CSE 332: Comparison Sorts

Michael Lee

Wednesday, February 1, 2016

Why bother?

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

Why study sorting?

Tradeoffs

- There is no 'best' sorting algorithm.
- DIfferent sorts have different purposes/tradeoffs.

Tech interviews

- Nobody asks about sorting on interviews...
- ...but if it comes up, you look bad if you can't talk about it.

Definition: Comparison Sort

SORT is a computational problem with the following preconditions and postconditions:

Preconditions (input)

An array A of length n where each element is of type E

A consistent, total ordering on all elements of type E:

```
compare(a, b)
```

Postconditions (outputs)

For all $0 \leq i < j < n$, `A[k]` $\leq$ `A[j]`

Every item in the original array must be in the sorted array

A comparison sort algorithm solves SORT.

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 one

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")]
```

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

Overview of sorting algorithms

Simple sorts: $\mathcal{O}(n^2)$

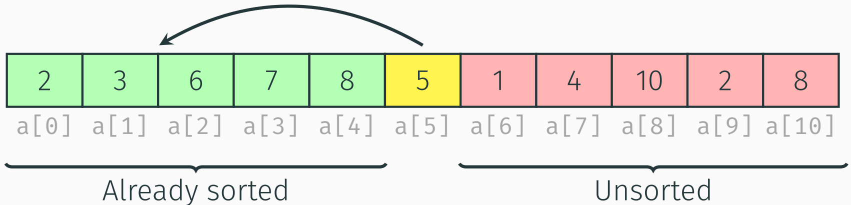
- Insertion sort
- Selection sort

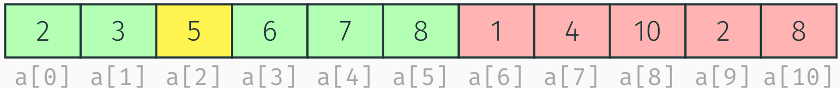
Fancy sorts: $\mathcal{O}(n \lg(n))$

- Heap sort
- Merge sort
- Quick sort

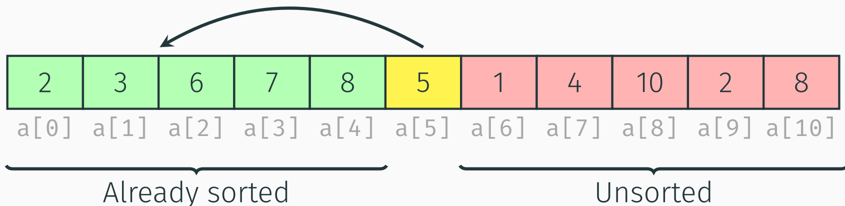
Insertion Sort

Insertion Sort





Insertion Sort



Pseudocode

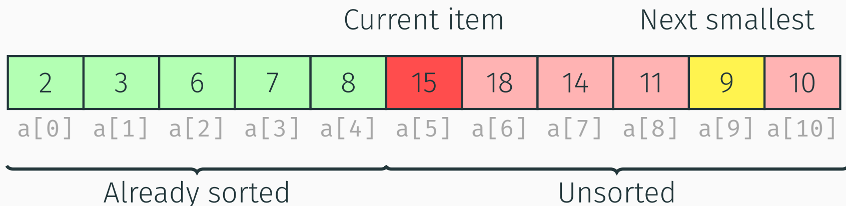
```
for (int i = 0; i < n; i++) {
    // Find index to insert into
    int newIndex = findPlace(i);

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

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

Insertion Sort: Analysis

- In the **worst case**, both `findPlace` and `shift` will need to do about i steps. Our runtime model is then $\sum^{n} i$ which will be in $\mathcal{O}(n^2)$.
- The **best case** is when the list is already sorted. Then, `findPlace` only needs to look at the last item and `shift` does nothing. So, the runtime is $\mathcal{O}(n)$.
- We don't really know how to analyze the **average case**, but it ends up being $\mathcal{O}(n^2)$.
- Insertion sort can be implemented to be either stable or unstable, depending on how we insert duplicate entries. It's usually implemented to be **stable**.
- Insertion sort **is in-place**.

