CSE100: Design and Analysis of Algorithms
Lecture 11 – Randomized Algorithms

Feb 22nd 2022

Randomized Algorithms,QuickSort

Randomized algorithms (review)

- 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.

e.g., **`Select`** with a random pivot is a randomized algorithm.

- Always works (aka, is correct).
- Probably fast.

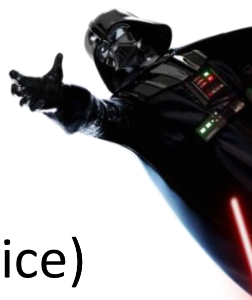
How do we measure the runtime of a randomized algorithm? (review)

Scenario 1

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



- 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.

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…)

Assume A has distinct entries

BogoSort (review)

Suppose that you can draw a random integer in {1,…,n} in time O(1). How would you randomly permute an array in-place in time O(n)?

Ollie the over-achieving ostrich

- **BogoSort**(A)
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

- Let $X_i = \begin{cases} 1 \text{ if A is sorted after iteration i} \\ 0 \text{ otherwise} \end{cases}$
- $E[X_i] = \frac{1}{n!}$
- $E[\text{number of iterations until A is sorted}] = n!$

MATH 32 Refresher

1. Let X be a random variable which is 1 with probability 1/100 and 0 with probability 99/100.
   a) E[X] = 1/100
   b) If $X_1, X_2, \dots X_n$ are iid copies of X, by linearity of expectation,

$$E\left[\sum_{i=1}^{n} X\_i\right] = \sum_{i=1}^{n} E[X_i] = \frac{n}{100}$$

   c) Let N be the index of the first 1. Then E[N] = 100.

iid: independent and identically distributed

To see part (c), either:

- You saw in MATH 32 that N is a geometric random variable, and you know a formula for that.
- Suppose you do the first trial. If it comes up 1 (with probability 1/100), then N=1. Otherwise, you start again except you've already used one trial. Thus:

$$E[N] = \frac{1}{100} \cdot 1 + \left(1 - \frac{1}{100}\right) \cdot (1 + E[N]) = 1 + \left(1 - \frac{1}{100}\right) E[N]$$

  Solving for E[N] we see E[N] = 100.
- (There are other derivations too).

MATH 32 Refresher 2

2. Let X_i be 1 iff A is sorted on iteration i.
   a) $E[X_i] = 1/n!$ since there are n! possible orderings of A and only one is sorted. (Suppose A has distinct entries).
   b) Let N be the index of the first 1. Then E[N] = n!.

Part (b) is similar to part (c) in previous slide:

- You saw in MATH 32 that N is a geometric random variable, and you know a formula for that. Or,
- Suppose you do the first trial. If it comes up 1 (with probability 1/n!), then N=1. Otherwise, you start again except you've already used one trial. Thus:

$$E[N] = \frac{1}{n!} \cdot 1 + \left(1 - \frac{1}{n!}\right) \cdot (1 + E[N]) = 1 + \left(1 - \frac{1}{n!}\right) E[N]$$

Solving for E[N] we see E[N] = n!

- (There are other derivations too).

Assume A has distinct entries

From your MATH 32 refresher exercise: BogoSort

Suppose that you can draw a random integer in {1,…,n} in time O(1). How would you randomly permute an array in-place in time O(n)?

Ollie the over-achieving ostrich

- **BogoSort**(A)
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

- Let $X_i = \begin{cases} 1 \text{ if A is sorted after iteration i} \\ 0 \text{ otherwise} \end{cases}$
- $E[X_i] = \frac{1}{n!}$
- $E[\text{number of iterations until A is sorted}] = n!$

Expected Running time of BogoSort

This isn't random, so we can pull it out of the expectation.

E[running time on a list of length n]

= E[(number of iterations) * (time per iteration)]

= (time per iteration) * E[number of iterations]

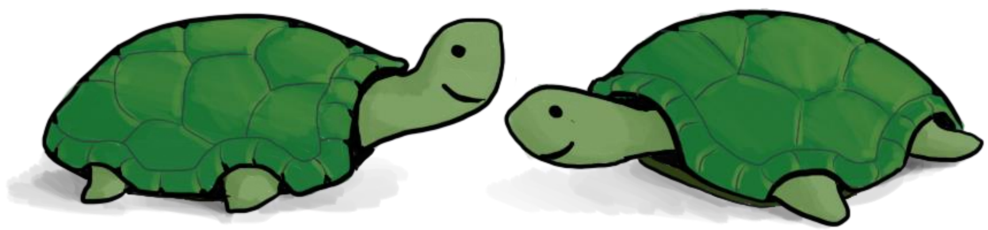
This is O(n) (to permute and then check if sorted)

We just computed this. It's n!.

