Problem Solving with Randomization

Zac Friggstad

NAPC - 2023

We'll discuss some key ideas (with examples) in effectively utilizing randomization to solve problems.

This is a bit of a whirlwind tour, the topic could have its own seminar series.

Comment: For some problems I list, a randomized approach is all I know. For some, there are also deterministic approaches but I found the randomized approach to be simpler for me.

To generate random numbers, use `mt19937`. This distributes much more "randomly" than `rand()` and is quite a bit faster.

Read `https://codeforces.com/blog/entry/61587` for more discussion. I just want to get to problem solving in these slides ☺.

Topic: Random Sampling

NWERC 2014 - Finding Lines

https://open.kattis.com/problems/findinglines

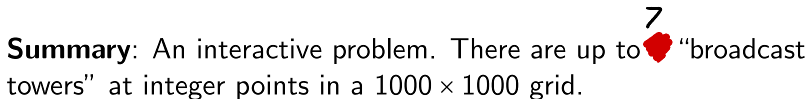
Sample 2 random points p, q. The probability they are distinct points on a line with 20% of the points is (nearly) at least $\frac{1}{5} \cdot \frac{1}{5}$.

In $O(n)$ time, can check that the line $\overline{pq}$ has $\geq 20\%$ of the points.

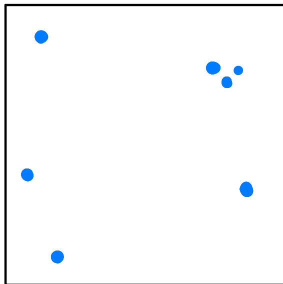
Repeating 200 times fails to detect the line with probability about $(1 - 1/25)^{200} \approx 0.03\%$.

NWERC 2022 - Dragonballs

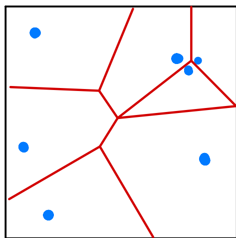
`https://open.kattis.com/problems/dragonballs`

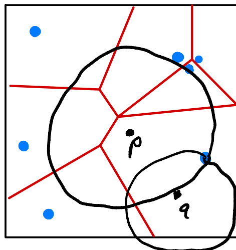


Summary: An interactive problem. There are up to 7 “broadcast towers” at integer points in a 1000×1000 grid.



Consider the Voronoi cells. At least one has $\sim\geq 1/7$ of the grid points (a negligible amount lie on the cell lines). So the probability two random points p, q lie in the same cell is at least roughly 1/49.



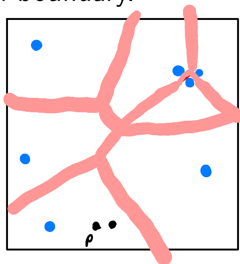


Query with p and q: if they are in the same cell then one of the intersections of the two circles will be a tower.

Each iteration involves ≤ 4 queries (p, q and the circle intersection points). So the expected number of iterations until all towers are identified is at most:

$$4 \cdot (49 + 36 + 25 + 16 + 9 + 4 + 1) = 560.$$

Even better. There are only $O(7 \cdot 1000)$ points that are within distance 1 of a voronoi boundary.



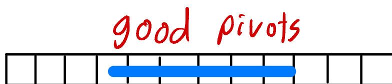
So a random $p = (x, y)$ is in the "interior" of a cell with probability $\frac{10^6 - O(7000)}{10^6} \geq 98\%$

So do as before, but with points p and $q = (x+1, y)$. Expected number of queries in total is at most

$$4 \cdot \frac{1}{0.98} \cdot 7 \leq 30.$$

Example of heuristic reasoning. **Quicksort**

- A random pivot is likely to cut the list into parts of size $\leq 3/4$.



- So even though not all iterations do this, it still happens often enough that the algorithm probably has logarithmic recursion depth, thus $O(n \log n)$ running time.

Caveat: The actual analysis you will probably find by searching for "randomized quicksort analysis" is likely different. But reasoning like this works in practice for competitive programming.

Other problems that can be solved with random sampling.