Selection Sort



Selection Sort

SELECT next min and swap with current

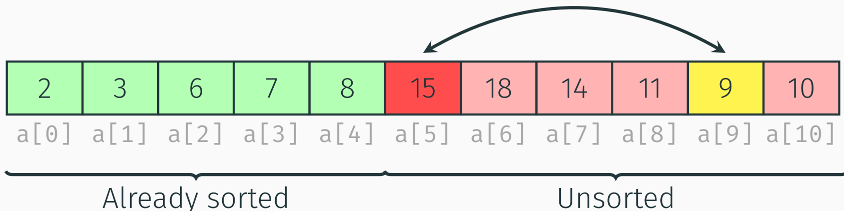
2	3	6	7	8	15	18	14	11	9	10
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Already sorted — Unsorted

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
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[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);
}
```

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

Selection Sort: Analysis

- In the **worst case**, `findNextMin` will need to do about $n - i$ steps per iteration. Our runtime model is then $\sum^{n} n - i$ which will be in $\mathcal{O}(n^2)$.
- Regardless of what the list looks like, we know nothing about the unsorted region so `findNextMin` must still scan the next $n - i$ items. So, the **best case** is the same as the worst case: $\mathcal{O}(n^2)$.
- The **average case** is therefore $\mathcal{O}(n^2)$.
- Same thing as insertion sort – we can choose if we want our implementation to be stable or not; so we might as well make it **stable**.
- Selection sort **is in-place**.

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);
}
```

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

Heap Sort: Analysis

- We know `buildHeap` is $\mathcal{O}(n)$ and `removeMin` is $\mathcal{O}(\lg(n))$ so the total runtime in the **worst case** is $\mathcal{O}(n \lg(n))$.
- The **best case** is the same as the worst case: $\mathcal{O}(n \lg(n))$.
- The **average case** is therefore $\mathcal{O}(n \lg(n))$.
- Heap sort is **not stable** – the heap methods don't respect the relative ordering of items that are considered the 'same' by the `compare` function.
- This version of heap sort is **not in-place**.

Can we do this in-place?

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

17	24	18	33	32	16	15	14	4	2	1
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Heap: a[0]–a[4]; Sorted region: a[5]–a[10]

Pseudocode

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

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

17	24	18	33	32	16	15	14	4	2	1
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Heap: a[0]–a[4]; Sorted region: a[5]–a[10]

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:

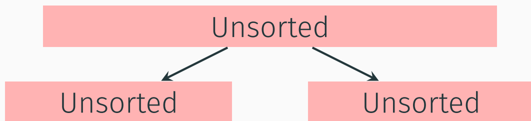
1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

Example template

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

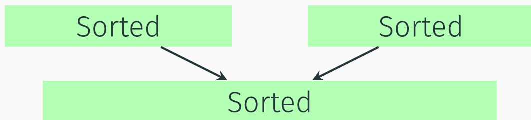
Merge sort: Core pieces

Divide: Split array roughly into half



Conquer: Return array when length ≤ 1

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.

