CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 13: Expected Runtime

Announcements

- WA2 due Sunday 10/1 @ 11:59PM
- Midterm #1 will be Monday 10/2 in class
  - Practice exams and WA1 answer key posted to the course website
  - See Piazza post for all the juicy details
  - Make sure you have AR appointments set up if necessary

Recap - Merge Sort

Divide: Split the sequence in half

$D(n) = \Theta(n)$ (can do in $\Theta(1)$)

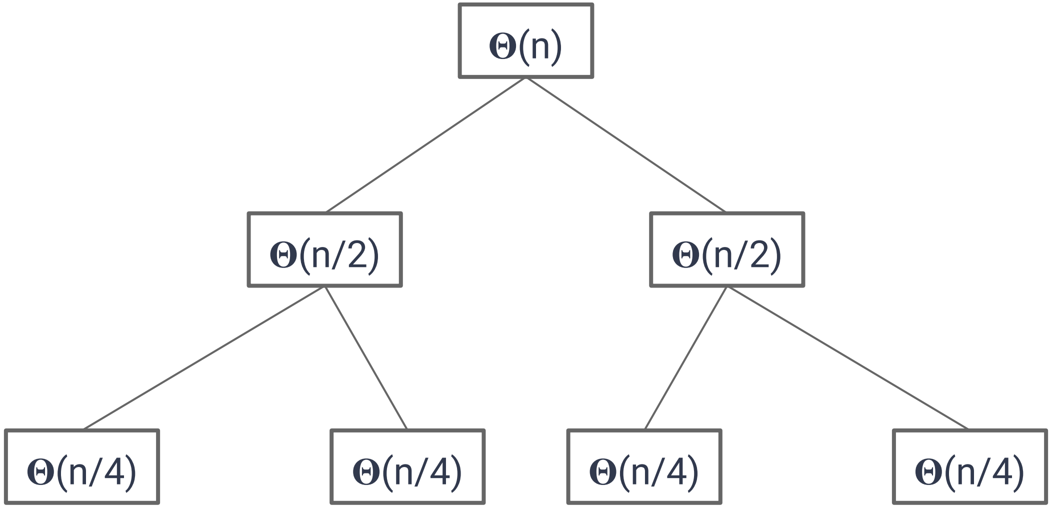
Conquer: Sort the left and right halves

$a = 2, b = 2, c = 1$

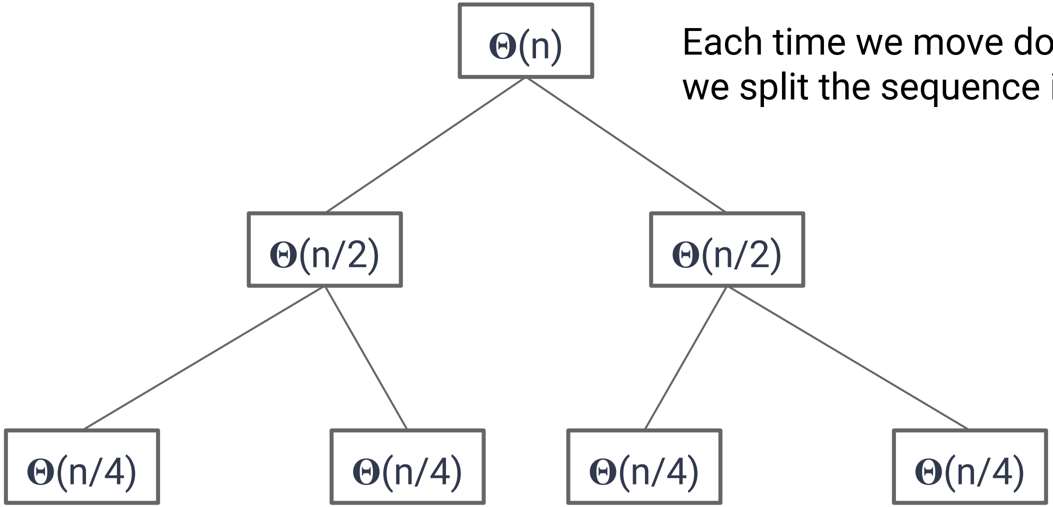
Combine: Merge halves together

$C(n) = \Theta(n)$

Merge Sort: Intuition

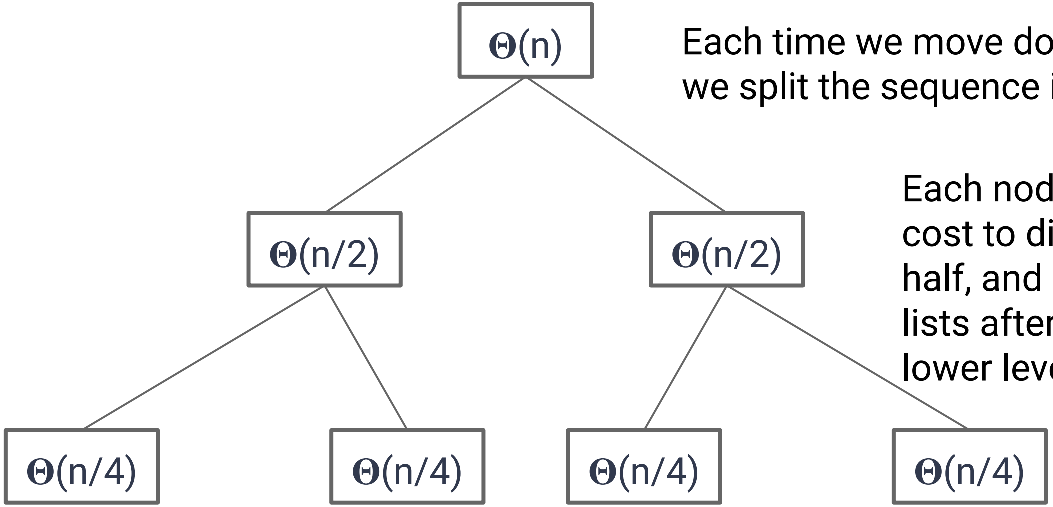


Merge Sort: Intuition



Each time we move down a level, we split the sequence in half

