CSC263 Winter 2020

Amortized Analysis, Quicksort & Randomized Algorithms

Week 6

Amortized Analysis

The **amortized sequence complexity** is the "average" cost per operation **over a sequence of operations**.

For a sequence of **m** operations:

$$\text{Amortized sequence complexity} = \frac{\text{worst-case } \textbf{sequence} \text{ complexity}}{m}$$

The MAXIMUM possible ***total*** cost of among all possible sequences of m operations

Amortized Analysis

The **amortized sequence complexity** is the "average" cost per operation **over the sequence**.

- Unlike average-case analysis, there is **NO** probability or expectation involved.

We do amortized analysis when we are interested in the total complexity of a **sequence** of operations.

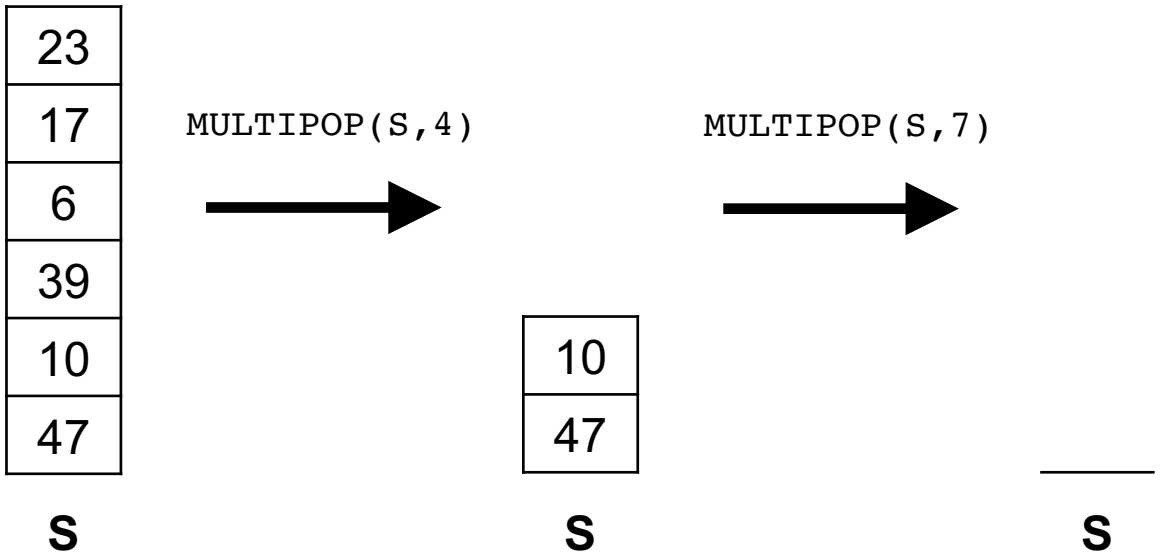
- Unlike in average-case analysis where we are interested in a **single** operation.

Example for Intuition

Stack with additional operation

- **PUSH(S, x)** push one item into the stack
- **POP(S)** pops the top item from S
- **MULTIPOP(S, k)** pop k top items from S

```
def MULTIPOP(S,k):
  while not STACK-EMPTY(S) and k>0:
    POP(S)
    k = k-1
```



Example for Intuition

We have a sequence of **N** operations consisting of **PUSH** and **MULTIPOP**.

What's the worst-case total runtime of the sequence?

(Start: S empty)

- with **N** operations we can push in at most **N** items into the stack
- MULTIPOP pops **N** elements in worst-case, which takes **N** steps.
- If all operations were MULTIPOP, each of which takes the worst N steps, the total runtime would be **$O(N^2)$**. Right?
- No! We have at most **N** elements, each of which is at most pushed once and popped **once**, so the total runtime of the sequence is **$O(N)$** !

Analysing the runtime of a sequence of operations
IS NOT
the runtime of a **single** operation **multiplied**
by the size of the sequence

especially when the sequence of operations are operating on a shared data structure and have interdependencies with each other.

So we need some special analysis methods

Amortized Analysis

An **amortized analysis** of a data structure
computes the maximum possible
average cost per operation
in a **sequence of operations**,
starting from some initial base state.

