Lecture 11: Streaming (Count Min Sketch and HyperLogLog Counting)

Sam McCauley

October 21, 2021

Williams College

Admin

- Assignment 4 out already
- Catching up on grading next few days
- Monday will be all about practice. (Murmurhash and practical hashing; Assignment 2 and Mini Midterm review; some efficiency techniques for Assignment 5.)
- Assignment schedule through end of semester posted
- Questions?

Really Large Data

- Netflix sends (so far as I can tell) about 500TB per minute on average to its customers
- Google's search index is over 100,000,000 GB
- Brazil Internet Exchange processes 7 trillion bits every second

Really Large Data

- Modern companies deal with extremely large data
- Can't even store all of it sometimes!
- If is possible to store, can be very difficult to access particular pieces

A Shift in Focus (Streaming)

- Up until now: nice self-contained instances; might fit in L3 cache; might fit in RAM
- In some situations: can't hope to do that
- Like you're sitting next to a stream that's constantly rushing past
- All you can do is sample pieces as they pass by

Streaming Model

- You receive a “stream” of N items one by one
- Stream is incredibly long; you can’t store all of the items
- Can’t move forward or backward either; just come in one at a time

Streaming Model

- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want
- You can only store $O(\log N)$ bytes of space, or maybe even $O(1)$
- What can we do in this situation?
- Note: very active area of research
- Today we'll look at two classic results

What We Really Want

- Much more extreme "compression" than a filter
- (Filter used a constant number of bits per item; we can't afford that)
- Today: two data structures
  - Count-min sketch: More aggressive than a filter. Good guarantees for counting how many times a given element occurred in a stream.
  - HyperLogLog: Only uses a few bytes. Estimates how many unique items appeared in the stream.

When to Use Streaming Algorithms?

- Data streams: network traffic, user inputs, telephone traffic, etc.
- Cache-efficiency! Streaming algorithms only require you to scan the data once.
    - N/B cache misses

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often
- Keep track of popular passwords
- Google uses an improved HyperLogLog to speed up searches
- Reddit uses HyperLogLog to estimate views of a post
- Facebook uses HyperLogLog to estimate number of unique visitors to site.

HyperLogLog at Facebook

FACEBOOK Engineering

POSTED ON DECEMBER 13, 2018 TO DATA INFRASTRUCTURE, OPEN SOURCE

HyperLogLog in Presto: A significantly faster way to handle cardinality estimation

"Doing this with a traditional SQL query on a data set as massive as the ones we use at Facebook would take days and terabytes of memory... With HLL, we can perform the same calculation in 12 hours with less than 1 MB of memory."

Count-Min Sketch

Count-Min Sketch

Goal:

- Maintain a data structure on a stream of items
- At any time, estimate how frequently a given item appeared

Example

You see the following items one by one:

adhesiveflawlessclosedadhesivedescrib

Example

- Now, answer questions of the form: how many times did some item x_i occur in the stream?

- Example: how many times did adhesive appear? How about closed?
    - (2 times and 3 times respectively)

Formally

- See a stream of elements $x_1, \dots x_N$, each from a universe U[1]
- For some element $q \in U$, estimate how many i exist with $x_i = q$?
- Today: pretty decent guess using $\left\lceil \frac{e}{\varepsilon} \right\rceil \lceil \ln(1/\delta) \rceil \lceil \log_2 N \rceil$ bits of space
    - ε and δ are parameters we can use to adjust the error
    - Don't depend on N, or $|U|$

[1]Like in the last lecture, this is just a requirement to make sure that we can hash them.

How would you do this with what you know right now?

- Keep a hash table with all elements
- Increment a counter each time you see an element
- $O(N)$ space, $O(1)$ time per query
- Pretty efficient! But we want way way less space.

Sketching: A first attempt

- Randomly sampling:
  - Keep $N/100$ slots
  - For each item, with probability $1/100$, write the item down.
- If an item appears k times in the stream, we see it $k/100$ times in expectation.

Sketching: A first attempt

- If an item appears k times in the stream, we see it $k/100$ times in expectation.
- So, if we wrote an item down w times, we can estimate that it probably occurred $100w$ times in the stream.

Sketching: A first attempt

What are some downsides to this approach?

