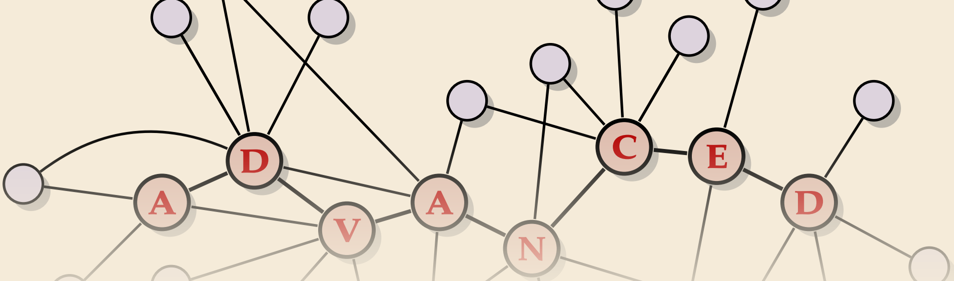


7 Randomization Basics

10 June 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version `2025-06-10 10:03`

Outline

7 Randomization Basics

7.1 Motivation

Computational Lottery?

- If we are faced with solving an NP-hard problem and known smart algorithms are too slow, we likely have to compromise on what "solving" means.
- Classical algorithms are *always* and *exactly* correct.

$\rightsquigarrow$ Here: Let's compromise on "always", i. e., allow algorithms to occasionally **fail**!

Computational Lottery?

- If we are faced with solving an NP-hard problem and known smart algorithms are too slow, we likely have to compromise on what "solving" means.
- Classical algorithms are *always* and *exactly* correct.

⇝ Here: Let's compromise on "always", i. e., allow algorithms to occasionally **fail**!

ϟ A *deterministic* algorithm A that fails on input x will ***always*** fail for x.

  - ⇝ What if we require a solution for such an input x? We get **nothing** from A!
  - Must use a form of *nondeterminism*.

Computational Lottery?

- If we are faced with solving an NP-hard problem and known smart algorithms are too slow, we likely have to compromise on what “solving” means.

- Classical algorithms are *always* and *exactly* correct.

⇝ Here: Let’s compromise on “always”, i. e., allow algorithms to occasionally **fail**!

ϟ A *deterministic* algorithm A that fails on input x will ***always*** fail for x.

  ⇝ What if we require a solution for such an input x? We get **nothing** from A!

  - Must use a form of *nondeterminism*.

- ***Randomization:*** Use *random bits* to guide computation.

⇝ *Instead of always failing on some rare inputs, we rarely fail on any input.*

can make this arbitrarily rare

Why Could Randomization Help?

- Main intuitive reason: (can be) much easier to be 99.999999% correct than 100%
  How can this manifest itself?
  - **Faster and simpler algorithms**
    Random choice can allow to sidestep tricky edge cases
  - We can use **fingerprinting** (a.k.a. checksums) hashing
    Cheap surrogate question, mostly correct, but sometimes wrong.
  - Protect against **adversarial inputs**
    We make our (algorithm's) behavior unpredictable, so it us harder to exploit us.

Why Could Randomization Help?

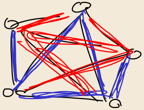
- Main intuitive reason: (can be) much easier to be 99.999999% correct than 100%
  How can this manifest itself?
  - **Faster and simpler algorithms**
    Random choice can allow to sidestep tricky edge cases
  - We can use **fingerprinting** (a.k.a. checksums)
    Cheap surrogate question, mostly correct, but sometimes wrong.
  - Protect against **adversarial inputs**
    We make our (algorithm's) behavior unpredictable, so it us harder to exploit us.

- Also: *probabilistic method* for proofs
  - Goal: Prove existence of discrete object with some property
  - Idea: Design randomized algorithm to find one

  $\rightsquigarrow$ If algorithm succeeds with prob. > 0, object must exist!

Ramsey theory

complete graph on n vertices



Claim:

$\exists$ monochromatic clique

of size $\geq R(n)$

$R(n) \approx \lg n$

Randomized Algorithms vs. Average-Case Analysis

Average-Case Analysis

- algorithm is **deterministic**
  same input, same computation

Randomized Algorithm (here)

- algorithm is **not** deterministic
  same input, potentially different comp.

Randomized Algorithms vs. Average-Case Analysis

Average-Case Analysis

- algorithm is **deterministic**
  same input, same computation
- input is chosen according to some **probability distribution**

Randomized Algorithm (here)

- algorithm is **not** deterministic
  same input, potentially different comp.
- input is chosen **adversarially** (worst-case inputs)

(

oblivious adversary

(can't see random bits)

Randomized Algorithms vs. Average-Case Analysis

Average-Case Analysis

- algorithm is **deterministic**
  same input, same computation
- input is chosen according to some **probability distribution**
- cost given as expectation over inputs

Randomized Algorithm (here)

- algorithm is **not** deterministic
  same input, potentially different comp.
- input is chosen **adversarially** (worst-case inputs)
- cost given as expectation over random choices of algorithm

Randomized Algorithms vs. Average-Case Analysis

Average-Case Analysis

- algorithm is **deterministic**
  same input, same computation
- input is chosen according to some **probability distribution**
- cost given as expectation over inputs

Randomized Algorithm (here)

- algorithm is **not** deterministic
  same input, potentially different comp.
- input is chosen **adversarially** (worst-case inputs)
- cost given as expectation over random choices of algorithm

example: sorting by first shuffle

Confusingly enough, the analysis (technique) is often the same!

