451: Amortized Analysis

G. Miller, K. Sutner
Carnegie Mellon University
2020/09/15

There are two standard ways of assessing running time:

Worst-Case Analysis: Assume everyting goes wrong in the worst possible way.

Average-Case Analysis: Assume a probability distribution on the instances, and take the average of the running times.

In a sense, average-case is more interesting, but significantly more challenging. For example, `quicksort` would be entirely useless if it were not for excellent average case behavior.

Any reasonable model $\mathcal{M}$ of computation comes with a notion of a "single step." Hence we can define

$$T_{\mathcal{M}}(x) = \text{length of computation of } \mathcal{M} \text{ on } x$$

This is perfectly fine in principle, but causes a bit of friction in the RealWorld$^{\mathrm{TM}}$: mathematical models of computation don't quite match the notion of "single step" in a real digital computer.

We will exploit this shamelessly and systematically ignore multiplicative constants in order to simplify calculations. Knuth might have some objections, but that's not fair.

Counting steps for individual inputs is often too cumbersome, one usually lumps together all inputs of the same size:

$$T_{\mathcal{M}}(n) = \max\big(\, T_{\mathcal{M}}(x) \mid x \text{ has size } n \,\big)$$

This is worst case complexity. Alternatively we could try to determine

$$T^{\text{avg}}_{\mathcal{M}}(n) = \sum\big(\, p_x T_{\mathcal{M}}(x) \mid x \text{ has size } n \big)$$

the average case complexity, where p_x is the probability of instance x.

BTW, getting probability distributions that conform to practical use is often very hard. Smale got a lot of flack for his analysis of the simplex algorithm.

In most algorithms one has to deal with a sequence π of operations (say, push and pop on a stack):

$$\pi : \pi_1, \pi_2, \pi_3, \ldots, \pi_{m-1}, \pi_m$$

Each operation transforms the underlying data structure and has some associated cost $\text{cost}(\pi_i)$.

$$S_0 \xrightarrow{\pi_1} S_1 \xrightarrow{\pi_2} S_2 \xrightarrow{\pi_3} \ldots \xrightarrow{\pi_{m-1}} S_{m-1} \xrightarrow{\pi_m} S_m$$

It may very well happen that some of the operations are quite expensive, but overall the cumulative cost is not bad, since most operations are cheap. What really matters is the overall cost of the whole sequence π, rather than the individual costs.

Neither worst-case nor average-case analysis addresses this properly.

Consider a sequence

$$S_0 \xrightarrow{\pi_1} S_1 \xrightarrow{\pi_2} S_2 \xrightarrow{\pi_3} \ldots \xrightarrow{\pi_{m-1}} S_{m-1} \xrightarrow{\pi_m} S_m$$

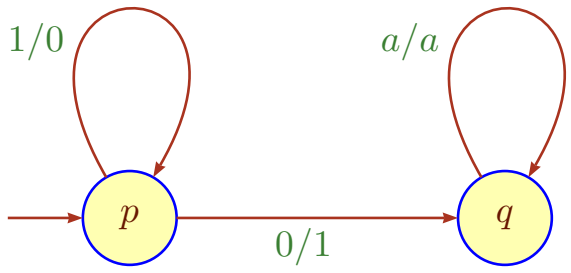
It makes sense to compute the amortized cost of the operations. For a single sequence π this simply means

$$\mathrm{cost_a}(\pi) = \frac{1}{m} \sum_{i=1}^{m} \mathrm{cost}(\pi_i).$$

Alas, for this to be useful we have to deal with all possible sequences. Typically, we need to make sure that every expensive operation must be preceded by lots of cheap ones.

So this is also a kind of average, but requires no probability distribution.

Here is a simple Mealy machine:



This transducer implements the successor function on binary strings (LSD first). E.g.,

