CSE 373: Floyd’s buildHeap algorithm; divide-and-conquer

Michael Lee
Wednesday, Feb 7, 2018
Warmup:

Insert the following letters into an empty binary min-heap. Draw the heap’s internal state in both tree and array form:

\[c, b, a, a, a, c\]
Warmup: 

Insert the following letters into an empty binary min-heap. Draw the heap’s internal state in both tree and array form:

c, b, a, a, a, c

In tree form:

```
    a
   /|
  a  b
 /|
c a c
```
Warmup:

Insert the following letters into an empty binary min-heap. Draw the heap’s internal state in both tree and array form:

\[ c, b, a, a, a, c \]
The array-based representation of binary heaps

Take a tree:

A
   /\    /
  B   C  D
 /\   /\  /\  
E F G H I J K L

How do we find parent?

parent \(i\) = \(\lfloor \frac{i-1}{2} \rfloor\)

The left child?

leftChild \(i\) = 2 \(i\) + 1

The right child?

leftChild \(i\) = 2 \(i\) + 2

And fill an array in the level-order of the tree:
The array-based representation of binary heaps

Take a tree:

And fill an array in the **level-order** of the tree:
The array-based representation of binary heaps

Take a tree:

How do we find parent?

\[ \text{parent}(i) = \left\lfloor \frac{i - 1}{2} \right\rfloor \]

The left child?

\[ \text{leftChild}(i) = 2i + 1 \]

The right child?

\[ \text{leftChild}(i) = 2i + 2 \]

And fill an array in the **level-order** of the tree:

\[
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
A & B & C & D & E & F & G & H & I & J & K & L &  &  & \\
\end{array}
\]
Finding the last node

If our tree is represented using an array, what’s the time needed to find the last node now?
Finding the last node

If our tree is represented using an array, what’s the time needed to find the last node now?

$\Theta(1)$: just use `this.array[this.size - 1]`. 
Finding the last node

If our tree is represented using an array, what’s the time needed to find the last node now?

$\Theta(1)$: just use `this.array[this.size - 1]`.

...assuming array has no 'gaps'. (Hey, it looks like the structure invariant was useful after all)
Re-analyzing insert

How does this change runtime of insert?

Runtime of insert:

\[
\text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}
\]

...which is:

\[
1 + 1 + \text{numSwaps} \times 1
\]

Observation:
when percolating, we usually need to percolate up a few times! So, \(\text{numSwaps} \approx 1\) in the average case, and \(\text{numSwaps} \approx \text{height} = \log(n)\) in the worst case!
Re-analyzing insert

How does this change runtime of insert?

Runtime of insert:

\[ \text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime} \]

...which is:

\[ 1 + 1 + \text{numSwaps} \times 1 \]
Re-analyzing insert

How does this change runtime of insert?

Runtime of insert:

\[
\text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}
\]

...which is:

\[
1 + 1 + \text{numSwaps} \times 1
\]

**Observation:** when percolating, we usually need to percolate up a few times! So, \( \text{numSwaps} \approx 1 \) in the average case, and \( \text{numSwaps} \approx \text{height} = \log(n) \) in the worst case!
Re-analyzing removeMin

How does this change runtime of removeMin?
Re-analyzing removeMin

How does this change runtime of removeMin?

Runtime of removeMin:

\[ \text{findLastNodeTime} + \text{removeRootTime} + \text{numSwaps} \times \text{swapTime} \]

...which is:

\[ 1 + 1 + \text{numSwaps} \times 1 \]
Re-analyzing removeMin

How does this change runtime of removeMin?

Runtime of removeMin:

\[ \text{findLastNodeTime} + \text{removeRootTime} + \text{numSwaps} \times \text{swapTime} \]

...which is:

\[ 1 + 1 + \text{numSwaps} \times 1 \]

**Observation:** unfortunately, in practice, usually must percolate all the way down. So \( \text{numSwaps} \approx \text{height} \approx \log(n) \) on average.
Deadlines:

▶ Partner selection: Fri, Feb 9
▶ Part 1: Fri, Feb 16
▶ Parts 2 and 3: Fri, Feb 23

Make sure to...

