CS 2

Introduction to Programming Methods

Sorting

Why Study Sorting?

A Useful Invariant

- Binary Search only works **if the array is sorted**
- BSTs are **based around the idea of sorting the input**

"Local" vs. "Global" Views of Data

- All of our data structure so far only gave us a local view:
  - Heaps gave us a view of the max or min
  - Stacks and Queues gave us a view of most/least recent
  - Dictionaries give us a view of "associated data"
- A "global" view tells us how the elements all interact with each other
- There is no "best" sorting algorithm: most sorts have a purpose

What is SORT?

SORT is the computational problem with the following requirements:

Inputs

- An array `A` of `E` data of length L.
- A consistent, total ordering on all elements of type `E`:

  `compare(a, b)`

Post-Conditions

- For all $0 \leq i < j < L$, `A[i]` $\leq$ `A[j]`
- Every element originally in the array must be somewhere in the resulting array.

An algorithm that solves this computational problem is called a **Comparison Sort**.

Properties of Sorting Algorithms

There are several important properties sorting algorithms

Definition (In-Place Sorting)

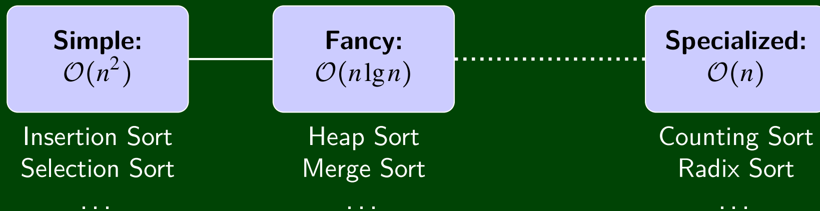
A sorting algorithm is **in-place** if we don't require (more than $\mathcal{O}(1)$) extra space to do the sort.
It's a useful property, because:

- The less memory we use the better...

Definition (Stable Sorting)

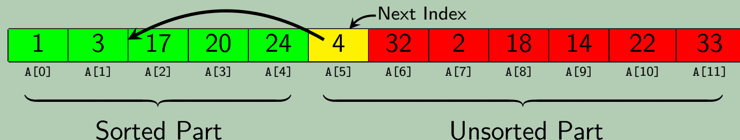
A sorting algorithm is **stable** if the order of any **equal** elements remains the same.
It's a useful property, because:

- We often want to first sort by one index and then another.
- Two objects might be equal but not completely duplicates.



There are a lot of different sorting algorithms out there!

We're not going to cover **all** of them, but we will cover the ones that demonstrate clear advantages in one way or another.

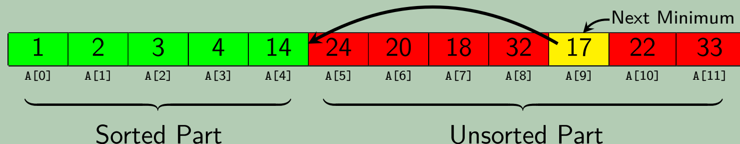


Algorithm

```
1 // i is "# of elements sorted"
2 for (i = 0; i < n; i++) {
3    swap(i, findPlace(i));
4    // shift everything after i over
5 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

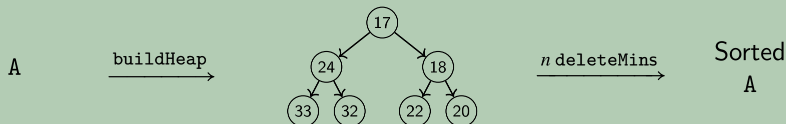


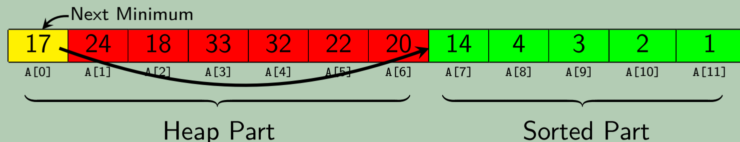
Algorithm

```
1 // i is "# of elements sorted"
2 for (i = 0; i < n; i++) {
3    swap(i, findMin(i, n));
4 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?





Algorithm

```
1 E[] A = buildHeap();
2 for (i = 0; i < n; i++) {
3    swap(n – i – 1, A.deleteMin());
4 }
```

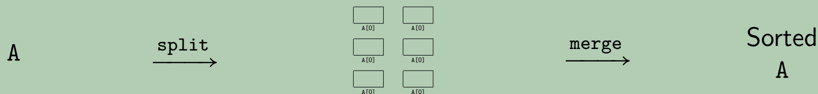
Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Divide and Conquer is a very useful algorithmic technique. It consists of multiple steps:

1. **Divide** the input into smaller pieces (recursively)
2. **Conquer** the individual pieces as base cases
3. **Combine** the finished pieces together (recursively)

```
1  algorithm(input) {
2     if (small enough) {
3        return conquer(input);
4     }
5     pieces = divide(input);
6     for (piece in pieces) {
7        result = combine(result, algorithm(piece));
8     }
9     return result;
10 }
```