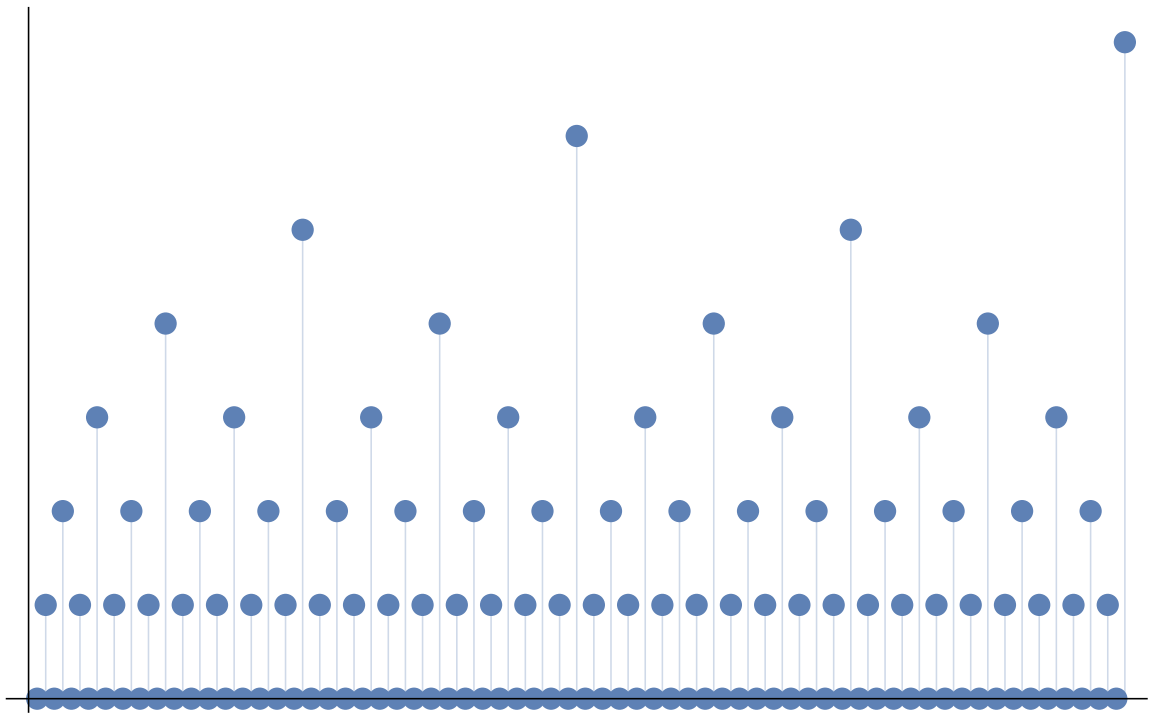
$$11110010 \rightsquigarrow 00001010$$

Used as a counter on k-bit strings, this resets at $2^k - 1 \rightsquigarrow 0$.

If the bits are stored in an array, we could just operate in place and stop when state q is reached (or we run out of bits).

In this array setting, how many steps does it take to count to n? Assuming, of course, $n < 2^k$.

The obvious simple-minded answer is $O(n \log n)$. But that's a wild overestimate:



Counting more carefully (and ignoring floors) we get

- first bit changes $n/2$ times
- second bit changes $n/4$ times
- ...
- the ith bit changes $n/2^i$ times.

For simplicity assume $n = 2^k$. Total number of bit-flips:

$$\sum_{i \leq k} i\, n/2^i = 2^{k+1} - k - 2 \approx 2n$$

Thus, the amortized cost is 2 per operation.

Note that there is only one sequence of operations of length n in this case, so this is deceptively simple.

For a binary counter, essentially the same argument also establishes the average cost of an increment operation, assuming a uniform distribution over all possible counter values (sounds a bit fishy, but it won't matter in this case).

$$\mathsf{cost}_{\mathsf{avg}} = 1/2^k \sum_{i=1}^{k} i\, 2^{k-i} = \sum_{i=1}^{k} i\, 2^{-i} = \frac{2^{k+1} - k - 2}{2^k} \approx 2.$$

So this is essentially the same as amortized.

Warning: this is emphatically not the case in general.

One can implement a queue Q using two stacks A and B:

- **enqueue** by pushing into A,
- **dequeue** by popping from B;
  if B is empty, first transfer all elements from A to B.

Suppose we have a sequence of m operations on the simulated queue.

So dequeue may be $O(m)$ steps, but clearly that requires lots of enqueue operations first. Thus $O(m^2)$ is a bad overestimate.

Say we have e enqueues, d dequeues, t transfers. Assume that a single transfer has cost 2: one each for pop and push. But recall, in so many ways $2 = 1$.

We clearly have

$$e + d = m$$
$$t \leq m$$

so that

$$e + d + t \leq 3m.$$

So we have an amortized cost of 3 per operation.

Vanilla Dijkstra requires

- n inserts
- n delete-mins
- m promotions

In a standard binary heap these are all $O(\log n)$, so we get $O((m+n)\log n)$.

