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

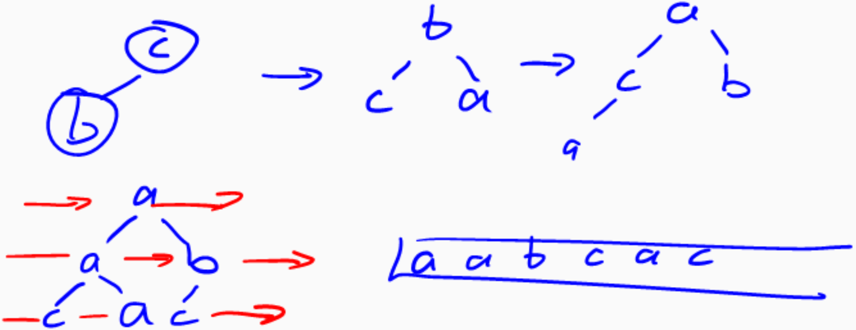
Michael Lee

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

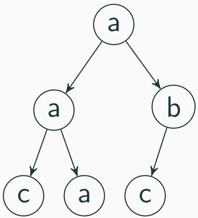


Warmup:

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

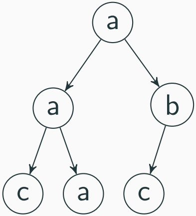


Warmup:

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

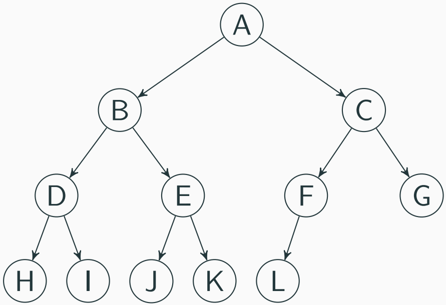


In array form

0	1	2	3	4	5	6	7
a	a	b	c	a	c		

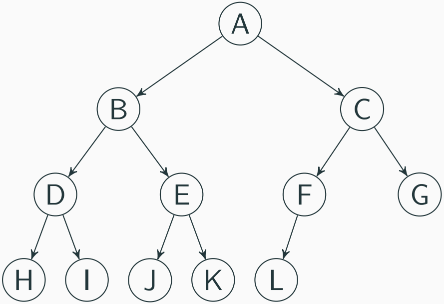
The array-based representation of binary heaps

Take a tree:



The array-based representation of binary heaps

Take a tree:

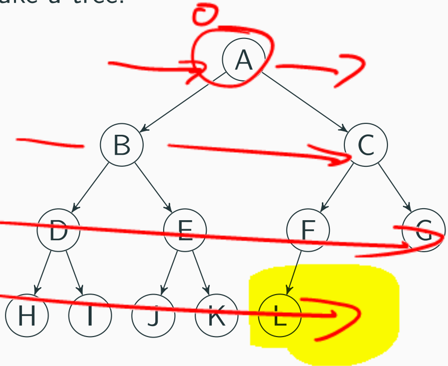


And fill an array in the **level-order** of the tree:

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			

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:

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			

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`?

Re-analyzing insert

How does this change runtime of `insert`?

Runtime of insert:

$$\texttt{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}$$

...which is:

$$1 + 1 + \text{numSwaps} \times 1$$

Re-analyzing insert

How does this change runtime of `insert`?

Runtime of insert:

$$\texttt{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}$$

...which is:

$$1 + 1 + \text{numSwaps} \times 1$$

Observation: when percolating, we usually need to percolate up a few times! So, numSwaps ≈ 1 in the average case, and numSwaps $\approx$ 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 numSwaps $\approx$ height $\approx \log(n)$ on average.

Project 2

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

Grades

Some stats about the midterm:

- Mean and median ≈ 80 (out of 100)
- Standard deviation ≈ 13

Grades

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.

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

An interesting extension

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?

An interesting extension

Idea 1: just call **insert** n times – total runtime of $\Theta\left(n\log(n)\right)$

An interesting extension

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".

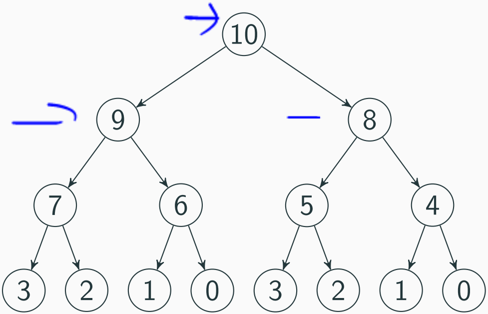
Floyd's buildHeap algorithm

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

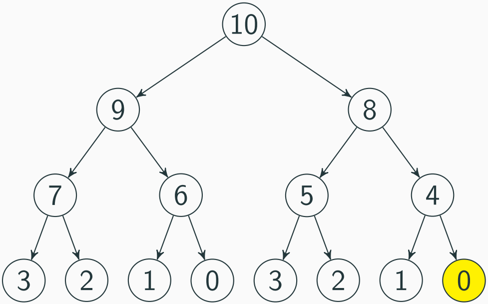
A visualization:



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

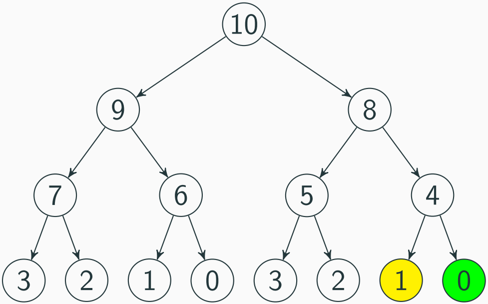
A visualization:



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

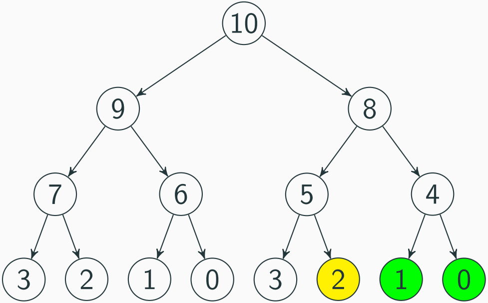
A visualization:



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

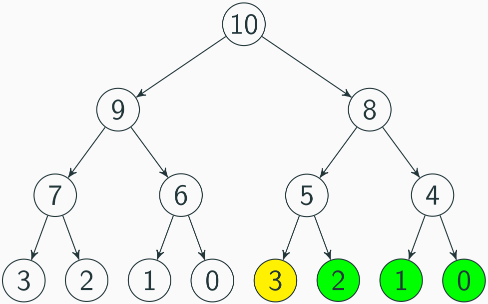
A visualization:



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

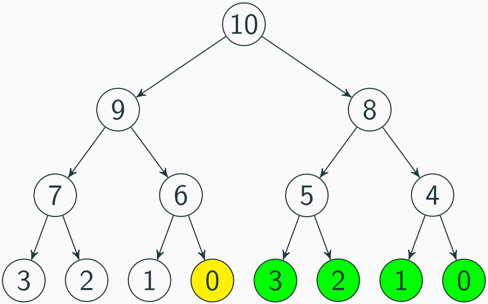
A visualization:



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

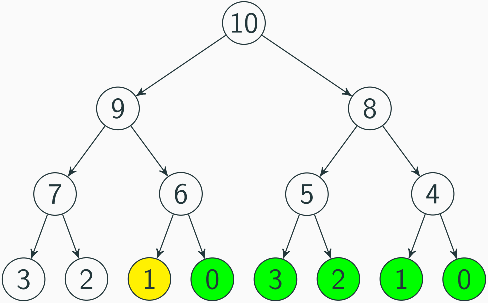
A visualization:



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

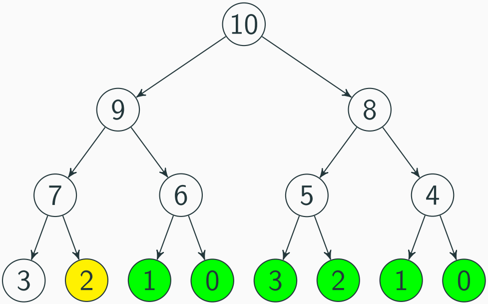
A visualization:



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

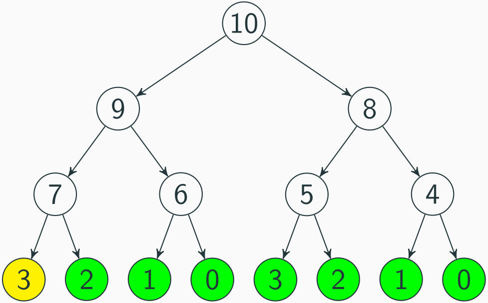
A visualization:



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

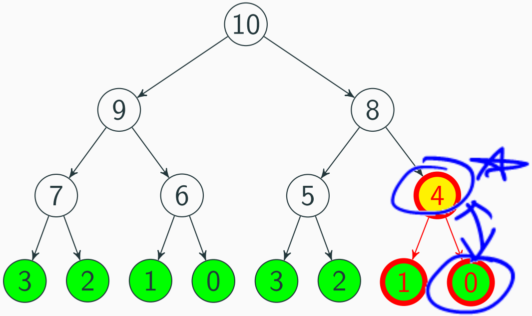
A visualization:



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

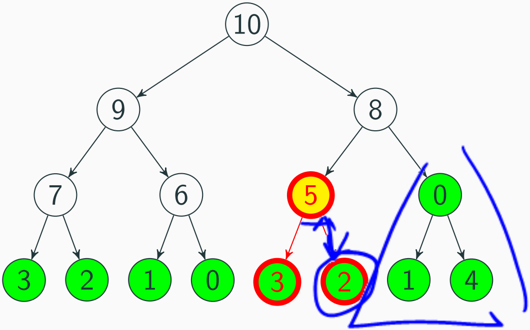
A visualization:



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

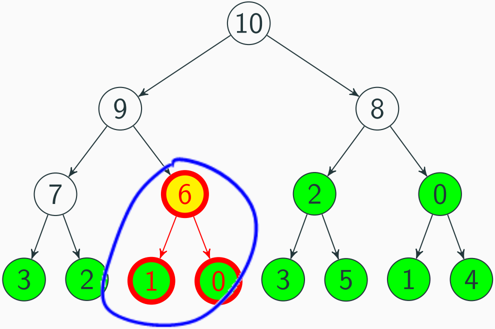
A visualization:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	8	7	6	5	0	3	2	1	6	3	2	1	4	

Floyd's buildheap algorithm: example

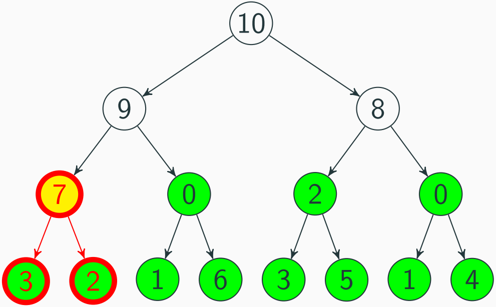
A visualization:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	8	7	6	2	0	3	2	1	6	3	5	1	4	

Floyd's buildheap algorithm: example

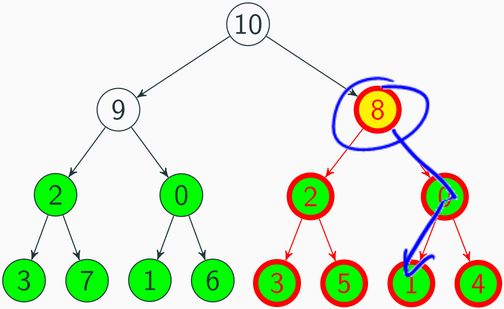
A visualization:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	8	7	0	2	0	3	2	1	6	3	5	1	4	

Floyd's buildheap algorithm: example

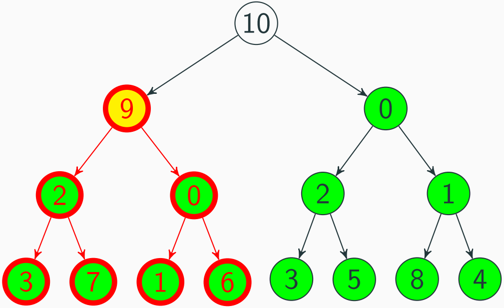
A visualization:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	8	2	0	2	0	3	7	1	6	3	5	1	4	

Floyd's buildheap algorithm: example

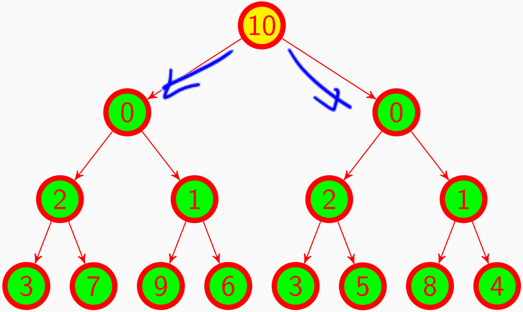
A visualization:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	9	0	2	0	2	1	3	7	1	6	3	5	8	4	

Floyd's buildheap algorithm: example

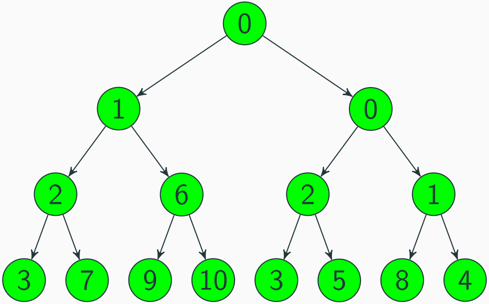
A visualization:



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	

Floyd's buildheap algorithm: example

A visualization:



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	

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?

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?

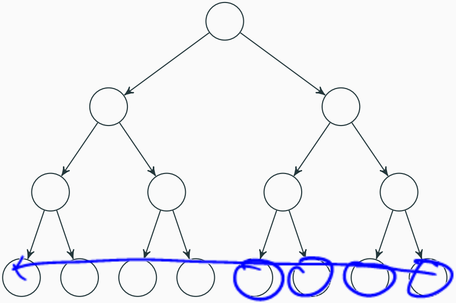
Yes – algorithm is $\mathcal{O}(n \log(n))$, but with a more careful analysis, we can show it's $\mathcal{O}(n)$!

