CE203
ساختمان داده ها و الگوریتم ها

سجاد شیرعلی شهرضا

پاییز 1400

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

جلسه هشتم:
الگوریتم های تصادفی

دوشنبه، 26 مهر 1400

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 5
- تشکیل کلاس تدریسیاری در روز پنجشنبه 29 مهر (به جای روز چهار شنبه 28 مهر)
  - از ساعت 10:45 الی 12:15

خلاصه ای از آنچه گذشت ...

LINEAR SELECTION: THE BIG IDEA

Select a pivot: **Median of Medians**

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME

Select a pivot: **Median of (sub)Medians**

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME

O(n)
Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

Divide the original list into $\lceil n/5 \rceil$ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME

O(n)
Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

- Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)
- Find the **sub-median** of each small group (3rd smallest out of the 5)
- Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

T(n/5)
Recursive work: we call SELECT on an array of size n/5

Partition around pivot

Recurse!

LINEAR SELECTION: RUNTIME

O(n)

Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

Divide the original list into $\lceil n/5 \rceil$ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

T(n/5)

Recursive work: we call SELECT on an array of size n/5

Partition around pivot

Recurse!

T(7n/10)

Recursive work: we call SELECT on either L or R (size $\leq 7n/10$)

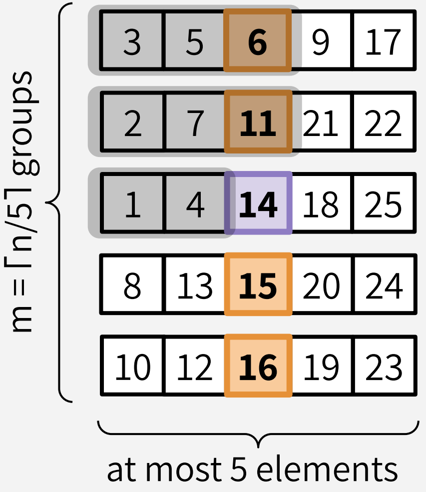
WAIT: WHERE DID WE GET 7n/10?

At the end of last lecture, we proved this claim:

$$3n/10 - 6 \le \texttt{len(L)} \le 7n/10 + 5$$

$$3n/10 - 6 \le \texttt{len(R)} \le 7n/10 + 5$$

this is because
`len(L)` + `len(R)` = n-1,
so if
3n/10 - 6 ≤ `len(L)`
then
`len(R)` **≤ 7n/10 + 5**



We asked ourselves:

<u>At least</u> how many elements are guaranteed to be <u>smaller</u> than the median of medians?

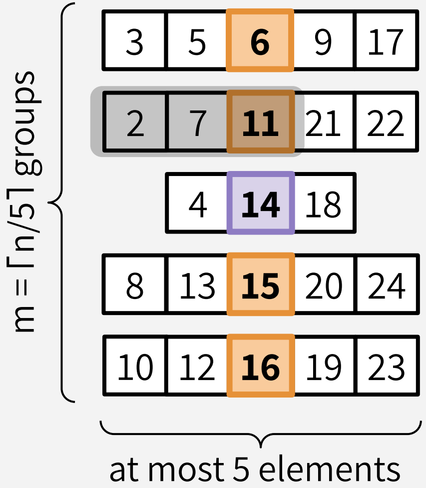
The shaded region denotes the only elements that are *guaranteed* to be smaller than **14** (the median of medians). We counted that up, took care of some off-by-one errors just to be safe (i.e. just to make sure we're underestimating), and we got **3n/10 - 6**!

(DETAILS IF YOU'RE CURIOUS)

At the end of last lecture, we proved this claim:

$$3n/10 - 6 \le \texttt{len(L)} \le 7n/10 + 5$$

$$3n/10 - 6 \le \texttt{len(R)} \le 7n/10 + 5$$



3 elements from each (non-leftover) group that has a **median** smaller than the **median of medians** + ~~2 elements from the group containing the **median of medians**~~

$$3 \cdot (\lceil m/2 \rceil - 1 - 1) \cancel{+ 2}$$

To exclude the group with the **median of medians**

To exclude any of those groups that might be a "leftover" group!

The group with the **median of medians** might be a "leftover" group! Might as well just get rid of the +2 to be safe

LINEAR SELECTION: RUNTIME

O(n)
Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Find the **median** of all the **sub-medians** (via recursive call to SELECT!!)

T(n/5)
Recursive work: we call SELECT on an array of size n/5

Partition around pivot

Recurse!