But with a Fibonacci heap we can get the amortized cost of promote down to constant, and the time changes to

$$O(m + n \log n)$$

There are three standard approaches to determining amortized time.

Aggregate This is just a euphemism for the brute force counting we have done so far.

Accounting "Overpay" for some operations, and use the surplus to pay for other "deficit" operations.

Potential Use a potential function to modify the actual cost, so as to simplify the calculation.

Psychology: A lot of people seem to respond very well to arguments couched in terms of finances, which makes the accounting method rather popular. Of course, it's all semantic sugar.

Say the actual cost of the ith operation is $C_i = \text{cost}(\pi_i)$. We use instead an amortized cost $C'_i = \text{cost}'(\pi_i)$ that may be larger or smaller than C_i.

- If $C'_i > C_i$ then we place the difference into a savings account.
- If $C'_i < C_i$ then we pay for the difference from the savings account.

For this to work we need the current credit to be nonnegative at all times:

$$\sum_{i \leq k} C'_i - C_i \geq 0$$

for all $k \leq m$. If an deficit operation comes along we need enough money in the bank.

Here is a way to assign amortized cost for the binary counter problem. We look at the two possible bit-flips separately.

op	C	C'	Δ
$0 \to 1$	1	2	1
$1 \to 0$	1	0	-1

This works since starting at $00\ldots00$ we need to perform a flip $0 \to 1$ before we can do a $1 \to 0$.

Table: counter, total Δ, account.

0000	1	1
1000	0	1
0100	1	2
1100	-1	1
0010	1	2
1010	0	2
0110	1	3
1110	-2	1
0001	1	2
1001	0	2
0101	1	3
1101	-1	2
0011	1	3
1011	0	3
0111	1	4
1111	-4	0
0000	-	-

Remember the clever modification that allows Dijkstra's algorithm to work on graphs with negative costs: the key idea is to force every edge to carry non-negative cost by setting

$$\mathsf{cost}_\Phi(u,v) = \mathsf{cost}(u,v) + \Phi(v) - \Phi(u)$$

using a potential function $\Phi : V \to \mathbb{R}$. Because sums telescope, this carries over to paths.

And, we can compute a suitable potential function relatively cheaply by finding shortest paths in the augmented network.

Our problem is to compute the amortized cost of a sequence of operations (analogous to a path) and do this for **all** possible sequences:

$$\frac{1}{m}\sum_{i=1}^{m}\mathsf{cost}(\pi_i).$$

At least, we need an upper bound for the sum.

Alas, in many cases, the individual cost $C_i = \mathsf{cost}(\pi_i)$ can be very erratic, so that the summation is difficult to carry out.

It would be nice to have some way to smooth things out.

We replace the actual cost by a modified cost using a potential function:

$$\Phi : \text{states} \to \mathbb{R}$$

$$\text{cost}_\Phi(\pi) = \text{cost}(\pi) + \underbrace{\Phi(S') - \Phi(S)}_{\Delta\Phi} \qquad \text{where } S \xrightarrow{\pi} S'$$

Summing over sequences of operations, we get a telescoping sum and it follows that

$$\sum_{i=1}^{m} \text{cost}_\Phi(\pi_i) = \sum_{i=1}^{m} \text{cost}(\pi_i) + \underbrace{\Phi(S_m) - \Phi(S_0)}_{\Delta\Phi}.$$

Contrary to appearances, the sum on the left may actually be easier to evaluate. And we can revover the actual cost by subtracting the difference in potential.

Note: the potential function only plays a rôle in the analysis of the algorithm, it is emphatically not part of the data structure.

Needless to say, the crucial problem with this method is to find the right potential function. This calls for an Ansatz: make a clever guess, and verify it works.

In other words, there is no simple universal method to construct potential functions, it's a bit of a black art and usually requires experimentation.

Typically we will choose Φ so that some of the terms in the actual cost function cost(π) are cancelled.

Sometimes counting things that change is a good idea.

For the binary counter from above the states of the system are simply all the k-bit numbers, and we can define

$$\Phi(S) = \text{ number of 1's in } S = \text{ digit sum of } S$$

The real cost of an increment operation is the number of initial 1's, plus one.

So suppose $S = \overbrace{111\ldots11}^{r}\,0\,x$ and let the number of 1's in x be s.