▶ Find a different partner for project 3
▶ ...or email me and petition to keep your current partner
Some stats about the midterm:

- Mean and median $\approx 80$ (out of 100)
- Standard deviation $\approx 13$
Common questions:

- I want to know how to do better next time
  Feel free to schedule an appointment with me.
Grades

Common questions:

- I want to know how to do better next time
  Feel free to schedule an appointment with me.
- **How will final grades be curved?**
  Not sure yet.
Common questions:

- I want to know how to do better next time
  Feel free to schedule an appointment with me.

- How will final grades be curved?
  Not sure yet.

- I want a midterm regrade.
  Wait a day, then email me.
Common questions:

- **I want to know how to do better next time**
  Feel free to schedule an appointment with me.

- **How will final grades be curved?**
  Not sure yet.

- **I want a midterm regrade.**
  Wait a day, then email me.

- **I want a regrade on a project or written homework**
  Fill out regrade request form on course website.
We discussed how to implement **insert**, where we insert one element into the heap.
An interesting extension

We discussed how to implement **insert**, where we insert one element into the heap.

What if we want to insert *n* different elements into the heap?
Idea 1: just call insert $n$ times – total runtime of $\Theta(n \log(n))$
Idea 1: just call insert $n$ times – total runtime of $\Theta(n \log(n))$

Can we do better?

Yes! Possible to do in $\Theta(n)$ time, using “Floyd’s buildHeap algorithm”.
Floyd’s buildHeap algorithm

The basic idea:

- Start with an array of all $n$ elements
- Start traversing *backwards* – e.g. from the bottom of the tree to the top
- Call `percolateDown(...) per each node`
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
10 9 8 7 6 5 4 3 2 1 0 3 2 1 0
```
Floyd’s buildheap algorithm: example

A visualization:

```
  10
 /   \
 9   8
 /     /
7   6   4
 /     /  \
3   2   5   1
|   |   |   |
10  9  8  7  6  5  4  3  2  1  0  3  2  1  0
```

10   9   8   7   6   5   4   3   2   1   0   3   2   1  0
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:
Floyd’s buildheap algorithm: example

A visualization:

```
0
1 0
2 1 6
3 7 9 10
2 3 5 8 4
1

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
10 0 0 2 1 2 1 3 7 9 6 3 5 8 4
```
Wait... isn’t this still $n \log(n)$?

We look at $n$ nodes, and we run percolateDown(...) on each node, which takes $\log(n)$ time... right?
Floyd’s buildheap algorithm

Wait... isn’t this still $n \log(n)$?

We look at $n$ nodes, and we run percolateDown(...) on each node, which takes $\log(n)$ time... right?

Yes – algorithm is $O(n \log(n))$, but with a more careful analysis, we can show it’s $O(n)$!
Question: How much work is percolateDown actually doing?
**Question:** How much work is percolateDown actually doing?

work \( (n) \approx n^2 \cdot 1 + n^4 \cdot 2 + n^8 \cdot 3 + \cdots \)
Analyzing Floyd’s buildheap algorithm

**Question:** How much work is percolateDown actually doing?

- (4 nodes) × (2 work)
- (8 nodes) × (1 work)

<table>
<thead>
<tr>
<th>n</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

work \( n \approx n^2 \cdot 1 + n^4 \cdot 2 + n^8 \cdot 3 + \cdots \)
Question: How much work is percolateDown actually doing?

(8 nodes) × (1 work)
(4 nodes) × (2 work)
(2 nodes) × (3 work)
Analyzing Floyd’s buildheap algorithm

**Question:** How much work is percolateDown actually doing?

![Diagram of a binary tree](image)

- (1 node) × (4 work)
- (2 nodes) × (3 work)
- (4 nodes) × (2 work)
- (8 nodes) × (1 work)

What's the pattern?

work \( \approx n^2 \cdot 1 + n^4 \cdot 2 + n^8 \cdot 3 + \cdots \)
Analyzing Floyd’s buildheap algorithm

**Question:** How much work is percolateDown actually doing?

What’s the pattern?

- (1 node) × (4 work)
- (2 nodes) × (3 work)
- (4 nodes) × (2 work)
- (8 nodes) × (1 work)

\[
\text{work} (n) \approx n^2 \cdot 1 + n^4 \cdot 2 + n^8 \cdot 3 + \cdots
\]
Question: How much work is percolateDown actually doing?

(1 node) \times (4 work)

(2 nodes) \times (3 work)

(4 nodes) \times (2 work)

(8 nodes) \times (1 work)

What’s the pattern?

\[
\text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots
\]
Analyzing Floyd’s buildheap algorithm

We had:

\[
\text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots
\]
Analyzing Floyd’s buildheap algorithm

We had:

\[ \text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots \]

Let’s rewrite bottom as powers of two, and factor out the \( n \):

\[ \text{work}(n) \approx n \left( \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \right) \]
Analyzing Floyd’s buildheap algorithm

We had:

$$\text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots$$

Let’s rewrite bottom as powers of two, and factor out the $n$:

$$\text{work}(n) \approx n \left( \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \right)$$

Can we write this in summation form? Yes.

$$\text{work}(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i}$$
Analyzing Floyd’s buildheap algorithm

We had:

\[ \text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots \]

Let’s rewrite bottom as powers of two, and factor out the \( n \):

\[ \text{work}(n) \approx n \left( \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \right) \]

Can we write this in summation form? Yes.

\[ \text{work}(n) \approx n \sum_{i=1}^{\infty} \frac{i}{2^i} \]

What is \( ? \) supposed to be?
Analyzing Floyd’s buildheap algorithm

We had:

\[
\text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots
\]

Let’s rewrite bottom as powers of two, and factor out the \(n\):

\[
\text{work}(n) \approx n \left( \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \right)
\]

Can we write this in summation form? Yes.

\[
\text{work}(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i}
\]

What is \(?\) supposed to be? It’s the height of the tree: so \(\log(n)\).
(Seems hard to analyze...)
Analyzing Floyd’s buildheap algorithm

We had:

\[ \text{work}(n) \approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots \]

Let’s rewrite bottom as powers of two, and factor out the \( n \):

\[ \text{work}(n) \approx n \left( \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots \right) \]

Can we write this in summation form? Yes.

\[ \text{work}(n) \approx n \sum_{i=1}^{\infty} \frac{i}{2^i} \]

What is \( ? \) supposed to be? It’s the height of the tree: so \( \log(n) \).
(Seems hard to analyze...) So let’s just make it infinity!

\[ \text{work}(n) \approx n \sum_{i=1}^{\infty} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} \]
Analyzing Floyd’s buildheap algorithm

Strategy: prove the summation is upper-bounded by something even when the summation goes on for infinity.

If we can do this, then our original summation must definitely be upper-bounded by the same thing.

\[
\text{work}(n) \approx n \sum_{i=1}^{\infty} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i}
\]
Analyzing Floyd’s buildheap algorithm

Strategy: prove the summation is upper-bounded by something even when the summation goes on for infinity.

If we can do this, then our original summation must definitely be upper-bounded by the same thing.

\[ \text{work}(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} \]

Using an identity (see page 4 of Weiss):

\[ \text{work}(n) \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} = n \cdot 2 \]
Analyzing Floyd’s buildheap algorithm

Strategy: prove the summation is upper-bounded by something even when the summation goes on for infinity.

If we can do this, then our original summation must definitely be upper-bounded by the same thing.

\[
\text{work}(n) \approx n \sum_{i=1}^{\infty} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i}
\]

Using an identity (see page 4 of Weiss):

\[
\text{work}(n) \leq n \sum_{i=1}^{\infty} \frac{i}{2^i} = n \cdot 2
\]

So buildHeap runs in $O(n)$ time!
Analyzing Floyd’s buildheap algorithm

Lessons learned:

- Most of the nodes near leaves (almost $\frac{1}{2}$ of nodes are leaves!)
  So design an algorithm that does less work closer to ‘bottom’
Analyzing Floyd’s buildheap algorithm

Lessons learned:

▶ Most of the nodes near leaves (almost $\frac{1}{2}$ of nodes are leaves!)
  So design an algorithm that does less work closer to ‘bottom’
▶ More careful analysis can reveal tighter bounds
Analyzing Floyd’s buildheap algorithm

Lessons learned:

▶ Most of the nodes near leaves (almost $\frac{1}{2}$ of nodes are leaves!) So design an algorithm that does less work closer to ‘bottom’
▶ More careful analysis can reveal tighter bounds
▶ Strategy: rather then trying to show $a \leq b$ directly, it can sometimes be simpler to show $a \leq t$ then $t \leq b$. (Similar to what we did when finding $c$ and $n_0$ questions when doing asymptotic analysis!)
Analyzing Floyd’s buildheap algorithm

What we’re skipping

- How do we merge two heaps together?
What we’re skipping

- How do we merge two heaps together?
- Other kinds of heaps (leftist heaps, skew heaps, binomial queues)