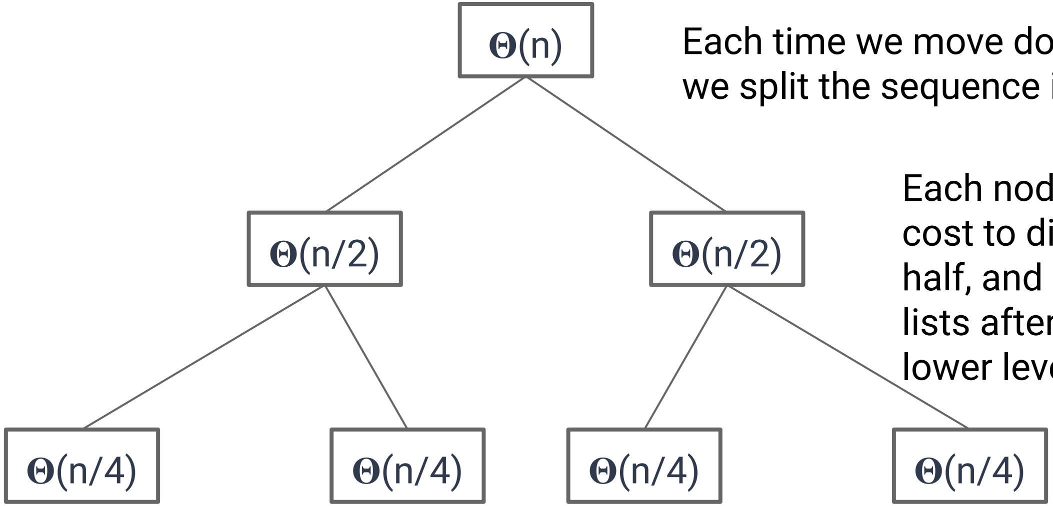
Merge Sort: Intuition



Each time we move down a level, we split the sequence in half

Each node is labeled with the total cost to dividing the sequence in half, and combining the sorted lists after they are sorted by the lower levels

Merge Sort: Intuition



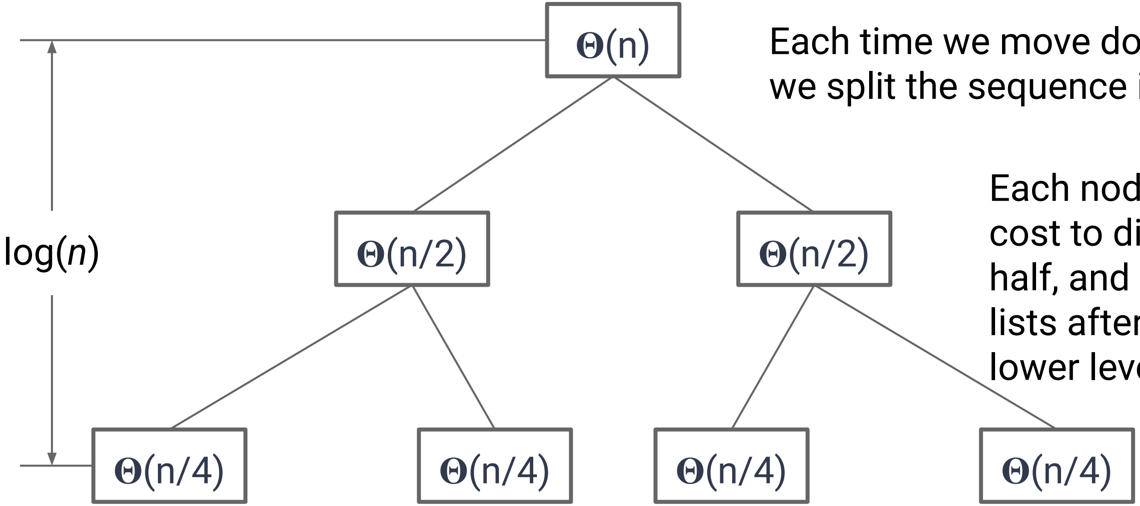
Each time we move down a level, we split the sequence in half

Each node is labeled with the total cost to dividing the sequence in half, and combining the sorted lists after they are sorted by the lower levels

Notice the total cost of each level is always $\Theta(n)$

Merge Sort: Intuition

Because we divide in half at each level, we have log(*n*) levels



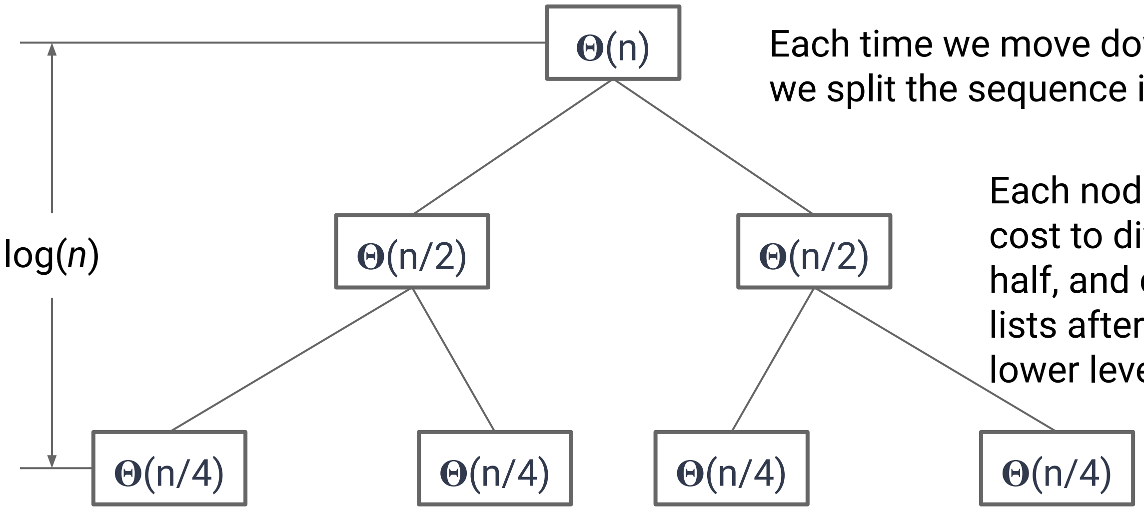
Each time we move down a level, we split the sequence in half