T(7n/10)
Recursive work: we call SELECT on either L or R (size ≤ ~7n/10)

LINEAR SELECTION: RUNTIME

O(n)
Non-recursive "shallow" work!

Select a pivot: **Median of (sub)Medians**

Divide the original list into ⌈n/5⌉ groups (each group has ≤ 5 elements)

Find the **sub-median** of each small group (3rd smallest out of the 5)

Fi

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

T(n/5)
Recursive work: we call SELECT on an array of size n/5

Partition around pivot

Recurse!

T(7n/10)
Recursive work: we call SELECT on either L or R (size ≤ 7n/10)

LINEAR SELECTION: RUNTIME

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

We also solved this recurrence using the Substitution Method!

$\Downarrow$

O(n)

Worst-case Runtime!

PSEUDOCODE & RUNTIME

```
SELECT(A,k):
    if len(A) == 1:
        return A[0]
    p = MEDIAN_OF_MEDIANS(A)
    L, R = PARTITION(A,p)
    if len(L) == k-1:
        return p
    else if len(L) > k-1:
        return SELECT(L, k)
    else if len(L) < k-1:
        return SELECT(R, k-len(L)-1)
```

O(n) work outside of recursive calls

(base case, set-up within MEDIAN_OF_MEDIANS, partitioning)

T(n/5) work hidden in this recursive call

(remember, MEDIAN_OF_MEDIANS calls SELECT on ⌈n/5⌉-size array)

T(7n/10) work hidden in this recursive call

7n/10 is the maximum size of either L or R (this is what the median-of-medians technique guarantees us)!

LINEAR SELECTION: THE BIG IDEA

Select a pivot: **Median of Medians**

Partition around pivot

Recurse!

Median of Medians is really cool! The math was a little detailed, but worth the time to digest so that you're 110% convinced that the technique does give a ~7n/10 bound on the max size of either L or R. Solving the recurrence can be done via Substitution Method. SELECT as a whole is an amazing display of Divide-and-Conquer!

سوال؟

الگوریتم های تصادفی

الگوریتم تصادفی چیست؟ چگونه میتوان آن را تحلیل کرد؟

WHAT IS A RANDOMIZED ALGORITHM?

- An algorithm that incorporates randomness as part of its operation.
- Basically, we'll make random choices during the algorithm:
  - Sometimes, we'll just hope that it works!
  - Other times, we'll just hope that our algorithm is fast!
- Let's formalize this...

LAS VEGAS vs. MONTE CARLO

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

LAS VEGAS vs. MONTE CARLO

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

MONTE CARLO ALGORITHMS

Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

LAS VEGAS vs. MONTE CARLO

LAS VEGAS ALGORITHMS

Guarantees correctness!

But the runtime is a random variable.
(i.e. there's a chance the runtime could take awhile)

We'll focus on these algorithms for now
(BogoSort, QuickSort, QuickSelect)

MONTE CARLO ALGORITHMS

Correctness is a random variable.
(i.e. there's a chance the output is wrong)

But the runtime is guaranteed!

We'll see some examples of these later!

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

WORST-CASE RUNNING TIME

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

WORST-CASE RUNNING TIME

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input,
then *you* run your randomized algorithm

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input,
then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

~~"Expected value over possible inputs"~~

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

~~"The worst possible input"~~

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

"Expected value over *dice outcomes*"

EXPECTED RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *you* run your randomized algorithm

The running time is a **random variable** (depends on the randomness that your algorithm employs), so we can reason about the ***expected running time***

In both cases, we are still thinking about the *WORST-CASE INPUT*

"The worst possible *dice outcomes*"

WORST-CASE RUNNING TIME

Scenario: you publish your algorithm and a bad guy picks the input, then *the bad guy chooses the randomness* ("fixes the dice") in your randomized algorithm

The running time is **not random** (we know how the bad guy will choose the randomness to make our algorithm suffer the most), so we can reason about the ***worst-case running time***

RUNTIME FOR RANDOMIZED ALGS

“Expected
val

EXPECTED RUNNING TIME

p

Don’t get confused!!!

Even with randomized algorithms, we are still considering the *WORST CASE INPUT*, regardless of whether we’re computing expected or worst-case runtime.