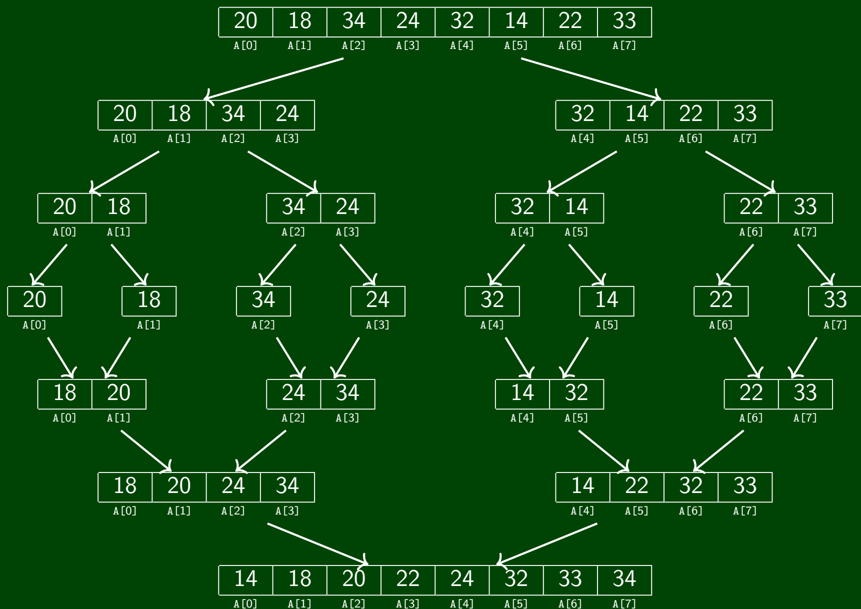


Algorithm

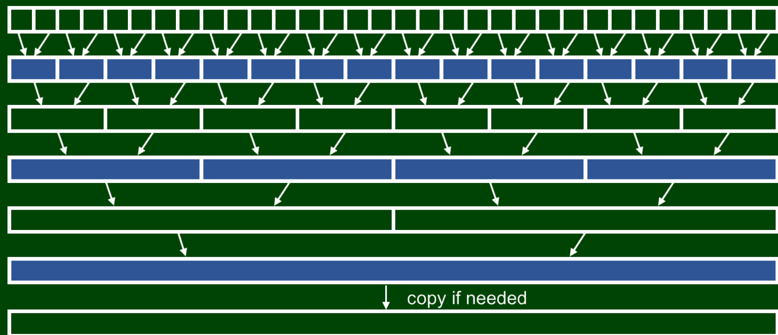
```
1 sort(A) {
2    if (A.length < 2) {
3       return A;
4    }
5    return merge(
6       sort(A[0, ..., mid]),
7       sort(A[mid + 1, ...])
8    );
9 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?


20 18 34 24 32 14 22 33
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]
20 18 34 24
A[0] A[1] A[2] A[3]
32 14 22 33
A[4] A[5] A[6] A[7]
20 18
A[0] A[1]
34 24
A[2] A[3]
32 14
A[4] A[5]
22 33
A[6] A[7]
20
A[0]
18
A[1]
34
A[2]
24
A[3]
32
A[4]
14
A[5]
22
A[6]
33
A[7]
18 20
A[0] A[1]
24 34
A[2] A[3]
14 32
A[4] A[5]
22 33
A[6] A[7]
18 20 24 34
A[0] A[1] A[2] A[3]
14 22 32 33
A[4] A[5] A[6] A[7]
14 18 20 22 24 32 33 34
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

The standard merge sort copies the array **at every step**. This is super slow! We can do better.



In this version, we allocate a **single auxiliary array** and swap between it and the original on each stage.

This is easier iteratively!

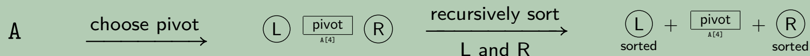
In general, we've been sorting with **arrays**, but what about **linked lists?**

An Approach

- Convert to an array ($\mathcal{O}(n)$)
- Sort ($\mathcal{O}(n \lg(n))$)
- Convert to a list ($\mathcal{O}(n)$)

But, we can actually **do merge sort directly on a list**! (This is not true for heapsort or quicksort!)

Mergesort is also a good choice for external sorting, because the linear merges minimize disk accesses.

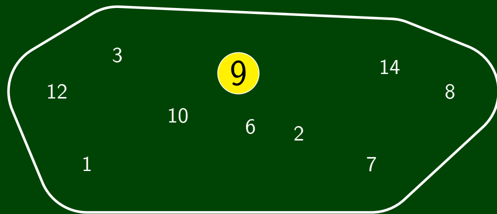


Algorithm

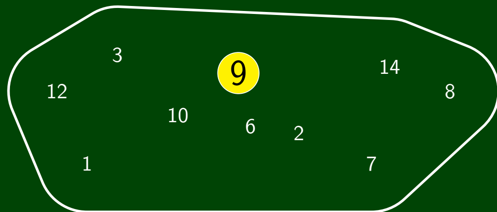
```
1  sort(A) {
2     if (A.length < 2) {
3        return A;
4     }
5
6     pivot = choosePivot(A);
7     left = sort(getLess(A, pivot));
8     right = sort(getGreater(A, pivot));
9     return left + pivot + right;
10 }
```

Runtime and Analysis

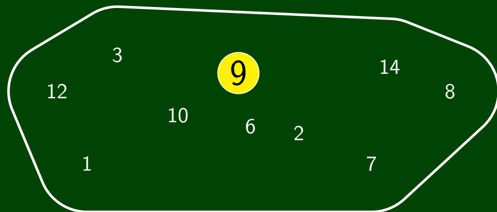
- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?



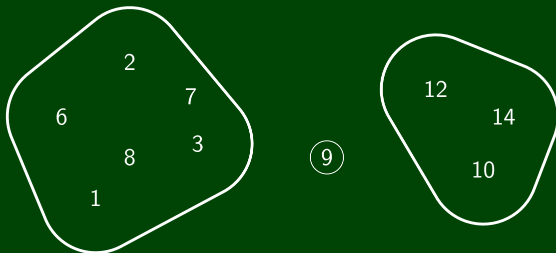
3
9
14
12
8
10
6
2
1
7

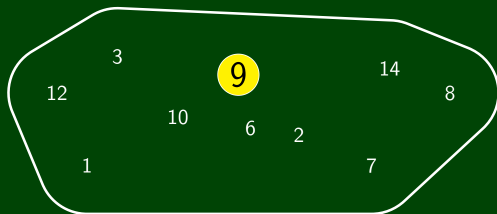


Partition based on `pivot = 9`

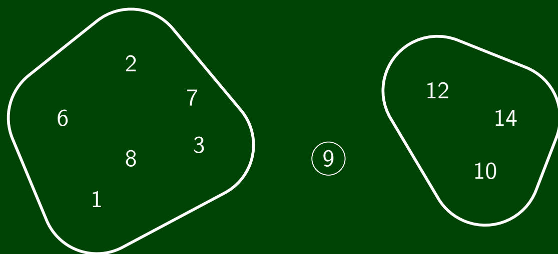


Partition based on `pivot = 9`



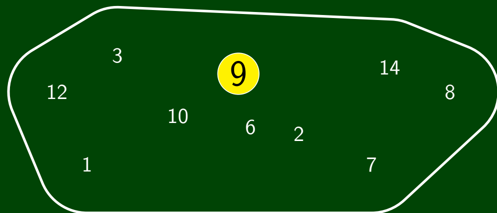


Partition based on `pivot = 9`

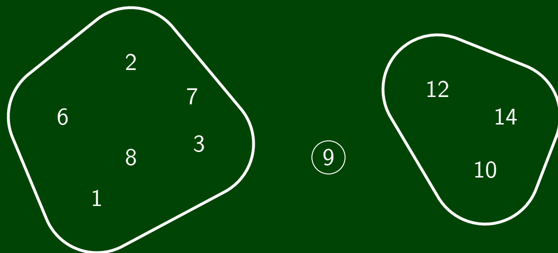


Recursively Sort Halves

Fancy Sorting: Quick Sort



Partition based on `pivot = 9`

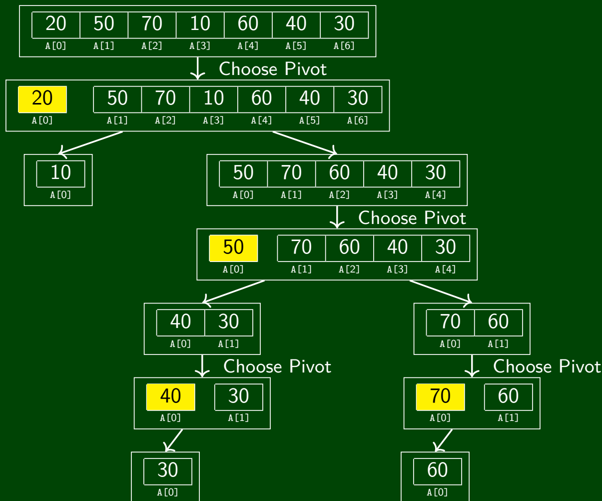


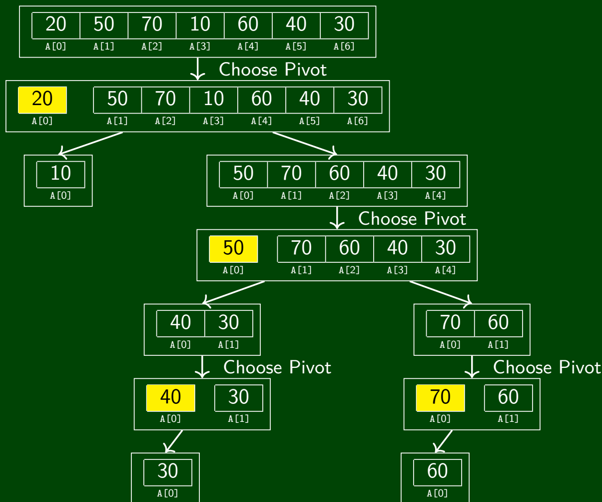
Recursively Sort Halves