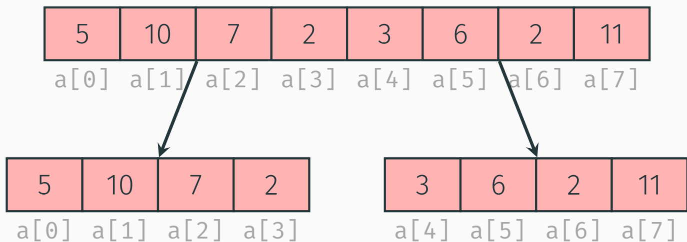
Pseudocode

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

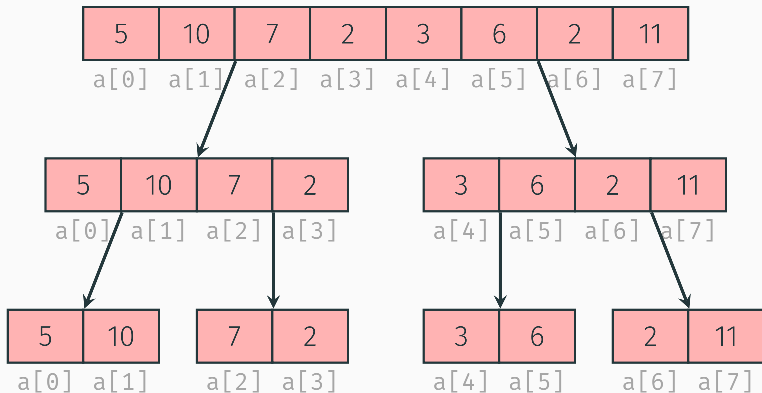
Merge sort: Example

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

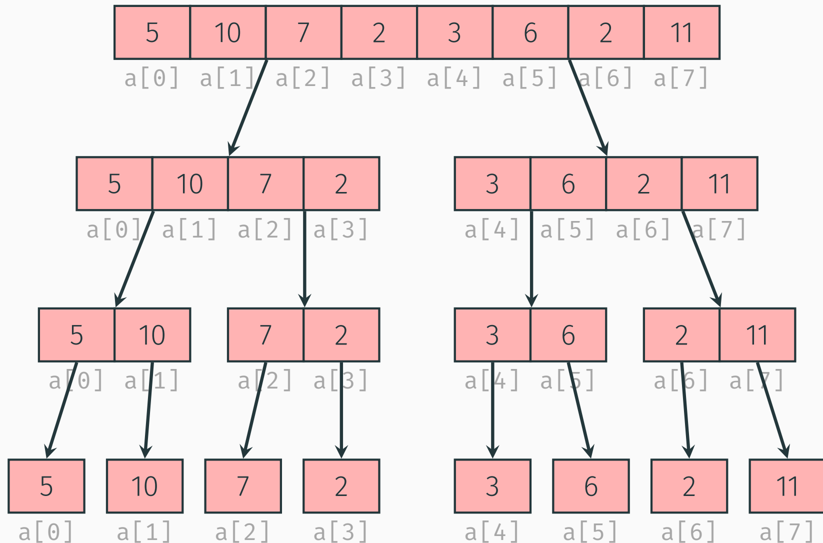
Merge sort: Example



Merge sort: Example

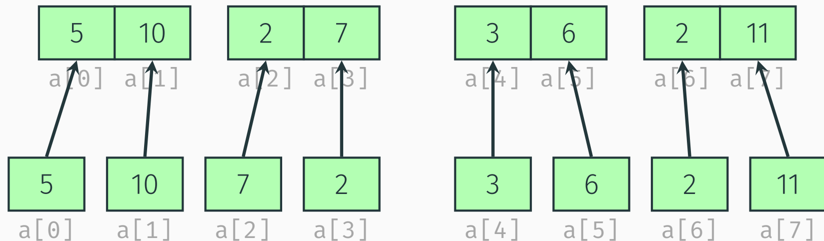


Merge sort: Example

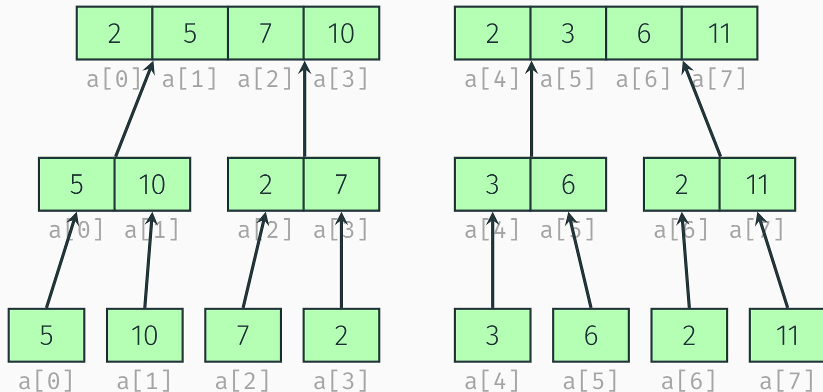


Merge sort: Example

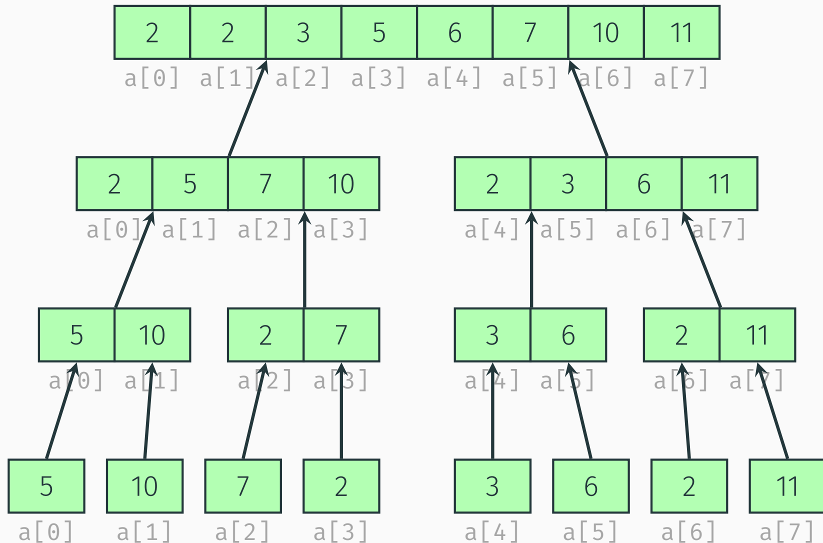
Merge sort: Example



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?

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:

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

With some work, we see the runtime is $\mathcal{O}(n \lg(n))$.

Stability and In-place

If we implement the `merge` function correctly, merge sort will be **stable**.

However, `merge` must construct a new array to contain the output, so merge sort is **not in-place**.