Expected runtime ***IS NOT*** runtime when given an expected input! We are taking the expectation over the random choices that our algorithm would make, ***NOT*** an expectation over the distribution of possible inputs.

make our algorithm suffer the most), so we can reason about the ***worst-case running time***

سوال؟

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}$, $\mathbf{X_2}$, …, $\mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^n X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^n X_i] = \sum_{i=1}^n \mathbb{E}[X_i] = \frac{n}{100}$

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

b. Suppose you draw *n* independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

c. Suppose I draw independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, and I stop when I see the first "**1**". Let N be the last index that we draw. What is the expected value of N?

QUICK PROBABILITY EXERCISE

X is a Bernoulli/indicator random variable which is **1** with probability 1/100 and **0** with prob. 99/100.

a. What is the expected value $\mathbb{E}[X]$?

$$\mathbb{E}[X] = 1\left(\frac{1}{100}\right) + 0\left(\frac{99}{100}\right) = \frac{1}{100}$$

b. Suppose you draw n independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, distributed like X. What is the expected value $\mathbb{E}[\sum_{i=1}^{n} X_i]$?

By linearity of expectation: $\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \frac{n}{100}$

c. Suppose I draw independent random variables $\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_n}$, and I stop when I see the first "**1**". Let N be the last index that we draw. What is the expected value of N?

N is a *geometric random variable* .
We can use the formula:

$$\mathbb{E}[N] = \frac{1}{p} = \frac{1}{1/100} = 100$$

GEOMETRIC RANDOM VARIABLE

If **N** represents "number of trials/attempts",
and **p** is the probability of "success" on each trial, then:

$$\mathbb{E}[N] = \frac{1}{p}$$

GEOMETRIC RANDOM VARIABLE

If **N** represents "number of trials/attempts",
and **p** is the probability of "success" on each trial, then:

$$\mathbb{E}[N] = \frac{1}{p}$$

$$\begin{aligned}\mathbb{E}[N] &= 1(p) + (1 + \mathbb{E}[N])(1-p) \\ &= p + (1-p) + (1-p)\mathbb{E}[N] \\ &= 1 + (1-p)\mathbb{E}[N]\end{aligned}$$

$$\begin{aligned}\mathbb{E}[N](1-(1-p)) &= 1 \\ \mathbb{E}[N](p) &= 1 \\ \mathbb{E}[N] &= \frac{1}{p}\end{aligned}$$

سوال؟

مرتب سازی شانسی!

یک نمونه آموزشی از الگوریتم های تصادفی!

BOGOSORT

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
            if A[i] > A[i+1]:
                sorted = False
        if sorted:
            return A
```

This randomly permutes A (assume it takes O(n) time)

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

What is the expected number of iterations?

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Probability that $\mathbf{X_i = 1}$ (A is sorted) = $\mathbf{1/n!}$

since there are n! possible orderings of A and only one is sorted (assume A has distinct elements) ⇒ $E[X_i] = 1/n!$

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

What is the expected number of iterations?

Let $\mathbf{X_i}$ be a Bernoulli/Indicator variable, where

- $\mathbf{X_i = 1}$ if A is sorted on iteration i
- $\mathbf{X_i = 0}$ otherwise

Probability that $\mathbf{X_i = 1}$ (A is sorted) = **1/n!**

since there are n! possible orderings of A and only one is sorted (assume A has distinct elements) ⇒ $E[X_i] = 1/n!$

E[# of iterations/trials] = 1/(prob. of success on each trial)

= 1/(1/n!) = **n!**

BOGOSORT: EXPECTED RUNTIME

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

E[runtime on a list of length n]

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

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

= O(n) * **E**[# of iterations]

= O(n) * (n!)

= O(n * n!)

= ***REALLY REALLY BIG***

BOGOSORT: WORST-CASE RUNTIME?

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

BOGOSORT: WORST-CASE RUNTIME?

```
BOGOSORT(A):
    while True:
        A.shuffle()
        sorted = True
        for i in [0,...,n-2]:
        if A[i] > A[i+1]:
            sorted = False
        if sorted:
        return A
```

Worst-case runtime =

$$\infty$$

This is as if the "bad guy" chooses all the randomness in the algorithm, so each shuffle could be unlucky… forever...

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 (leave it up to randomness)

WORST-CASE RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. Bad guy “rolls” the dice (will choose the randomness in the worst way possible)

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 (leave it up to randomness)

WORST-CASE RUNNING TIME

1. You publish your randomized algorithm
2. Bad guy picks an input
3. Bad guy “rolls” the dice (will choose the randomness in the worst way possible)

Don’t use BogoSort.

سوال؟