

Advanced Data Structure and Algorithm

Randomized algorithms and QuickSort

Randomized algorithms

- We make some random choices during the algorithm.
- We hope the algorithm works.
- We hope the algorithm is fast.

For today we will look at algorithms that always work and are probably fast.



Today



- How do we analyze randomized algorithms?
- An randomized algorithms for sorting.
 - QuickSort

How do we measure the runtime of a randomized algorithm?

Scenario 1

1. You publish your algorithm.
2. Bad guy picks the input.
3. You run your randomized algorithm.




- In **Scenario 1**, the running time is a **random variable**.
 - It makes sense to talk about **expected running time**.
- In **Scenario 2**, the running time is **not random**.
 - We call this the **worst-case running time** of the randomized algorithm.

Scenario 2

1. You publish your algorithm.
2. Bad guy picks the input.
3. Bad guy chooses the randomness (fixes the dice) and runs your algorithm.



Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
 - QuickSort 

QuickSort

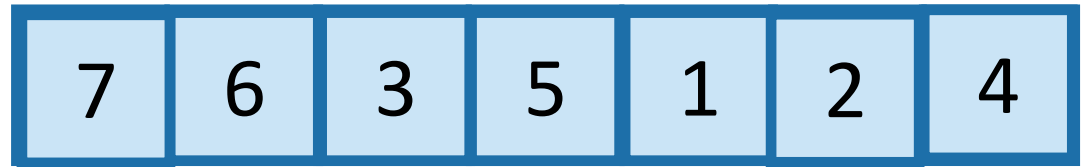
- Expected runtime $O(n \log(n))$.
- Worst-case runtime $O(n^2)$.
- In practice works great!
 - (More later)

Quicksort

We want to sort this array.

For the rest of the lecture, assume all elements of A are distinct.

First, pick a “pivot.”
Do it at random.



Next, partition the array into “bigger than 5” or “less than 5”

random pivot!

This PARTITION step takes time $O(n)$.
(Notice that we don't sort each half).

Arrange them like so:

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:



PseudoPseudoCode for what we just saw

- QuickSort(A):
 - **If** $\text{len}(A) \leq 1$:
 - **return**
 - Pick some $x = A[i]$ at random. Call this the **pivot**.
 - **PARTITION** the rest of A into:
 - L (less than x) and
 - R (greater than x)
 - Replace A with [L, x, R] (that is, rearrange A in this order)
 - QuickSort(L)
 - QuickSort(R)

Assume that all elements
of A are distinct. How
would you change this if
that's not the case?



How would you do all this in-place?
Without hurting the running time?



Running time?

- $T(n) = T(|L|) + T(|R|) + O(n)$
- In an ideal world...
 - if the pivot splits the array exactly in half...

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

- We've seen that a bunch:

$$T(n) = O(n \log(n)).$$



Red flag

- **Slow** Sort(A):
 - If $\text{len}(A) \leq 1$:
 - return

We can use the same argument to prove something false.

- **Pick the pivot x to be either max(A) or min(A), randomly**
 - \\ We can find the max and min in $O(n)$ time

- PARTITION the rest of A into:

- L (less than x) and
- R (greater than x)

- Replace A with [L, x, R] (that is, rearrange A in this order)

- **Slow** Sort(L)

- **Slow** Sort(R)

- Same recurrence relation:

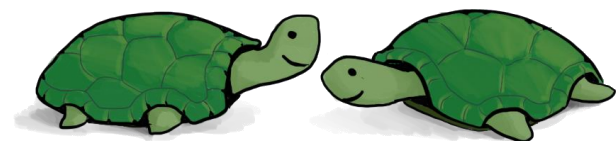
$$T(n) = T(|L|) + T(|R|) + O(n)$$

- We still have $E[|L|] = E[|R|] = \frac{n-1}{2}$
- But now, one of |L| or |R| is always $n-1$.
- You check: Running time is $\Theta(n^2)$, with probability 1.

The expected running time of SlowSort is $O(n \log(n))$.

Proof:^{*}

What's wrong???



- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

***Disclaimer: this proof is wrong.**

What's wrong?