Methods for Amortized Analysis

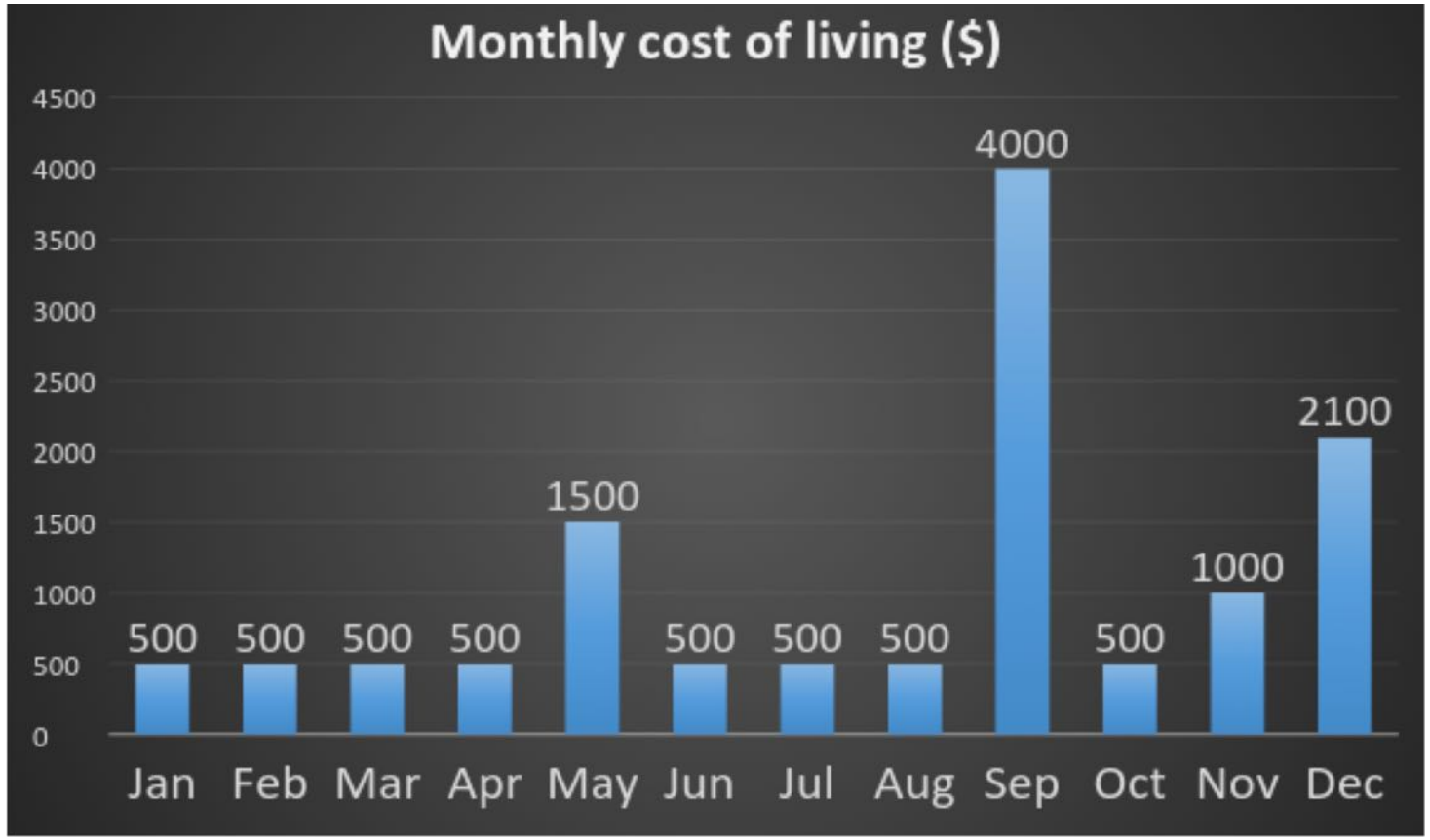
- Aggregate Method
- Accounting Method
- Potential Method