Codeforces 364 - Ghd
`https://codeforces.com/problemset/problem/364/D`

NERC 2022-23 - Amazing Trick
`https://codeforces.com/contest/1773/problem/A`

BOI 2018 - Worm Worries
`https://open.kattis.com/problems/worm`
Caveat: Some of the groups can (must?) be solved with a randomized algorithm but other groups need a different approach.

Topic: Random Constructions

With some constructive problems, things might seem so "forgiving" that a random construction could work.

NAC 2022 - Word Ladder

`https://open.kattis.com/problems/wordladder3`

Randomized Construction

- Start with any length-10 string.
- Pick a random index and change it to a random character in the most recent string to generate the next string. Reject if it is a string in the sequence or is "adjacent" to a string too early in the sequence.
- Repeat until the sequence is long enough.

Intuition: It just seems super-unlikely that you will get "stuck".
Determinstic Variant: Fix a seed. Make sure it works on your machine for $N = 5\,000$.

For random construction problems you should consider things like:

- Is there enough “slack” in the constraints of the problem that each step will probably succeed? Otherwise a random approach may not work (or needs to be refined).
- Checking if your recent sample is “ok” before proceeding with the next one.
- You may have to “backtrack” a few steps if you seem stuck.

Other constructions I have solved with randomization.

Codeforces 1722 - Even-Odd XOR
`https://codeforces.com/contest/1722/problem/G`

Codeforces 907 - Seating of Students
`https://codeforces.com/contest/907/problem/D`

UAPC 2023 - Sneaky Exploration
`https://open.kattis.com/problems/sneakyexploration`

Topic: Colour Coding

KTH Challenge 2013 - vacuum

https://open.kattis.com/problems/vacuum

The problem at the link does not need a randomized algorithm.

But consider this variant: $n \leq 200\,000$ and the sizes can be huge.

An observation

If there was only one bin, it can be solved in $O(n \log n)$ time. Sort items by size:

- For each item, binary search to find the largest other item that it could fit with.

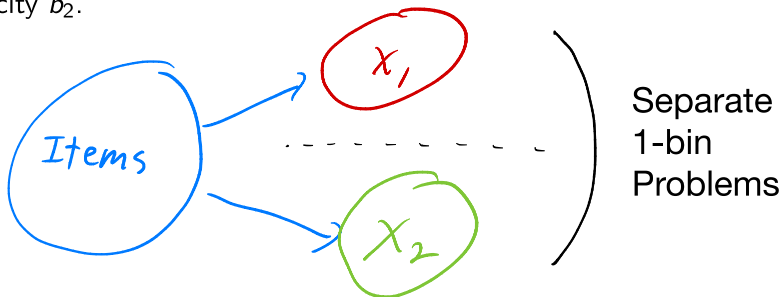
1, 5, 6, 6, 8, 9, 12, 14, 18

Cap = 21

- **or** can even do in $O(n)$ time after sorting using a simultaneous forward/backward linear scan.

What about 2 bins?

Randomly partition the items into two groups X_1, X_2. Run the $O(n \log n)$ algorithm for X_1 with capacity b_1 and also for X_2 with capacity b_2.



With probability at least $1/2^4$, this works since each of the items in the optimum solution will be added to the right part X_1 or X_2 with probability $1/2$.

Can improve the probability a bit by also trying X_2 with b_1 and X_1 with b_2: the probability of success is at least $1/8$ now.

I'm borrowing the term colour coding from the field Parameterized Algorithms, which has many examples of colour coding.
Some more academic examples are here:
`https://tcs.rwth-aachen.de/lehre/FPT/WS2014/slides.pdf`

Basic Idea

If the actual number of objects the optimum cares about is small, then perhaps a random grouping/colouring of the objects into fewer groups will help.

This may reduces the complexity of a combinatorial search for the right items.

