CS 261: Data Structures

Week 2: Dictionaries and hash tables

Lecture 2a: Overview, probability, and hash functions

David Eppstein
University of California, Irvine

Spring Quarter, 2024

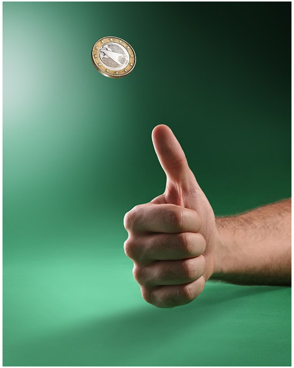
Quick review of probability

Basic concepts

Random event: Something that might or might not happen

Probability of an event: a number, the fraction of times that an event will happen over many repetitions of an experiment

$$\Pr[X]$$



More basic concepts

Discrete random variable: Can take one of finitely many values, with (possibly different) probabilities summing to one

Uniformly random variable: Each value is equally likely

Independent variables: Knowing the value of any subset doesn't help predict the rest (their probabilities stay the same)

Expected values

If R is any function $R(X)$ of the random choices we're making, its expected value is just weighted average of its values:

$$E[R] = \sum_{\text{outcome } X} \Pr[X] R(X)$$

Linearity of expectations: For all collections of functions R_i,

$$\sum_i E[R_i] = E\left[\sum_i R_i\right]$$

(It expands into a double summation, can do sums in either order)

Analysis of random algorithms

We will analyze algorithms that use randomly-chosen numbers to guide their decisions

...but on inputs that are not random (usually worst-case)

The amount of time (number of steps) that such an algorithm takes is a random variable

Most common measure of time complexity: $E[\text{time}]$

Also used: "with high probability", meaning that the probability of seeing a given time bound is $1 - o(1)$ (tends to one in the limit as n gets large)

Expected number of occurrences

Suppose that we have a collection of events (things that might happen once), and let P_i be the probability that event i happens

Then, if X_i is the number of times event i happens:

- $X_i = 0$ with probability $(1 - P_i)$: it didn't happen
- $X_i = 1$ with probability P_i: it did happen
- $E[X_i] = (1 - P_i) \cdot 0 + P_i \cdot 1 = P_i$

Linearity of expectation $\Rightarrow$

$$E[\text{number of events that happen}] = E[\sum X_i] = \sum E[X_i] = \sum P_i$$

Expected number of events that happen = sum of probabilities

Does not require the events to be independent!

Markov's inequality and the union bound

If X is any non-negative random variable, and $a \geq 0$, then

$$\Pr[X \geq a] \leq \frac{E[X]}{a}$$

because otherwise the contribution to $E[X]$ from outcomes with value $\geq a$ would be too large

Corollary: Suppose we have a collection of events with probabilities P_i, again possibly non-independent

$\Pr[\text{at least one event happens}] \leq \sum P_i$

(apply Markov with $X = \#$ events and $a = 1$)

Chernoff bound: Intuition

Suppose you have a collection of random events
(independent, but possibly with different probabilities)

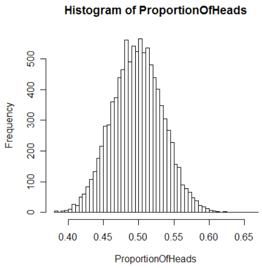
Define a random variable X: how many of these events happen

Main idea: X is very unlikely to be far away from $E[X]$

Example: Flip a coin n times

X = how many times do you flip heads?

Very unlikely to be far away from $E[X] = n/2$



1000 trials, 200 coins

Chernoff bound (multiplicative form)

Let X be a sum of independent 0–1 random variables

Then for any $c > 1$,

$$\Pr[X \geq c\,E[X]] \leq \left(\frac{e^{c-1}}{c^c}\right)^{E[X]}$$

(where $e \approx 2.718281828\ldots$, "Euler's constant")

Similar formula for the probability that X is at most $E[X]/c$

Three special cases of Chernoff

$$\Pr[X \geq c\,E[X]] \leq \left(\frac{e^{c-1}}{c^c}\right)^{E[X]}$$

If c is constant, but $E[X]$ is large $\Rightarrow$
probability is an exponentially small function of $E[X]$

If $E[X]$ is constant, but c is large $\Rightarrow$
probability is $\leq 1/c^{\Theta(c)}$, even smaller than exponential in c

If c is close to one ($c = 1 + \delta$ for $0 < \delta < 1$) $\Rightarrow$
probability is $\leq e^{-\delta^2 E[X]/3}$

Dictionaries

What is a dictionary?