Each node is labeled with the total cost to dividing the sequence in half, and combining the sorted lists after they are sorted by the lower levels

Notice the total cost of each level is always $\Theta(n)$

Merge Sort: Intuition

Because we divide in half at each level, we have log(n) levels



Each time we move down a level, we split the sequence in half

Each node is labeled with the total cost to dividing the sequence in half, and combining the sorted lists after they are sorted by the lower levels

Notice the total cost of each level is always $\Theta(n)$

Hypothesis: The cost of merge sort is $n \log(n)$

Merge Sort: Proof by Induction

Base Case: $T(1) \le c\ 1 \log(1)$

~~$c_0 \le 0$~~

$T(2) \le c\ 2 \log(2)$

True for any $c > c_0 / 2$

Merge Sort: Proof by Induction

Assume: $T(n/2) \leq c\,(n/2)\log(n/2)$

Show: $T(n) \leq cn\log(n)$

Merge Sort: Proof by Induction

Assume: $T(n/2) \leq c\,(n/2)\log(n/2)$

Show: $T(n) \leq cn\log(n)$

$$2 \cdot T(\frac{n}{2}) + c_1 + c_2 n \leq cn\log(n)$$

Merge Sort: Proof by Induction

Assume: $T(n/2) \leq c\,(n/2)\log(n/2)$

Show: $T(n) \leq cn\log(n)$

$$2 \cdot T(\frac{n}{2}) + c_1 + c_2 n \leq cn\log(n)$$

By the assumption, and transitivity, we just need to show:

$$2c\frac{n}{2}\log\left(\frac{n}{2}\right) + c_1 + c_2 n \leq cn\log(n)$$

Merge Sort: Proof by Induction

Assume: $T(n/2) \leq c\,(n/2)\log(n/2)$

Show: $T(n) \leq cn\log(n)$

$$2 \cdot T(\frac{n}{2}) + c_1 + c_2 n \leq cn\log(n)$$

By the assumption, and transitivity, we just need to show:

$$2c\frac{n}{2}\log\left(\frac{n}{2}\right) + c_1 + c_2 n \leq cn\log(n)$$

$$cn\log(n) - cn\log(2) + c_1 + c_2 n \leq cn\log(n)$$

Merge Sort: Proof by Induction

Assume: $T(n/2) \leq c\,(n/2)\log(n/2)$

Show: $T(n) \leq cn\log(n)$

$$2 \cdot T(\frac{n}{2}) + c_1 + c_2 n \leq cn\log(n)$$

By the assumption, and transitivity, we just need to show:

$$2c\frac{n}{2}\log\left(\frac{n}{2}\right) + c_1 + c_2 n \leq cn\log(n)$$

$$cn\log(n) - cn\log(2) + c_1 + c_2 n \leq cn\log(n)$$

$$c_1 + c_2 n \leq cn\log(2)$$

Merge Sort: Proof by Induction

$$c_1 + c_2 n \leq cn \log(2)$$

Merge Sort: Proof by Induction

$$c_1 + c_2 n \leq cn\log(2)$$

$$\frac{c_1}{n\log(2)} + \frac{c_2}{\log(2)} \leq c$$

Merge Sort: Proof by Induction

$$c_1 + c_2 n \leq cn\log(2)$$

$$\frac{c_1}{n\log(2)} + \frac{c_2}{\log(2)} \leq c$$

Which is true for any

$$n_0 \geq \frac{c_1}{\log(2)} \quad \text{and} \quad c > \frac{c_2}{\log(2)} + 1$$

Benefits of a Sorted List

So in ***O*(*n* log(*n*))** we can sort a list using the merge sort algorithm…

But how does that benefit us?

Binary vs Linear Search

Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take?

Binary vs Linear Search

Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take? ***O*(*n*)**, we have to check all ***n*** elements

This is called a **Linear Search** (it takes linear time)

Binary vs Linear Search

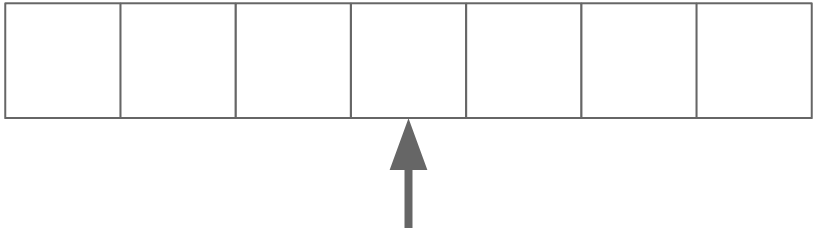
Consider searching for a particular value in an **Array** (or **ArrayList**)...

How long does that search take? ***O*(*n*)**, we have to check all ***n*** elements

This is called a **Linear Search** (it takes linear time)

What if our list is sorted? Can we do better?

