04. Randomization

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Big Idea: Randomization

Big idea: a *randomized* algorithm deliberately makes random choices

- con: behavior and/or performance becomes *stochastic* (unpredictable)
- pro: other aspects can get better (speed, simplicity)
- often algorithm gets faster/simpler but analysis gets harder
- (this is a win)

A deterministic algorithm is not randomized.

Recap: Quicksort

Recall:

- every sorting algorithm takes $\Omega(n \log n)$ time
- merge sort takes $\Theta(n \log n)$ worst-case time and $\Theta(n)$ space
- **quicksort synopsis:**
  - recursive divide-and-conquer
  - pick random pivot p
  - rearrange list into two zones: $\leq p$ and $> p$
  - recurse onto each zone
- quicksort
  - is randomized
  - $\Theta(n \log n)$ expected time
  - $\Theta(n^2)$ worst-case time
  - $\Theta(\log n)$ space

Recap: Analyzing Randomized Algorithms

For each randomized algorithm we make two separate efficiency claims:

1. expected time: the focus
2. worst-case time: disclaimer; secondary consideration

"Quicksort takes $\Theta(n \log n)$ expected time and $\Theta(n^2)$ worst-case time."

Definition of Expected Time

Definition:

$$\text{algo.'s expected time} = E[\text{algo.'s worst-case time}]$$

For random variable X with outcomes $x_1, \ldots, x_n$ of probability $p_1, \ldots, p_n$,

$$E[X] = \sum_{\text{outcome } x_i} p_i x_i$$

So

$$\text{algo.'s expected time} = \sum_{\text{sequence of random choices } i} p_i \cdot (\text{worst-case time steps given } i)$$

The Hiring Problem

```
1: function DETERMINISTIC-HIRE-ASSISTANT(A)
2:     best = NIL
3:     for x in A do
4:         if best is still NIL or x is better than best then
5:             best = x
6:         end if
7:     end for
8:     return best
9: end function
```

Analyze the number of **reassignments** (line 5)

Adversarial Analysis

Adversarial analysis: Proof strategy for worst-case analysis

"Adversary"

- is a fictional opponent character
- seeks to show algorithm is inefficient
- has full knowledge of algorithm pseudocode
- picks least-flattering input

Big idea:

- randomization makes the adversary's job harder
- so improves the algorithm's expected performance

Adversarial Analysis of Deterministic Hiring

```
function DETERMINISTIC-HIRE-ASSISTANT(A)
    best = NIL
    for x in A do
        if best is still NIL or x is better than best then
            best = x
        end if
    end for
    return best
end function
```

Q: How can an adversary arrange A to maximize reassignments?

Q: What is the worst-case number of reassignments?

Randomized Hiring

```
function RANDOMIZED-HIRE-ASSISTANT(A)
    Randomly permute A                                   ▷ only change
    best = NIL
    for x in A do
        if best is still NIL or x is better than best then
            best = x
        end if
    end for
    return best
end function
```

Observe

- the same worst-case scenario exists
- **but**, the adversary cannot force it to occur

Randomized Hiring Analysis

Define

$$X_i = \{1 \text{ if best is reassigned in iteration } i, 0 \text{ otherwise}\}.$$

Observe

$$X_i = 1 \text{ when the } i\text{th element is the maximum so far}$$

and since A is permuted randomly,

$$Pr\{X_i = 1\} = 1/i \text{ so } E[X_i] = 1/i,$$

and the total number of reassigns is

$$X = 1/1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n \in O(\log n).$$

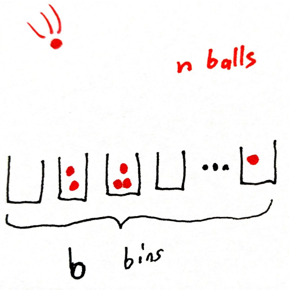
$\#$ reassignments is $O(\log n)$ expected and $\Theta(n)$ worst-case.