But: Implications are quite different; randomization is much more versatile and robust.

7.2 Randomized Selection

Separation Example

- Before we introduce randomization more formally, let's see a successful example
- Here, not a "hard" problem, but a showcase where randomization makes something possible that is *provably*

Introductory Example – Quickselect

Selection by Rank

- **Given:** array $A[0..n)$ of numbers and number $k \in [0..n)$.

but 0-based & counting dups

- **Goal:** find element that would be in position k if A was sorted (kth smallest element).
  - $k = \lfloor n/2 \rfloor \rightsquigarrow$ median; $k = \lfloor n/4 \rfloor \rightsquigarrow$ lower quartile
    $k = 0 \rightsquigarrow$ minimum; $k = n - \ell \rightsquigarrow$ ℓth largest

Introductory Example – Quickselect

Selection by Rank

- **Given:** array $A[0..n)$ of numbers and number $k \in [0..n)$.
- **Goal:** find element that would be in position k if A was sorted (kth smallest element).

  but 0-based & counting dups

  - $k = \lfloor n/2 \rfloor \rightsquigarrow$ median; $k = \lfloor n/4 \rfloor \rightsquigarrow$ lower quartile
    $k = 0 \rightsquigarrow$ minimum; $k = n - \ell \rightsquigarrow$ ℓth largest

```
procedure quickselect(A[0..n), k):
    l := 0;  r := n
    while r − l > 1
        b := random pivot from A[l..r)
        j := partition(A[l..r), b)
        if j ≥ k then r := j − 1
        if j ≤ k then l := j + 1
    return A[k]
```

- simple algorithm:
  determine rank of random element,
  recurse

$\rightsquigarrow$ $O(n)$ time in expectation

over random choices

- worst case: $\Theta(n^2)$
- $O(n)$ also possible deterministically,
  but algorithms is more involved

median of medians

A closer look at Selection

While all within $\Theta(n)$, we do get a strict separation for selecting the median.

Theorem 7.1 (Bent & John (1985))

Any **deterministic** comparison-based algorithm for finding the median of n elements uses at least $\mathbf{2}\boldsymbol{n} - o(n)$ comparisons in the worst case. ◀

Proof omitted.

A closer look at Selection

While all within $\Theta(n)$, we do get a strict separation for selecting the median.

Theorem 7.1 (Bent & John (1985))

Any **deterministic** comparison-based algorithm for finding the median of n elements uses at least $\mathbf{2n} - o(n)$ comparisons in the worst case. ◀

Proof omitted.

The following weaker result is easier to see:

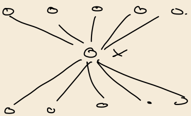
Theorem 7.2 (Blum et al. (1973))

Any deterministic comparison-based algorithm for finding the median of n elements uses at least $\underbrace{n-1}_{\checkmark} + \underbrace{(n-1)/2}_{\checkmark} \sim 1.5n$ comparisons in the worst case. ◀

Proof: Two types of comparisons

(1) certificate comparisons

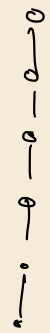
$n-1$



always necessary for correct algorithm

$n = 11$

sorting



n elements

n-1 comps

A Median Adversary

Proof (Theorem 7.2):

(2) "nonessential" comparisons

(not part of certificate)

in particular, comparison between L and S

m = true median

$L = \{x : x > m\}$

$S = \{x : x < m\}$

$(|S| = |L|)$

Given a deterministic algorithm A, we (the adversary) try to answer comparison queries by A in the least useful way (for A)

Here: maintain elements in 3 sets, S, L and U (undecided)

initially all in U

query "$x \overset{?}{<} y$"

if x and y not in same set, answer $S < U < L$

$\left.\begin{array}{l} x, y \in S \\ x, y \in L \end{array}\right\}$ arbitrary answer

$x, y \in U$ $\quad$ $x < y$, put x to S, y into L

$\Rightarrow$ created <u>one</u> non-essential cmp for A

remove 2 elements from U

$\Rightarrow \geq \frac{n-1}{2}$ non-essential comparisons $\square$

Randomized Selection

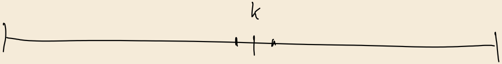
- Can prove: Randomized Quickselect uses in expectation $\sim (2\ln 2 + 2)n \approx 3.39n$ comparisons to find the median
- But we can do better!