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
 - The expected number of items on each side of the pivot is half of the things.
- If that occurs, the running time is $T(n) = O(n \log(n))$.
 - Since the relevant recurrence relation is $T(n) = 2T\left(\frac{n-1}{2}\right) + O(n)$
- Therefore, the expected running time is $O(n \log(n))$.

This argument says:

$$T(n) = \text{some function of } |L| \text{ and } |R| \quad \checkmark$$

$$E[T(n)] = E[\text{some function of } |L| \text{ and } |R|] \quad \checkmark$$

$$E[T(n)] = \text{some function of } E|L| \text{ and } E|R| \quad \times$$

***That's not how
expectations work!***



Instead

- We'll have to think a little harder about how the algorithm works.

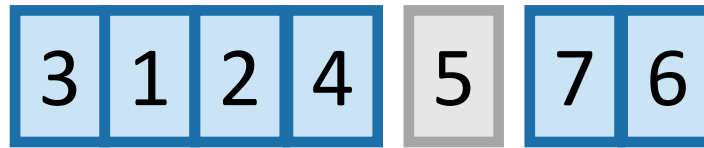
Next goal:

- Get the same conclusion, correctly!

Example of recursive calls

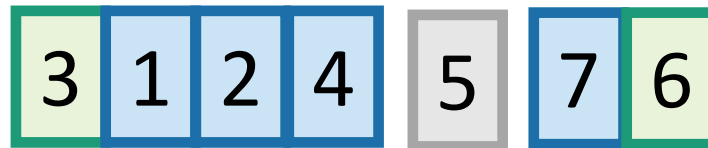


Pick 5 as a pivot



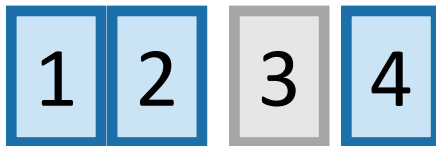
Partition on either side of 5

Recurse on [3142]
and pick 3 as a pivot.



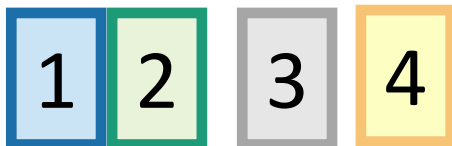
Recurse on [76] and
pick 6 as a pivot.

Partition
around 3.

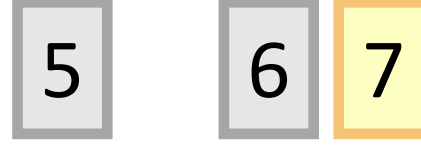


Partition on
either side of 6

Recurse on
[12] and
pick 2 as a
pivot.

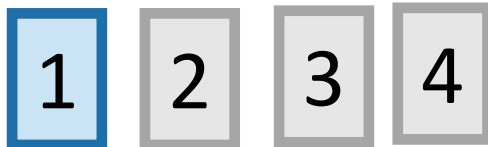


Recurse on
[4] (done).

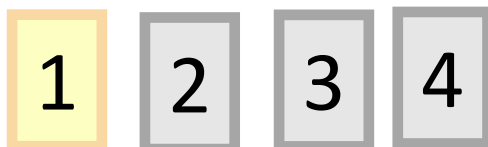


Recurse on [7], it has
size 1 so we're done.

partition
around 2.

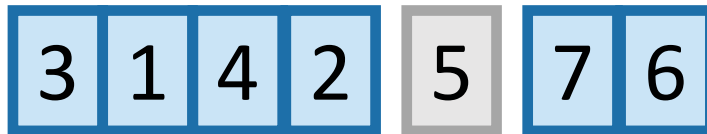


Recurse on
[1] (done).



How long does this take to run?

- We will count the number of **comparisons** that the algorithm does.
 - This turns out to give us a good idea of the runtime. (Not obvious).
- How many times are any two items compared?



In the example before, everything was compared to 5 once in the first step....and never again.



But not everything was compared to 3.
5 was, and so were 1,2 and 4.
But not 6 or 7.

Each pair of items is compared either 0 or 1 times. Which is it?

7	6	3	5	1	2	4
---	---	---	---	---	---	---

Let's assume that the numbers in the array are actually the numbers 1,...,n

Of course this doesn't have to be the case! It's a good exercise to convince yourself that the analysis will still go through without this assumption.