- It's pretty loose. If our counter is just one off, that changes our guess by $+100$
- Could have a fairly frequent item that we never write down.
- Can't guarantee much about our estimate

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries of at least $\lceil \log N \rceil$ bits
- Hash function h for A
- When we see an item x_i:
  - Increment $A[h(x_i)]$
- How can we query?

Counters of length $\lceil \log N \rceil$ to have room

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences
  - How much does it overestimate by?
  - Each of N items hashes to same slot with probability ε, so $N\varepsilon$ in expectation

Since we always increase this counter when we see $x_i = q$

But, also increase it when $h(x_i) = h(q)$, but $x_i \neq q$

Second attempt: hash counts

Expectation is not that great!

- Let's say we have two items; A appears 100 times and B appears 900
- Query A: with probability $1 - \varepsilon$ we get 100; with probability ε we get 1000

What do we really want?

- To guarantee a high-quality answer, we want to say that the solution is *likely* to be close to correct.
    - We want concentration bounds!
- How can you increase the reliability of a random process?
- For example, let's say we're rolling a die. We want to be sure we see a 6 at least once. How can we do that?
- Of course: roll the die many times!

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T
- T has $\lceil \ln(1/\delta) \rceil$ rows
- Each row consists of $\lceil e/\varepsilon \rceil$ slots
- Different hash function for each row

We'll come back to δ later.

The e is important for the analysis.

Inserts

To insert x_i:

- For $j = 0 \dots \lceil \ln(1/\delta) \rceil - 1$:
  - Increment $T[j][h_j(x_i)]$

We now have $\lceil \ln(1/\delta) \rceil$ counters for each item. How can we query?

Queries

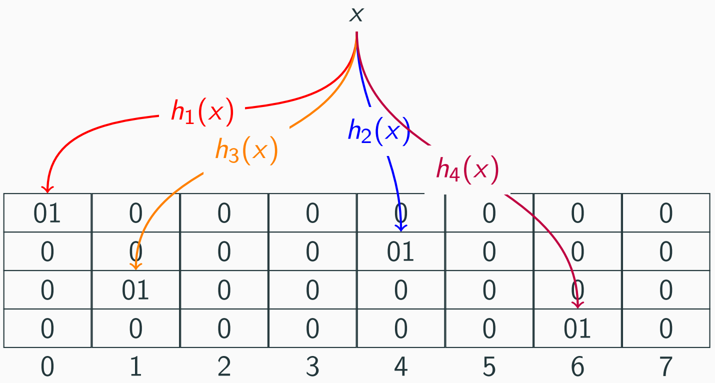
Each entry is an *overestimate*.

- Find $\min_j T[j][h_j(x_i)]$.

Count-Min Sketch

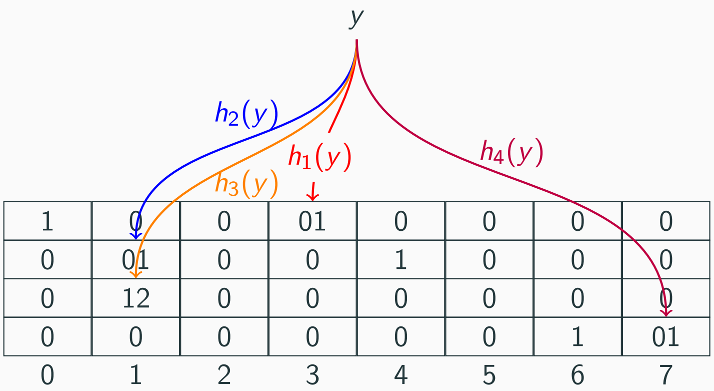
- Table T with $\lceil \ln(1/\delta) \rceil$ rows, each with $\lceil e/\varepsilon \rceil$ columns. Cells of size $\lceil \log N \rceil$
- $\lceil \ln(1/\delta) \rceil$ hash functions; one for each row
- To insert x: increment $T[j][h_j(x)]$ for all $j = 0, \ldots \lceil \ln(1/\delta) \rceil - 1$
- To query q: return $\min_{j \in \{0, \ldots, \lceil \ln(1/\delta) \rceil - 1\}} T[j][h_j(q)]$