Thus $S' = \overbrace{000\ldots00}^{r}\,1\,x$ and we have

$$\begin{aligned}\mathsf{cost}_\Phi(\pi) &= \mathsf{cost}(\pi) + \Phi(S') - \Phi(S)\\ &= (r+1) + (s+1) - (r+s) = 2\end{aligned}$$

Let k be the digit sum of m and it follows that

$$\sum_{i=1}^{m} \mathsf{cost}(\pi_i) = 2m - k.$$

Thus, the amortized cost is bounded by 2.

The states are certain pairs of stacks (A, B). Actually, we only need their sizes. At any rate, define

$$\Phi(A, B) = 2 \cdot \text{ size of } A.$$

Write aA to indicate that a is at the top. Then

$$\begin{aligned}
\text{cost}_\Phi(\mathsf{enq}) &= 1 + \Phi(aA, B) - \Phi(A, B) \\
&= 1 + 2(|A| + 1) - 2|A| = 3 \\
\text{cost}_\Phi(\mathsf{deq}) &= 1 + \Phi(A, B) - \Phi(A, bB) \\
&= 1 + 2|A| - 2|A| = 1
\end{aligned}$$

or, in the transfer case,

$$\begin{aligned}
\text{cost}_\Phi(\mathsf{deq}) &= (2|Aa| + 1) + \Phi(\mathsf{nil}, A) - \Phi(Aa, \mathsf{nil}) \\
&= (2|Aa| + 1) + 0 - 2|Aa| = 1.
\end{aligned}$$

Hence the amortized cost is bounded by 3.

Think of a table data structure, say, an array. Operations are fast as long as we do not exceed the capacity of the table. We would like to have an option to, say, double the capacity (move everything over, deallocate the old structure).

We can think of this as a discrete dynamical system, with parameters:

capacity the maximum number of entries, and

size the actual number of entries.

We write n for the capacity and $s \leq n$ for the size. α denotes the load factor $s/n \leq 1$. Operations:

insert Insert an element, $s \rightsquigarrow s + 1$.

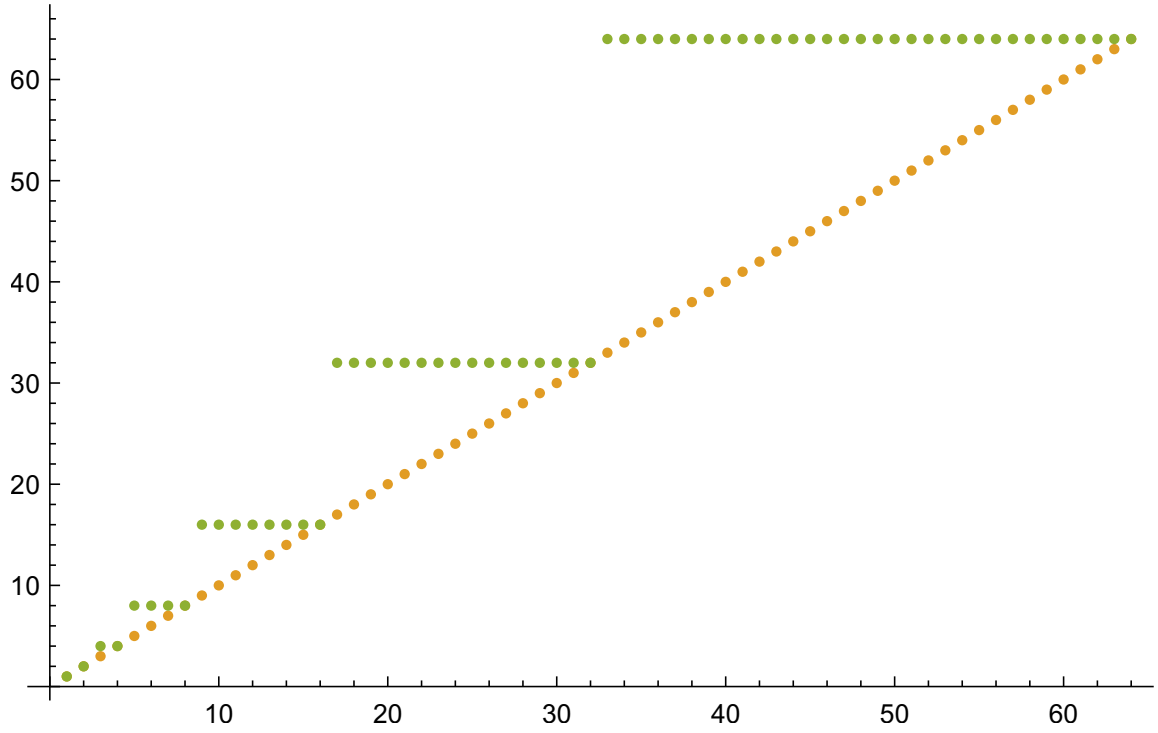
grow Grow the table, $n \rightsquigarrow 2n$.