  not covered in lecture, read CLRS Chapter 17

Amortized Analysis

Real-life Intuition

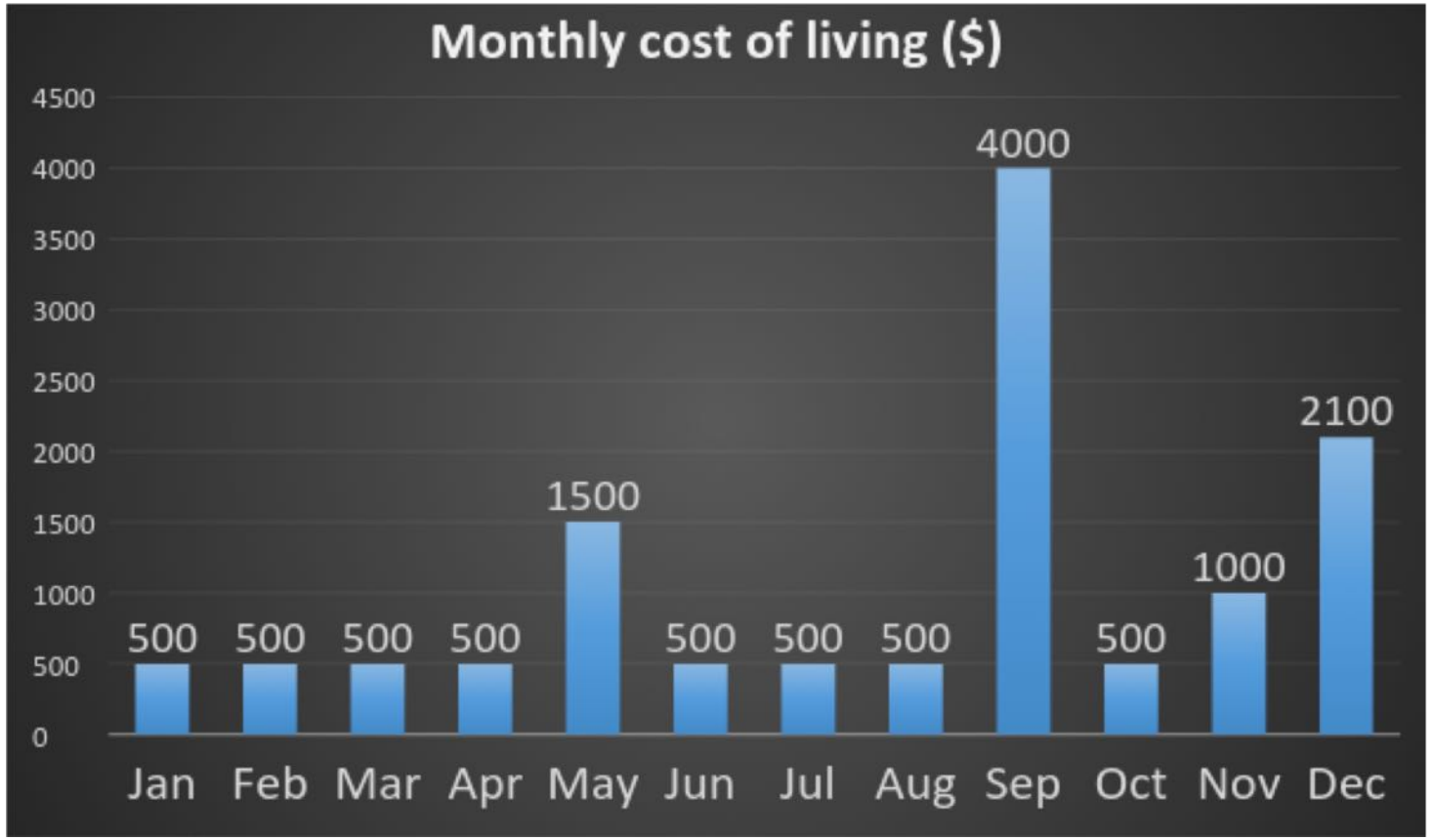
Monthly cost of living, a sequence of 12 operations



Aggregate method

What is the amortized cost per month (operation)?

Build **sum** of the costs of sequence of operations
and **divide** by the number of operations,
to determine the **average cost per operation**.



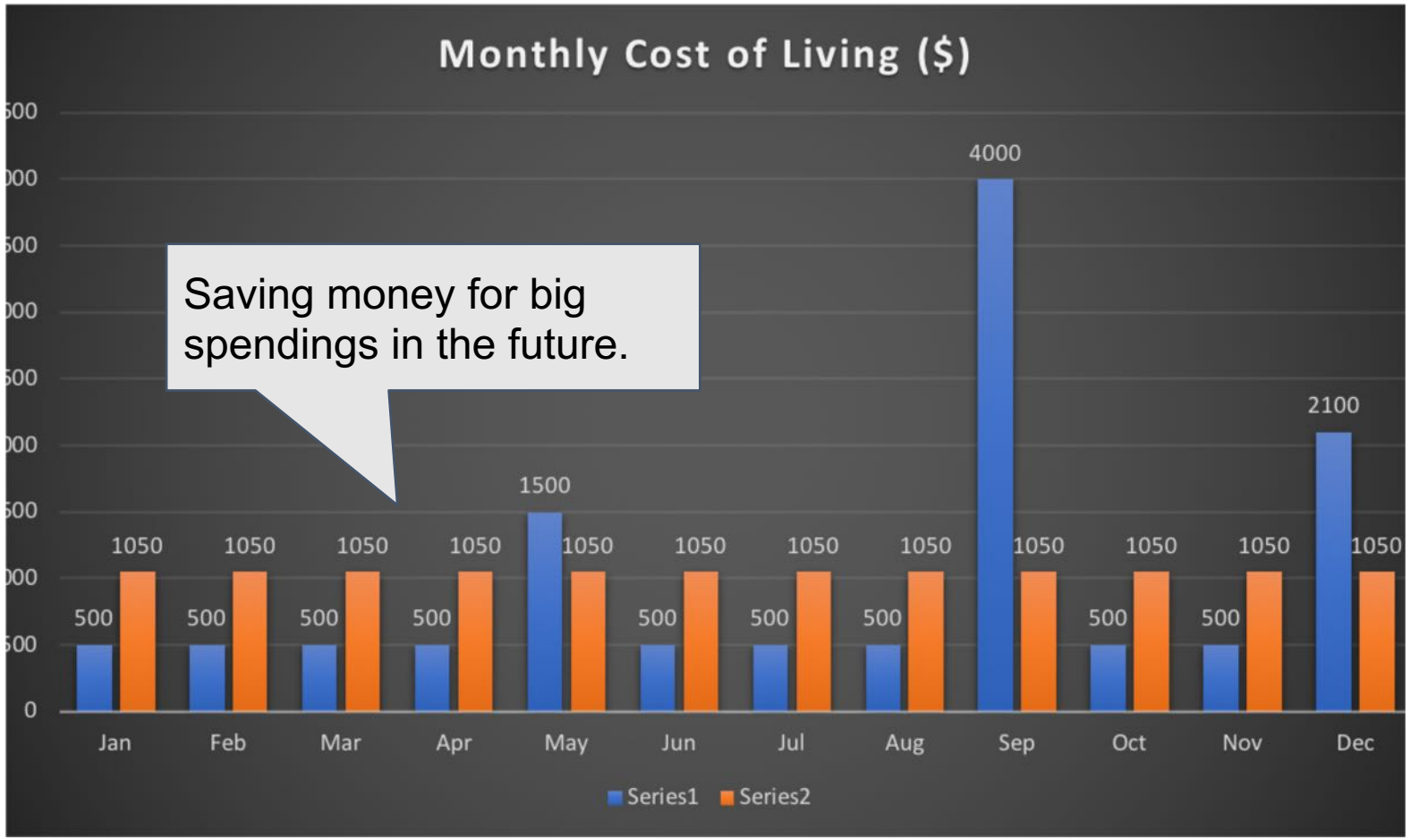
Aggregate Method: sum of all months' spending is $12,600 divided by 12 months