- **Whether or not a,b are compared** is a random variable, that depends on the choice of pivots. Let's say

$$X_{a,b} = \begin{cases} 1 & \text{if } a \text{ and } b \text{ are ever compared} \\ 0 & \text{if } a \text{ and } b \text{ are never compared} \end{cases}$$

- In the previous example $X_{1,5} = 1$, because item 1 and item 5 were compared.
- But $X_{3,6} = 0$, because item 3 and item 6 were NOT compared.

Counting comparisons

- The number of comparisons total during the algorithm is

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b}$$

- The expected number of comparisons is

$$E \left[\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b} \right] = \sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}]$$

using linearity of expectations.

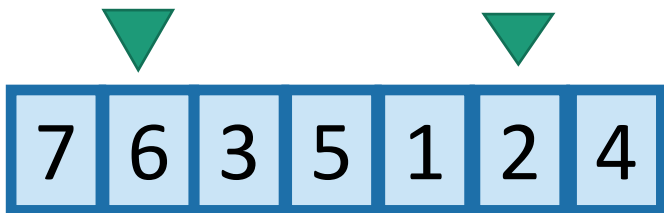
Counting comparisons

expected number of comparisons:

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}]$$

- So we just need to figure out $E[X_{a,b}]$
- $E[X_{a,b}] = P(X_{a,b} = 1) \cdot 1 + P(X_{a,b} = 0) \cdot 0 = P(X_{a,b} = 1)$
 - (using definition of expectation)
- So we need to figure out:

$P(X_{a,b} = 1)$ = the probability that a and b are ever compared.



Say that $a = 2$ and $b = 6$. What is the probability that 2 and 6 are ever compared?



This is exactly the probability that either 2 or 6 is first picked to be a pivot out of the highlighted entries.



If, say, 5 were picked first, then 2 and 6 would be separated and never see each other again.

Counting comparisons

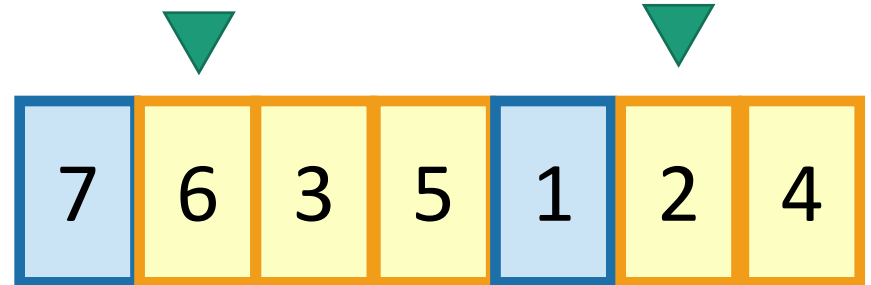
$$P(X_{a,b} = 1)$$

= probability a,b are ever compared

= probability that one of a,b are picked first out of all of the $b - a + 1$ numbers between them.

2 choices out of $b-a+1$...

$$= \frac{2}{b - a + 1}$$



Aside:

Why don't we care about 1 and 7?

In a bit more detail:

- Let $S = \{a, a+1, \dots, b\}$
- $P\{a, b \text{ are ever compared}\}$
 $= \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S \mid \text{stuff}\} \cdot P\{\text{stuff}\}$

where the sum is over all the stuff that does not involve S .

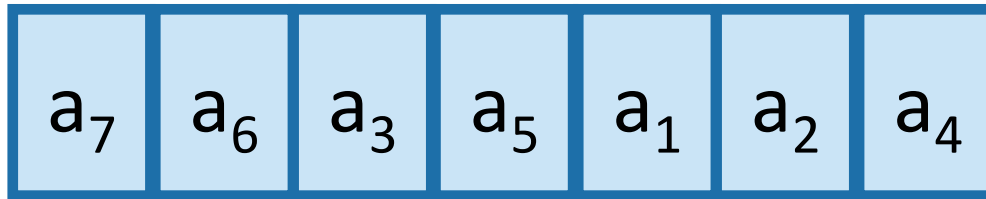
- But since that stuff is independent of what happens with S , this is equal to:

$$\begin{aligned} &= \sum_{\text{stuff}} P\{a \text{ or } b \text{ picked first out of } S\} \cdot P\{\text{stuff}\} \\ &= P\{a \text{ or } b \text{ picked first out of } S\} \cdot \sum_{\text{stuff}} P\{\text{stuff}\} \\ &= P\{a \text{ or } b \text{ picked first out of } S\} \\ &= 2/|S| \end{aligned}$$

Aside:

Why can we assume that the elements of the array are $\{1, 2, \dots, n\}$?

