{heapify}(A, 1) \]

endwhile

Running time:

\[ \text{build-heap: } O(N) \]
\[ \text{while-loop: } O(\log N + \log(N-1) + \ldots + \log 3 + \log 2) = O(N \log N) \]
\[ \text{Total running time } = O(N \log N). \]
extract_max(A):

this operation assumes that A[1..n] is a heap and n ≥ 1.

max = A[1];
n = n-1;
heapify(A, i);
return max

time = O(1) + time for heapify(A, i)
    = O(1) + O(\log n)
    = O(\log n)
We have discussed max-heaps.

Symmetric: min-heap

\[ A[\text{parent}(i)] \leq A[i], \ 1 < i \leq n \]

1. \rightarrow \ \text{minimum (A)}
2. \rightarrow \ \text{decrease_key}(A, i, x) \ ; \ \text{assumes} \ x \leq A[i]
8. \rightarrow \ \text{extract-min (A)}