Example Insert



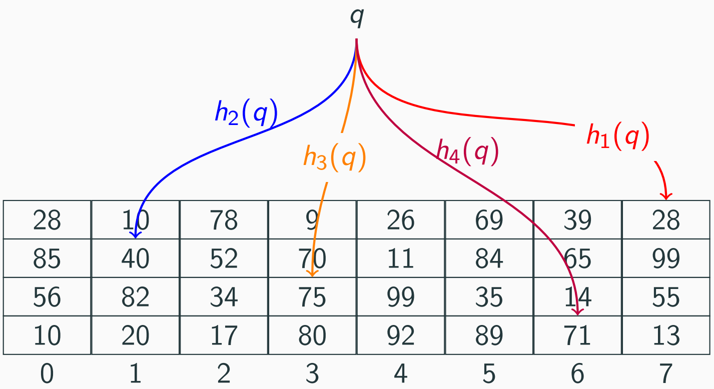
01	0	0	0	0	0	0	0
0	0	0	0	01	0	0	0
0	01	0	0	0	0	0	0
0	0	0	0	0	0	01	0
0	1	2	3	4	5	6	7

Example Insert



1	0	0	01	0	0	0	0
0	01	0	0	1	0	0	0
0	12	0	0	0	0	0	0
0	0	0	0	0	0	1	01
0	1	2	3	4	5	6	7

Example Query



28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

The estimated number of occurrences for q is 28.

Count-Min Sketch Guarantee: Lower bound

So $o_q = \min_j T[j][h_j(q)]$

- On query q, let's say the filter returns that there were o_q occurrences

- In reality, the correct answer is $\widehat{o_q}$ occurrences

- First: always have $\widehat{o_q} \leq o_q$.

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.
- We know that for any j, $\mathrm{E}[T[j][h_j(q)]] \leq \widehat{o_q} + \frac{\varepsilon N}{e}$
- That is to say: guess is off by $\frac{\varepsilon N}{e}$ in expectation
- On Assignment 4, you'll prove that for any positive random variable X, $\Pr[X \geq e\mathrm{E}[X] \leq 1/e]$
- So the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?
  - Yes. (For each row, we select a new hash and start over)
- What is $\Pr\left[\min_j T[j][h_j(q)]\right] \geq \widehat{o_q} + \varepsilon N$?
- Only fails if cell is too big in every row! Occurs with probability

$$\left(\frac{1}{e}\right)^{\#\text{ rows}} = \left(\frac{1}{e}\right)^{\lceil \ln 1/\delta \rceil} \leq \delta$$

Count-Min Sketch Bounds

- $\left\lceil \frac{e}{\varepsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil \lceil \log_2 N \rceil$ bits of space

- For any query q, if the filter returns o_q and the actual number of occurrences is $\widehat{o_q}$, then with probability $1 - \delta$:

$$\widehat{o_q} \leq o_q \leq \widehat{o_q} + \varepsilon N.$$

Count-Min Sketch

- Small sketch (size based on error rate)
- Always overestimates count
- Bound on overestimation is based on stream length

Parameters in Assignment CMS

- 300 entries in each row, 4 rows
- 32-bit counters (wasteful!)
- 7.3MB of data summarized in 4.8KB
- Really accurate still: in 1.2 million word stream, can estimate num occurrences of each word within ± 1500
- Often more accurate! Also: feel free to try 1000 or 10000 entries per row; it gets quite accurate

Hyper Log Log Counting

Setting up

- Count-min sketch takes up a lot of space!
- OK not really. But, it stores a lot of information about the stream
- Common question: how many unique elements are there in the stream?

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements
- (Compare to CMS: stores approximately how many there are of *each* element)

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements
- To do this exactly: need dictionary of all elements we've already seen.
- How can you count unique elements approximately? Challenge: don't want to double-count when we see an element twice.

Cool way to solve this

- Let's hash each item as it comes in
- Then instead of a list of items, we get a list of random hashes
- Idea: let's look at a rare event in these hashes. The more often it happens, the more distinct hashes we must be seeing!
- In particular: how many 0s does each hash end with?

Hashes ending in 0s

- What is the probability that a hash ends in 10 0's? Answer: 1/1024
- So if we only see two distinct hashes, it's very unlikely that either will end in 10 0's.
- If we see $2^{10} = 1024$ distinct hashes, it's pretty likely that one will end with 10 0's.
- Note "distinct!" All of this comes back to estimating how many *unique* elements there are. Unique elements give a new hash, and a new opportunity for many zeroes. Non-unique elements don't give a new hash.

Example

You see the following hashes one by one:

0010001010101001001011001011110

How many unique items were there?

Example 2

You see the following hashes one by one:

0010001010101001001011001011110

How many unique items were there? Was it more or less than the last one?