Here we tacitly assume that the first insert creates a table with $n = s = 1$.

Since we double capacity only when the load factor is already 1 we have the invariant

$$2^{k-1} < s \leq n = 2^k.$$

So m inserts lead to state $(2^k, m)$ where $k = |m-1|$, writing $|x|$ for the number of binary digits of x.



Clearly **insert** is $O(1)$ unless a **grow** is needed: in this case we move from state $(2^k, 2^k)$ to state $(2^{k+1}, 2^k + 1)$.

grow is expensive at, say, $n = 2^k$ steps. But these operations are rare and require many inserts first. Again, we are ignoring multiplicative constants here (whose exact values we don't know to begin with).

The cost of all the **grow** operations is $\sum_{i<k} 2^i = 2^k - 1$.

Adding the m inserts, we get $2^k + m - 1 < 2^{k+1}$.

So amortized cost is still $O(1)$.

Can we use potentials for the array analysis? What is a suitable potential function?

As usual, let's try an Ansatz:

$$\Phi(n, s) = 2s - n$$

We need to analyze the modified cost for a state transition $S \xrightarrow{\pi} S'$

$$\mathsf{cost}_\Phi(\pi) = \mathsf{cost}(\pi) + \Phi(S') - \Phi(S)$$

In our case, we only have to deal with $\pi =$ **insert**. But, there really are two cases depending on whether we are at capacity or not.

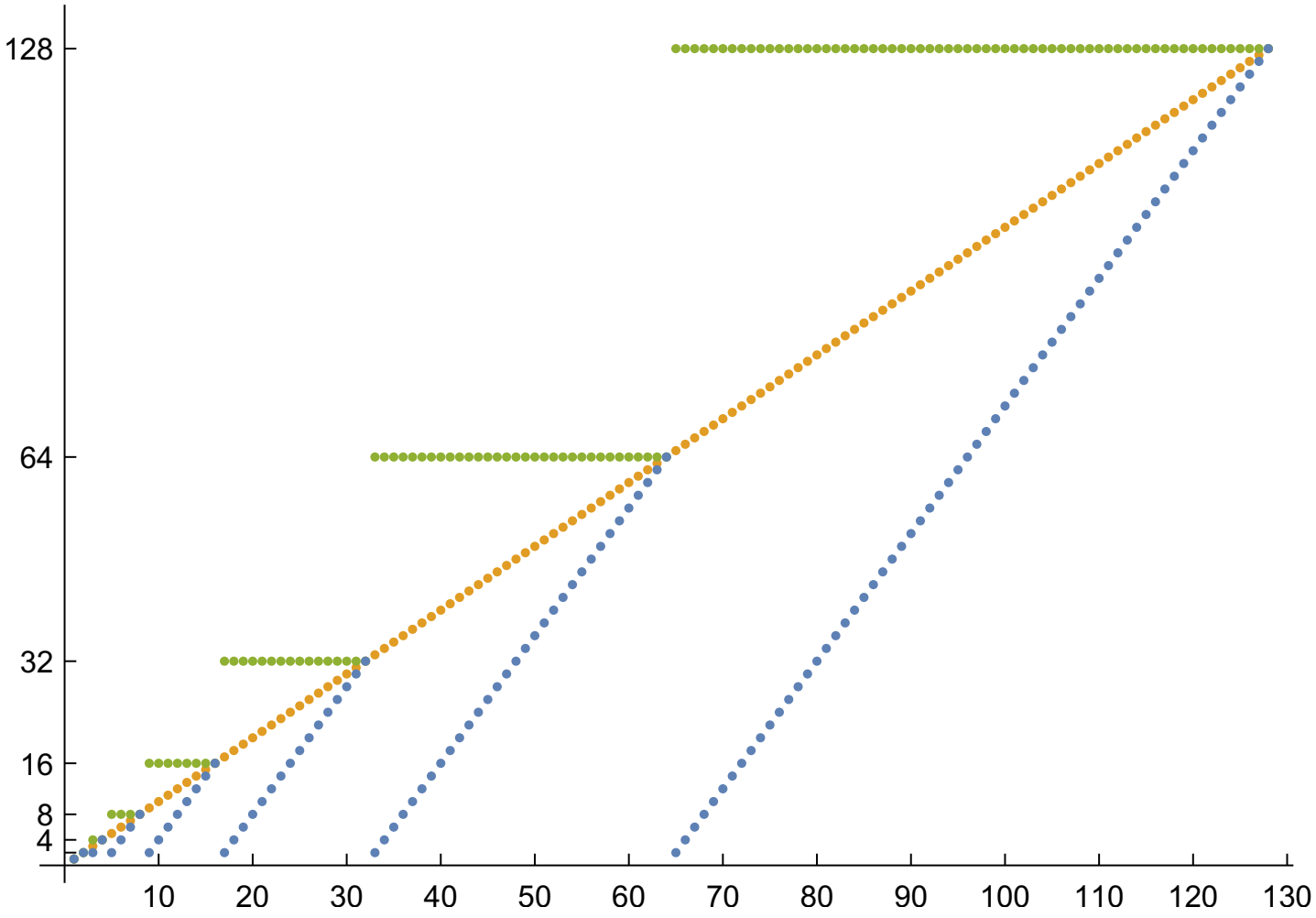
If the **insert** is plain we have $S = (n, s)$ and $S' = (n, s+1)$, so