- More generally, say the elements of the array are $a_1 < a_2 < \dots < a_n$, so the array looks like:



- Then we'd do exactly the same thing, except we'd focus on the subscripts instead of the values. For example, the probability that a_2 and a_6 are ever compared is the probability that a_2 or a_6 are picked as a pivot before a_3, a_4 , or a_5 are.

All together now...

Expected number of comparisons

- $E\left[\sum_{a=1}^{n-1} \sum_{b=a+1}^n X_{a,b}\right]$ This is the expected number of comparisons throughout the algorithm
- $= \sum_{a=1}^{n-1} \sum_{b=a+1}^n E[X_{a,b}]$ linearity of expectation
- $= \sum_{a=1}^{n-1} \sum_{b=a+1}^n P(X_{a,b} = 1)$ definition of expectation
- $= \sum_{a=1}^{n-1} \sum_{b=a+1}^n \frac{2}{b-a+1}$ the reasoning we just did

- This is a big nasty sum, but we can do it.
- We get that this is less than $2n \ln(n)$.

Do this sum!



Almost done

- We saw that $E[\text{number of comparisons}] = O(n \log(n))$
- Is that the same as $E[\text{running time}]$?
- In this case, **yes**.
- We need to argue that the running time is dominated by the time to do comparisons.
- QuickSort(A):
 - If $\text{len}(A) \leq 1$:
 - **return**
 - Pick some $x = A[i]$ at random. Call this the **pivot**.
 - **PARTITION** the rest of A into:
 - L (less than x) and
 - R (greater than x)
 - Replace A with [L, x, R] (that is, rearrange A in this order)
 - QuickSort(L)
 - QuickSort(R)

What have we learned?

- The expected running time of QuickSort is $O(n \log(n))$

Worst-case running time

- Suppose that an adversary is choosing the “random” pivots for you.
- Then the running time might be $O(n^2)$
 - Eg, they’d choose to implement SlowSort
 - In practice, this doesn’t usually happen.



