Amortized Analysis

- Spreading out big costs over time
- Doing a bunch of operations — how long does each one take on average?

---

n	0	1	2	3	4	5	6	7	8	...	16
flips		1	2	1	3	1	2	1	4		
T(n) = total		1	3	4	7	8	10	11	15		31

for powers of 2, $T(n) = 2n - 1$

Seems like $T(n) < 2n$, in general.

(1, 2, 1, 3, 1, 2, 1, 4, ... ruler function)

IF SO, then average # of bit flips per increment

$$= \frac{\text{total}}{n} = \frac{T(n)}{n} < \frac{2n}{n} = 2.$$

Hence each increment takes $\Theta(1)$ on average (amortized).

How to actually prove this formally?

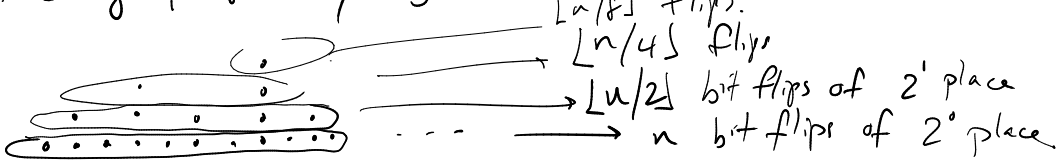
---

(1) Direct calculation.

Come up w/ formula for total cost, show it is < something.

$$T(n) = \underbrace{1 + 2 + 1 + 3 + 1 + 2 + 1 + 4 + \cdots}_{n}$$

often: change perspective / regroup / reorder ...?



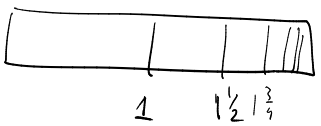
So $T(n) = n + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \cdots + 1$ ←

$\leq n + \lfloor \frac{n}{2} \rfloor + \lfloor \frac{n}{4} \rfloor + \cdots$

$$\leq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right)$$

$$= 2n.$$



---

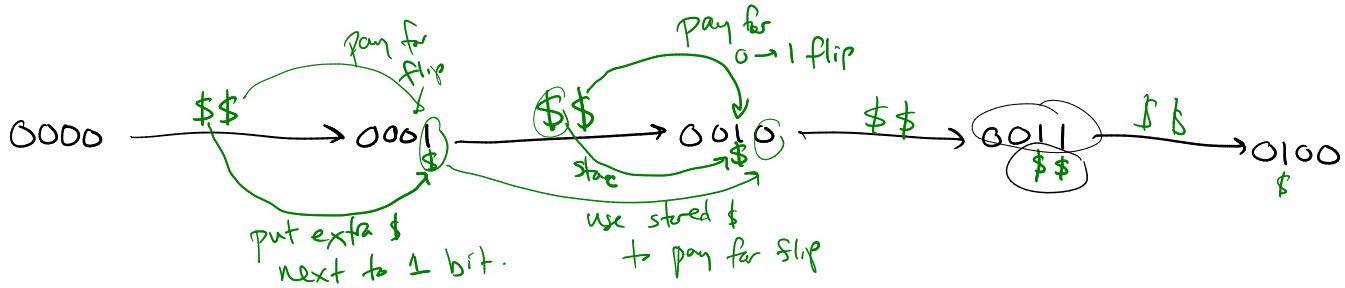
(2) Accounting method.

e.g. Running a binary counter service

- People can ask for value of their counter
- " " " to increment.
- Each bit flip costs us $1.
- How much should we charge per increment operation so we always have enough to cover our costs?

Claim: we can charge $2 per increment.

(Hence incrementing costs $\leq \$2$ on average, ie. $\Theta(1)$).



General idea:

- Every increment we flip exactly one $0 \to 1$.
  - Use 1$ to pay for the flip and save 1$ next to the 1.
- Every 1 that needs to be flipped to 0 will have $ stored next to it, which can be used to pay for the flip

(Aside: this shows $T(n) = 2n - \#n$ where $\#n$ = number of 1 bits in n's binary representation).

Example: extensible arrays.

Want to store a sequence of values + support 2 operations:

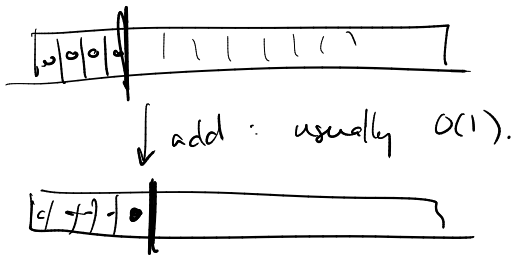
① Add new value to the end

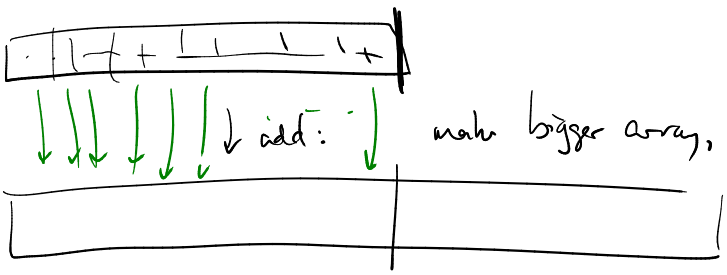
② Get value at a given index.