2019 ICPC Mid-Central Regional - Dragonball II

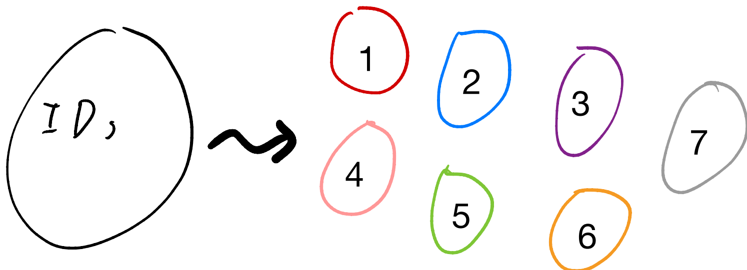
`https://open.kattis.com/problems/dragonball2`

Notice: `1~~4~~6 seconds`

Observation: If there are only 7 distinct IDs, say $\{1, 2, 3, 4, 5, 6, 7\}$ to begin with, we can solve this with Dijkstra's algorithm on the graph with vertices $V \times 2^{\{1,2,\ldots,7\}}$: i.e. where are we at and which IDs have we already collected?

Colour Coding

For up to 1000 different IDs, randomly map each distinct ID to $\{1, 2, \ldots, 7\}$. The probability the 7 distinct IDs collected by the optimum solution map to 7 distinct colours is $\frac{7!}{7^7} \approx 0.006$.



Unfortunately, this isn't good enough because the probability is too small. Would need too many repeated trials to make this work, eg. even 100 trials wouldn't bring the success probability up to 50%.

Boosting the Probability with More Colours

Instead of 7 colours, use more.

Randomly map each ID to a value in $\{1, 2, \ldots, 15\}$. The graph size increases to ≈ 3.2 million nodes and ≈ 32 million edges, but that's still doable multiple times within 14 seconds.

Find the cheapest path to a state $(v, S) \in V \times 2^{\{1,2,\ldots,15\}}$ with $|S| \geq 7$.

The probability the 7 balls collected by the optimum solution are mapped to different colours is much better: $\frac{15!}{(15-7)!} \cdot \frac{1}{15^7} \approx 0.189$. Repeating 30 times yields a 99.8% chance of success.

Topic: Designing a good hash function

RMC 2020 - Typo

`https://open.kattis.com/problems/typo`

One idea is to use a hashing scheme mapping strings to integers such that for any string t and any index i, we can quickly compute the hash of string resulting from deleting the i'th character from t.

If so, then do this for the 10^6 characters across all strings and see if any of the resulting hashes equals the hash of an input string. Linear time!

But how to do the hashing?

View the string s as a polynomial $h_s(x)$.

Coefficients == ascii values of the characters.

Example: $s = \texttt{apples}$

$$h_s(x) = \texttt{a} \cdot x^0 + \texttt{p} \cdot x^1 + \texttt{p} \cdot x^2 + \texttt{l} \cdot x^3 + \texttt{e} \cdot x^4 + \texttt{s} \cdot x^5$$

Two polynomials are distinct if and only if their corresponding strings are distinct.

Pick a large prime P and sample a single random integer $z \neq 0 \pmod{P}$.

The hash of string s is then the value $h_s(z) \pmod{p}$.

Effectiveness

- For two distinct strings s, t of length at most d, their polynomials have degree at most d. If $P >$ 'Z', the polynomials will remain distinct $\pmod{P}$.
- Then $h_s(x) - h_t(x) \pmod{P}$ will still be a nonzero polynomial. Thus, will have at most d roots.
- **Therefore**: the probability s and t hash to the same value (i.e. $h_s(z) \equiv h_t(z) \pmod{P}$) is at most $\frac{d}{P-1}$.

We can pick the prime P to be close to 2^{63} if we work with 128-bit integers, i.e. `__uint128_t` or `__int128_t`

Flexibility

We can compute all hashes for **all** prefixes of s in $O(|s|)$ time.

Then, to compute the hash after deleting a single character i just do a bit of algebra using these hashes and modular inverses.

Example: Deleting the letter `l` from `apples`. The hash is just:

$$h_{\texttt{appes}}(z) = \texttt{a} \cdot z^0 + \texttt{p} \cdot z^1 + \texttt{p} \cdot z^2 + z^{-1} \cdot (\texttt{e} \cdot z^4 + \texttt{s} \cdot z^5)$$

$$= z^{-1} \cdot (h_{\texttt{apples}}(z) - h_{\texttt{app}}(z)) + h_{\texttt{app}}.$$

If we precomputed $z^{-1} \pmod{P}$ and precomputed the hashes of all prefixes of strings, computing the hash of `appes` takes $O(1)$ time.