And now on to sorting...
Why study sorting?

Why not just use Collections.sort(...)?
Why study sorting?

Why not just use Collections.sort(...)?

- You should just use Collections.sort(...)

A vehicle for talking about a technique called "divide-and-conquer"

Different sorts have different purposes/tradeoffs.

(General purpose sorts work well most of the time, but you might need something more efficient in niche cases)

It's a "thing everybody knows".
Why study sorting?

Why not just use Collections.sort(...)?

- You should just use Collections.sort(...)
- A vehicle for talking about a technique called “divide-and-conquer”
Why not just use Collections.sort(...)?

- You should just use Collections.sort(...)
- A vehicle for talking about a technique called “divide-and-conquer”
- Different sorts have different purposes/tradeoffs. (General purpose sorts work well most of the time, but you might need something more efficient in niche cases)
Why study sorting?

Why not just use `Collections.sort(...)`?

- You should just use `Collections.sort(...)`
- A vehicle for talking about a technique called “divide-and-conquer”
- Different sorts have different purposes/tradeoffs. (General purpose sorts work well most of the time, but you might need something more efficient in niche cases)
- It’s a “thing everybody knows”.
Two different kinds of sorts:

**Comparison sorts**

Works by *comparing* two elements at a time.

Assumes elements in list form a *consistent, total ordering*:
Types of sorts

Two different kinds of sorts:

**Comparison sorts**

Works by **comparing** two elements at a time.

Assumes elements in list form a **consistent, total ordering**:

Formally: for every element $a$, $b$, and $c$ in the list, the following must be true.

- If $a \leq b$ and $b \leq a$ then $a = b$
- If $a \leq b$ and $b \leq c$ then $a \leq c$
- Either $a \leq b$ is true, or $b \leq a$ is true (or both)
Two different kinds of sorts:

**Comparison sorts**

Works by *comparing* two elements at a time.

Assumes elements in list form a *consistent, total ordering*:

Formally: for every element $a$, $b$, and $c$ in the list, the following must be true.

- If $a \leq b$ and $b \leq a$ then $a = b$
- If $a \leq b$ and $b \leq c$ then $a \leq c$
- Either $a \leq b$ is true, or $b \leq a$ is true (or both)

Less formally: the `compareTo(...)` method can’t be broken.
Two different kinds of sorts:

**Comparison sorts**

Works by **comparing** two elements at a time.

Assumes elements in list form a **consistent, total ordering**:

Formally: for every element \(a\), \(b\), and \(c\) in the list, the following must be true.

- If \(a \leq b\) and \(b \leq a\) then \(a = b\)
- If \(a \leq b\) and \(b \leq c\) then \(a \leq c\)
- Either \(a \leq b\) is true, or \(b \leq a\) is true (or both)

Less formally: the `compareTo(...)` method can’t be broken.

Fact: comparison sorts will run in \(\mathcal{O}(n \log(n))\) time at best.
Two different kinds of sorts:

**Niche sorts (aka “linear sorts”)**

Exploits certain properties about the items in the list to reach faster runtimes (typically, $O(n)$ time).
Two different kinds of sorts:

<table>
<thead>
<tr>
<th>Niche sorts (aka “linear sorts”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploits certain properties about the items in the list to reach faster runtimes (typically, $\mathcal{O}(n)$ time).</td>
</tr>
<tr>
<td>Faster, but less general-purpose.</td>
</tr>
</tbody>
</table>
Two different kinds of sorts:

**Niche sorts (aka “linear sorts”)**

Exploits certain properties about the items in the list to reach faster runtimes (typically, $O(n)$ time).

Faster, but less general-purpose.

We’ll focus on comparison sorts, will cover a few linear sorts if time.
## In-place sort

A sorting algorithm is **in-place** if it requires only $\mathcal{O}(1)$ extra space to sort the array.

- Usually modifies input array
- Can be useful: lets us minimize memory
## Stable sort

A sorting algorithm is **stable** if any **equal** items remain in the same relative order before and after the sort.

- **Observation**: We sometimes want to sort on some, but not all attribute of an item.
- Items that ’compare’ the same might not be exact duplicates.
- Sometimes useful to sort on one attribute first, then another.
Stable sort: Example

Input:

▶ Array: [(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
▶ Compare function: compare pairs by number only
Stable sort: Example

Input:

- Array: [(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
- Compare function: compare pairs by number only

Output; stable sort:

[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]

Output; unstable sort:

[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]

26
Stable sort: Example

Input:

- Array: [(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
- Compare function: compare pairs by number only

Output; stable sort:

[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]

Output; unstable sort:

[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]

26
Overview of sorting algorithms

There are many sorts...

Quicksort, Merge sort, In-place merge sort, Heap sort, Insertion sort, Intro sort, Selection sort, Timsort, Cubesort, Shell sort, Bubble sort, Binary tree sort, Cycle sort, Library sort, Patience sorting, Smoothsort, Strand sort, Tournament sort, Cocktail sort, Comb sort, Gnome sort, Block sort, Stackoverflow sort, Odd-even sort, Pigeonhole sort, Bucket sort, Counting sort, Radix sort, Sortsor, Burstsort, Flashtsort, Postman sort, Bead sort, Simple pancake sort, Spaghetti sort, Sorting network, Bitonic sort, Bogosort, Stooge sort, Insertion sort, Slow sort, Rainbow sort...
Overview of sorting algorithms

There are many sorts...

Quicksort, Merge sort, In-place merge sort, Heap sort, Insertion sort, Intro sort, Selection sort, Timsort, Cubesort, Shell sort, Bubble sort, Binary tree sort, Cycle sort, Library sort, Patience sorting, Smoothsort, Strand sort, Tournament sort, Cocktail sort, Comb sort, Gnome sort, Block sort, Stackoverflow sort, Odd-even sort, Pigeonhole sort, Bucket sort, Counting sort, Radix sort, Spreadsort, Burstsort, Flashsort, Postman sort, Bead sort, Simple pancake sort, Spaghetti sort, Sorting network, Bitonic sort, Bogosort, Stooge sort, Insertion sort, Slow sort, Rainbow sort...

...we’ll focus on a few
Insertion Sort

Current item

2 3 6 7 8 5 1 4 10 2 8


Already sorted

Unsorted
Insertion Sort

INSERT current item into sorted region

2 3 6 7 8 5 1 4 10 2 8


Already sorted

Unsorted
Insertion Sort

INSERT current item into sorted region

Already sorted

Unsorted
Insertion Sort

**INSERT** current item into sorted region

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

Already sorted

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

Unsorted