⟹ amortized cost is $1,050 per month.

Accounting Method

Instead of calculating the average cost, we think about the cost from a **different angle**, i.e.,

How much money do I need to **earn** each month in order to **keep living**? That is, be able to pay for the spending every month and **never become broke**.



Accounting method: if I **earn** $1,050 per month from Jan to Dec, I will never become broke (assuming earnings are paid at the beginning of month).

So the **amortized cost**: $1,050

Aggregate vs Accounting

- Aggregate method gives each type of operation the same amortized cost (the average cost)
- Accounting method is more flexible
  - Each type of operation can be assigned a **different** amortized cost
  - Works even when the sequence of operations is not concretely defined
  - Gives more interesting insights for data structure design

Amortized Analysis on Dynamic Arrays

Case Study

Problem Description

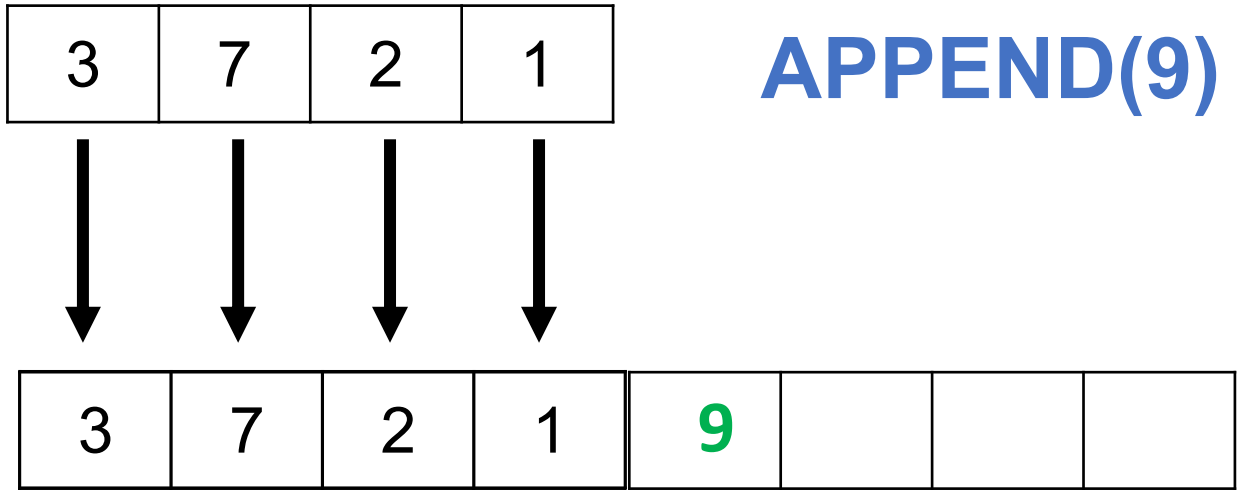
- Think of an **array** initialized with a **fixed** number of slots, which supports **APPEND** operations.
- When we APPEND too many elements, the array would be **full** and we need to **expand** the array (increase its size).
- Requirement: the array must be using **one** **contiguous block** of memory all the time.