Merge sort: Elaborations

Problem: we can sort only arrays; what about linked lists?

- **Solution 1:** copy the linked list into an array first.
- **Solution 2:** merge sort can run directly on linked lists! (This is not true of heapsort or quicksort).

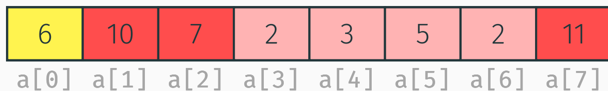
Problem: we're constantly copying and creating new arrays at each level.

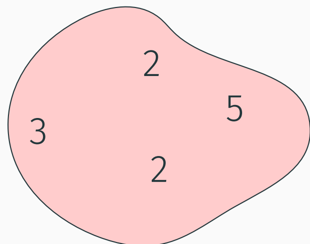
- **Solution:** create a single auxiliary array and swap between it and the original on each level.

Problem: our merge sort algorithm isn't in-place.

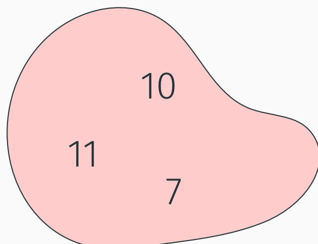
- ...ok, this is complicated – not within scope of class.

Quick sort: Divide step





Pivot

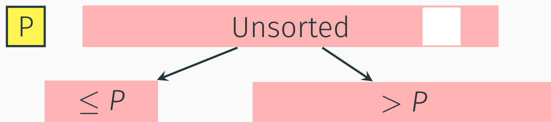


Numbers $\leq$ pivot

Numbers > pivot

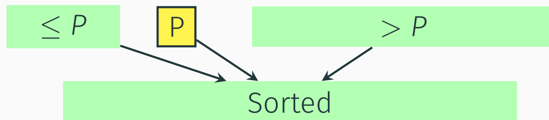
Quick sort: Core pieces

Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1

Combine: Combine sorted portions and the pivot



Quick sort: Summary

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.

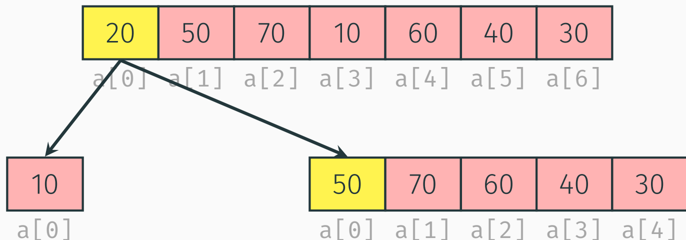
Pseudocode

```
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;
    }
}
```