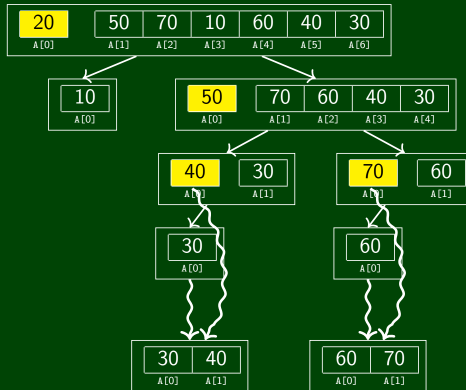
1	2	3	6	7	8
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

9

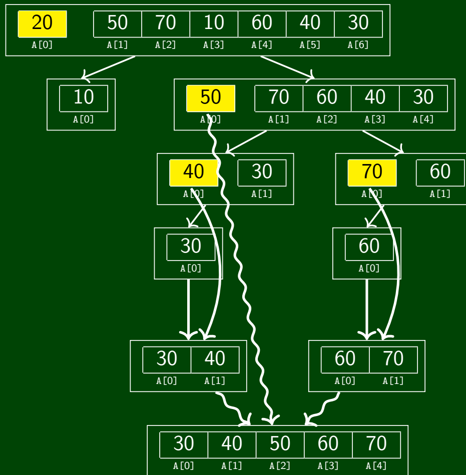
10	12	14
R[0]	R[1]	R[2]


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

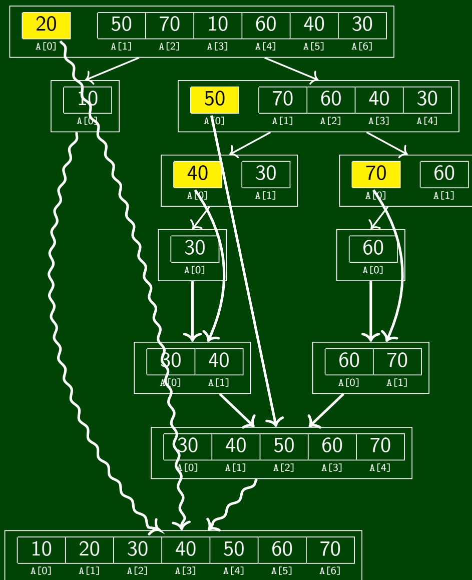

20
50
70
10
60
40
30
A[0]
A[1]
A[2]
A[3]
A[4]
A[5]
A[6]
Choose Pivot
20
A[0]
50
70
10
60
40
30
A[1]
A[2]
A[3]
A[4]
A[5]
A[6]
10
A[0]
50
70
60
40
30
A[0]
A[1]
A[2]
A[3]
A[4]
Choose Pivot
50
A[0]
70
60
40
30
A[1]
A[2]
A[3]
A[4]
40
30
A[0]
A[1]
70
60
A[0]
A[1]
Choose Pivot
Choose Pivot
40
A[0]
30
A[1]
70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



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


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

Fancy Sorting: Quick Sort



We now have the general idea of Quick Sort, but there are some remaining questions:

How do we choose the pivot?

How do we partition the array?

Best Pivot?

If we had our choice of pivots, which one would we choose?

Best Pivot?

If we had our choice of pivots, which one would we choose?

Median

The median will halve the problem each recursive call.

Best Pivot?

If we had our choice of pivots, which one would we choose?

Median

The median will halve the problem each recursive call.

Worst Pivot?

If an adversary chose our pivot (to make the algorithm take as long as possible), which one would they choose?

QuickSort: Good and Bad Pivots

Best Pivot?

If we had our choice of pivots, which one would we choose?

Median

The median will halve the problem each recursive call.

Worst Pivot?

If an adversary chose our pivot (to make the algorithm take as long as possible), which one would they choose?

Minimum or **Maximum**

This will decrease the problem size by only **one** each recursive call.

Pivot Strategies

There are several “standard” strategies to choose a pivot:

1. Choose the first/last element of the array
   - Very fast!
   - Bad, because real-world data is usually “mostly sorted”
2. Random choice
   - Generation can be slow
   - Good, because there’s no easy worst case
3. Median of first, middle, and last elements
   - Works well in practice

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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]

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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]

$\mathbf{1} < 6$ $\mathbf{8} > 6$

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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]