Can quickly hash other string manipulations this way (eg. deleting an entire substring, concatenating two strings).

But will this be good enough to avoid all false positives (i.e. hash collisions)?

Quick reasoning to convince yourself it is good

With typos, we have up to 10^6 input strings and 10^6 strings we try deleting a character from.

Each string has length $\leq 10^6$ so the probability of a false positive between an input string and a single string we get by deleting a character is at most $\frac{10^6}{P}$.

Therefore, the probability that some comparison is bad is at most (# pairs of strings we are worried about colliding) $\times \frac{10^6}{P} \leq \frac{10^{18}}{P}$.

This analysis is very coarse since we cannot simultaneously have 10^6 strings in the input and have a large number of them with length 10^6. So it will be much better!

Another way to boost the probability of correctness is to consider sampling a few different z and running the hash checks for each z.

With some problems, you can even remove doubt by explicitly doing string comparison when a collision. For example, with Typo false positives are so rare and you can stop if you do find a typo so you can afford to check if the strings are == if their hashes collide.

There are many other problems that can be solved using string hashing. Even **KMP** itself can be replaced by string hashing. It is really powerful!

Topic: Breaking patterns in data

Buckeye Programming Competition 2022 - Spooky Scary Skeletons
`https://buckeyecode.club/problem/view/bpc22spookyscaryskeletons`

1 2 1 4 2 3 1 3

spooky spooky

not spooky

We can compute the bitwise XOR of any subarray by maintaining prefix XOR sums.

1. If all integers appeared an even number of times in the subarray, its XOR would be 0.

   1 2 1 4 5 3 1 5 2 3 1 4 XORS to 0

2. If there was exactly one integer appearing an odd number of times, the subarray XOR would be that integer.

   5 1 5 6 5 1 6 5 6 1 1 XORS to 6

3. But it is not so clear what we get if there are multiple integers appearing an odd number of times.

Simple idea: Map each integer b in the input to a random random 64-bit value $h(b)$.

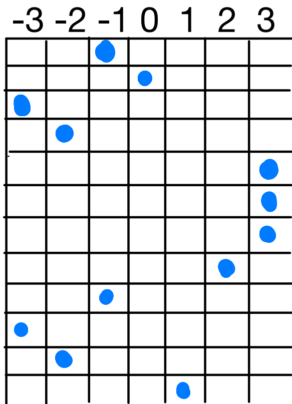
For any query as in Case 3, the XOR is now very unlikely one of the $h(b)$ values: at most $\frac{n}{2^{64}}$.

Over all q queries, the probability at least one of them fails is then at most $\frac{Q \cdot N}{2^{64}}$. Small enough for this range of Q and N.

Topic: Random Walks

300iq Singapore Contest (2019) - Zero Sum

`https://qoj.ac/contest/947/problem/4238`



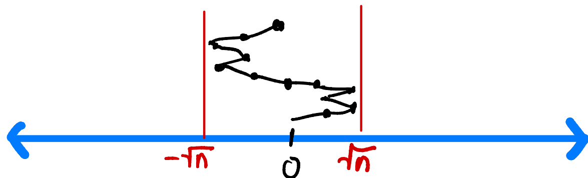
n = 35,000 rows

Standard dynamic programming: let $f(i,c)$ be the best way to select from the first i rows such that the column index sum is c.

$$f(i,c) = \min_{-k \le j \le k} A[i][j] + f(i-1, c-j).$$

Best we can say for c is $|c| \le n \cdot k/2$. So this takes $O((n \cdot k)^2)$ time. Too slow!

Concept: If you add n random ± 1 values, $|\text{sum of values}|$ is probably about $O(\sqrt{n})$. That is, the standard deviation is $\sqrt{n}$.



Generalizing, if the values are random between $-k$ and k, the standard deviation is $k \cdot \sqrt{n}$.

Idea: Randomly shuffle the rows. We can think[1] of the column indices in the optimal solution as being random from $-k$ to k.