$= O(n \cdot n!)$

= REALLY REALLY BIG.

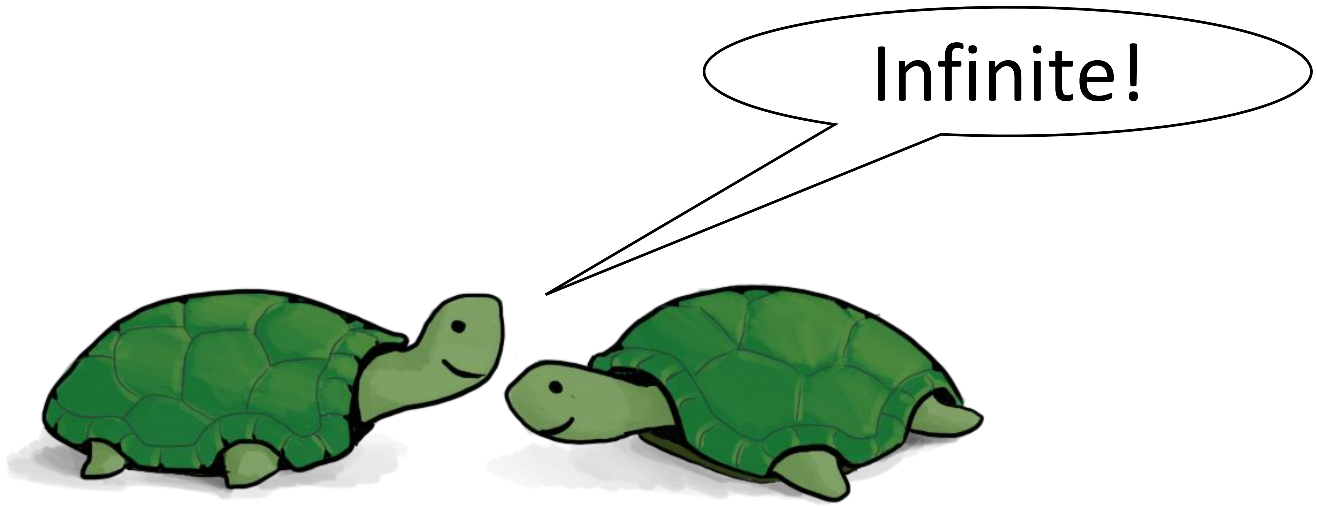
Worst-case running time of BogoSort?



Think-Pair-Share Terrapins!

- **BogoSort**(A)
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

Worst-case running time of BogoSort?



Think-Pair-Share Terrapins!

- **BogoSort**(A)
  - **While** true:
    - Randomly permute A.
    - Check if A is sorted.
    - **If** A is sorted, **return** A.

What have we learned?

- Expected running time:
  1. You publish your randomized algorithm.
  2. Bad guy picks an input.
  3. You get to roll the dice.

- Worst-case running time:
  1. You publish your randomized algorithm.
  2. Bad guy picks an input.
  3. Bad guy gets to “roll” the dice.

- Don’t use bogoSort.

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…)

a better randomized algorithm: QuickSort

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

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

Quicksort

We want to sort this array.

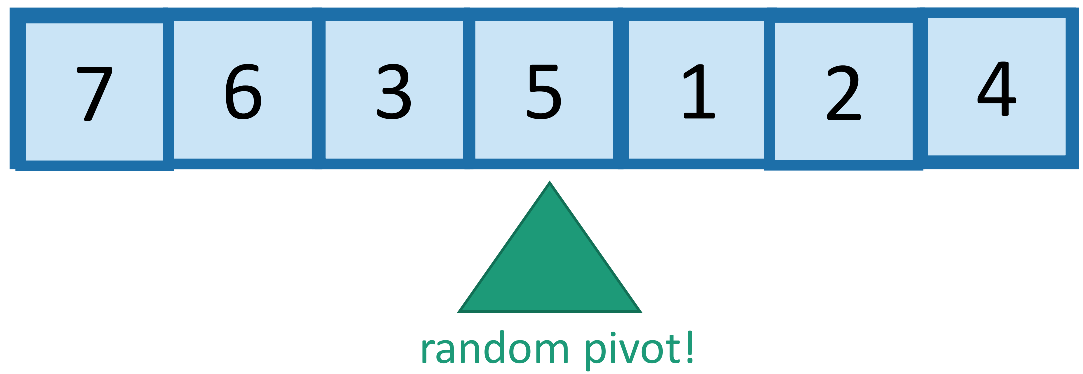
7	6	3	5	1	2	4

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

Quicksort

We want to sort this array.