Randomization Patterns

Randomization pattern: approach for using randomization, along with analysis

Best from random order pattern: best only gets reassigned expected $O(\log n)$ times, worst case $\Theta(n)$ times

Balls and Bins



Story to help think about probabilities:

- b bins that can hold balls
- throw n balls
- a ball is equally likely to fall into each bin
- corresponds to a game called *plinko*

Balls and Bins Q & A

Answers to questions:

- *Q: After n throws, how many balls does a given bin have?* expected n/b
- *Q: How many throws before a given bin has a ball?* expected b
- *Q: How many throws before every bin has a ball?* expected $b \ln b \in \Theta(b \log b)$

Application: Web Server Load Balancer

- scenario: we have b webservers, n requests coming in, need to route each request to one of the servers $1, \ldots, b$
- adversary could make expensive requests, so if we take turns in a deterministic way, we are vulnerable to a denial-of-service attack
  - example: round-robin
- So route requests randomly somehow
- Idea: choose a **random** server in $\{1, \ldots, b\}$
- Worst-case user experience depends on the heaviest-loaded server

With High Probability (WHP)

Terminology: Let $p(n)$ be the probability that event X occurs as a function of n; then X occurs *with high probability (w.h.p. or WHP)* when

$$\lim_{n \to \infty} p(n) = 1.$$

Examples of $p(n)$ bounds considered w.h.p.: $O(1 - \frac{1}{n})$, $O(1 - \frac{1}{n^2})$, $O(1 - \frac{1}{n \log n})$.

Typically, of the form $O(1 - \frac{1}{n^k})$ for $k \geq 1$.

Intuition: for sufficiently large n, event X is practically certain.

Analyzing Random Load Balancing

- suppose $n = b$
- suppose the #balls in a bin is its *load*
- high load is bad
- *After n throws, what is the maximum load?*

$$\frac{\log n}{\log \log n} + o(1) \text{ w.h.p.}$$

- Note that for $n \geq 2$,

$$\frac{\log n}{\log \log n} \ll \log n.$$

The Power of Two Random Choices

- **two random choices**: pick two bins at random, put the ball in *the less-loaded bin;* maximum load becomes

$$\frac{\log \log n}{\log 2} + \Theta(1) \text{ w.h.p.}$$

- Note that $\frac{\log \log n}{\log 2} \ll \frac{\log n}{\log \log n}$; the choosing dramatically decreases maximum load
- Intuition: what random events lead to a high load in each scenario?
- generally, if we make d **random choices**, maximum load is

$$\frac{\log \log n}{\log d} + \Theta(1) \text{ w.h.p.}$$

- very slow-growing, nearly constant

Load Balancing Patterns

Balance load with one random choice: for n balls in $\Theta(n)$ bins, expected load is $\Theta(1)$ and maximum load is $\Theta(\frac{\log n}{\log \log n})$ w.h.p.

Balance load with d random choices: expected load is still $\Theta(1)$, and maximum load is $\Theta(\frac{\log \log n}{\log d})$ w.h.p.

Trade-off:

- one random choice: choosing bin involves only one random number, $\Theta(1)$ time, and does not involve state of bins; *but* load can be more uneven
- d random choices: choosing bin involves querying the state of $\Theta(d)$ servers, but load is distributed **extremely** evenly

Streaks

- suppose we flip a fair coin, so $Pr\{\text{heads}\} = Pr\{\text{tails}\} = \frac{1}{2}$
- *streak:* sequence of the same result (seq. of heads, or seq. of tails)
- *Q: After n flips, what is the longest streak?*
- A: expected length of the longest streak is $\Theta(\log n)$

Hash Tables

Review hash tables

- can store a **set** of *keys*
- or a **map** from keys to arbitrary *values*
- keys must *hashable:* either integers, or can be mapped deterministically to integers (e.g. strings, floats, tuples of hashable objects, etc.)
- a search, insert, or delete operation takes $\Theta(1)$ expected time and $\Theta(n)$ worst-case time
- many variants with trade-offs: chaining vs. open addressing, universal vs. tabular functions, cuckoo, robin hood, etc.

Hash Set Operations

Operation	Pseudocode Example	Exp. Time	W.C. Time
Create empty hash set	`S = HashSet()`	$\Theta(1)$	$\Theta(1)$
Insert an element	`S.insert(x)`	$\Theta(1)$	$\Theta(n)$
Remove an element	`S.remove(x)`	$\Theta(1)$	$\Theta(n)$
Search for an element	`if S.contains(x):`	$\Theta(1)$	$\Theta(n)$

Recall: A math set cannot have duplicates, so inserting the same x multiple times leaves only one copy in the set.

Hash Map Operations

Operation	Pseudocode Example	Exp. Time	W.C. Time
Create empty hash map	`M = HashMap()`	$\Theta(1)$	$\Theta(1)$
Insert k, v	`M.insert(k, v)`	$\Theta(1)$	$\Theta(n)$
Remove key k	`S.remove(k)`	$\Theta(1)$	$\Theta(n)$
Search for key k	`if S.contains(k):`	$\Theta(1)$	$\Theta(n)$
Lookup key k	`S.get(k)`	$\Theta(1)$	$\Theta(n)$

Recall: Each key must be distinct, so re-inserting a new value for key k overwrites the old value.