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3
- Notice that only one hash in the second example ended with 0
  - Extremely unlikely if there were 14 different elements!
- One of the items in the first example ended with 4 0's
  - Unlikely if there were 3 elements!

Intuitive loglog counting

- Let's say that the hash ending with the most 0s has k 0s at the end
- Any given hash has k 0s with probability $1/2^k$
- So it seems that, there are probably something like 2^k items
- But if we're just off by 1 or 2 zeroes, that affects our answer by a lot!

Improving reliability

- How do we improve the consistency of a random process? Repeat!
- Hash each item first to one of several counters
- For each counter, keep track of $1 +$ the maximum number of 0s at end of any item hashed to that counter
- For CMS, we took the min. What do we do here to combine the estimates?
- Answer: It's complicated. (And outside the scope of the course.)

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M
- Hash each item as it comes in. Then:
  - Get an index i, consisting of the lowest $\log_2 m$ bits of $h(x)$. Shift off these bits.
  - Look at the remaining bits. Let z be the number of zeroes. If $z + 1 > M[i]$, set $M[i] = z + 1$
- **Make sure to add 1 to your count of the number of zeroes**

Getting an Estimate

- At the end, we have an array M, each containing a count
- Let
$$Z = \sum_{i=0}^{m-1} \left(\frac{1}{2}\right)^{M[i]}.$$
- Let b be a bias constant.[2] For $m = 32$, $b = .697$.
- Return bm^2/Z.

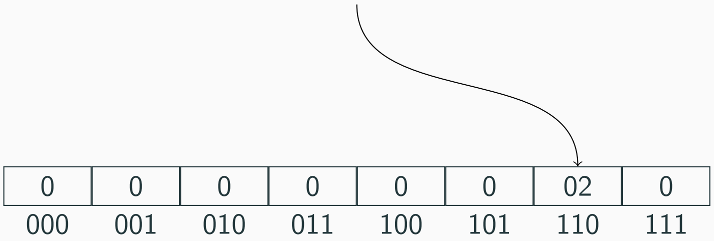
[2]You have to look this constant up.

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 01000100011111011111101010110$$

index $= 110$ Remaining: $01000100011111011111101010$



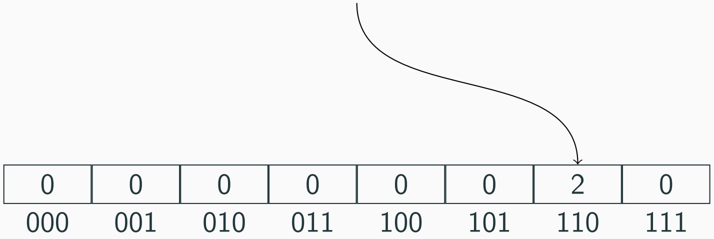
The remaining hash ends with 1 zero, so we want to store 2. The counter stores less than 2, so we store it.

Example (with $m = 8$; in practice m is higher)

x_2

$h(x_2) = 01111000110010000111101001 0110$

index $= 110$ Remaining: $01111000110010000111101001 0$



0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

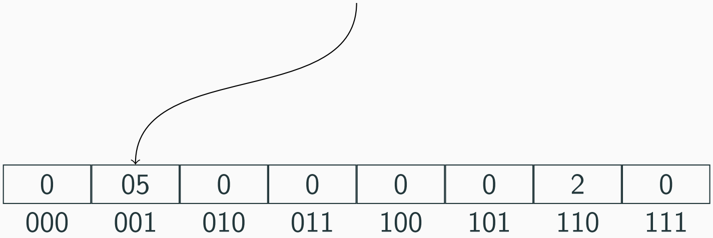
The remaining hash ends with 1 zero, so we want to store 2. The counter stores 2, so we keep it as-is.

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 11001101110110000001101000000 1$$

index = 001 Remaining: 11001101110110000001101000000



The remaining hash ends with 4 zeroes, so we want to store 5.

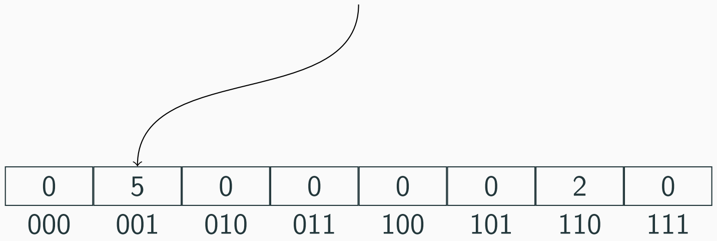
The counter stores 0, so we store 5 in the slot.