How do we do the **expansion** so that we have good performance with a sequence of APPENDs?

One Way to Expand

If the array is full, APPEND is called

- Create a new array of double the size
- Copy the all elements from old array to new array
- Append the element



Amortized Analysis of Expand

Now consider a dynamic array initialized with **size 1** and a sequence of **m APPEND** operations on it.

Analyse the amortized cost per operation

Assumption: only count array assignments, i.e.,
***append** an element and **copy** an element*

Using the Aggregate Method

Cost sequence would be like:

$$c_i = \begin{cases} i+1, & \text{if } i \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

Copy 2 append 1

1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, …

Copy 1 append 1

Copy 4 append 1

Copy 8 append 1

Cost sequence concretely defined, sum-and-divide can be done, but we want to do something more interesting…

Using the Accounting Method

How much money do we need to **earn** at each operation, so that all future costs can be paid for?

How much money to earn for **each APPEND'ed element** ?

I hope that's enough, so that I will never become broke.

Son, you're going to be appended to the array. Here is some money. Use it to pay all your future expenses (like cost of appending, being copied, etc).

Using the Accounting Method

How much money do we need to **earn** at each operation, so that all future costs can be paid for?

How much money to earn for **each APPEND'ed element** ?

$1 ?

$2 ?

$3 ?

$log m ?

$m ?

Earn $1 for each appended element

This $1 (the “append-dollar”) is spent when appending the element.

But, when we need to copy this element to a new array (when expanding the array), we don’t any money to pay for it --

BROKE!

Earn $2 for each appended element

$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

What if the element is copied for a second time (when expanding the array for a second time)?

BROKE!

Earn $3 for each appended element

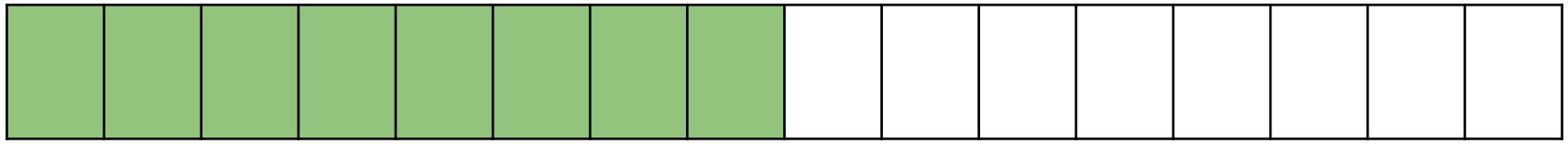
$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

$1 (the “recharge-dollar”) is used to **recharge** **the old elements** that have spent their “copy-dollars”.

NEVER BROKE!

$1 (the “recharge-dollar”) is used to **recharge** the old elements that have used their “copy-dollar”.



Old elements who have used their “copy-dollars”

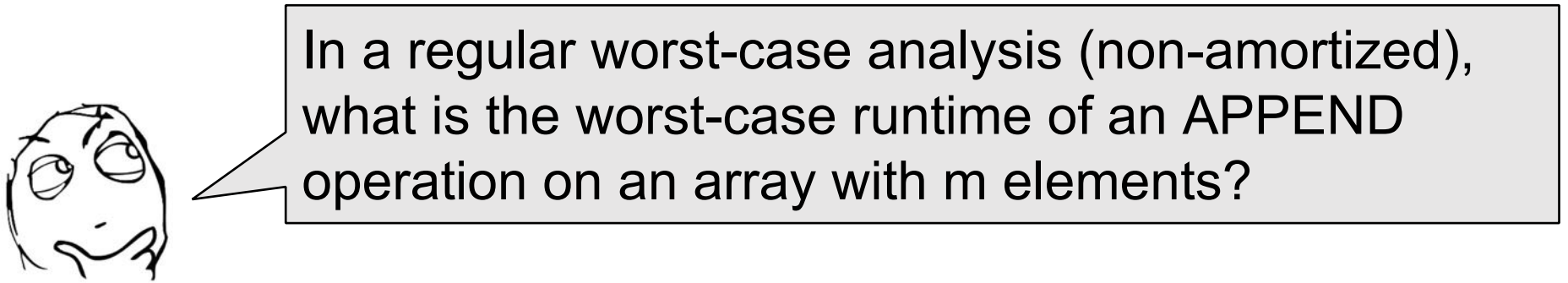
New elements each of whom **spares** $1 for recharging one old element’s “copy-dollar”.

There will be enough new elements who will spare **enough money** for **all** the old elements, because the way we expand – **TWICE the size**

In Summary

If we earn $3 upon each APPEND it is enough money to pay for all costs in the sequence of APPEND operations.

In other words, for a sequence of **m** APPEND operations, the amortised cost per operations is **3**, which is in O(1).