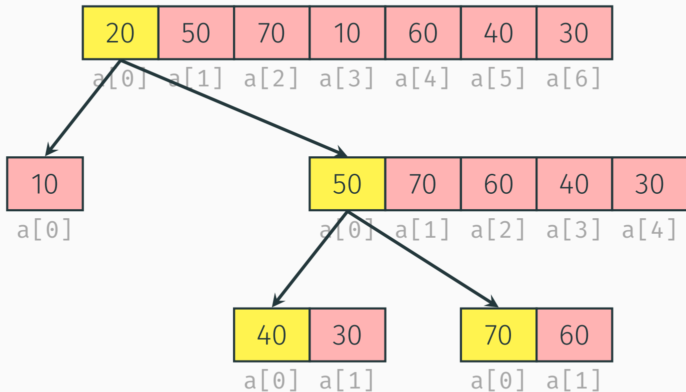
Quick sort: Example

20	50	70	10	60	40	30
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

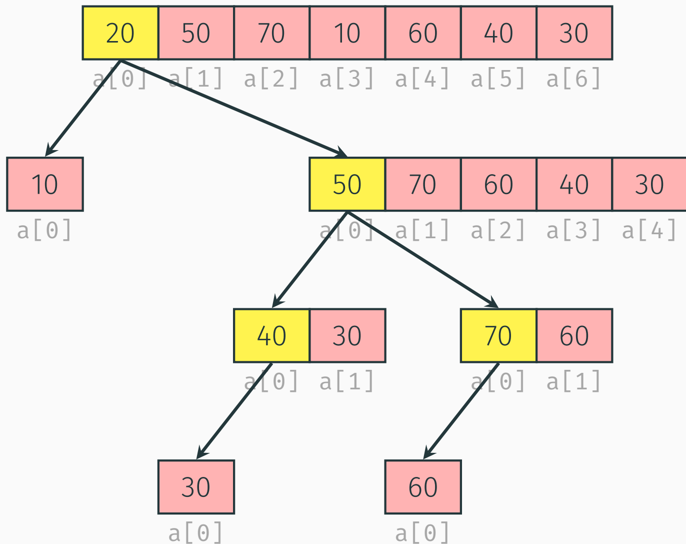
Quick sort: Example



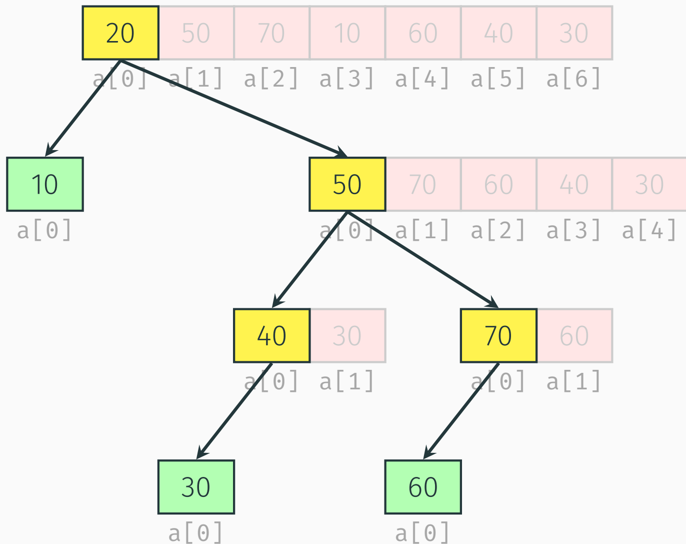
Quick sort: Example



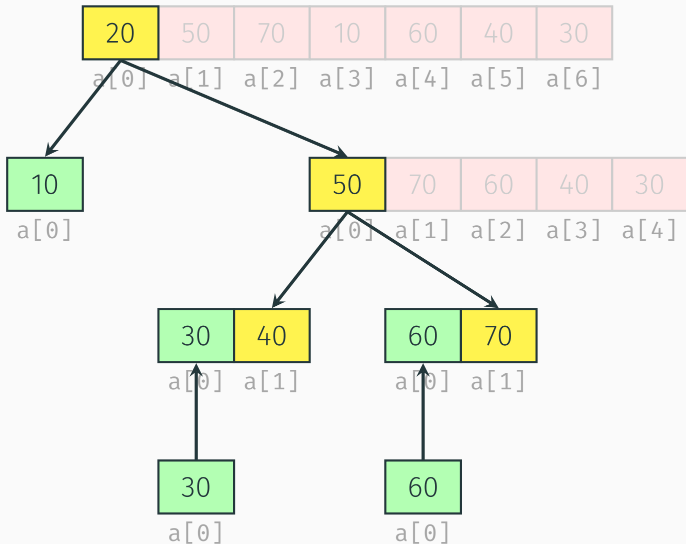
Quick sort: Example



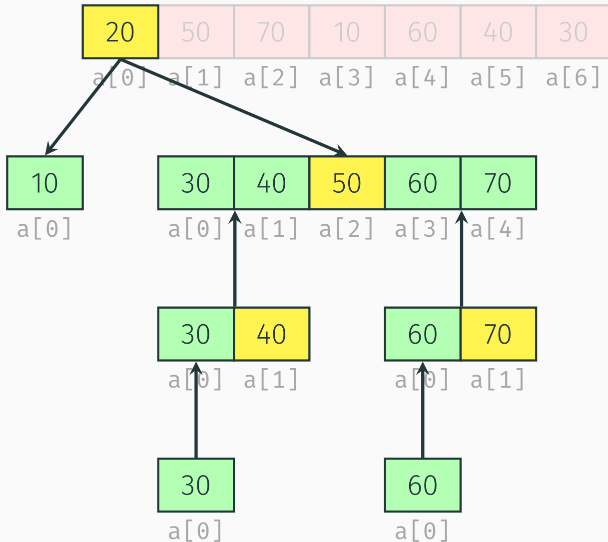
Quick sort: Example



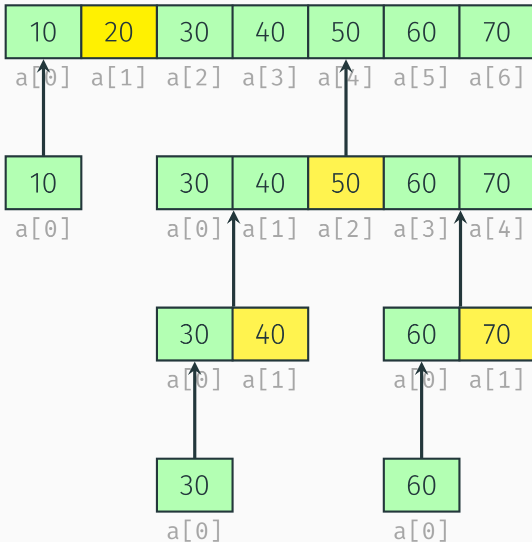
Quick sort: Example



Quick sort: Example



Quick sort: Example



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 $\Theta(n \lg(n))$.