Analyzing Floyd's buildheap algorithm

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

Analyzing Floyd's buildheap algorithm

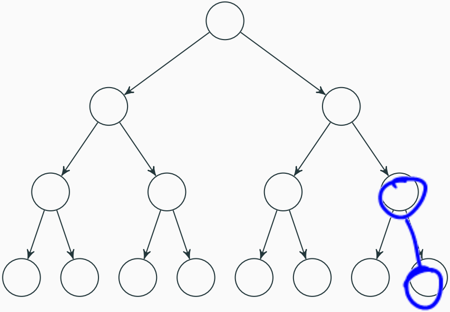
Question: How much work is `percolateDown` actually doing?



(8 nodes) $\times$ (1 work)

Analyzing Floyd's buildheap algorithm

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

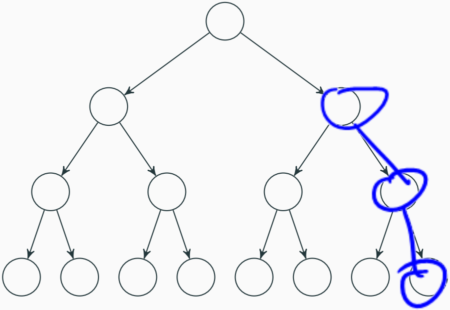


(4 nodes) $\times$ (2 work)

(8 nodes) $\times$ (1 work)

Analyzing Floyd's buildheap algorithm

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



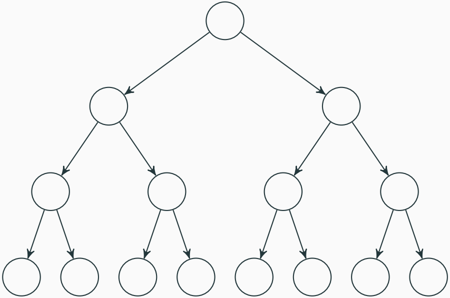
(2 nodes) × (3 work)

(4 nodes) × (2 work)

(8 nodes) × (1 work)

Analyzing Floyd's buildheap algorithm

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



(1 node) $\times$ (4 work)

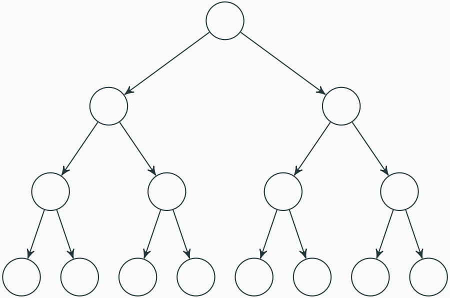
(2 nodes) $\times$ (3 work)

(4 nodes) $\times$ (2 work)

(8 nodes) $\times$ (1 work)

Analyzing Floyd's buildheap algorithm

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



(1 node) $\times$ (4 work)

(2 nodes) $\times$ (3 work)

(4 nodes) $\times$ (2 work)

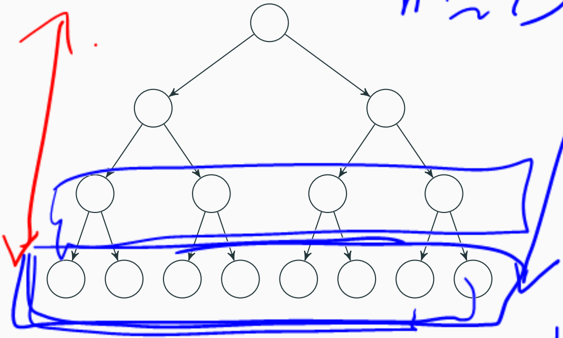
(8 nodes) $\times$ (1 work)

What's the pattern?

Analyzing Floyd's buildheap algorithm

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

$n \approx 15$



(1 node) $\times$ (4 work)

(2 nodes) $\times$ (3 work)

(4 nodes) $\times$ (2 work)

(8 nodes) $\times$ (1 work)

What's the pattern?

$\frac{15}{2} \approx 8 \quad \frac{15}{4} \approx$

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

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:

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

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.

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

$$\frac{i}{2^i}$$

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

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.

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

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.

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

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!

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

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.

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

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.

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

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 $\mathcal{O}(n)$ 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?

Analyzing Floyd's buildheap algorithm

What we're skipping

- How do we merge two heaps together?
- Other kinds of heaps (leftist heaps, skew heaps, binomial queues)

On to sorting

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(...)`

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 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)

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".

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**:

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)

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)

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

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)

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

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

Types of sorts

Two different kinds of sorts:

Niche sorts (aka "linear sorts")

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

Types of sorts

Two different kinds of sorts:

Niche sorts (aka "linear sorts")

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

Faster, but less general-purpose.

Types of sorts

Two different kinds of sorts:

Niche sorts (aka "linear sorts")

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

Faster, but less general-purpose.

We'll focus on comparison sorts, will cover a few linear sorts if time.

More definitions

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

More definitions

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:

1 2

- 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")]`

Stable sort: Example

Input:

1 2

- 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")]`

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...