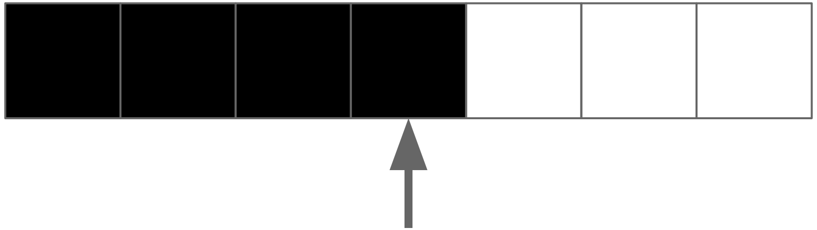
Binary vs Linear Search



Check the middle element (which we can access in constant time)

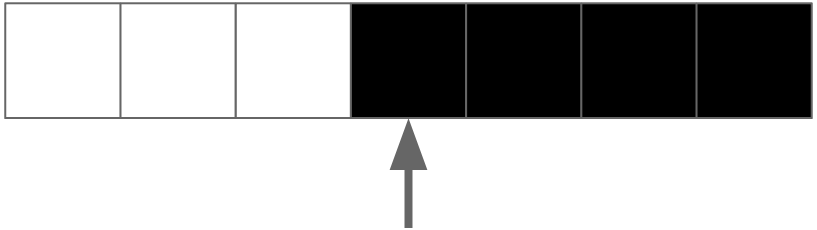
Binary vs Linear Search

We can ignore half the list



Check the middle element (which we can access in constant time)
If it is smaller than what we are looking for, then our target must be to the right (because our list is sorted)

Binary vs Linear Search



We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)

Binary vs Linear Search

We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

Binary vs Linear Search

We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

What is the runtime to search in this fashion?

Binary vs Linear Search

We can ignore half the list

Check the middle element (which we can access in constant time)
If it is larger than what we are looking for, then our target must be to the left (because our list is sorted)
Repeat this process recursively with the remaining elements

What is the runtime to search in this fashion? ***O*****(log(*****n*****))**

Binary vs Linear Search

Linear search:

- Removes one element from consideration each step, ***O*(*n*)**
- Does not require list to be sorted
- Does not require constant time random access

Binary search:

- Removes half of the elements from consideration each step, ***O*(log(*n*))**
- Requires list to be sorted
- Requires constant time random access

Merge Sort

Where is all of the "work" being done?

Merge Sort

Where is all of the "work" being done?

The combine step

Merge Sort

Where is all of the "work" being done?

The combine step

Can we put the work in the divide step instead?

QuickSort

Idea: What if we divide our sequence around a particular value?

What value would we like to choose?

QuickSort

Idea: What if we divide our sequence around a particular value?