$\mathbf{4} < 6$ $\mathbf{7} > 6$

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move $<$ `pivot` to the front and $>$ `pivot` to the end:

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]

$\mathbf{9} < 6$ $\mathbf{2} > 6$

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

$\mathbf{9} < 6$ swap $\mathbf{2} > 6$

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

$\mathbf{2} < 6$ $\quad$ $\mathbf{9} > 6$

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

$\mathbf{0} < 6$ $\mathbf{5} > 6$

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

$\mathbf{3} < 6$ $\mathbf{5} > 6$

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

5 < 6

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move $<$ `pivot` to the front and $>$ `pivot` to the end:

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

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, 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]

Move `pivot` to front:

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]

Move < `pivot` to the front and > `pivot` to the end:

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

Put `pivot` in middle:

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

QuickSort Analysis

QuickSort Analysis

Best Case

The best case is that the pivot is always the **median**. Then, we get two recursive calls each of size $n/2$.

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

So, the best case behavior is $\mathcal{O}(n \lg(n))$.

Worst Case

QuickSort Analysis

Best Case

The best case is that the pivot is always the **median**. Then, we get two recursive calls each of size $n/2$.

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

So, the best case behavior is $\mathcal{O}(n \lg(n))$.

Worst Case

The worst case is that the pivot is always the **minimum** or the **maximum**. Then, we get one recursive call of size $n-1$.

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

So, the worst case behavior is $\mathcal{O}(n^2)$.

Average Case

With a random pivot, on average we get $\mathcal{O}(n \lg(n))$ behavior.

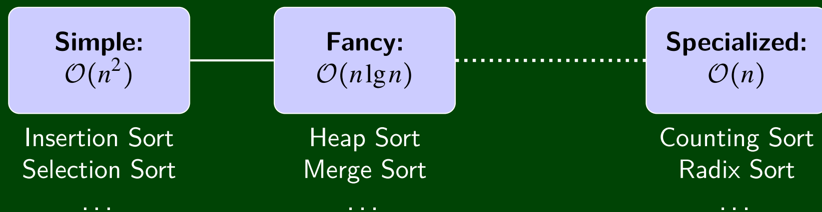
QuickSort Cutoff

For small n, the recursion is a waste. The constants on quick/merge sort are higher than the ones on insertion/selection sort **for small n**.

The solution is to switch to a different algorithm for small sub-problems. For sorting, $n < 10$ is a good choice.

For example:

```
1 void quicksort(int[] arr, int lo, int hi) {
2 if (hi lo < CUTOFF) {
3     insertionSort(arr,lo,hi);
4 }
5 else {
6    ...
7 }
```

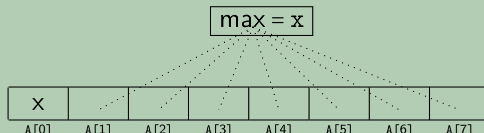


We've discussed some sorting methods

They all happened to be $\Omega(n \lg n)$. Can we do better?

Upper Bound

```
1 int findMax(int[] arr) {
2    int max = arr[0];
3    for (i = 0; i < arr.length; i++) {
4       if (arr[i] > max) {
5          max = arr[i];
6       }
7    }
8    return max;
9 }
```



This algorithm takes **at most** $n-1$ comparisons. So, $n-1$ is an **upper bound** for the **MAXIMUM** problem.

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.

Since the algorithm uses fewer than $n-1$ comparisons, there must be some element of the input (call it x) that wasn't compared to some non-empty group of elements (say y is in that group):

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.
Since the algorithm uses fewer than $n-1$ comparisons, there must be some element of the input (call it x) that wasn't compared to some non-empty group of elements (say y is in that group):

a_0	y	a_2	x	a_3	a_4	a_5	a_6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Consider two distinct values for x:

- $x = \min(a_0, a_1, a_2, \ldots, a_n) - 1$
- $x = \max(a_0, a_1, a_2, \ldots, a_n) + 1$

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.
Since the algorithm uses fewer than $n-1$ comparisons, there must be some element of the input (call it x) that wasn't compared to some non-empty group of elements (say y is in that group):

a_0	y	a_2	x	a_3	a_4	a_5	a_6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Consider two distinct values for x:

- $x = \min(a_0, a_1, a_2, \ldots, a_n) - 1$
- $x = \max(a_0, a_1, a_2, \ldots, a_n) + 1$

Notice that, to be correct, the algorithm must output **different** answers based on the value of x.

Lower Bounds are much more difficult to prove. We must show that **any** algorithm that solves the problem has to do something.

Lower Bound (Proof #1)

Consider an algorithm that solves the **MAXIMUM** problem in **fewer** than $n-1$ comparisons.
Since the algorithm uses fewer than $n-1$ comparisons, there must be some element of the input (call it x) that wasn't compared to some non-empty group of elements (say y is in that group):

a_0	y	a_2	x	a_3	a_4	a_5	a_6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Consider two distinct values for x:

- $x = \min(a_0, a_1, a_2, \ldots, a_n) - 1$
- $x = \max(a_0, a_1, a_2, \ldots, a_n) + 1$

Notice that, to be correct, the algorithm must output **different** answers based on the value of x.

But it never compares x and y! So, it must always output the same thing on otherwise identical arrays.

Bounding The MAXIMUM Problem

Key Ideas

- Must be able to output any valid answer (every index is the max for **some** input)

Bounding The MAXIMUM Problem

Key Ideas

- Must be able to output any valid answer (every index is the max for **some** input)
- The only computations that give information about the correct answer are the **comparisons**

Bounding The MAXIMUM Problem

Key Ideas

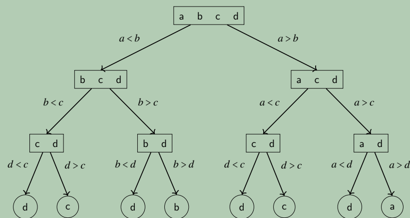
- Must be able to output any valid answer (every index is the max for **some** input)
- The only computations that give information about the correct answer are the **comparisons**
- Must only have **one** valid possibility remaining before answering

Key Ideas

- Must be able to output any valid answer (every index is the max for **some** input)
- The only computations that give information about the correct answer are the **comparisons**
- Must only have **one** valid possibility remaining before answering

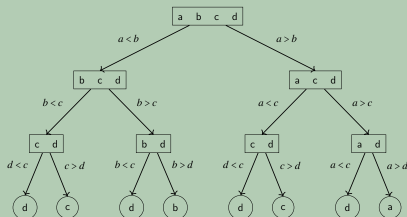
Decision Tree

Consider the comparisons some (arbitrary) algorithm makes:



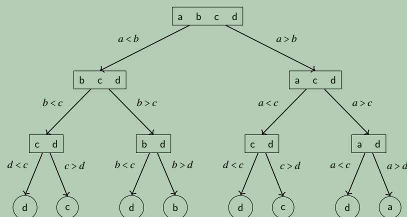
This is a **decision tree**. The nodes have **the remaining valid possibilities**. The edges represent **making a comparison**.

Lower Bound (Proof #2)



- Every valid output (element of the array) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

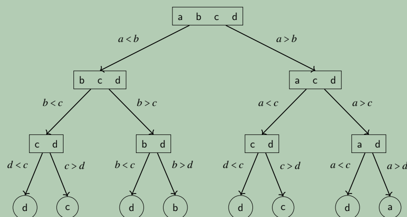
Lower Bound (Proof #2)



- Every valid output (element of the array) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

Lower Bound (Proof #2)

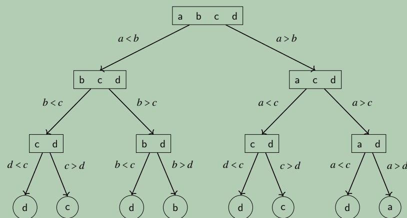


- Every valid output (element of the array) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

A single comparison can rule out (at most) one output.

Lower Bound (Proof #2)



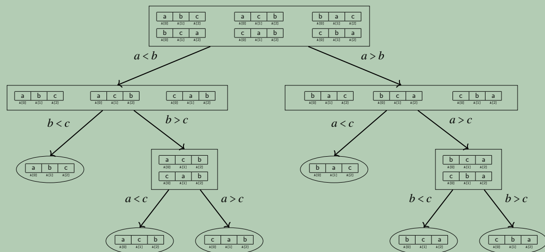
- Every valid output (element of the array) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

A single comparison can rule out (at most) one output.

We begin with n possibilities and each comparison rules out **at most one**. So, the **minimum length** of the **longest path** is $n-1$.

Lower Bound for Sorting



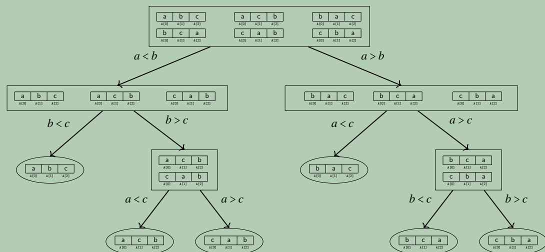