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

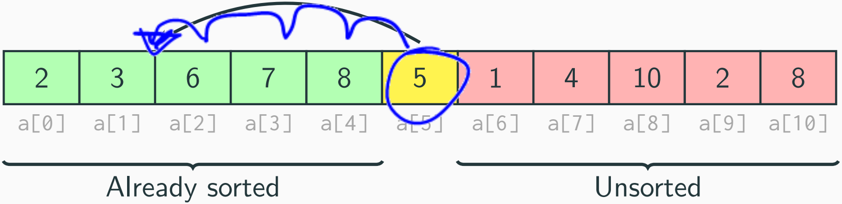
2	3	6	7	8	5	1	4	10	2	8
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Already sorted

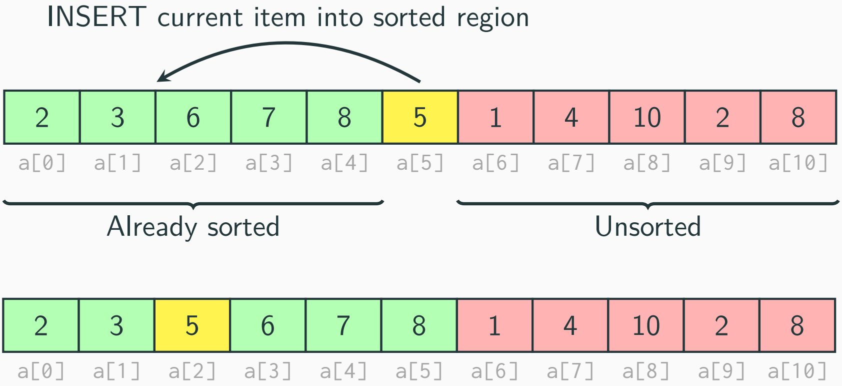
Unsorted

Insertion Sort

INSERT current item into sorted region



Insertion Sort



Insertion Sort

INSERT current item into sorted region

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	6	7	8	5	1	4	10	2	8

Already sorted   Unsorted

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	5	6	7	8	1	4	10	2	8

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	5	6	7	8	1	4	10	2	8

Insertion Sort

INSERT current item into sorted region

1, 2, 3, 4, 5, 6

2	3	6	7	8	5	1	4	10	2	8
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Already sorted (a[0]–a[4]) Unsorted (a[6]–a[10])

Pseudocode

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

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

BS

$O(n^2)$ ▶ Worst case runtime?

$O(n)$ ▶ Best case runtime?

depends ▶ Average runtime?

▶ Stable? ✓

▶ In-place? ✓

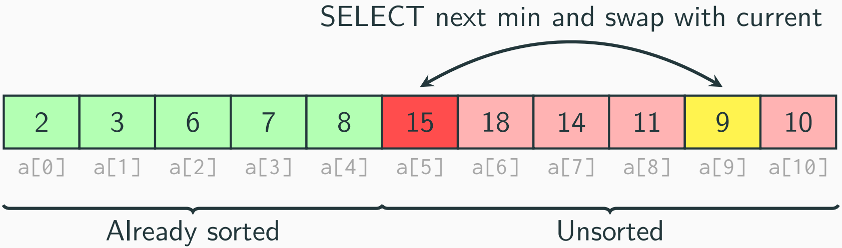
$n(\log(n)+n) \in O(n^2)$

Selection Sort

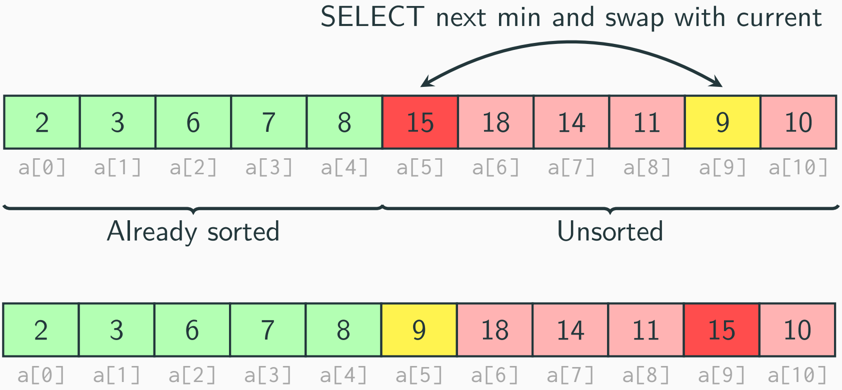
					Current item				Next smallest	
2	3	6	7	8	15	18	14	11	9	10
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Already sorted (a[0]–a[4]) Unsorted (a[5]–a[10])