What value would we like to choose? **Median**

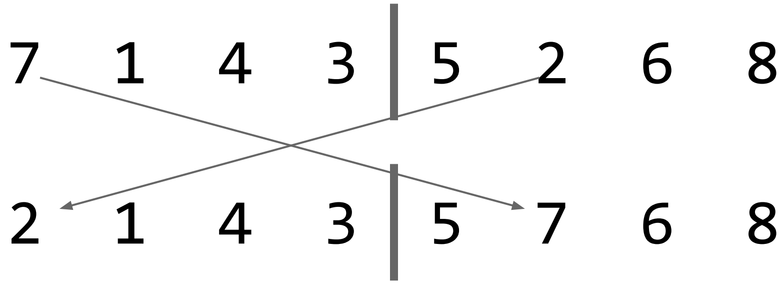
QuickSort: Idealized Version

7 1 4 3 5 2 6 8

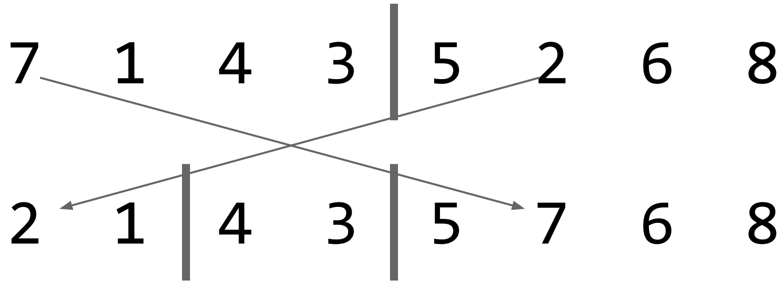
QuickSort: Idealized Version

7 1 4 3 | 5 2 6 8

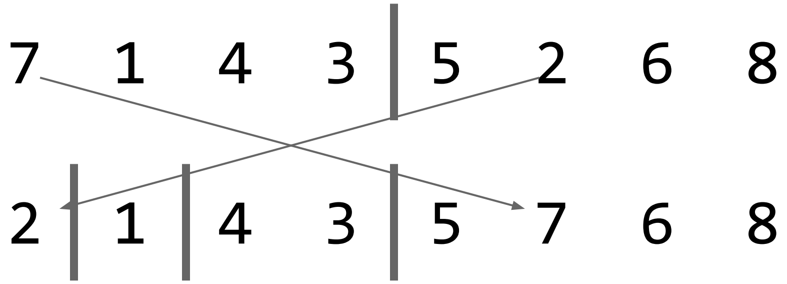
QuickSort: Idealized Version



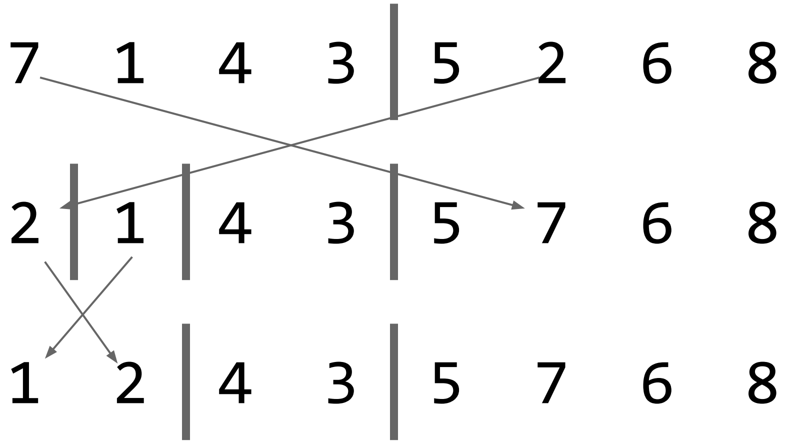
QuickSort: Idealized Version



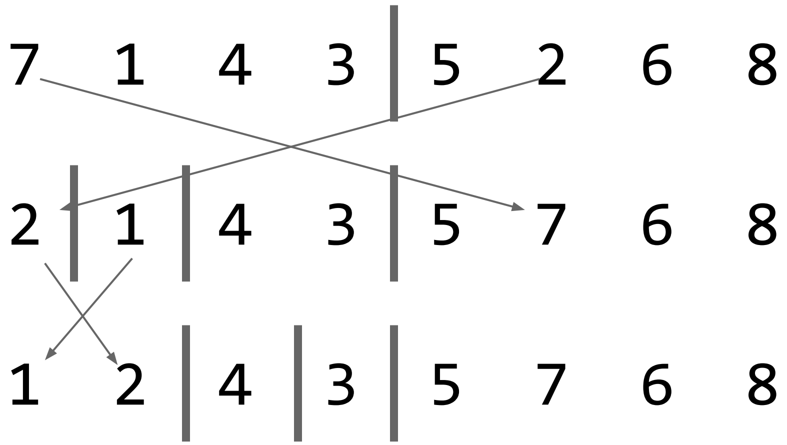
QuickSort: Idealized Version



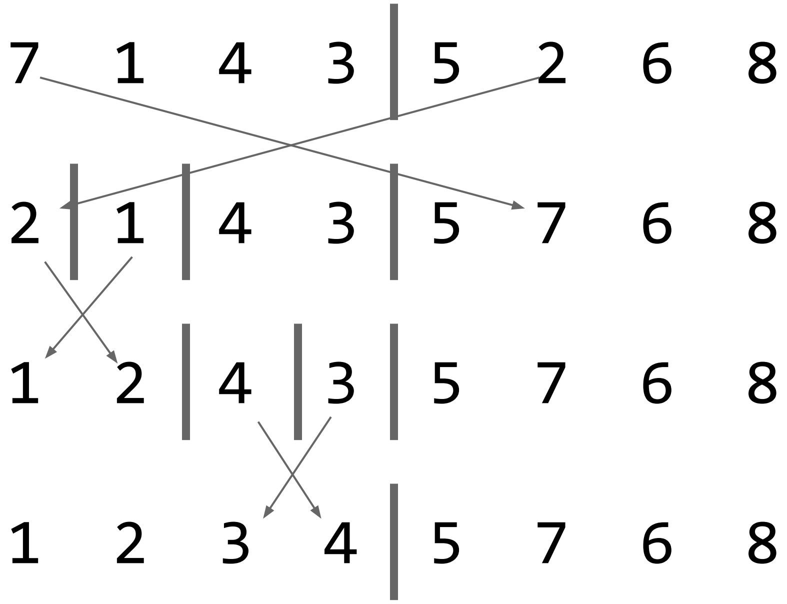
QuickSort: Idealized Version



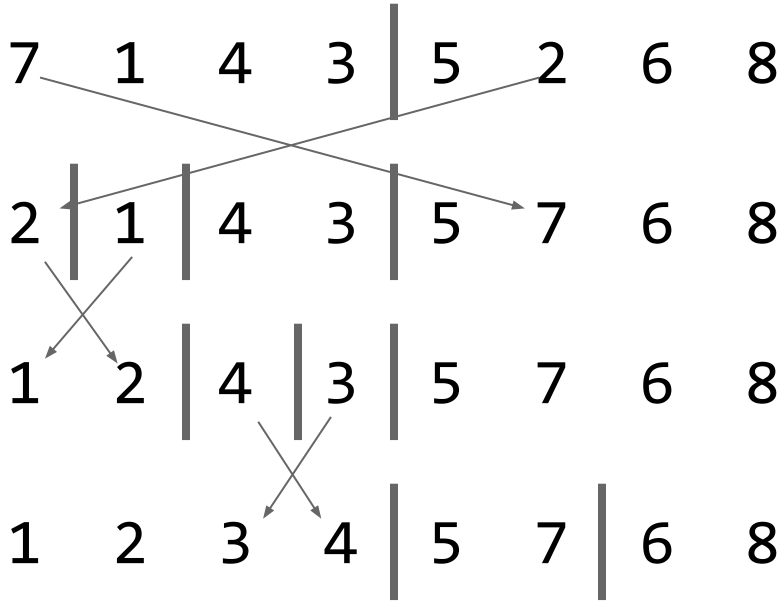
QuickSort: Idealized Version



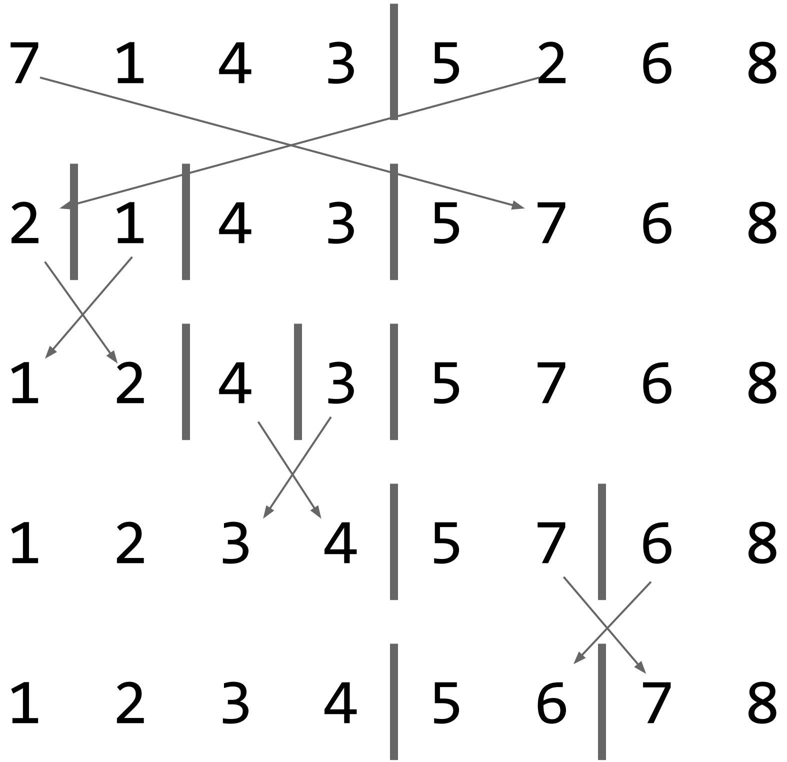
QuickSort: Idealized Version



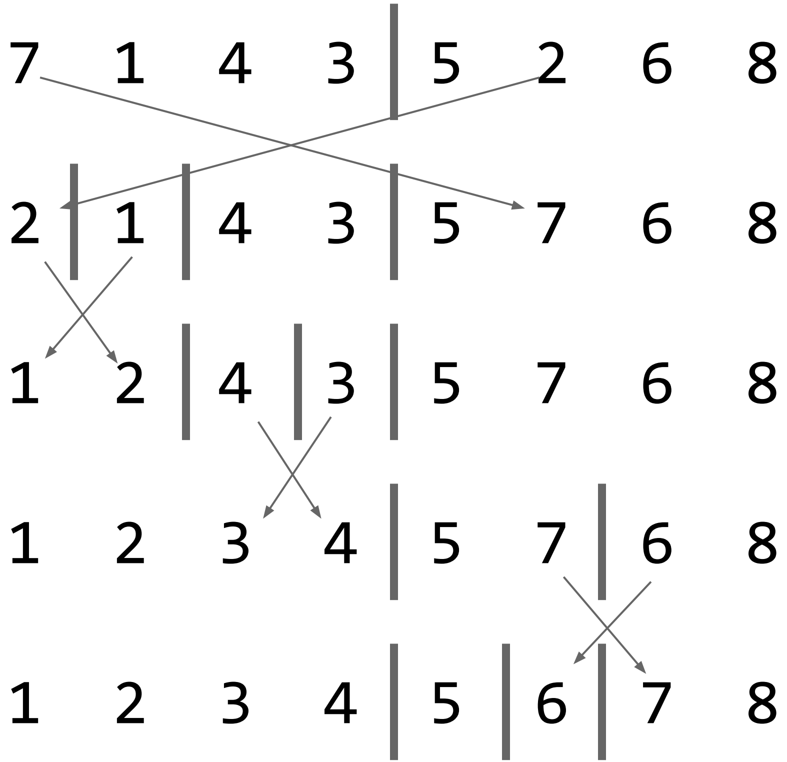
QuickSort: Idealized Version



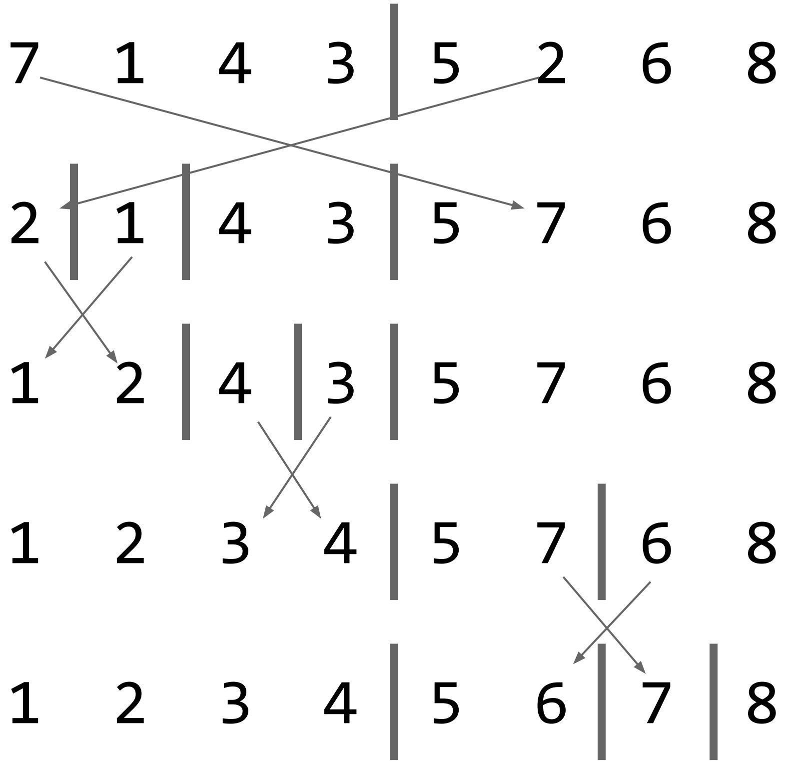
QuickSort: Idealized Version



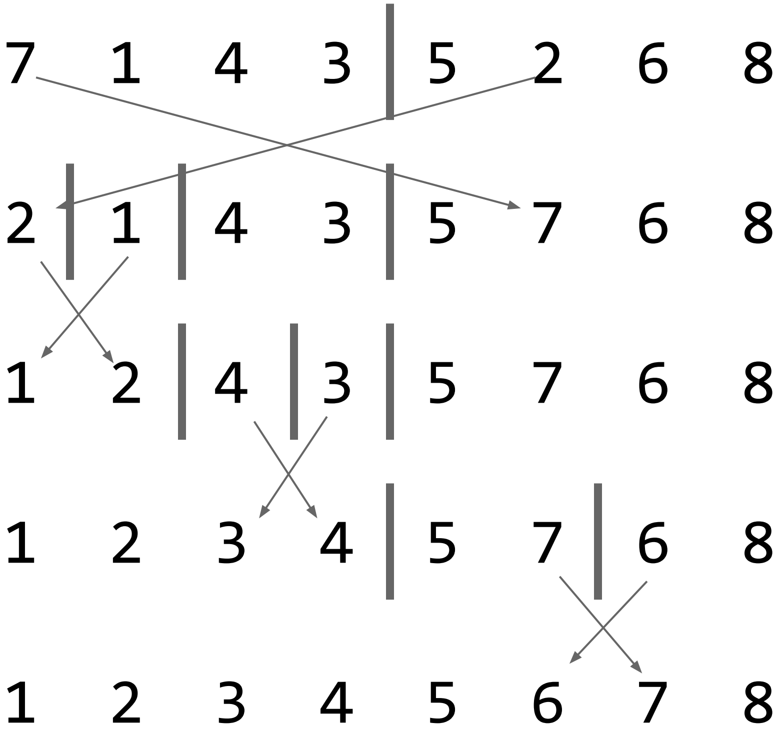
QuickSort: Idealized Version



QuickSort: Idealized Version



QuickSort: Idealized Version



QuickSort: Idealized Algorithm

To sort an array of size *n*:

1. Pick a *pivot* value (median?)
2. Swap values until:
   a. elements at [1, *n*/2) are ≤ pivot
   b. elements at [*n*/2, *n*) are > pivot
3. Recursively sort the lower half
4. Recursively sort the upper half

Great! So…how do we find the median…?

Great! So…how do we find the median…?

Finding the median takes O(n log(n)) for an unsorted array :(

QuickSort: Hypothetical

Imagine a world where we can obtain a pivot in $O(1)$.

Now what is our complexity?

QuickSort: Hypothetical

Imagine a world where we can obtain a pivot in *O*(1).