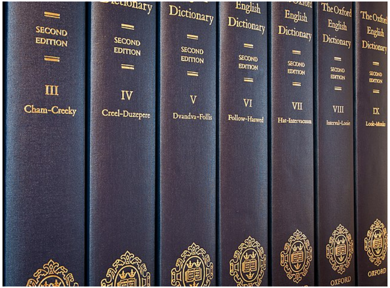
Maintain a collection of key–value pairs

- Keys are often integers or strings but can be other types
- At most one copy of each different key
- Values may be any type of data

Operations update the collection and look up the value associated with a given key

Dictionary in everyday usage:
book of definitions of words.

Its keys are words and its
values are their definitions



Example application

We've already seen one in week 1!

In the depth-first-search example, we represented the input graph as a dictionary with keys = vertices and values = collections of neighboring vertices

The algorithm didn't need to know what kind of object the vertices are, only that they are usable as keys in the dictionary

Dictionary API

Create new empty dictionary

Look up key k and return the associated value

(Exception if k is not included in the collection)

Add key–value pair (k, x) to the collection

(Replace old value if k is already in the collection)

Check whether key k is in the collection and return Boolean result

Enumerate pairs in the collection

Dictionaries in Java and Python

Python: dict type

- Create: `D = {key1: value1, key2: value2, ...}`
- Get value: `value = D[key]`
- Set value: `D[key] = value`
- Test membership: `if key in D:`
- List key–value pairs: `for key, value in D.items():`

Java: HashMap

Similar access methods with different syntax

Python example: Counting occurrences

```
def count_occurrences(sequence):
    D = {}              # Create dictionary
    for x in sequence:
        if x in D:      # Test membership
            D[x] += 1   # Get and then set
        else:
            D[x] = 1    # Set
    return D
```

Non-hashing dictionaries: Association list

Store unsorted collection of key–value pairs

- Very slow (each get/set must scan whole dictionary),
  $O(n)$ time per operation where n is # key–value pairs
- Can be ok when you know entire dictionary will have size $O(1)$
- We will use these as a subroutine in hash chaining (Thursday)

Non-hashing dictionaries: Direct addressing

Use key as index into an array

- Only works when keys are small integers
- Wastes space unless most keys are present
- Fast: $O(1)$ time to look up a key or change its value
- Important as motivation for hashing

Non-hashing dictionaries: Search trees

Binary search trees, B-trees, tries, and flat trees

- We'll see these in weeks 6 and 7!

  (Until then, you won't need to know much about them)
- Unnecessarily slow if you need only dictionary operations (searching for exact matches)
- But they can be useful for other kinds of query (inexact matches)

Hashing

Hash table intuition

The short version:

Use a hash function $h(k)$ to map keys to small integers

Use direct addressing with key $h(k)$ instead of k itself

Two complications:

Where does the hash function come from? (today)

What do we do when two keys have the same hash function value? (Thursday)

Hash table intuition

Maintain a (dynamic) array A whose cells store key–value pairs

Construct and use a hash function $h(k)$ that "randomly" scrambles the keys, mapping them to positions in A

Store key–value pair (k, x) in cell $A[h(k)]$ and do lookups by checking whether the pair stored in that cell has the correct key

When table doubles in size, update h for the larger set of positions

All operations: $O(1)$ amortized time (assume computing h is fast)

Complication: Collisions. What happens when two keys k_1 and k_2 have the same hash value $h(k_1) = h(k_2)$?

Hash functions: Perfect hashing

Sometimes, you can construct a function h that maps the n keys one-to-one to the integers $0, 1, \ldots n-1$

By definition, there are no collisions!

Works when set of keys is small, fixed, and known in advance, so can spend a lot of time searching for perfect hash function

Example: reserved words in programming languages / compilers

Use fixed (not dynamic) array A of size n, store key–value pair (k, x) in cell $A[h(k)]$ (include value so can detect invalid keys)

Hash functions: Random

Standard assumption for analysis of hashing:

The value of $h(k)$ is a random number,
independent of the values of h on all the other keys

Not actually true in most applications of hashing

Results in this model are not mathematically valid
for non-randomly-chosen hash functions

(nevertheless, analysis tends to match practical performance
because many nonrandom functions behave like random)

This assumption is valid for Java IdentityHashMap:
Each object has hash value randomly chosen at its creation

Hash functions: Cryptographic

There has been much research in cryptographic hash functions that map arbitrary information to large integers (e.g. 512 bits)