- Every valid output (**??????**) must be a leaf
- Some decision tree **completely** represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

A single comparison can rule out (at most) **???????** output.

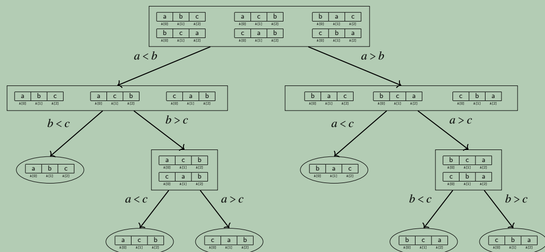
We begin with **????** possibilities and each comparison rules out **??????**.
So, the **minimum length** of the **longest path** is **????**

Filling In The Blanks



- What are the outputs?

Filling In The Blanks

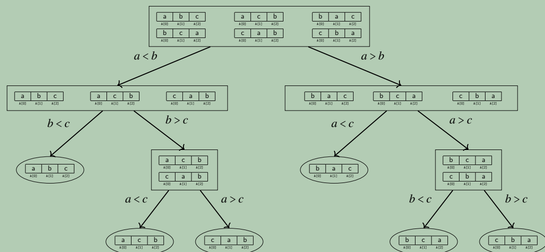


- What are the outputs?

The outputs are permutations of the input:
abc, acb, bac, bca, cab, cba

- How many of them are there?

Filling In The Blanks



- What are the outputs?

  The outputs are permutations of the input:
  abc, acb, bac, bca, cab, cba

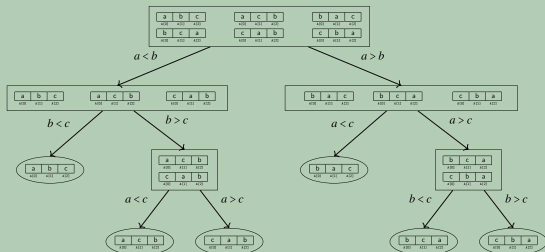
- How many of them are there?

  There are $n!$ permutations of n items:

  ___ n choices ___ $n-1$ choices ___ $n-2$ choices $\cdots$ ___ 1 choice

- How many outputs removed per comparison (min gives us max len)?

Filling In The Blanks



- What are the outputs?

  The outputs are <u>permutations</u> of the input:
  abc, acb, bac, bca, cab, cba

- How many of them are there?

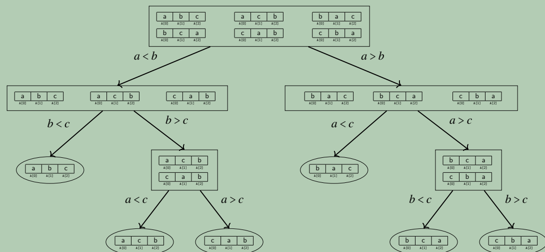
  There are $n!$ permutations of n items:

  $\underline{\hspace{2cm}}$ n choices, $\underline{\hspace{2cm}}$ $n-1$ choices, $\underline{\hspace{2cm}}$ $n-2$ choices $\cdots$ $\underline{\hspace{2cm}}$ 1 choice

- How many outputs removed per comparison (min gives us max len)?

  Every output either goes left or right.
  So, one side has $\geq x/2$ and the other has $\leq x/2$.

Lower Bound for Sorting



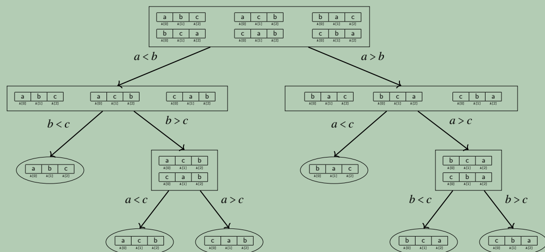
- Every valid output (**permutations of `A`**) must be a leaf
- Some decision tree completely represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

A single comparison can rule out (at most) **half** of the outputs.

We begin with $n!$ possibilities and each comparison rules out at most **half of the remaining ones**. So, the minimum length of the longest path is:

Lower Bound for Sorting



- Every valid output (**permutations of** `A`) must be a leaf
- Some decision tree completely represents the execution of **any** algorithm that solves this problem
- The algorithm must get to a **leaf** before stopping

We must show that the worst input takes at least $f(n)$ comparisons. So, we're asking **how long** the **minimum length of the longest path** is.

A single comparison can rule out (at most) **half** of the outputs.

We begin with $n!$ possibilities and each comparison rules out at most **half of the remaining ones**. So, the minimum length of the longest path is:

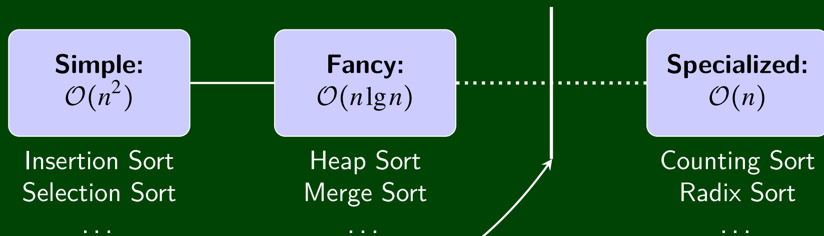
$$\lg(n!).$$

(Asymptotic) Lower Bound for Sorting

We've now shown that the comparison sorting problem is $\Omega(\lg(n!))$. It turns out that this is actually $\Omega(n\lg(n))$:

$$\begin{aligned}
\lg(n!) &= \lg(n(n-1)(n-2)\ldots 1) && \text{[Def. of } n!\text{]} \\
&= \lg(n) + \lg(n-1) + \ldots \lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2}-1\right) + \ldots \lg(1) && \text{[Prop. of Logs]} \\
&\geq \lg(n) + \lg(n-1) + \ldots + \lg\left(\frac{n}{2}\right) \\
&\geq \left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) \\
&= \left(\frac{n}{2}\right)(\lg n - \lg 2) \\
&= \frac{n\lg n}{2} - \frac{n}{2} \\
&\in \Omega(n\lg(n))
\end{aligned}$$

It follows that $\Omega(n\lg(n))$ is a lower bound for the sorting problem!



There are a lot of comparison based sorts, but they can't break the lower bound of $\Omega(n \lg n)$

But what about algorithm that **don't use comparisons**!

Bounded Set Returns!

Remember the assumption we made for the `BoundedSet` ADT?

`BoundedSet` ADT

Data	**Set** of **numerical** keys where $0 \le k \le B$ for some $B \in \mathbb{N}$
`insert(`**key**`)`	Adds **key** to set
`find(`**key**`)`	Returns `true` if **key** is in the set and false otherwise
`delete(`**key**`)`	Deletes **key** from the set

The only difference between `Set` and `BoundedSet` is that `BoundedSet` comes with an upper bound of B.

Suppose we have integers between 1 and B (just like `BoundedSet`). How could we go about sorting them?

Remember the assumption we made for the `BoundedSet` ADT?

`BoundedSet` ADT

`Data`	**Set** of **numerical** keys where $0 \leq k \leq B$ for some $B \in \mathbb{N}$
`insert(`**key**`)`	Adds **key** to set
`find(`**key**`)`	Returns `true` if **key** is in the set and false otherwise
`delete(`**key**`)`	Deletes **key** from the set

The only difference between `Set` and `BoundedSet` is that `BoundedSet` comes with an upper bound of B.

Suppose we have integers between 1 and B (just like `BoundedSet`). How could we go about sorting them?

Counting Sort

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Example

Input: $5\ 1\ 3\ 3\ 2\ 1\ 3\ 4\ 5\ 1\ 1$ $(B = 5)$

- Initialize the array:

A[0]	A[1]	A[2]	A[3]	A[4]

- Loop through the elements:

A[0]	A[1]	A[2]	A[3]	A[4]
4	1	3	1	2

- Loop through the indices

Output: $1\ 1\ 1\ 1\ 2\ 3\ 3\ 3\ 4\ 5\ 5$

Counting Sort Analysis

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Analysis

- Best Case?
- Worst Case?
- Why doesn't the sorting lower bound apply?

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Analysis

- Best Case?

$$\mathcal{O}(n+B)$$

- Worst Case?

$$\mathcal{O}(n+B)$$

- Why doesn't the sorting lower bound apply?

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Analysis

- Best Case?

$$\mathcal{O}(n+B)$$

- Worst Case?

$$\mathcal{O}(n+B)$$

- Why doesn't the sorting lower bound apply?

  It's not a comparison sort! We actually didn't use comparisons at all!
- When should we use Counting Sort?

Counting Sort Analysis

Counting Sort

Assuming all data is **ints** between 1 and B:

- Create an `int` array of size B
- Loop through the elements and increment their counts
- Then, loop through the array and output each element found

Analysis

- Best Case?

$$\mathcal{O}(n+B)$$

- Worst Case?

$$\mathcal{O}(n+B)$$

- Why doesn't the sorting lower bound apply?

  It's not a comparison sort! We actually didn't use comparisons at all!
- When should we use Counting Sort?

  We should use Counting Sort when $n \approx B$.

Radix Sort

- Choose a "number" representation (e.g. $(100)_{10} = (1100100)_2 = (d)_{128}$)
- For each digit from least significant to most significant, do a **stable sort** (why stable?)

Usually for the sorting step, we use **counting sort**.

Example

478		721		003		003
537		003		009		009
009		143		721		038
721	Sort Yellow →	537	Sort Yellow →	537	Sort Yellow →	067
003		067		038		143
038		478		143		478
143		038		067		537
067		009		478		721

Radix Sort

- Choose a "number" representation (e.g. $(100)_{10} = (1100100)_2 = (d)_{128}$). Say base B.
- For each digit from least significant to most significant, do a **stable COUNTING sort**. Say there are P passes.

Analysis

- Best Case?
- Worst Case?
- Should we use radix sort?

Radix Sort

- Choose a "number" representation (e.g. $(100)_{10} = (1100100)_2 = (d)_{128}$). Say base B.
- For each digit from least significant to most significant, do a **stable COUNTING sort**. Say there are P passes.

Analysis

- Best Case?

$$\mathcal{O}(P(B+n))$$

- Worst Case?

$$\mathcal{O}(P(B+n))$$

- Should we use radix sort?

Counting Sort: Analysis

Radix Sort

- Choose a "number" representation (e.g. $(100)_{10} = (1100100)_2 = (d)_{128}$). Say base B.
- For each digit from least significant to most significant, do a **stable COUNTING sort**. Say there are P passes.

Analysis

- Best Case?

$$\mathcal{O}(P(B+n))$$

- Worst Case?

$$\mathcal{O}(P(B+n))$$

- Should we use radix sort?

  Consider Strings of English letters up to length 15:

  - Radix Sort will take 15(52 + n)
  - For $n < 33{,}000$, $n \lg n$ wins.

Possibly the most useful application of sorting is as a form of **pre-processing**. We sort the input in $\mathcal{O}(n \lg n)$ and then solve the actual problem using the sorted data. (e.g. if we expect to do more than $\mathcal{O}(n)$ finds, the sorting step is worth it)

Big CS Idea!

To make a repeated operation easier, do an expensive **pre-processing step** once. You saw this with DFAs and String Matching in CSE 311 as well!

The remaining slides are kind of neat and interesting, but we won't cover them in lecture. Feel free to look at them on your own.

The **median** problem has already come up. Let's explore it more!

SELECT is the computational problem with the following requirements:

Inputs

- An array `A` of `E` data of length L and a number $0 \le k < L$.
- A consistent, total ordering on all elements of type `E`:

  `compare(a, b)`

Post-Conditions

- The array remains unchanged.
- Let `B` be the ordering that **SORT** would return. We return `B`(k).

Solving SELECT(k)

- Copy `A` into `B`
- Sort `B`
- Return (`B`(k)

Awesome, except this is $\mathcal{O}(n \lg n)$

An Algorithm to Solve SELECT

Solving **SELECT**(k)

- Copy `A` into `B`
- Sort `B`
- Return (`B`(k)

Awesome, except this is $\mathcal{O}(n \lg n)$

Another idea, instead of "sorting", only sort the parts we need.

QuickSort: A Reminder

- Choose a pivot in `A`: p
- Partition `A` into two arrays: `SMALLER` and `LARGER`
- QuickSort `SMALLER`.
- QuickSort `LARGER`.
- `SMALLER` $+ [p] +$ `LARGER` is a sorted array.

Solving **SELECT**(k)

- Copy `A` into `B`
- Sort `B`
- Return (`B`(k)

Awesome, except this is $\mathcal{O}(n \lg n)$

Another idea, instead of "sorting", only sort the parts we need.

QuickSort: A Reminder

- Choose a pivot in `A`: p
- Partition `A` into two arrays: `SMALLER` and `LARGER`
- QuickSort `SMALLER`.
- QuickSort `LARGER`.
- `SMALLER` $+ [p] +$ `LARGER` is a sorted array.

Idea: To find the k-th element, do we need to recurse on both sides?