Now what is our complexity?

$$T_{quicksort}(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) + 0 & \textbf{otherwise} \end{cases}$$

QuickSort: Hypothetical

Imagine a world where we can obtain a pivot in *O*(1).

Now what is our complexity?

$$T_{quicksort}(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) + 0 & \textbf{otherwise} \end{cases}$$

Compare to Merge Sort:

$$T_{mergesort}(n) = \begin{cases} \Theta(1) & \textbf{if } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(1) + \Theta(n) & \textbf{otherwise} \end{cases}$$

QuickSort: Attempt #2

So how can we pick a pivot value (in $O(1)$ time)?

QuickSort: Attempt #2

So how can we pick a pivot value (in O(1) time)?

Idea: Pick it randomly! On average, half the values will be lower.

QuickSort: Attempt #2

To sort an array of size *n*:

1. Pick a value at random as the *pivot*
2. Swap values until the array is subdivided into:
    a. *low*: array elements < *pivot*
    b. *pivot*
    c. *high:* array elements > pivot
3. Recursively sort *low*
4. Recursively sort *high*

QuickSort: Runtime

What is the worst-case runtime?

QuickSort: Worst-Case Scenario

What if we always pick the worst pivot?

`[8,7,6,5,4,3,2,1]`

QuickSort: Worst-Case Scenario

What if we always pick the worst pivot?

`[8,7,6,5,4,3,2,1]`

`[7,6,5,4,3,2,1],8,[]`

QuickSort: Worst-Case Scenario

What if we always pick the worst pivot?

`[8,7,6,5,4,3,2,1]`

`[7,6,5,4,3,2,1],8,[]`

`[6,5,4,3,2,1],7,[],8`

QuickSort: Worst-Case Scenario

What if we always pick the worst pivot?