So now c only needs to index between, say, $\pm 10 \cdot \sqrt{n} \cdot k$ and the DP algorithm now runs in $O(n^{1.5} \cdot k^2)$ time: fast enough!

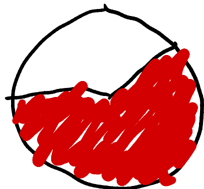
---

[1] This isn't completely precise since the column indices are dependent, but one can make this "heuristic" argument more formal.

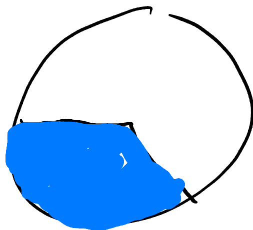
KTH Challenge 2014 - Pizza Problems

`https://open.kattis.com/problems/pizzaproblems`

∃ awesome pizza

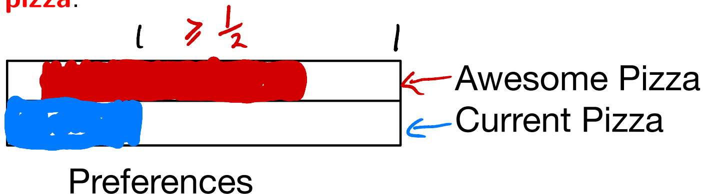


Find a good pizza



Let's say the pizza we forgot about is the **awesome pizza**.

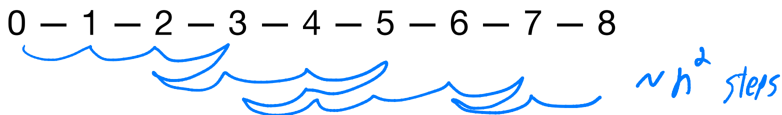
Notice for any pizza that satisfies $\leq 1/3$ of someone's preferences, at least half of their unsatisfied preferences were satisfied by the **awesome pizza**.



So if we pick one of the unsatisfied preferences randomly and toggle that topping, our current pizza gets one step closer to the **awesome pizza**.

Iterate this until everyone is $> 1/3$ satisfied by your proposed pizza.

Why does this work? A random walk on a path $0, 1, \ldots, n$ will take at most n^2 steps in expectation to reach node n.



In our setting, the nodes on the "path" indicate how many toppings are in agreement between our pizza and the **awesome pizza**.

Each iteration, we have at least a 1/2 chance of agreeing more with the awesome pizza i.e. taking a random step to the right in this graph.

With only 250 toppings, the expected number of topping changes is at most 250^2. Fast!

Randomized Algorithms Courses

Everything I talked about should be covered in a standard course on randomized algorithms. While most offerings get more theoretical than what we need in CP, they do provide a good foundation for thinking this way.

Consider perusing the notes of a randomized algorithms course or a textbook in this topic.

(tiny font so they fit... just click on them)

https://www.cs.cmu.edu/afs/cs/academic/class/15859-f04/www/

https://ocw.mit.edu/courses/6-856j-randomized-algorithms-fall-2002/pages/lecture-notes/

https://www.amazon.com/Randomized-Algorithms-Rajeev-Motwani/dp/0521474655

https://www.amazon.com/Probability-Computing-Randomized-Algorithms-Probabilistic/dp/0521835402/

Short list of other useful facts + **keywords** to search

- **Markov Chain of an Undirected Graph**: The expected number of steps for a random walk (starting anywhere) to reach all nodes is at most $2 \cdot |V| \cdot |E|$.

- **Schwartz-Zippel Lemma**: If you plug random values into the variables of a nonzero, degree-d multi-variate polynomial, the probability it vanishes is at most $\frac{d}{|S|}$ where S is the set you sampled values from.

- **Tutte Matrix**: For a graph G (not necessarily bipartite), pick random values $z_e \pmod P$ for the edges. Construct the adjacency matrix except use $A_{u,v} = z_e$ and $A_{v,u} = -z_e$ for each edge $e = (u, v)$ (the order of u and v doesn't matter).

  Then $\det(A) \pmod P$ will always be 0 if G has no perfect matching, and will be nonzero with probability at least $1 - |V|/P$ if G has a perfect matching.