QuickSelect(A, k)

- Choose a pivot in A: p
- Partition A into two arrays: SMALLER and LARGER
- Since we know how big SMALLER and LARGER are, we know the final index of p. Call this x.
- If $k = \texttt{x}$, return p.
- If $k < \texttt{x}$, return QuickSelect(SMALLER, k)
- If $k > \texttt{x}$, return QuickSelect(LARGER, $k - \texttt{x}$)

Analysis

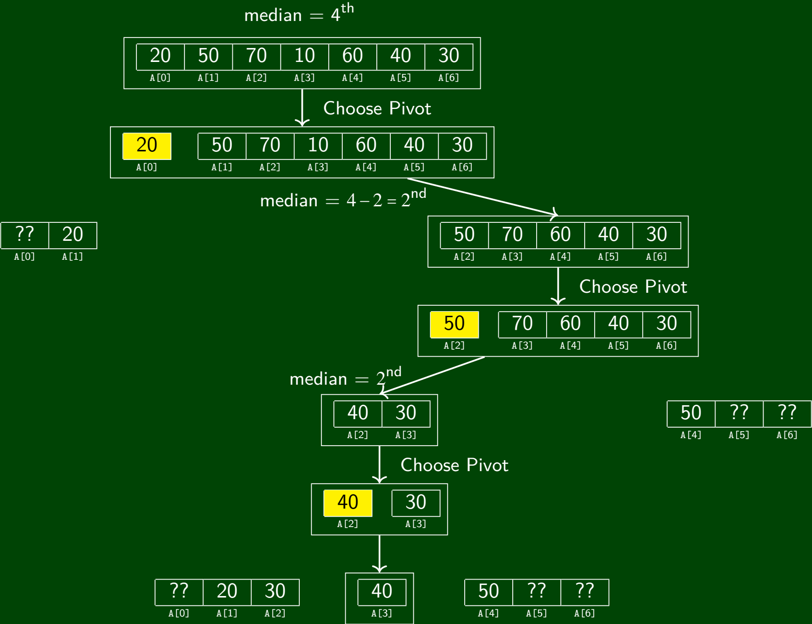
- Best Case:
- Worst Case:
- (Average Case is $\mathcal{O}(n)$)

QuickSelect(A, k)

- Choose a pivot in A: p
- Partition A into two arrays: SMALLER and LARGER
- Since we know how big SMALLER and LARGER are, we know the final index of p. Call this x.
- If $k = \texttt{x}$, return p.
- If $k < \texttt{x}$, return QuickSelect(SMALLER, k)
- If $k > \texttt{x}$, return QuickSelect(LARGER, $k - \texttt{x}$)

Analysis

- Best Case: $T(n) = T(n/2) + cn$ (So, $\mathcal{O}(n)$)
- Worst Case: $T(n) = T(n-1) + cn$ (So, $\mathcal{O}(n^2)$)
- (Average Case is $\mathcal{O}(n)$)


median = 4th
20 50 70 10 60 40 30
A[0] A[1] A[2] A[3] A[4] A[5] A[6]
Choose Pivot
20
A[0]
50 70 10 60 40 30
A[1] A[2] A[3] A[4] A[5] A[6]
median = 4 – 2 = 2nd
?? 20
A[0] A[1]
50 70 60 40 30
A[2] A[3] A[4] A[5] A[6]
Choose Pivot
50
A[2]
70 60 40 30
A[3] A[4] A[5] A[6]
median = 2nd
40 30
A[2] A[3]
50 ?? ??
A[4] A[5] A[6]
Choose Pivot
40
A[2]
30
A[3]
?? 20 30
A[0] A[1] A[2]
40
A[3]
50 ?? ??
A[4] A[5] A[6]

Deterministic QuickSelect (Median-of-Medians)

Median-of-Medians

- Split `A` into $g = n/5$ groups of 5 elements.
- Sort each group and find the medians: $m_1, m_2, \ldots, m_{n/5}$
- Find p: the median of the medians (recursively. . .)
- Separate the input into two groups `SMALLER` and `LARGER` and recurse on the appropriate piece

This algorithm is "basically" **QuickSelect**, but with a special pivot.

Analysis

The key to this algorithm is that whichever side we recurse on is at least $3/10$ of the input. Here's why:

- Consider `SMALLER`.

Deterministic QuickSelect (Median-of-Medians)

Median-of-Medians

- Split `A` into $g = n/5$ groups of 5 elements.
- Sort each group and find the medians: $m_1, m_2, \ldots, m_{n/5}$
- Find p: the median of the medians (recursively...)
- Separate the input into two groups `SMALLER` and `LARGER` and recurse on the appropriate piece

This algorithm is "basically" **QuickSelect**, but with a special pivot.

Analysis

The key to this algorithm is that whichever side we recurse on is at least $3/10$ of the input. Here's why:

- Consider `SMALLER`. We know that at least $g/2$ of the groups have a median $\geq p$. Of the 5 elements in each of these groups, since the median is $\geq p$, 3 of them are $\geq p$ (possibly including the median). Putting this together, we have $3(g/2) = 3((n/5)/2) = 3n/10$ elements $\geq p$. This means we **know** we will discard at least this many. So, the maximum number of elements we could recurse on is $7n/10$.
- The other case is symmetric.

Median-of-Medians

- Split `A` into $g = n/5$ groups of 5 elements.
- Sort each group and find the medians: $m_1, m_2, \ldots, m_{n/5}$
- Find p: the median of the medians (we're gonna do this recursively. . .)
- Separate the input into two groups `SMALLER` and `LARGER` and recurse on the appropriate piece

Solving The Recurrence

So, putting all this together gives us the recurrence

$$T(n) \leq \mathcal{O}(5 \lg 5)\left(\frac{n}{5}\right) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

$$= cn \quad + T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right)$$

$$= cn \quad + \left(\frac{2n}{10} + T\left(2\left(\frac{2n}{10}\right)\right) + T\left(7\left(\frac{2n}{10}\right)\right)\right)$$

$$+ \left(\frac{7n}{10} + T\left(2\left(\frac{7n}{10}\right)\right) + T\left(7\left(\frac{7n}{10}\right)\right)\right)$$

$$= cn \quad + \frac{9n}{10} + T\left(\frac{2^2 n}{10^2}\right) + 2T\left(7 \times 2 \times \left(\frac{n}{10^2}\right)\right) + T\left(\frac{7^2 n}{10^2}\right)$$

Solving The Recurrence

So, putting all this together gives us the recurrence

$$
\begin{aligned}
T(n) \leq \mathcal{O}(5 \lg 5)\left(\frac{n}{5}\right) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\
&= cn \quad + T\left(\frac{2n}{10}\right) + T\left(\frac{7n}{10}\right) \\
&= cn \quad + \left(\frac{2n}{10} + T\left(2\left(\frac{2n}{10}\right)\right) + T\left(7\left(\frac{2n}{10}\right)\right)\right) \\
&\quad\quad + \left(\frac{7n}{10} + T\left(2\left(\frac{7n}{10}\right)\right) + T\left(7\left(\frac{7n}{10}\right)\right)\right) \\
&= cn \quad + \frac{9n}{10} + T\left(\frac{2^2 n}{10^2}\right) + 2T\left(7 \times 2 \times \left(\frac{n}{10^2}\right)\right) + T\left(\frac{7^2 n}{10^2}\right) \\
&\leq cn \quad + \frac{9n}{10} + \frac{2^2 + 2(7 \times 2) + 7^2}{10^2} + \ldots \\
&= cn \quad + \frac{9n}{10} + \frac{9^2 n}{10^2} + \ldots \\
&= cn \quad \left(\sum_{i=0}^{\infty} 9^i 10^i\right) = cn\left(\frac{1}{1 - 9/10}\right) = 10cn
\end{aligned}
$$

Whoo hoo!