$$\Delta\Phi = (2(s+1) - n) - (2s - n) = 2.$$

If **grow** is needed we have $S = (n, n)$ and $S' = (2n, n+1)$. Hence

$$\Delta\Phi = (2(s+1) - 2n) - (2s - n) = 2 - n.$$

This is exactly what we need: ordinary inserts push the potential up, but **grow** inserts cause it to drop (hopfully not by too much).


128
64
32
16
8
4
10
20
30
40
50
60
70
80
90
100
110
120
130

It is tempting to avoid wasting lots of memory when the load factor becomes small: we should shrink the array and free up memory.

Careful, though, it's not a good idea to adopt the following simple-minded policy:

- When $\alpha = 1$, double.
- When $\alpha < 1/2$, shrink.

Why? First perform 2^k insertions, leading to the edge of chaos. Then do

insert, **delete**, **delete**, **insert**, **insert**, **delete**, **delete**, . . .

A quadratic disaster.

Wait: Only shrink by half when the load factor drops below $1/4$.

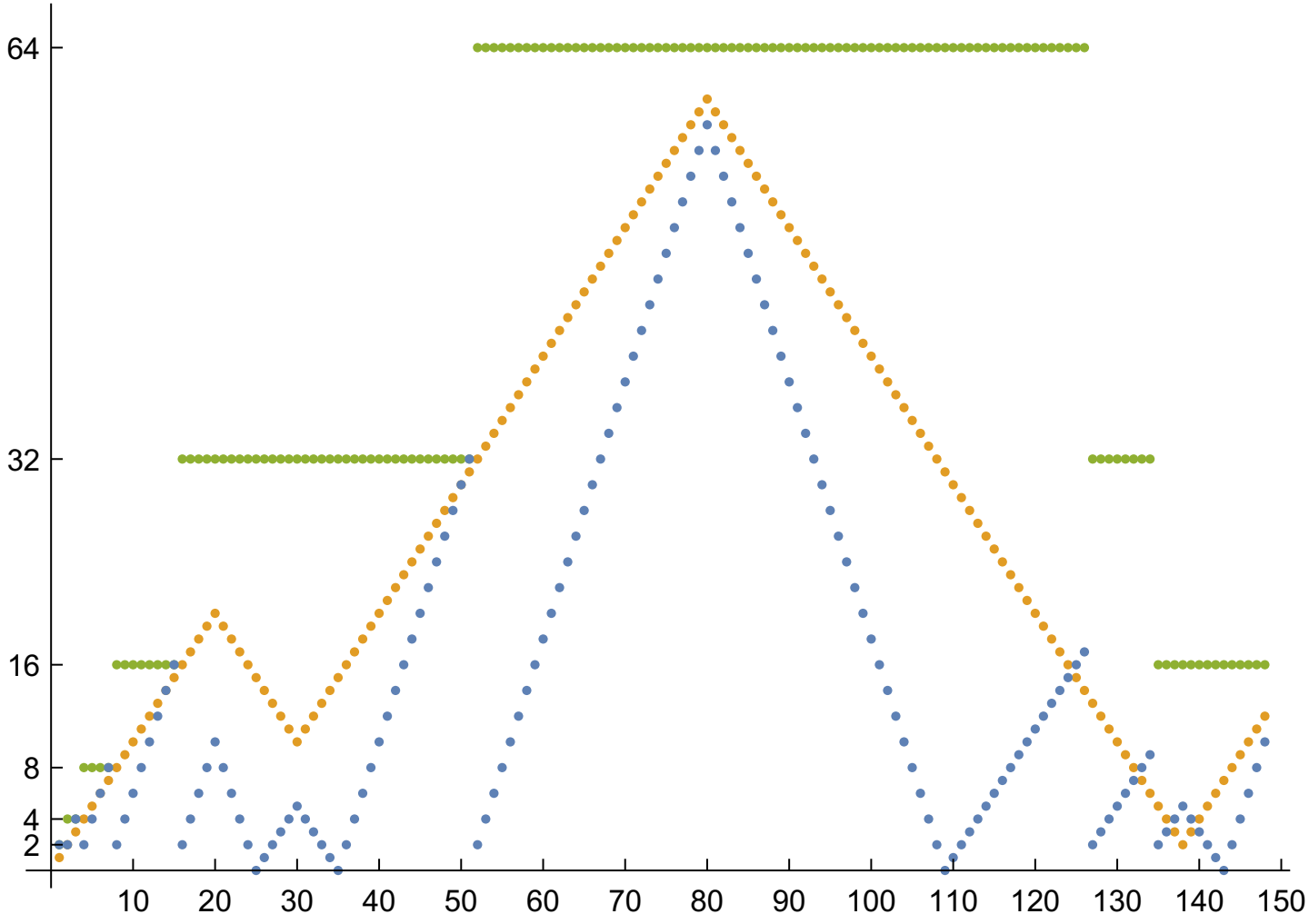
In this case, we have $1/4 \leq \alpha \leq 1$.

We need to modify our potential function to deal with this new situation

$$\Phi(n, s) = \begin{cases} 2s - n & \text{if } \alpha \geq 1/2, \\ n/2 - s & \text{otherwise.} \end{cases}$$

So, at the sweet spot $\alpha = 1/2$, the potential is 0. As we move away from $\alpha = 1/2$, we add to the potential to get read to pay for the next **grow** or **shrink**.