```
2  3  5  6  7  8  1  4  10  2  8
```
Insertion Sort

**Pseudocode**

```java
for (int i = 1; i < n; i++) {
    // Find index to insert into
    int newIndex = findPlace(i);

    // Insert and shift nodes over
    shift(newIndex, i);
}
```

- Worst case runtime?
- Best case runtime?
- Average runtime?
- Stable?
- In-place?
Selection Sort

Current item

Next smallest

2 3 6 7 8 15 18 14 11 9 10

Already sorted

Unsorted
Selection Sort

SELECT next min and swap with current

Already sorted

Unsorted
Selection Sort

SELECT next min and swap with current

2 3 6 7 8 15 18 14 11 9 10

Already sorted

3 6 7 8 9 18 14 11 15 10

Unsorted
Selection Sort

SELECT next min and swap with current

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>15</th>
<th>18</th>
<th>14</th>
<th>11</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

Already sorted

Unsorted
Selection Sort

SELECT next min and swap with current

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>18</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Already sorted

Unsorted

Pseudocode

```java
for (int i = 0; i < n; i++) {
    // Find next smallest
    int newIndex = findNextMin(i);

    // Swap current and next smallest
    swap(newIndex, i);
}
```

- Worst case runtime?
- Best case runtime?
- Average runtime?
- Stable?
- In-place?
Can we use heaps to help us sort?

**Pseudocode**

```
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = removeMin(input);
}
```
Can we use heaps to help us sort?

Idea: run buildHeap then call removeMin \( n \) times.
Can we use heaps to help us sort?

Idea: run `buildHeap` then call `removeMin` \( n \) times.

**Pseudocode**