Example (with $m = 8$; in practice m is higher)

$$x_4$$

$$h(x_4) = 100010011101101110110110111001$$

index $= 001$ Remaining: $100010011101101110110110111$



The remaining hash ends with 0 zeroes, so we want to store 1.

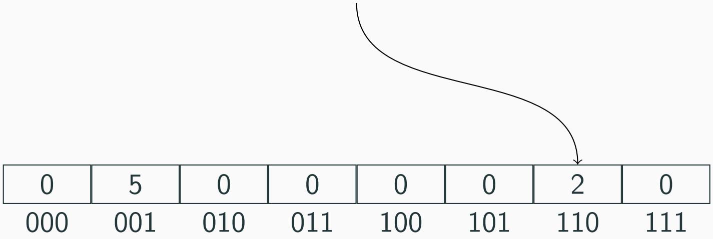
The counter stores 5, so we keep the slot as-is.

Example (with $m = 8$; in practice m is higher)

x_2

$h(x_2) = 01111000110010000111101 0010110$

index $= 110$ Remaining: 011110001100100001111010010110



The remaining hash ends with 1 zero, so we want to store 2. The counter stores 2, so we keep it as-is.

At the end of the day

Have an array of counters:

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

- Sum up $(1/2)^{M[j]}$ across all $j = 0$ to $m - 1$; store in Z
- Return bm^2/Z. Here $m = 8$. We would have to look up the value of b for 8. (No one does HyperLogLog with 8)

Discussion

- How big do our counters need to be?
- Need to be long enough to count the longest string of 0s in any hash
- Size $> \log\log$(number of distinct elements) (hence the *loglog* in the name)
- 8-bit counters are good enough, so long as the number of elements in your stream is less than the number of particles in the universe
- Note: one thing to be careful of is hash length. But 64 bit hashes should be good enough for any reasonable application (and 32 bits is usually fine)

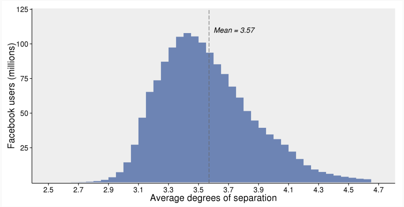
HLL in the Assignment

- We'll use $m = 32$ counters
- Bias constant is .697

HLL Beyond the Assignment

- HLL does poorly when the number of distinct items is not much more than m
- Or is very very high
- Google developed HyperLogLog++ to help deal with these problems
- Other known improvements as well

One More Cool Thing



- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph
    - In terms of “friend jumps”, how far away are the furthest people in the Facebook graph?
    - How far away are two people on average?
- Usually takes $O(n^2)$ time!
- Theirs is essentially linear time, gives extremely accurate results

Let’s Look at Assignment 4

Assignment 2 Optimizations

Assignment 2

- Lots of cool ideas!
- Some seem very nice but don't speed things up much.

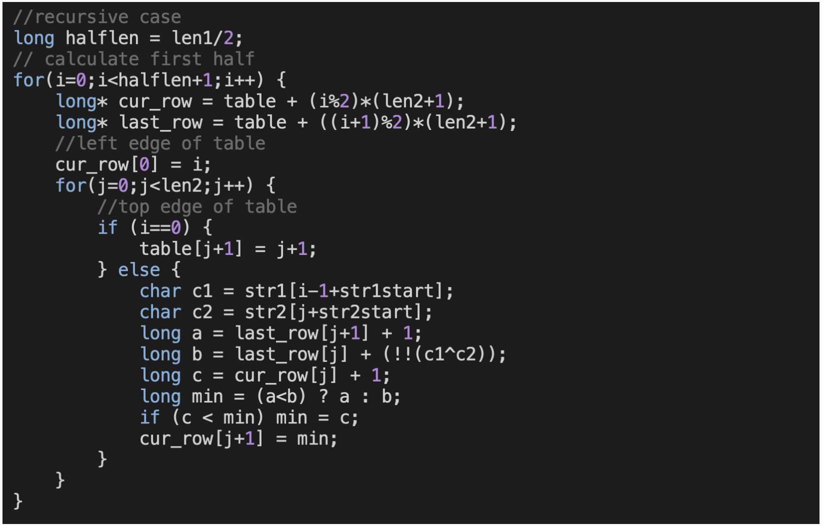
Assignment 2: Some ideas that seemed to work

- Large base case
  - Why is this good? (Hint: why was Hirshberg's a good idea in the first place?)
  - 300, 2048 both used
  - Both small enough that `len1 * len2` fits in cache
- Iterative version!
  - Recursive calls have overhead; can skip them
  - To be honest this seems like it should be a lower-order term to me
- Don't reverse strings?
  - Just doing the DP backwards might be faster(?)