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 $\Theta(n \lg(n))$.

Quick sort: Analysis

Stability

Quick sort is **not stable** – our partition step ends up disregarding and sometimes ignoring the existing relative ordering of duplicate elements.

In-place?

Quick sort is **in-place** – see next few slides for details!

Quick sort: Unresolved questions

How do we pick a pivot?

- Worst case? Pick the minimum or the maximum. The work will shrink by only 1 on each recursive call.
- Ideally? Pick the median. The work will split in half on each recursive call.

How do we partition?

Quick sort: Picking a pivot

How do we find the median?

- Idea: pick the first item in the array
  - Problem: what if the array is already sorted?
  - (Real world data often is partially sorted)
  - But hey, it's speedy ($bigO1$)
- Idea: try finding it by looping through the array
  - Problem: hard to implement, and expensive ($\mathcal{O}(n)$)

These seem like bad ideas :(

Quick sort: Picking a pivot

Other ideas:

- Idea: pick a random element
  - On average, guaranteed to do well – no easy worst case (take CSE 312 for details!)
  - Random number generation can sometimes be expensive/fraught with peril
- Idea: pick the median of first, middle, and last
  - Adversary could still construct malicious input
  - ...but works well in practice, and is efficient

These seem like good ideas :)

Quick sort: Unresolved questions

How do we pick a pivot?

How do we partition?

Quick sort: Partitioning (using median-of-three pivot)

Find the `lo`, `med`, and `hi`

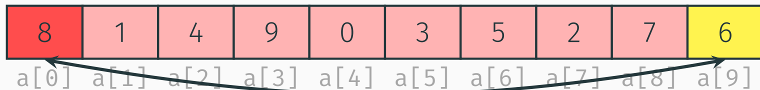
8	1	4	9	0	3	5	2	7	6
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Quick sort: Partitioning (using median-of-three pivot)

Find the `lo`, `med`, and `hi`

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

Find the median of the three and swap with front

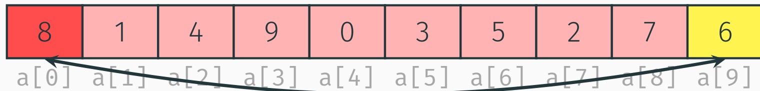


Quick sort: Partitioning (using median-of-three pivot)

Find the `lo`, `med`, and `hi`

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

Find the median of the three and swap with front

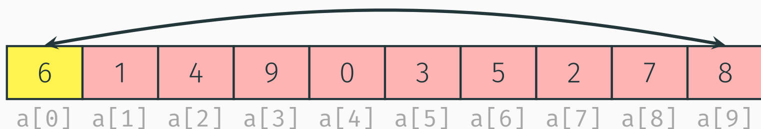


Final result: pivot is now at index 0

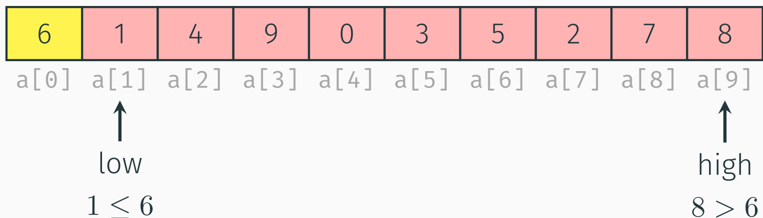
6	1	4	9	0	3	5	2	7	8
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

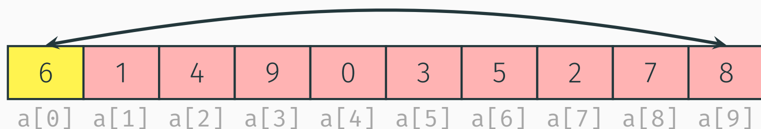


Partitioning:

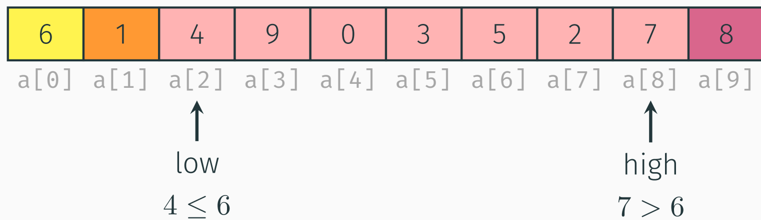


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

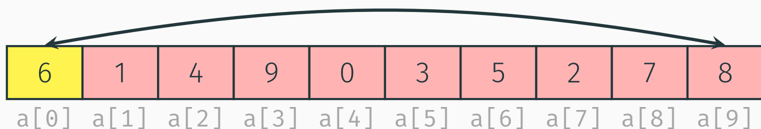


Partitioning:

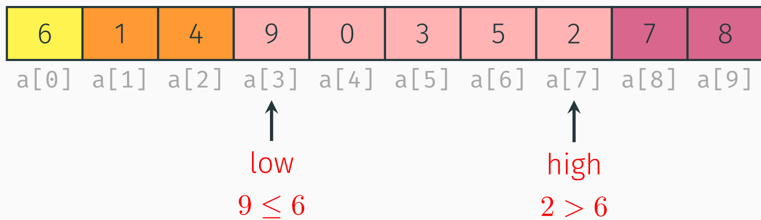


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

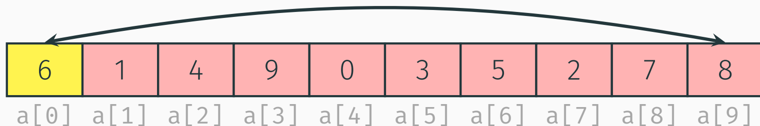


Partitioning:

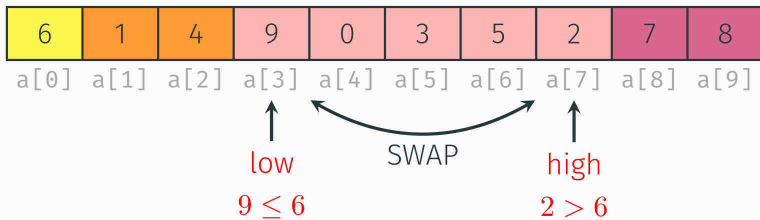


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

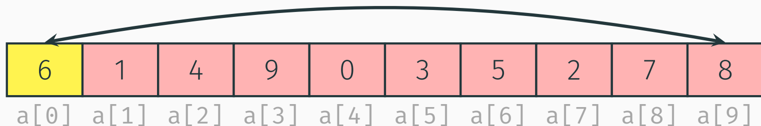


Partitioning:

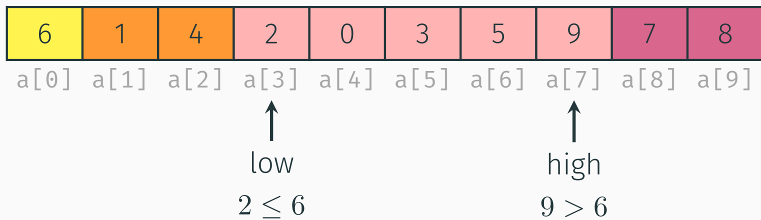


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

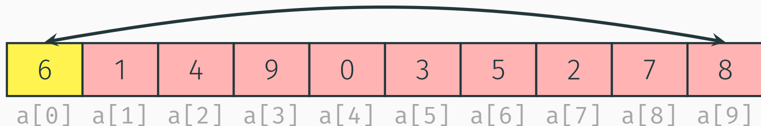


Partitioning:

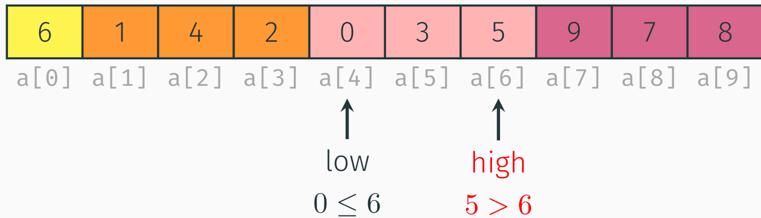


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

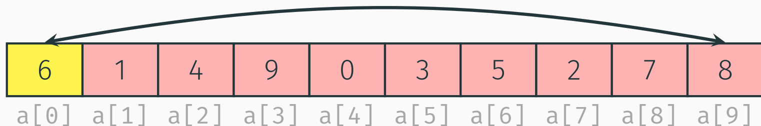


Partitioning:

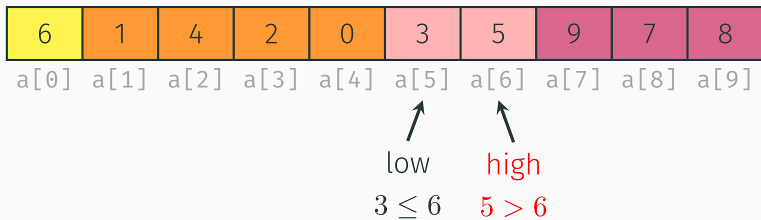


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

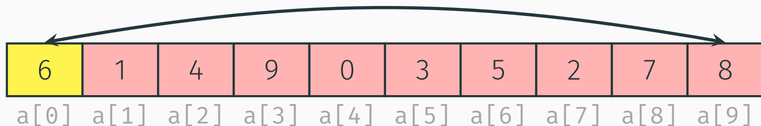


Partitioning:

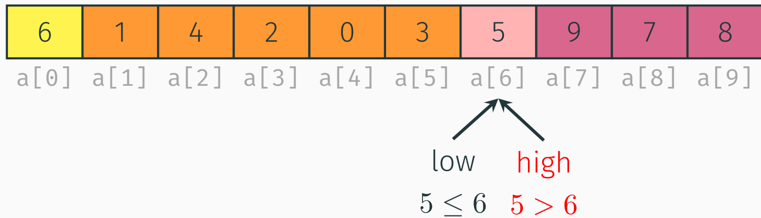


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

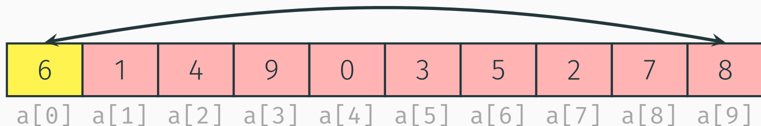


Partitioning:

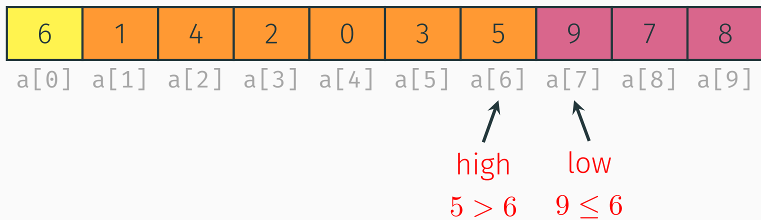


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

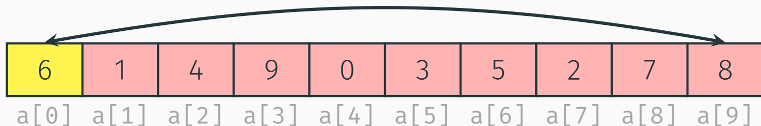


Partitioning:

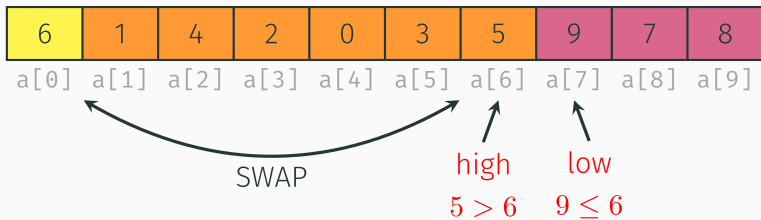


Quick sort: Partitioning (using median-of-three pivot)

Array after moving pivot:

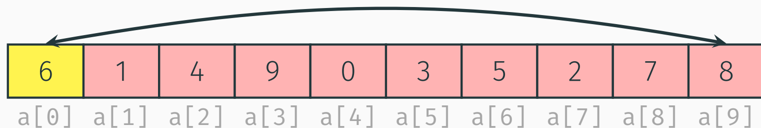


Partitioning:



Quick sort: Partitioning (using median-of-three pivot)

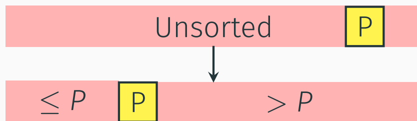
Array after moving pivot:



Partitioning:

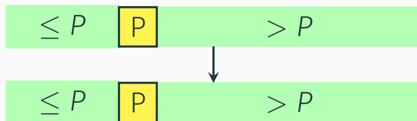
Quick sort: Core pieces revisited

Divide: Pick a pivot, partition in-place into groups



Conquer: When subarray is length ≤ 1, do nothing

Combine: Do nothing; already done!



Bored? Try looking up...

Adaptive sorts: Sorts that adapt to patterns in data

Hybrid sorts: Sorts that combine other sorts

Introsort: Like a better quick sort

- Switches to heap sort when we start heading towards the $\mathcal{O}(n^2)$ worst case
- You can implement this for above-and-beyond!

Timsort: Like a better merge sort (but really complex)

- Switches to insertion sort for small subarrays
- Packed with many optimizations to make merge more efficient (e.g. handling runs of pre-sorted numbers)