```java
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = removeMin(input);
}
```

- Worst case runtime?
- Best case runtime?
- Average runtime?
- Stable?
- In-place?
Heap Sort: In-place version

Can we do this in-place?

Pseudocode

E[] input = buildHeap(...);
for (int i = 0; i < n; i++) {
    input[n - i - 1] = removeMin(input);
}
Can we do this in-place?

Idea: after calling removeMin, input array has one new space. Put the removed item there.
Can we do this in-place?

Idea: after calling `removeMin`, input array has one new space. Put the removed item there.
Can we do this in-place?

Idea: after calling `removeMin`, input array has one new space. Put the removed item there.

Pseudocode

```java
E[] input = buildHeap(...);
for (int i = 0; i < n; i++) {
    input[n - i - 1] = removeMin(input);
}
```
Heap Sort: In-place version

Complication: when using in-place version, final array is reversed!

![Heap Sort Diagram](image-url)
Complication: when using in-place version, final array is reversed!

Several possible fixes:

1. Run reverse afterwards (seems wasteful?)
Complication: when using in-place version, final array is reversed!

Several possible fixes:

1. Run reverse afterwards (seems wasteful?)
2. Use a max heap
Complication: when using in-place version, final array is reversed!

Several possible fixes:

1. Run reverse afterwards (seems wasteful?)
2. Use a max heap
3. Reverse your compare function to emulate a max heap
Technique: Divide-and-Conquer

Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

Example template:

```plaintext
algorithm(input) {
  if (small enough) {
    CONQUER, solve, and return input
  } else {
    DIVIDE input into multiple pieces
    RECURSE on each piece
    COMBINE and return results
  }
}
```
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
2. *Conquer* the individual pieces (as base cases)
Technique: Divide-and-Conquer

Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
2. *Conquer* the individual pieces (as base cases)
3. *Combine* the results together (recursively)
Divide-and-conquer is a useful technique for solving many kinds of problems. It consists of the following steps:

1. *Divide* your work up into smaller pieces (recursively)
2. *Conquer* the individual pieces (as base cases)
3. *Combine* the results together (recursively)

**Example template**

```plaintext
algorithm(input) {
    if (small enough) {
        CONQUER, solve, and return input
    } else {
        DIVIDE input into multiple pieces
        RECURSE on each piece
        COMBINE and return results
    }
}
```
Divide:

Unsorted
**Divide:** Split array roughly into half

- Unsorted
  - Unsorted
  - Unsorted
Merge sort: Core pieces

**Divide:** Split array roughly into half

```
Unsorted

Unsorted
```

**Conquer:**
**Merge sort: Core pieces**

**Divide:** Split array roughly into half

```
Unsorted
```

```
Unsorted       Unsorted
```

**Conquer:** Return array when length $\leq 1$
Merge sort: Core pieces

**Divide:** Split array roughly into half

- Unsorted
- Unsorted
- Unsorted

**Conquer:** Return array when length $\leq 1$

**Combine:**

- Sorted
- Sorted
**Merge sort: Core pieces**

**Divide:** Split array roughly into half

- Unsorted
- Unsorted

**Conquer:** Return array when length \( \leq 1 \)

**Combine:** Combine two sorted arrays using merge

- Sorted
- Sorted

- Sorted
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged.

**Pseudocode**

```plaintext
sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}
```
## Merge sort: Example

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>
Merge sort: Example
Merge sort: Example
Merge sort: Example
## Merge sort: Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
Merge sort: Example