Since there is asymmetry, the deltas are 1 or 2.


64
32
16
8
4
2
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150

Case 1: $\alpha \geq 1/2$:

We are essentially in the same situation as in the first scenario when there were no deletions: $\text{cost}_\Phi(\textbf{insert}) = 2$.

Case 2: $\alpha, \alpha' < 1/2$

There is no change in capacity, so we have

$$\text{cost}_\Phi(\textbf{insert}) = 1 + (n'/2 - s') - (n/2 - s) = 0.$$

Case 3: $\alpha < 1/2 \leq \alpha'$

We have $s = n/2 - 1$, $s' = n'/2 = n/2$. So

$$\text{cost}_\Phi(\textbf{insert}) = 1 + (2s' - n') - (n/2 - s) = 0.$$

Case 1: $\alpha' \geq 1/2$:

There is no shrinking, so

$$\text{cost}_\Phi(\textbf{delete}) = 1 + (2s' - n') - (2s - n) = -1.$$

Case 2: $\alpha = 1/2$

Again, no change in capacity, so we have

$$\text{cost}_\Phi(\textbf{delete}) = 1 + (n'/2 - s') - (2s - n) = 2.$$

Case 3: $1/4 \leq \alpha', \alpha < 1/2$

Then

$$\text{cost}_\Phi(\textbf{delete}) = 1 + (n'/2 - s') - (n/2 - s) = 2.$$

Case 4: $\alpha' < 1/4 \leq \alpha$.

This triggers a shrink operation, so

$n' = n/2$, $s = n/4$, $s' = s - 1$.

Then

$$\begin{aligned}\text{cost}_\Phi(\textbf{delete}) &= (s+1) + (n'/2 - s') - (n/2 - s) \\ &= s + 1 + n/4 - s + 1 - n/2 + s \\ &= 2\end{aligned}$$

So, in all cases, $\text{cost}_\Phi(\textbf{delete}) = O(1)$.

Done.