Cool ideas that didn't speed things up much

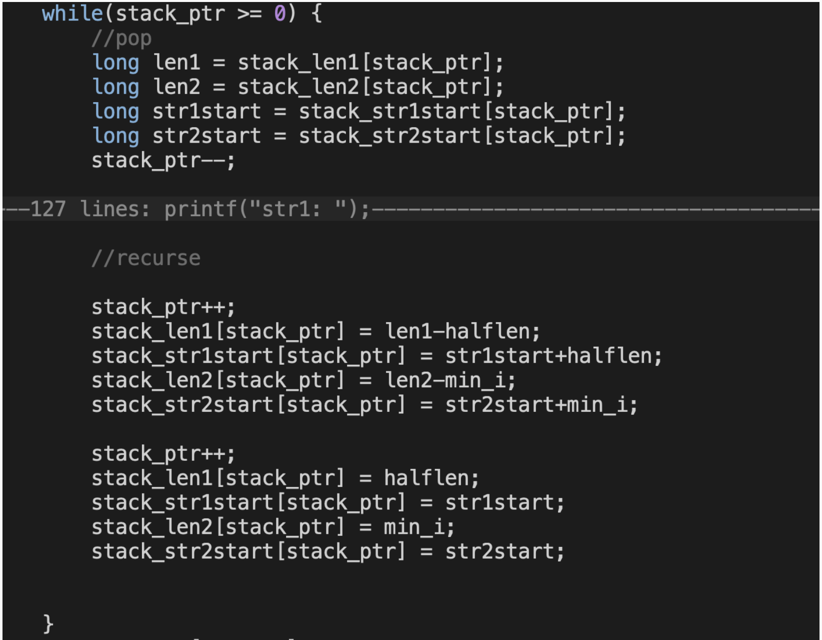
- Using 1 array instead of 2 when calculating the edit distance value
- Getting fancy with calculating the min
  - Two if statements seems fastest
  - I think: compiler is really good at optimizing that.
- Easiest way of passing around solution seemed to be continuously appending each piece as it's found

Code from best last year (calculating costs)



```
//recursive case
long halflen = len1/2;
// calculate first half
for(i=0;i<halflen+1;i++) {
    long* cur_row = table + (i%2)*(len2+1);
    long* last_row = table + ((i+1)%2)*(len2+1);
    //left edge of table
    cur_row[0] = i;
    for(j=0;j<len2;j++) {
        //top edge of table
        if (i==0) {
            table[j+1] = j+1;
        } else {
            char c1 = str1[i-1+str1start];
            char c2 = str2[j+str2start];
            long a = last_row[j+1] + 1;
            long b = last_row[j] + (!!(c1^c2));
            long c = cur_row[j] + 1;
            long min = (a<b) ? a : b;
            if (c < min) min = c;
            cur_row[j+1] = min;
        }
    }
}
```

Code from best last year (iterative Hirschberg's)



```
while(stack_ptr >= 0) {
    //pop
    long len1 = stack_len1[stack_ptr];
    long len2 = stack_len2[stack_ptr];
    long str1start = stack_str1start[stack_ptr];
    long str2start = stack_str2start[stack_ptr];
    stack_ptr--;

---127 lines: printf("str1: ");------------------------------------

    //recurse

    stack_ptr++;
    stack_len1[stack_ptr] = len1-halflen;
    stack_str1start[stack_ptr] = str1start+halflen;
    stack_len2[stack_ptr] = len2-min_i;
    stack_str2start[stack_ptr] = str2start+min_i;

    stack_ptr++;
    stack_len1[stack_ptr] = halflen;
    stack_str1start[stack_ptr] = str1start;
    stack_len2[stack_ptr] = min_i;
    stack_str2start[stack_ptr] = str2start;


}
```