```
  5   10
   a[0]  a[1]
```

```
  2   7
```

```
  3   6
```

```
  2   11
```
Merge sort: Example
Merge sort: Example
**Pseudocode**

```java
sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}
```

Best case runtime?  Worst case runtime?
Best and worst case

We always subdivide the array in half on each recursive call, and merge takes $\mathcal{O}(n)$ time to run. So, the best and worst case runtime is the same:

$$ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} $$
Best and worst case

We always subdivide the array in half on each recursive call, and merge takes $O(n)$ time to run. So, the best and worst case runtime is the same:

$$T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}$$

But how do we solve this recurrence?
Analyzing recurrences, part 2

We have: \[ T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise} 
\end{cases} \]

Problem: Unfolding technique is a major pain to do.

Next time:

- Tree method: requires a little work, but more general purpose
- Master method: very easy, but not as general purpose
Analyzing recurrences, part 2

We have: 

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

**Problem:** Unfolding technique is a major pain to do
Analyzing recurrences, part 2

We have: \( T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \)

**Problem:** Unfolding technique is a major pain to do

**Next time:** Two new techniques:

- Tree method: requires a little work, but more general purpose
- Master method: very easy, but not as general purpose
Quick sort: Divide step

6 10 7 2 3 5 2 11

Quick sort: Divide step

6 10 7 2 3 5 2 11

Pivot
Quick sort: Divide step

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>


Numbers \( \leq \) pivot

Pivot
Quick sort: Divide step

Pivot

Numbers ≤ pivot

Numbers > pivot
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- **P**
- Unsorted
- \( \leq P \)
- \( > P \)
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- \( P \)
- Unsorted
- \( \leq P \)
- \( > P \)

**Conquer:**
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

![Diagram showing partitioning into groups based on pivot P.](image)

**Conquer:** Return array when length $\leq 1$
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- \( P \)
- \( \leq P \)
- \( > P \)

**Conquer:** Return array when length \( \leq 1 \)

**Combine:**

- \( \leq P \)
- \( P \)
- \( > P \)
Quick sort: Core pieces

**Divide:** Pick a pivot, partition into groups

- Unsorted
  - $P \leq P$
  - $P > P$

**Conquer:** Return array when length $\leq 1$

**Combine:** Combine sorted portions and the pivot
Core idea: Pick some item from the array and call it the pivot. Put all items smaller in the pivot into one group and all items larger in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

Pseudocode

sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
Quick sort: Example

<table>
<thead>
<tr>
<th>20</th>
<th>50</th>
<th>70</th>
<th>10</th>
<th>60</th>
<th>40</th>
<th>30</th>
</tr>
</thead>
</table>
Quick sort: Example

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th>50</th>
<th>70</th>
<th>10</th>
<th>60</th>
<th>40</th>
<th>30</th>
</tr>
</thead>
</table>
Quick sort: Example

20 50 70 10 60 40 30


10

50 70 60 40 30

10

50 70 60 40 30
Quick sort: Example
Quick sort: Example
Quick sort: Example

```
20  50  70  10  60  40  30

10
  a[0]

50  70  60  40  30

40  30
  a[0]  a[1]

70  60
  a[0]  a[1]
```
Quick sort: Example

20 50 70 10 60 40 30

10

40 30

30

70 60

60

50

10 50 70 10 60 40 30


10

40 30

30

70 60

60
Quick sort: Example
Quick sort: Example

20

10

30 40

30

50

70 60 40 30

70 60 40 30

60 70

60

30

10


a[0]  a[1]
Quick sort: Example

```
20
  10

30  40  50  60  70

30  40

60  70
```

```
50  70  10  60  40  30


a[0]  a[1]

a[0]
```
Quick sort: Example

10  20  30  40  50  60  70


10
a[0]

30  40  50  60  70


30  40
a[0]  a[1]

60  70
a[0]  a[1]

30
a[0]

60
a[0]
Quick sort: Analysis

Pseudocode

sort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}

Best case runtime?  Worst case runtime?
Best case analysis

In the best case, we always pick the median element.

\[
T(n) = \begin{cases} 
2T(n/2) + n & \text{if } n > 1 \\
1 & \text{otherwise}
\end{cases}
\]

So, the best-case runtime is \( \Theta(n \lg(n)) \)
**Quick sort: Analysis**

### Best case analysis
In the best case, we always pick the median element, the best-case runtime is $\Theta(n \lg(n))$.

### Worst case analysis
In the worst case, we always end up picking the minimum or maximum element.

$$T(n) = \begin{cases} 
T(n-1) + n & \text{if } n > 1 \\
1 & \text{otherwise}
\end{cases}$$

So, the worst-case runtime is $\Theta(n^2)$. 
## Quick sort: Analysis

### Best case analysis

In the *best* case, we always pick the *median* element, so the best-case runtime is $\Theta (n \log(n))$.

### Worst case analysis

In the *worst* case, we always end up picking the *minimum* or *maximum* element, so, the worst-case runtime is $\Theta (n^2)$.

### Average case runtime

Usually, we’ll pick a *random* element, which makes the runtime $\Theta (n \log(n))$. 