First, pick a "pivot."
Do it at random.

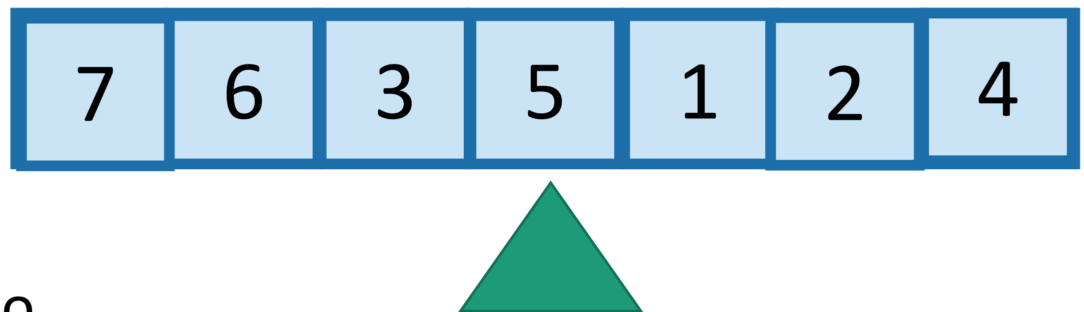


For the rest of the lecture, assume all elements of A are distinct.

Quicksort

We want to sort this array.

First, pick a "pivot."
Do it at random.



random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

Quicksort

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

We want to sort this array.

7	6	3	5	1	2	4

First, pick a "pivot."
Do it at random.

random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

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

Quicksort

We want to sort this array.

7	6	3	5	1	2	4

First, pick a "pivot."

Do it at random.

random pivot!

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Next, partition the array into "bigger than 5" or "less than 5"

3	1	2	4

L = array with things smaller than A[pivot]

7	6

R = array with things larger than A[pivot]

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

Quicksort

We want to sort this array.

7	6	3	5	1	2	4

First, pick a "pivot."
Do it at random.

random pivot!

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Next, partition the array into "bigger than 5" or "less than 5"

Arrange them like so:

3 1 2 4

L = array with things smaller than A[pivot]

7 6

R = array with things larger than A[pivot]

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.

7	6	3	5	1	2	4

random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Arrange them like so:

3 1 2 4   5   7 6

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Quicksort

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

We want to sort this array.

First, pick a "pivot."
Do it at random.

7	6	3	5	1	2	4

random pivot!

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Next, partition the array into "bigger than 5" or "less than 5"

Arrange them like so:

3 1 2 4 5 7 6

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:

Quicksort

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

We want to sort this array.

First, pick a "pivot."
Do it at random.

7 6 3 5 1 2 4

random pivot!

Next, partition the array into "bigger than 5" or "less than 5"

This PARTITION step takes time O(n). (Notice that we don't sort each half). [same as in SELECT]

Arrange them like so:

3 1 2 4   5   7 6

L = array with things smaller than A[pivot]

R = array with things larger than A[pivot]

Recurse on L and R:

1 2 3 4   5   6 7

PseudoPseudoCode for what we just saw

Lab 05 asks for an implementation of this algorithm.

- QuickSort(A):
  - **If** len(A) <= 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? (We'll see later…)

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:

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

The expected running time of QuickSort is O(nlog(n)).

Proof:*

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - The expected number of items on each side of the pivot is half of the things.

Remember, we are assuming all elements of A are distinct

Aside
why is $E[|L|] = \frac{n-1}{2}$?

- $E[|L|] = E[|R|]$
  - by symmetry
- $E[|L| + |R|] = n - 1$
  - because L and R make up everything except the pivot.
- $E[|L|] + E[|R|] = n - 1$
  - By linearity of expectation
- $2E[|L|] = n - 1$
  - Plugging in the first bullet point.
- $E[|L|] = \frac{n-1}{2}$
  - Solving for $E[|L|]$.

The expected running time of QuickSort is O(nlog(n)).

Proof:*

- $E[|L|] = E[|R|] = \frac{n-1}{2}$.
  - 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.**

Red flag

We can use the same argument to prove something false.

- **Slow** Sort(A):
  - **If** len(A) <= 1:
    - **return**
  - **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:

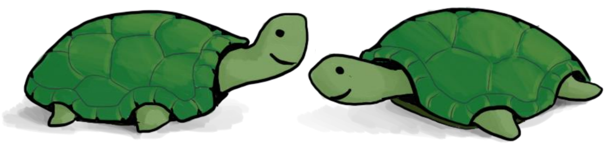
$$T(n) = T(|L|) + T(|R|) + O(n)$$

- 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(nlog(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:

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

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

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

That’s not how expectations work!

Plucky the Pedantic Penguin