① alone: use linked list, but then ② is $O(n)$.

② alone: use array, but then ① is $O(n)$ (allocate new array etc.)

Solution: make big array, keep track of how much space we are using.

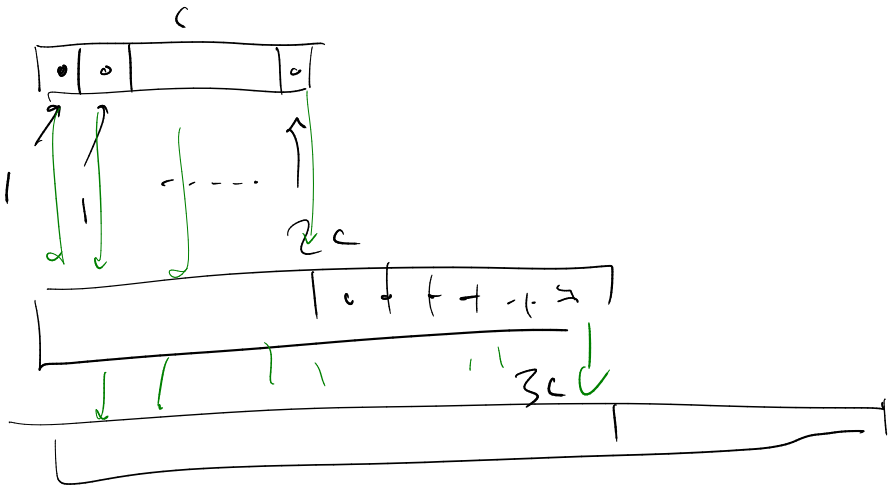




Q: how do we decide how big to make bigger array?

- Double size ←
- Increase by 1 — obviously terrible, append takes $O(n)$.
- Triple size ←
- Increase by 1000? ←

(1) Add c to size when resizing?



$T(n)$ = total cost of n consecutive appends

$$= \underbrace{1+1+\dots+1}_{c} + c + \underbrace{1+1+\dots+1}_{c} + 2c$$

(c: copy)

$$+ \underbrace{1+\dots+1}_{c} + 3c + \dots$$

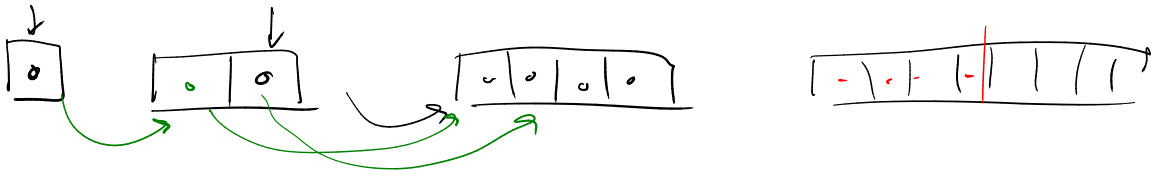
$$\approx n + \left(c + 2c + 3c + \dots \frac{n}{c}\cdot c\right)$$

$$= n + c\left(1+2+3+\dots \frac{n}{c}\right)$$

$$= n + c \cdot \Theta\left(\left(\frac{n}{c}\right)^2\right) = \Theta(n^2).$$

So on average, each append takes $\Theta(n^2)/n = \Theta(n)$.

(2) Double size?



$$1 + \underline{1} + 1 + \underline{2} + 1 + 1 + \underline{4} + 1 + 1 + \dots$$

total cost : $T(n) \approx n + (1 + 2 + 4 + 8 + \dots + n)$

$= n + (2n - 1)$

$= 3n - 1$ -- $\Theta(n)$.

So, on average, each append takes $\Theta(1)$.

(3) Square the size? Sizes : $2 \to 2^2 \to 2^4 \to 2^8 \to \dots$

2^{2^k}

$T(n) = n + (2 + 2^2 + 2^4 + 2^8 + \dots \; n)$

$< n + 2n$

Suppose most operations take $O(1)$ but every k^{th} operation takes $O(k)$ when k is a perfect square.

$T(n) = 1^2 + 1 + 1 + 2^2 + 1 \dots + 1 + 3^2 + 1 \dots 1 + 4^2 \dots$

$= n + (1^2 + 2^2 + 3^2 + \dots + k^2)$ (if $k^2 = n$)

$= n + \frac{k(k+1)(2k+1)}{6}$

$= n + \Theta(k^3)$ $\longrightarrow$ $k^3 = (\sqrt{n})^3 = n^{\frac{3}{2}}$

$= n + \Theta(n^{\frac{3}{2}}) = \Theta(n^{3/2})$.

What if we can add + remove from extensible array, want to make it bigger/smaller?

Bad idea:

- Double size when adding to full array
- Halve size when removing + array is 1/2 full

Better idea:

- Double when full
- Halve when array is 1/4 full