Randomized Selection

- Can prove: Randomized Quickselect uses in expectation $\sim (2\ln 2 + 2)n \approx 3.39n$ comparisons to find the median
- But we can do better!

```
procedure floydRivest(A[ℓ..r), k):
    n := r − ℓ
    if n < n_0 return quickselect(A, k)
    s := (1/2) n^{2/3}  // all numbers to be rounded
    sd := (1/2) √(ln(n) s (n − s) / n)
    S[0..s) := random sample from A
    k̂ := s k/n
    p := floydRivest(S, k̂ − sd)
    q := floydRivest(S, k̂ + sd)
    (i, j) := partition A around p_0 and p_1
    if i == k return A[i]
    if j == k return A[j]
    if k < i return floydRivest(A[ℓ..i), k)
    if k > j return floydRivest(A[j..r), k)
    return floydRivest(A[i..j), k)
```

- Variant of Quickselect with huge sample
- Analysis sketch:
  - partition costs $1.5n$ comparisons



Randomized Selection

- Can prove: Randomized Quickselect uses in expectation $\sim (2\ln 2 + 2)n \approx 3.39n$ comparisons to find the median
- But we can do better!

procedure floydRivest($A[\ell..r]$, k):
    $n := r - \ell$
    if $n < n_0$ **return** quickselect(A, k)
    $s := \frac{1}{2} n^{2/3}$ *// all numbers to be rounded*
    $sd := \frac{1}{2}\sqrt{\ln(n)s(n-s)/n}$
    $S[0..s)$:= random sample from A
    $\hat{k} := s\frac{k}{n}$
    p := floydRivest(S, $\hat{k} - sd$)
    q := floydRivest(S, $\hat{k} + sd$)
    (i, j) :=partition A around p_0 and p_1
    if $i == k$ **return** $A[i]$
    if $j == k$ **return** $A[j]$
    if $k < i$ **return** floydRivest($A[\ell..i)$, k)
    if $k > j$ **return** floydRivest($A[j..r)$, k)
    return floydRivest($A[i..j)$, k)

- Variant of Quickselect with huge sample
- Analysis sketch:
  - partition costs $1.5n$ comparisons
  - Everything on sample has cost $o(n)$
  - by the choice of parameters, with prob $1 - o(1)$:
    (a) $i < k < j$ after partition
    (b) $j - i = o(n)$

  $\rightsquigarrow$ all recursive calls expected $o(n)$ cost

Randomized Selection

- Can prove: Randomized Quickselect uses in expectation $\sim (2\ln 2 + 2)n \approx 3.39n$ comparisons to find the median
- But we can do better!

---

procedure floydRivest($A[\ell..r]$, k):
    $n := r - \ell$
    if $n < n_0$ **return** quickselect(A, k)
    $s := \frac{1}{2} n^{2/3}$ *// all numbers to be rounded*
    $sd := \frac{1}{2}\sqrt{\ln(n)s(n-s)/n}$
    $S[0..s)$:= random sample from A
    $\hat{k} := s\frac{k}{n}$
    p := floydRivest(S, $\hat{k} - sd$)
    q := floydRivest(S, $\hat{k} + sd$)
    (i, j) :=partition A around p_0 and p_1
    if $i == k$ **return** $A[i]$
    if $j == k$ **return** $A[j]$
    if $k < i$ **return** floydRivest($A[\ell..i)$, k)
    if $k > j$ **return** floydRivest($A[j..r)$, k)
    return floydRivest($A[i..j)$, k)

- Variant of Quickselect with huge sample
- Analysis sketch:
  - partition costs $1.5n$ comparisons
  - Everything on sample has cost $o(n)$
  - by the choice of parameters, with prob $1 - o(1)$:
    (a) $i < k < j$ after partition
    (b) $j - i = o(n)$
  - $\rightsquigarrow$ all recursive calls expected $o(n)$ cost

$\rightsquigarrow$ Randomized median selection with $1.5n + o(n)$ comparisons

$\rightsquigarrow$ Separation from deterministic case!

Power of Randomness

- Selection by Rank shows two aspects of randomization:
  - A simpler algorithm by avoiding edge cases (like an initial order giving bad pivots)
  - Protection against adversarial inputs
    (inputs constructed with knowledge about the algorithm)

    Here randomization provably more powerful than any thinkable deterministic algorithm!

    constant factor for #cmps

Power of Randomness

- Selection by Rank shows two aspects of randomization:
  - A simpler algorithm by avoiding edge cases (like an initial order giving bad pivots)
  - Protection against adversarial inputs
    (inputs constructed with knowledge about the algorithm)

    Here randomization provably more powerful than any thinkable deterministic algorithm!

    constant factor for #cmps

- What can we gain for (NP-)hard problems?
- But first, let's define things properly.