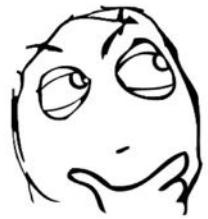


By performing the amortised analysis, we showed that "**double the size when full**" is a good strategy for expanding a dynamic array, since it's amortised cost per operation is O(1).

In contrast, "**increase size by 100 when full**" would not be a good strategy.

It will cause O(n) amortised runtime.

Why?



Takeaway

Amortized analysis provides us valuable insights into the design of the expansion strategy of dynamic arrays.

It is a powerful tool for data structures design.

How about Shrinking?

For a complete implementation of Dynamic Array, we also need a strategy for shrinking.

If we keep deleting elements and there are many unused space in the array, we want to shrink.

When to shrink?

Amortised analysis tells we shrink when the array is ¼ full, this ensures DELETE operations have amortised runtime O(1).

More details in CLRS Chapter 17.4

This type of strategy for expanding and shrinking is widely used by hash table implementations

- Java HashMap: expand when load factor is larger than ¾
- Python Dict: expand when load factor is larger than ⅔

By dynamically resizing, the hash table can maintain a constant load factor therefore guarantees constant lookup time.

7 min

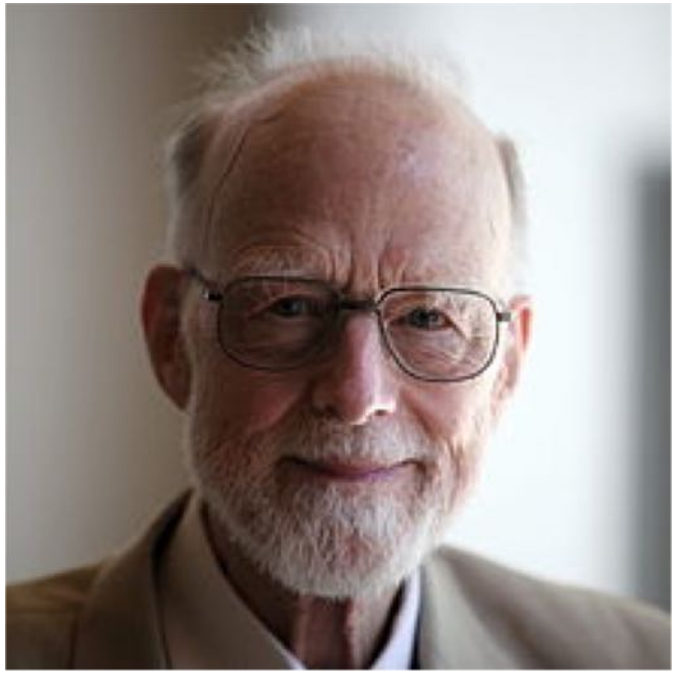
QuickSort

Background

Invented by **Tony Hoare** in 1960

(aka Sir Charles Antony Richard Hoare)

Very commonly used sorting algorithm. When **implemented well**, it is typically faster than **merge sort** and **heapsort**.