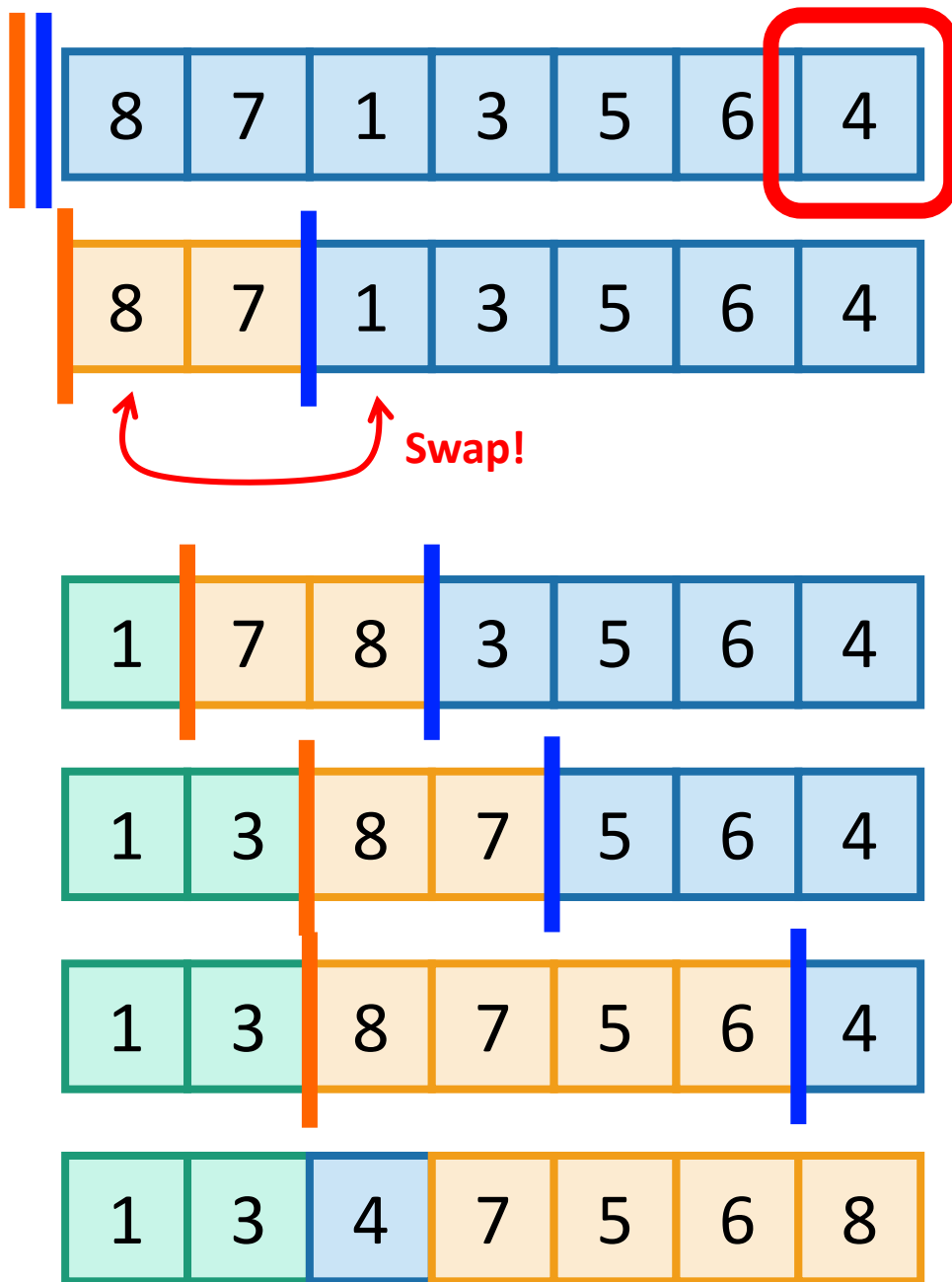
A note on implementation

- Our pseudocode is easy to understand and analyze, but is not a good way to implement this algorithm.

- QuickSort(A):
 - If $\text{len}(A) \leq 1$:
 - **return**
 - Pick some $x = A[i]$ at random. Call this the **pivot**.
 - **PARTITION** the rest of A into:
 - L (less than x) and
 - R (greater than x)
 - Replace A with [L, x, R] (that is, rearrange A in this order)
 - QuickSort(L)
 - QuickSort(R)



- Instead, implement it **in-place** (without separate L and R)

A better way to do Partition




Pivot

Choose it randomly, then swap it with the last one, so it's at the end.

Initialize  and 

Step  forward.

When  sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

QuickSort vs MergeSort

*What if you want $O(n \log(n))$ worst-case runtime and stability? Check out “Block Sort” on Wikipedia!

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	<ul style="list-style-type: none">Worst-case: $O(n^2)$Expected: $O(n \log(n))$	Worst-case: $O(n \log(n))$
Used by	<ul style="list-style-type: none">Java for primitive typesC qsortUnixg++	<ul style="list-style-type: none">Java for objectsPerl
In-Place? (With $O(\log(n))$ extra memory)	Yes, pretty easily	Not easily* if you want to maintain both stability and runtime. (But pretty easily if you can sacrifice runtime).
Stable?	No	Yes
Other Pros	Good cache locality if implemented for arrays	Merge step is really efficient with linked lists

Understand this

These are just for fun.
(Not on exam).

Today

- How do we analyze randomized algorithms?
- A few randomized algorithms for sorting.
 - **BogoSort**
 - **QuickSort**
- **BogoSort** is a pedagogical tool.
- **QuickSort** is important to know. (in contrast with BogoSort...)

Recap

- How do we measure the runtime of a **randomized algorithm**?

- Expected runtime
- Worst-case runtime



- **QuickSort** (with a random pivot) is a randomized sorting algorithm.
 - In many situations, QuickSort is nicer than MergeSort.
 - In many situations, MergeSort is nicer than QuickSort.