Selection Sort



Selection Sort



Selection Sort

SELECT next min and swap with current

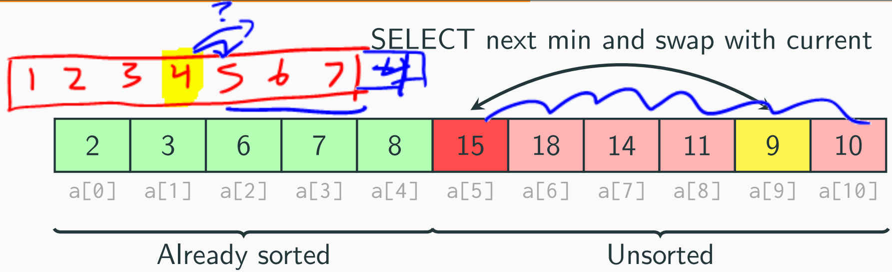
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	6	7	8	15	18	14	11	9	10

$\underbrace{\text{a[0] … a[4]}}_{\text{Already sorted}}$ $\underbrace{\text{a[5] … a[10]}}_{\text{Unsorted}}$

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	6	7	8	9	18	14	11	15	10

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
2	3	6	7	8	9	18	14	11	15	10

Selection Sort



Pseudocode

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

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

- $O(n^2)$ Worst case runtime?
- $O(n^2)$ Best case runtime?
- Average runtime?
- Stable?
- In-place?

Heap sort

Can we use heaps to help us sort?

Can we use heaps to help us sort?

Idea: run `buildHeap` then call `removeMin` n times.

Heap sort

Can we use heaps to help us sort?

Idea: run `buildHeap` then call `removeMin` *n* times.

Pseudocode

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

↳ log(n)

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

Heap Sort: In-place version

Can we do this in-place?

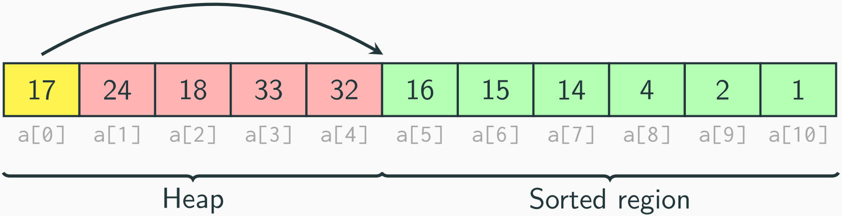
Heap Sort: In-place version

Can we do this in-place?

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

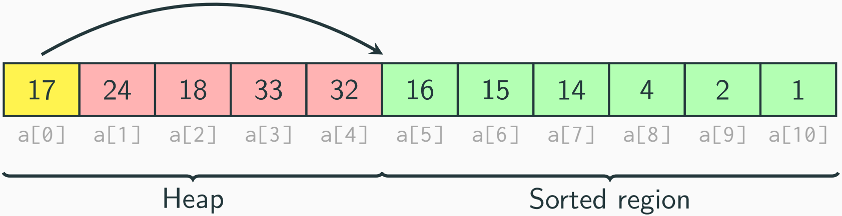
Can we do this in-place?

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.

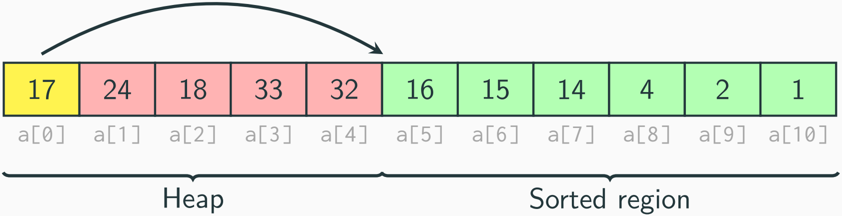


Pseudocode

```
E[] input = buildHeap(...);
for (int i = 0; i < n; i++) {
    input[n - i - 1] = removeMin(input);
}
```

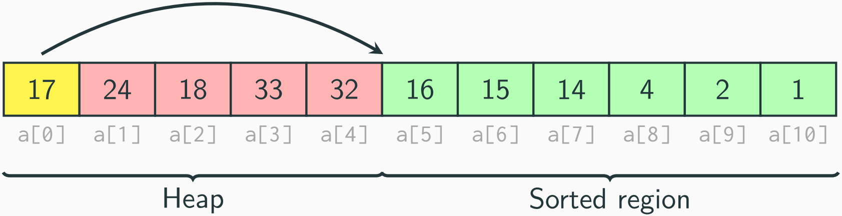
Heap Sort: In-place version

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



Heap Sort: In-place version

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

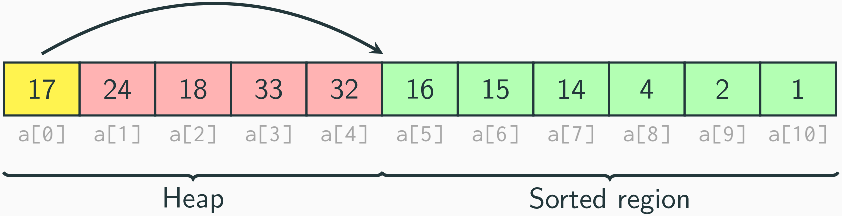


Several possible fixes:

1. Run `reverse` afterwards (seems wasteful?)

Heap Sort: In-place version

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

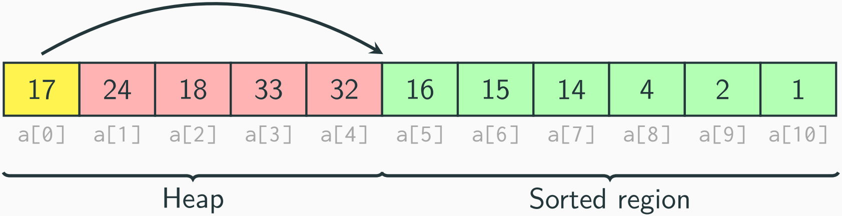


Several possible fixes:

1. Run `reverse` afterwards (seems wasteful?)
2. Use a max heap

Heap Sort: In-place version

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:

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)

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)

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)

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

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

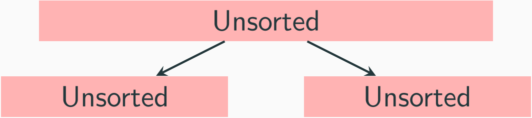
Merge sort: Core pieces

Divide:

Unsorted

Merge sort: Core pieces

Divide: Split array roughly into half



Merge sort: Core pieces

Divide: Split array roughly into half

Unsorted

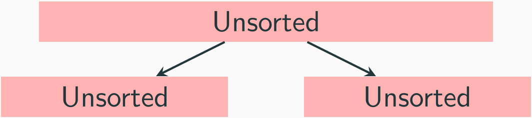
Unsorted | Unsorted

Conquer:

[3]

Merge sort: Core pieces

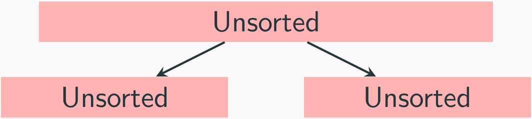
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1

Merge sort: Core pieces

Divide: Split array roughly into half

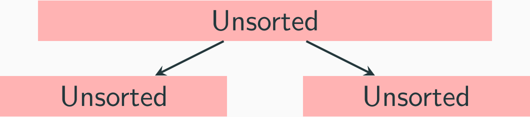


Conquer: Return array when length ≤ 1

Combine:

Merge sort: Core pieces

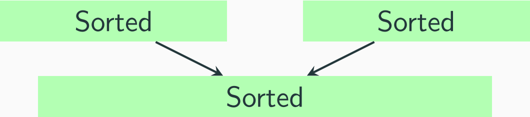
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1

$O(n)$

Combine: Combine two sorted arrays using `merge`



Merge sort: Summary

Core idea: split array in half, sort each half, merge back together.

If the array has size 0 or 1, just return it unchanged.