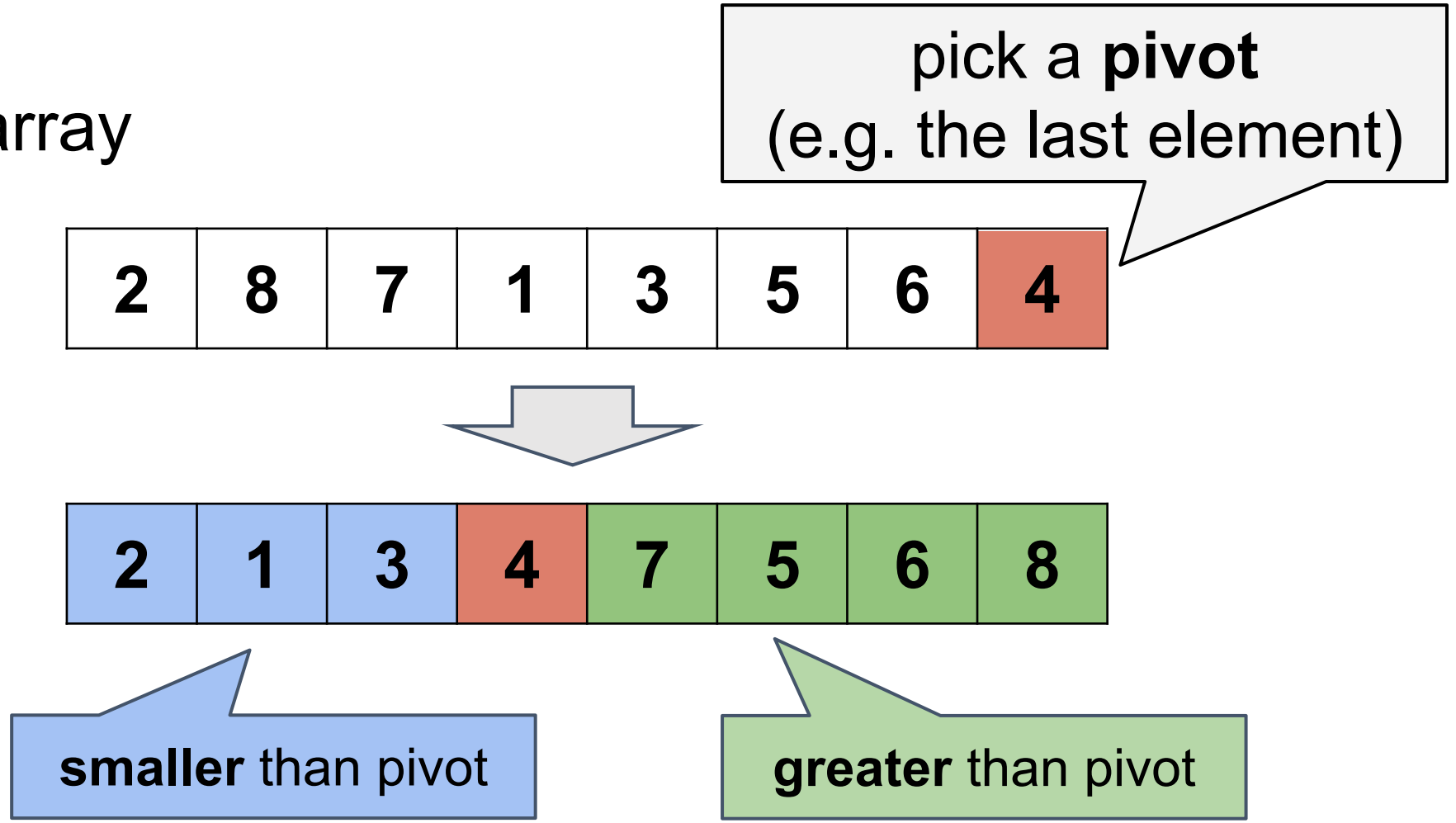
Invented **NULL pointer** in 1965. Apologized for it in 2009

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

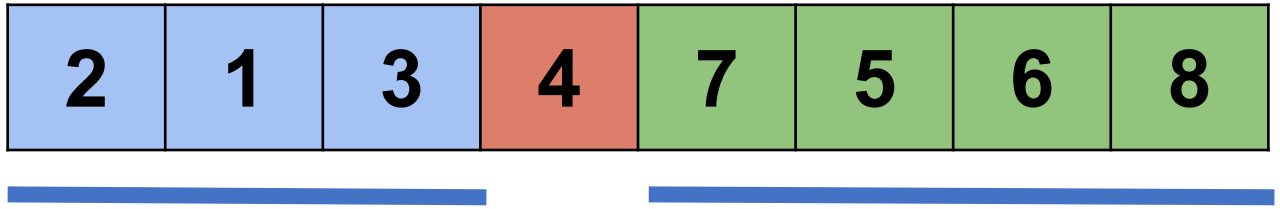
CLRS Chapter 7

QuickSort: The Idea

Partition an array

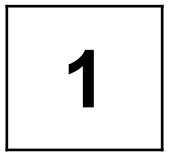


CLRS Chapter 7



Recursively partition the sub-arrays **before** and **after** the pivot.