Reduce-to-Hash-Tables Pattern

- make critical use of a hash set or hash map
- replace a $\Theta(n)$ loop with a $\Theta(1)$ expected-time hash table operation
- good: fast, simple
- bad: time efficiency becomes *expected*

Duplicate Removal Problem

duplicate removal problem
input: an array $A[1..n]$ of objects
output: a list D of the distinct elements of A (i.e. duplicates are removed)

Duplicate Removal – Baseline Pseudocode

```
function REMOVE-DUPLICATES-BASELINE(A)
    D = new list
    for a in A do
        already-present = False
        for d in D do
            if a == d then
                already-present = True
            end if
        end for
        if not already-present then
            D.add(a)
        end if
    end for
    return D
end function
```

Duplicate Removal – Baseline Analysis

- outer loop: n iterations
- inner loop: $\Theta(n)$ time
- total $\Theta(n^2)$ time
- bottleneck is the nested loops
- *Reduce-to-Hash-Tables Pattern:* replace the inner loop with a hash table operation

Duplicate Removal – Improved Algorithm

```
function REMOVE-DUPLICATES-RANDOMIZED(A)
    HS = HashSet()
    for x in A do
        if not HS.contains(x) then
            HS.insert(x)
        end if
    end for
    D = insert each element of HS into a list        ▷ match output: data type
    return D
end function
```

$\Theta(n)$ expected time, $\Theta(n^2)$ worst-case time.

Duplicate Removal – Analyze Trade-Offs

- baseline: $\Theta(n^2)$ time; more complicated; not dependent on hash tables knowledge
- randomized: $\Theta(n)$ expected time, $\Theta(n^2)$ worst-case time; simpler; depends on hash tables
- randomized is superior
- pay-off for learning about data structures!

Planning a Hash Set

- hash set stores *key-value* associations
- each key is linked to a value
- *key:* identity of a thing
- *value:* information associated with that thing
- insert, remove, search
- Strategy: fill in the blanks "Given __*key*__, update __*value*__ ."
- Strategy: write out concrete data in a table like

Key	**Value**
key 1	value 1
key 2	value 2
...	...

Mode Problem

duplicate removal problem
input: a list A of n elements
output: a most-frequently-ocurring element of A

Note:

- A is a list, not a set, so A may have duplicates
- in the event of ties, any of the ties may be output

Mode – Baseline Pseudocode

```
function MODE-BASELINE(A)
    mode = NIL
    mode-count = 0
    for a in A do
        a-count = 0
        for b in A do
            if b == a then
                a-count++
            end if
        end for
        if a-count > mode-count then
            mode = a
            mode-count = a-count
        end if
    end for
    return mode
end function
```

Mode – Baseline Analysis

- total $\Theta(n^2)$ time
- again, bottleneck is the nested loops
- *Reduce-to-Hash-Tables Pattern:* replace the inner loop with a hash table operation
- Pre-compute each element's count
- Hash map: "Given (an element), update (its count) ."
- Concrete data :

$$A = \langle 3, 1, 7, 1, 1, 1, 7 \rangle$$

Key	**Value**
3	1
1	4
7	2

Mode – Second Draft

Hash table is used to speed up first draft, but we haven't initialized it yet.

```
function MODE-RANDOMIZED(A)
    (somehow create and populate hash map ElementToCount)
    mode = A[0]
    mode-count = 0
    for a in A do
        a-count = ElementToCount.get(a)
        if a-count > mode-count then
            mode = a
            mode-count = a-count
        end if
    end for
    return mode
end function
```

Mode – Final Draft

```
function MODE-RANDOMIZED(A)
    ElementToCount = HashMap()
    for a in A do
        if ElementToCount.contains(a) then
            ElementToCount.set(a, ElementToCount.get(a) +1)
        else
            ElementToCount.set(a, 1)
        end if
    end for
    mode = A[0]
    mode-count = 0
    for a in A do
        a-count = ElementToCount.get(a)
        if a-count > mode-count then
            mode = a
            mode-count = a-count
        end if
    end for
    return mode
end function
```

Duplicate Removal – Analyze Trade-Offs

- baseline: $\Theta(n^2)$ time
- randomized:
  - first loop: $\Theta(n)$ expected, $\Theta(n^2)$ worst-case
  - second loop: same
  - everything else: $\Theta(1)$
  - total: $\Theta(n)$ expected, $\Theta(n^2)$ worst-case
  - like the maximum subarray algorithm, refactoring two nested loops into two sequential loops speeds things up
- again, randomized is superior