```
[8,7,6,5,4,3,2,1]

[7,6,5,4,3,2,1],8,[]

[6,5,4,3,2,1],7,[],8

[5,4,3,2,1],6,[],7,8
```

QuickSort: Worst-Case Scenario

What if we always pick the worst pivot?

```
[8,7,6,5,4,3,2,1]

[7,6,5,4,3,2,1],8,[]

[6,5,4,3,2,1],7,[],8

[5,4,3,2,1],6,[],7,8

...
```

QuickSort: Worst-Case Runtime

What is the worst-case runtime?

QuickSort: Worst-Case Runtime

What is the worst-case runtime?

$$T_{quicksort}(n) \in O(n^2)$$

QuickSort: Worst-Case Runtime

What is the worst-case runtime?

$$T_{quicksort}(n) \in O(n^2)$$

Remember: This is called the unqualified runtime…we don't take any extra context into account

QuickSort: Worst-Case Runtime

Is the worst case runtime representative?

QuickSort: Worst-Case Runtime

Is the worst case runtime representative?

No! (the actual runtime will almost always be faster)

QuickSort: Worst-Case Runtime

Is the worst case runtime representative?

No! (the actual runtime will almost always be faster)

But what **can** we say about runtime?

QuickSort

Let's say we pick Xth largest element for our pivot.

What is the runtime ($T(n)$)?

QuickSort

Let's say we pick Xth largest element for our pivot.

What is the runtime ($T(n)$)?

$$\begin{cases} T(0) + T(n-1) + \Theta(n) & \textbf{if } X = 1 \\ T(1) + T(n-2) + \Theta(n) & \textbf{if } X = 2 \\ T(2) + T(n-3) + \Theta(n) & \textbf{if } X = 3 \\ .. & \\ T(n-2) + T(1) + \Theta(n) & \textbf{if } X = n-1 \\ T(n-1) + T(0) + \Theta(n) & \textbf{if } X = n \end{cases}$$

Probabilities

How likely are we to pick X = k for any specific k?