Could be used for hash functions in dictionaries
by taking result modulo n

Any detectable difference between the results and a random function $\Rightarrow$ the cryptographic hash is considered broken

Too slow to be practical for most purposes

Hash functions: Fast, practical, and provable

It's possible to construct hash functions that are fast and practical

... and at the same time use them in valid mathematical analysis of hashing algorithms

Details depend on the choice of hashing algorithm

We'll see more on this topic in the rest of this lecture

k-independent hash functions

The problem

All analysis so far has assumed hash function is random

But that is rarely achievable in practice

- Cryptographic functions act like random but too slow
- IdentityHashMap not usable in all applications
  and doesn't allow changing to a new hash function

Many software libraries use ad-hoc hash functions that are arbitrary, but not random

- We can't prove anything about how well they work!

Instead, we want a function that

- Can be constructed using only a small seed of randomness
- Is fast (theoretically and in practice) to evaluate
- Can be proven to work well with hashing algorithms

k-independence

Choose function h randomly from a bigger family H of functions

If $H =$ all functions, h is uniformly random (previous assumption)

If H is smaller, h will be less random

Define H to be k-independent if every k-tuple of keys has independent outputs (every tuple of outputs is equally likely)

Bigger values of k give stronger independence guarantees

An example of a (bad) 1-independent hash function: choose one random number r and define h_r to ignore its argument and return r

So we are selecting a function randomly from the set $H = \{h_r\}$

Is k-independence enough?

We will see three algorithms next time: chaining, linear probing, cuckoo hashing

Expected-time analysis of hash chaining only pairwise collision probabilities

If we use a 2-independent hash function, these probabilities are the same as for a fully independent hash function

Expected-time analysis of linear probing has been done with 5-independent hashing [Pagh et al. 2009]

But there exist 4-independent hash functions designed to make linear probing bad [Pătrașcu and Thorup 2016]

Cuckoo hashing requires $\Theta(\log n)$-independence

Algebraic method for k-independence

From b-bit numbers (that is, $0 \leq \text{value} < 2^b$) to range $[0, N-1]$

Choose (nonrandom) prime number $p > 2^b$

Choose k random coefficients $a_0, a_1, \ldots a_{k-1} \pmod{p}$

$$h(x) = \left(\left(\sum_i a_i x^i \right) \bmod p \right) \bmod N$$

Works because, for every k-tuple of keys and every k-tuple of outputs, exactly one polynomial mod p produces that output

$O(k)$ arithmetic operations but multiplications can be slow

Tabulation hashing

Represent key x as a base-B number for an appropriate base B
$x = \sum_i x_i B^i$ with $0 \leq x_i < B$

E.g. for $B = 256$, x_i can be calculated by `(x>>(i<<3))&0xff`
(using only shifts and masks, no multiplication)

Let d be number of digits ($d = 4$ for 32-bit keys and $B = 256$)

Initialize: fill a $d \times B$ table T with random numbers

$h(x) =$ bitwise exclusive or of $T[i, x_i]$ ($i = 0, 1, \ldots d - 1$)

3-independent but not 4-independent; works anyway for linear probing and static cuckoo hashing [Pătrașcu and Thorup 2012]

References and image credits

Fangz. Proportion of heads on a fair coin, histogram plot. PD image, June 5 2008. URL https://commons.wikimedia.org/wiki/File:HistPropOfHeads.png.

mrpolyonymous. Volumes of the Second Edition of the Oxford English Dictionary. CC-BY image, April 21 2012. URL https://commons.wikimedia.org/wiki/File:OED2_volumes.jpg.

Mvolz. Candy graph experiment. CC-BY-SA image, September 25 2016. URL https://commons.wikimedia.org/wiki/File:Candy_graph_experiment.jpg.

Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. *SIAM Journal on Computing*, 39(3):1107–1120, 2009. doi: 10.1137/070702278.

Mihai Pătrașcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM*, 59(3):A14:1–A14:50, 2012. doi: 10.1145/2220357.2220361.

Mihai Pătrașcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. *ACM Transactions on Algorithms*, 12(1):A8:1–A8:27, 2016. doi: 10.1145/2716317.

ICMA Photos. Coin toss. CC-BY-SA image, June 17 2009. URL https://commons.wikimedia.org/wiki/File:Coin_Toss_(3635981474).jpg.

Rlyehable. D100. CC-BY-SA image, February 26 2013. URL https://commons.wikimedia.org/wiki/File:D100.jpg.