04. Randomization

CPSC 535 ~ Fall 2019

Kevin A. Wortman

California State University Fullerton

September 16, 2019

Randomization

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

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

E.g. quicksort, recall

- every sorting algorithm takes $\Omega(n \log n)$ time
- merge sort takes $\Theta(n \log n)$ worst-case time but $\Theta(n)$ temporary space
- quicksort is randomized, takes $\Theta(n \log n)$ expected time but only $\Theta(\log n)$ space *(in-place)*, better constant factors

Kinds of Time Bounds

Suppose algorithm A takes...

- $\Theta(n)$ deterministic worst-case time: for *every* input, A takes $\Theta(n)$ time
- $\Theta(n)$ average time: the mean time, *averaging over every possible input,* is $\Theta(n)$
  - only relevant when each input is equally likely
  - not true for e.g. sorting, maximum subarray
  - in principle we could take a weighted average, but we'd need to know the probability distribution of inputs, unlikely
- $\Theta(n)$ expected time: the mean time, *averaging over every sequence of random choices A could make*, is $\Theta(n)$
  - no assumption about input; still assume worst case
- by default, "$\Theta(n)$ time" means $\Theta(n)$ deterministic worst-case time

Deterministic versus Randomized Algorithms

If alg A is **deterministic**: its deterministic worst-case time and expected time bound are always the same

- technically, we can say linear search takes "$\Theta(n)$ expected time"
- but this is kind of misleading/distracting

A **is randomized**: usually expected-case is faster than worst-case

- (because expected-case is an average, worst-case is a maximum)
- hash table insert: $\Theta(1)$ expected time, $\Theta(n)$ worst-case time
- treap insert: $\Theta(\log n)$ expected time, $\Theta(n)$ worst-case time
- quicksort: $\Theta(n \log n)$ expected time, $\Theta(n^2)$ worst-case time

Multiplying Randomized Bounds

- Multiplying works normally
- If running A once takes $O(E)$ expected time and $O(W)$ worst-case time...
- ...then running A (k) times takes $O(kE)$ expected time and $O(kW)$ worst-case time.
- So
  - k hash table inserts takes $O(k)$ expected time and $O(kn)$ worst-case time
  - n hash table inserts takes $O(n)$ expected time and $O(n^2)$ worst-case time
  - n treap inserts takes $O(n \log n)$ expected time and $O(n^2)$ worst-case time

Adding Randomized Bounds

adding works normally with two caveats

1. the *expected* qualifier is "sticky"
   - $O(D)$ worst-case time $+$ $O(E)$ expected time $= O(\max\{D, E\})$ expected-time
   - $\implies$ insert n elements into hash table, then loop through hash table $= O(n)$ expected $+ O(n)$ worst-case $= O(n)$ expected
2. however, you have the option of using a randomized alg's worst-case bound
   - $\implies$ insert n elements into hash table, then sort elements with insertion sort $= O(n)$ expected $+ O(n^2)$ worst-case $= O(n^2)$ expected time
   - but we could also use hash tables' worst-case bound and say $= O(n^2)$ worst-case $+ O(n^2)$ worst-case $= O(n^2)$ worst-case

Ordinarily

- $O(T)$ worst-case time is better than $O(T)$ expected time
- faster expected-time is better than slower worst-case time

Maximum

```
function MAXIMUM(A)
    best = NIL
    for x in A do
        if best is still NIL or x > best then
            best = x
        end if
    end for
    return best
end function
```

- (CLRS calls this *hiring*, but it generalizes to any kind of find-the-best process.)
- Suppose the "best $= x$" step is expensive (e.g. moving your house).
- *Q: how many times is best reassigned in the best case?*
- *Q: what about the worst case?*

Maximum (continued)

```
function MAXIMUM(A)
    best = NIL
    for x in A do
        if best is still NIL or x > best then
            best = x
        end if
    end for
    return best
end function
```

A best-case: A in decreasing order; reassigned only once

A worst-case: A in increasing order; reassigned n times

Randomized Maximum

```
function RANDOMIZED-SMAXIMUM(A)
    permute A randomly                          ▷ only change
    best = NIL
    for x in A do
        if best is still NIL or x > best then
            best = x
        end if
    end for
    return best
end function
```

- best-case: luckily visit maximum first, only one reassign
- worst-case: unluckily visit in increasing order, reassign n times
- (same)
- **but what about the expected number of reassigns?**

Randomized Maximum 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

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

$\implies$ expected number of reassigns is $O(\log n)$.

Randomization Patterns

Randomization pattern: approach for using randomization, along with analysis

Best from random order pattern: maximum 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
- (sketch)
- corresponds to a game called *plinko*

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 $b \ln b \in \Theta(b \log b)$

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?*

$$\frac{\log n}{\log \log n} \text{ w.h.p.}$$

- (*w.h.p.* = with high probability = probability $O(1/n^k)$ for $k \geq 1$)

The Power of Two Random Choices

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

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

- generally, if we make d **random choices**, maximum load is

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

- almost constant; truly constant if we set $d \in \Omega(\log n)$

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 d random numbers, $\Theta(d)$ time, and needs to know current load of bins; but load is distributed very evenly

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
- $\therefore$ route requests randomly somehow
- choose a random server in $\{1, \ldots, b\}$
  - very simple
  - balls-and-bins: expect n/b requests/server, b requests before a given server is working, $\Theta(b \log b)$ requests before all servers working
  - maximum requests/server $\Theta(\frac{\log n}{\log \log n})$ w.h.p.
- choose two random servers, ask for their current load, route to the less-loaded server
  - good: better server utilization, maximum requests/server is lower at $\Theta(\frac{\log \log n}{\log 2})$ w.h.p.
  - bad: routing involves querying two servers for current load
  - trade-off: which is worse, spending time on these current-load queries, or letting some servers get more overloaded?

Application: Chained Hash Tables

- Recall *chained hash table:* use a random hash function to map each key to a list of collisions called a *chain*
- search or delete involves looping through one chain (also insert that checks for duplicates)
- chain length is expected $\Theta(1)$ but worst-case $\Theta(n)$
- (sketch)
- power of two random choices:
  - **two** random hash functions
  - to insert, find **two** random chains, add to the *shorter* chain
  - length is still $\Theta(1)$ expected but worst-case $\Theta(\log \log n)$ w.h.p.
  - better for applications intolerant to outliers
- could find $\Theta(\log n)$ random chains for worst-case $\Theta(1)$ chain length, but then table operations take $\Theta(\log n)$ time and we might as well use a binary search tree

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 $\Theta(\log n)$

Application: Skip Lists

- *skip list:* alternative to a binary search tree
- *layer 1* (bottom) is a sorted linked list
- layer 2: contains subset of layer 1; each layer-1 element is present with probability $\frac{1}{2}$
- layer i: contains subset of layer $(i-1)$; each lower element is present with probability $\frac{1}{2}$
- (sketch)
- search takes time $\Theta(\#\text{ layers})$
- $\#$ layers $=$ length of streak after n flips $= \Theta(\log n)$
- $\implies$ searching a skip list takes $\Theta(\log n)$ expected time

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.

Reduce to hash tables pattern:

- make critical use of a hash set or hash map
- good: fast, simple (when hash internals are encapsulated)
- bad: introduces expected qualifier

Application: Duplicate Removal

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

Baseline uses nested for loops and $\Theta(n^2)$ time. Reducing to hash tables:

```
function REMOVE-DUPLICATES(A)
    S = new hash map
    D = hew list
    for x in A do
        if not S.contains(x) then
            D.add(x)
            S.insert(x)
        end if
    end for
    return D
end function
```

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