Probabilities

How likely are we to pick X = k for any specific k?

P[X = k] = 1/n

Probability Theory (Great Class...)

If I roll a d6 (6-sided die) x times,

what is the average roll over all possible outcomes?

A single die roll

If I rolled a d6 1 time…

Roll	Probability	Outcome
⚀	1/6	1
⚁	1/6	2
⚂	1/6	3
⚃	1/6	4
⚄	1/6	5
⚅	1/6	6

Expected Value

The **Expected Value** of a random variable (ie the number rolled on the d6) is the sum of all outcomes times the probability of that outcome

$$\sum_i Probability_i \cdot Contribution_i$$

Expected Value

The **Expected Value** of a random variable (ie the number rolled on the d6) is the sum of all outcomes times the probability of that outcome

$$\sum_{i=1}^{6} \frac{1}{6} i = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = 3.5$$

Expected Value

The **Expected Value** of a random variable (ie the number rolled on the d6) is the sum of all outcomes times the probability of that outcome

$$\sum_{i=1}^{6} \frac{1}{6} i = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = 3.5$$

We refer to the expected value of a random variable as ***E[X]***

Independent Events

If we roll a d6 twice, does one roll affect the other?

Independent Events

If we roll a d6 twice, does one roll affect the other?

No. They are independent events.

Independent Events

If we roll a d6 twice, does one roll affect the other?

No. They are independent events.

If ***X*** and ***Y*** are independent then:

$$E[X+Y] = E[X] + E[Y]$$

Independent Events

If we roll a d6 twice, does one roll affect the other?

No. They are independent events.

If ***X*** and ***Y*** are independent then:

$$E[X+Y] = E[X] + E[Y]$$

If ***X*** and ***Y*** are our dice rolls, then $E[X+Y] = E[X] + E[Y] = 3.5 + 3.5 = 7$

QuickSort Runtime

Now we can write our runtime function in terms of random variables:

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ T(0) + T(n-1) + \Theta(n) & \textbf{if } n > 1 \wedge X = 1 \\ T(1) + T(n-2) + \Theta(n) & \textbf{if } n > 1 \wedge X = 2 \\ T(2) + T(n-3) + \Theta(n) & \textbf{if } n > 1 \wedge X = 3 \\ .. & \\ T(n-2) + T(1) + \Theta(n) & \textbf{if } n > 1 \wedge X = n-1 \\ T(n-1) + T(0) + \Theta(n) & \textbf{if } n > 1 \wedge X = n \end{cases}$$

QuickSort Runtime

…and convert it to the expected runtime over the variable ***X***

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ E[T(X-1) + T(n-X)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

QuickSort Runtime

…and convert it to the expected runtime over the variable ***X***

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ E[T(X-1)] + E[T(n-X)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

QuickSort Runtime

…and convert it to the expected runtime over the variable ***X***

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ E[T(X-1)] + E[T(n-X)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

This looks like the runtime of MergeSort, so now our hypothesis is that our **Expected Runtime** is ***n* log(*n*)**

Back to Induction

Hypothesis: $E[T(n)] \in O(n \log(n))$

Base Case

Base Case: $E[T(1)] \leq c\ (1 \log(1))$

Base Case

Base Case: $E[T(1)] \le c\ (1 \log(1))$

$E[T(1)] \le c\ (1 \cdot 0)$

Base Case

Base Case: $E[T(1)] \leq c\ (1 \log(1))$

$E[T(1)] \leq c\ (1 \cdot 0)$

$E[T(1)] \nleq 0$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\ (2 \log(2))$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\,(2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

$$T(0) + T(1) + 2c_1 \leq 2c$$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\,(2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

$$T(0) + T(1) + 2c_1 \leq 2c$$

$$2c_0 + 2c_1 \leq 2c$$

Base Case (Take 2)

Base Case (Take Two): $E[T(2)] \leq c\ (2 \log(2))$

$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$

$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$

$T(0) + T(1) + 2c_1 \leq 2c$

$2c_0 + 2c_1 \leq 2c$

True for any $c \geq c_0 + c_1$

Inductive Case

Assume: $E[T(n')] \leq c\,(n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\,(n \log(n))$

Inductive Case

Assume: $E[T(n')] \leq c\,(n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\,(n \log(n))$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

Inductive Case

Assume: $E[T(n')] \leq c\,(n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\,(n \log(n))$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} ci\log(i)\right) + c_1 \leq cn\log(n)$$

Inductive Case

Assume: $E[T(n')] \leq c\ (n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\ (n \log(n))$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} ci\log(i)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

$$cn\log(n) - c\log(n) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

$$cn\log(n) - c\log(n) + c_1 \leq cn\log(n)$$

$$c_1 \leq c\log(n)$$

QuickSort

So…is QuickSort $O(n \log(n))$...?

No!

What guarantees do you get?

If $f(n)$ is a Tight Bound

The algorithm always runs in $cf(n)$ steps

If $f(n)$ is a Worst-Case Bound

The algorithm always runs in at most $cf(n)$

If $f(n)$ is an Amortized Worst-Case Bound

n invocations of the algorithm **always** run in $cnf(n)$ steps

If $f(n)$ is an Average Bound

…we don't have any guarantees

What guarantees do you get?

If $f(n)$ is a Tight Bound
The algorithm always runs in $cf(n)$ steps

← Unqualified runtime

If $f(n)$ is a Worst-Case Bound
The algorithm always runs in at most $cf(n)$

If $f(n)$ is an Amortized Worst-Case Bound
n invocations of the algorithm **always** run in $cnf(n)$ steps

If $f(n)$ is an Average Bound
…we don't have any guarantees