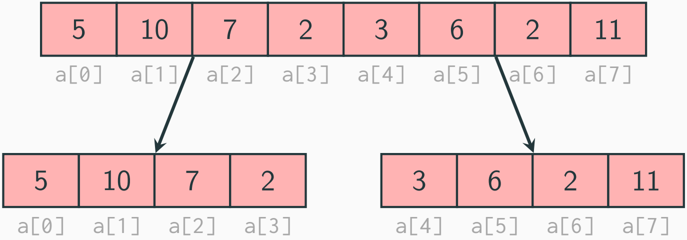
Pseudocode

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

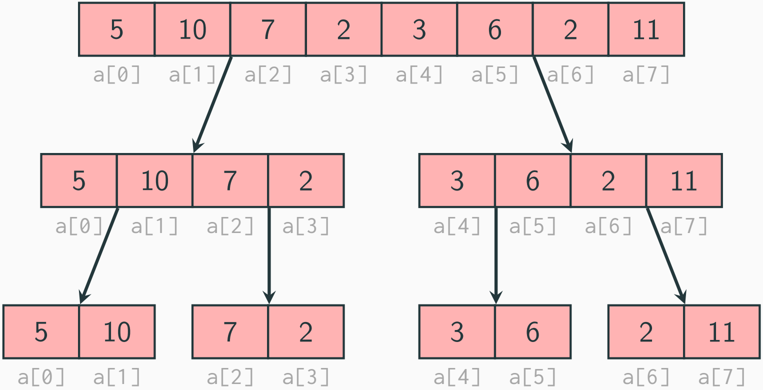
Merge sort: Example

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
5	10	7	2	3	6	2	11

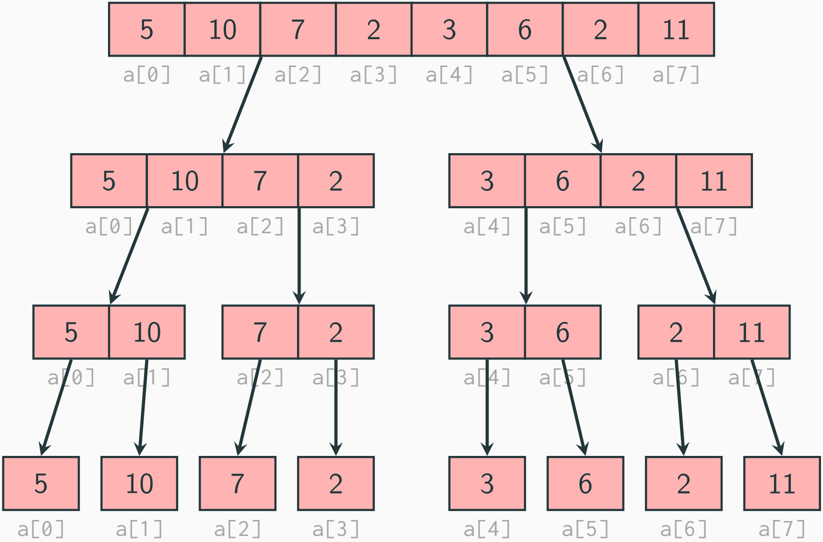
Merge sort: Example



Merge sort: Example



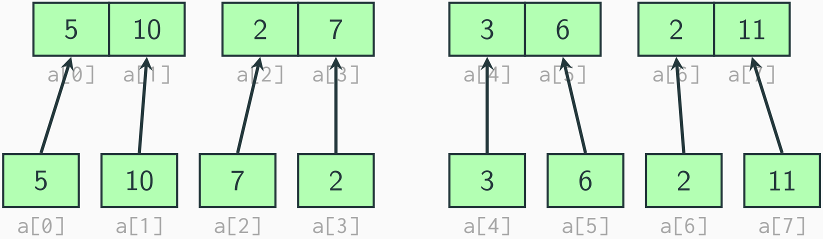
Merge sort: Example



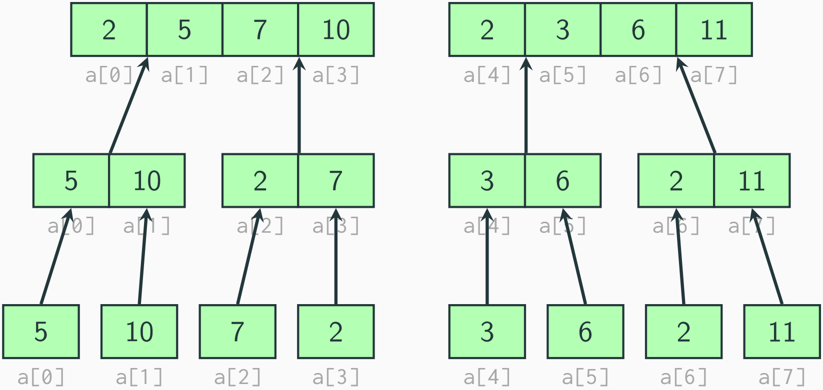
Merge sort: Example

5	10	7	2	3	6	2	11
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

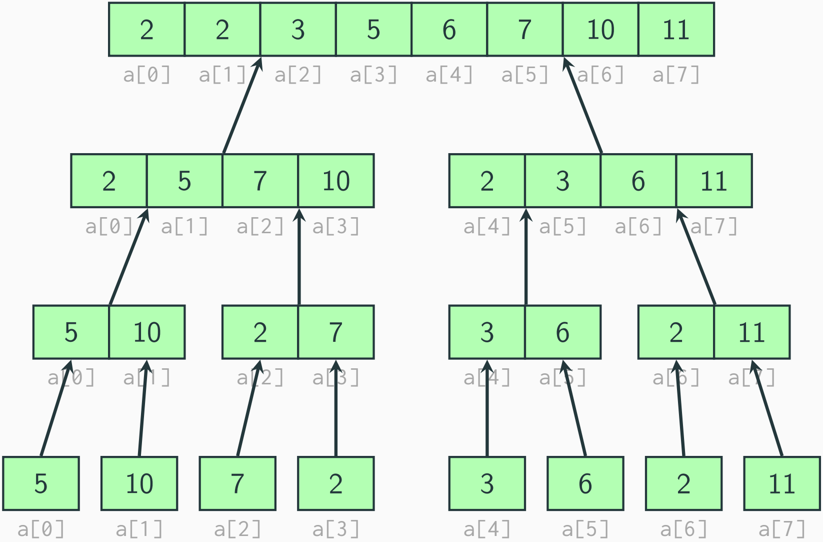
Merge sort: Example



Merge sort: Example



Merge sort: Example



Merge sort: Analysis

Pseudocode

```
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?

Merge sort: Analysis

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:

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

Merge sort: Analysis

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:

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

But how do we solve this recurrence?

Analyzing recurrences, part 2

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

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