Base case

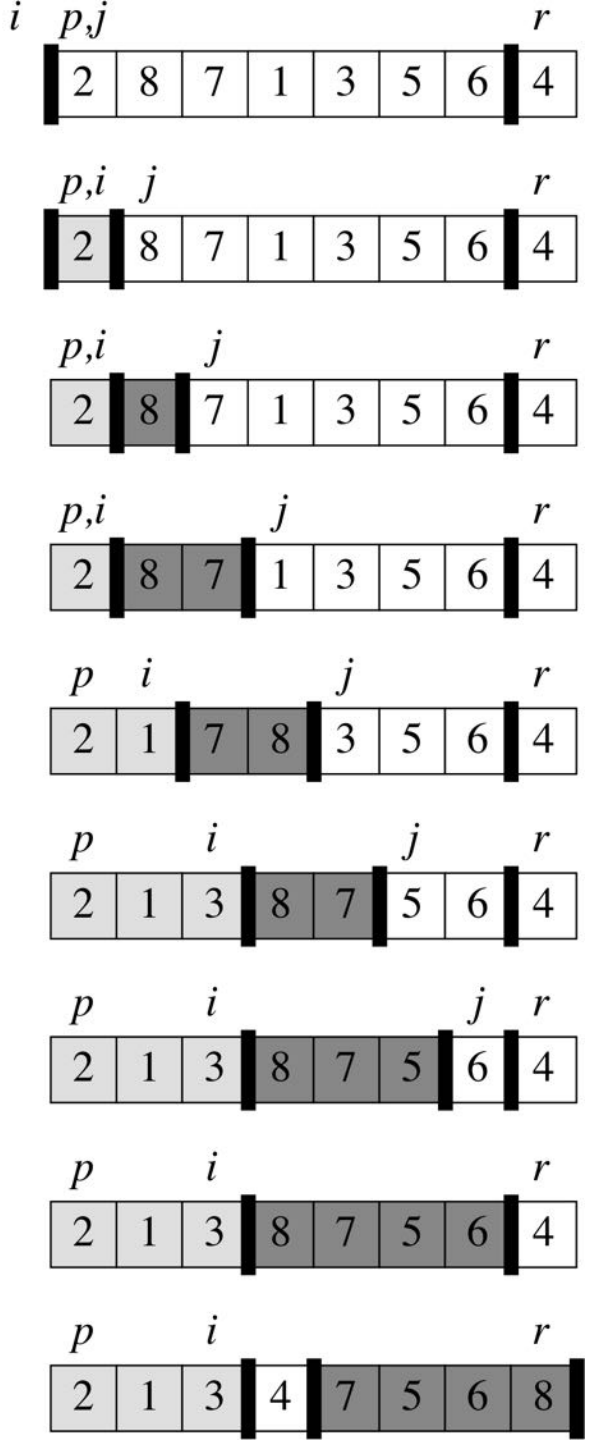


sorted

Pseudocode Quicksort

```
def quicksort(array,p,r):
  #initial call: quicksort(A, 1, A.length)
  if p < r:
    q <- partition(array, p, r)
  quicksort(array, p, q-1)
  quicksort(array, q+1, r)
```

```
def partition (array, p, r):
  pivot = array[r]
  i = p-1
  for j = p to r-1:
    if array[j] <= pivot:
      i=i+1
      exchange array[i] with array[j]
  exchange array[i+1] with array[r]
  return i+1
```



Worst-case Analysis of QuickSort

T(n): the total number of **comparisons** made

We will first prove T(n) = O(something)

Then we will prove T(n) = Ω(the same thing)

Therefore we conclude T(n) = Θ(the thing)

For simplicity, assume all elements are distinct

Claim 1

A pivot never goes into a sub-array on which a recursive call is made.

Each element in **A** can be chosen as **pivot** **at most once**.

Claim 2

That's what partition is all about - comparing with pivot.

Elements are **only** compared to **pivots**.

Claim 3

Any **pair** (a, b) of elements in A are compared **at most once**.

For a and b to be compared, one of them must be the pivot (Claim 2). This pivot will never be pivot again (Claim 1), and so can't participate in any more comparisons.

The total number of **comparisons** is **no more than** the **total number of pairs**.

The total number of **comparisons** is **no more than** the **total number of pairs**.

$$T(n) \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

$$T(n) \in \mathcal{O}(n^2)$$

Next, show $T(n) \in \Omega(n^2)$

Show $T(n) \in \Omega(n^2)$

i.e., the **worst-case** running time is **lower-bounded** by some cn^2

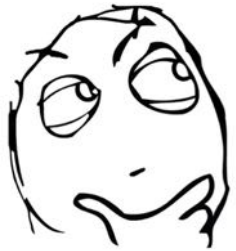
How do you show the **tallest** person in the room is **lower-bounded** by **1 meter**?

Just find **one** person who is taller than 1m!

Week 1

so, just find **one input** for which the running time is some cn^2

so, just find **one input** for which the running time is some cn^2



i.e., find one input that results in **awful** partitions (everything on one side).

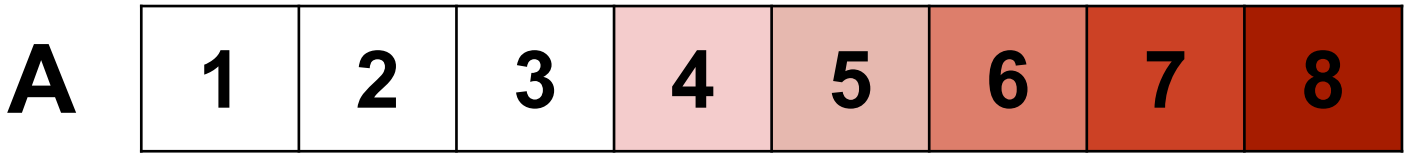
1	2	3	4	5	6	7	8

Remember that we always pick the last one as pivot.

IRONY

The **worst input** for `QuickSort` is an **already sorted array**.

Calculate the Number of Comparisons



Choose pivot **A[n]**, then **n-1** comparisons

Recurse to subarray, pivot **A[n-1]**, then **n-2** comps

Recursive to subarray, pivot **A[n-2]**, then **n-3** comps

...

Total # of comps: $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$

So, the worst-case runtime

$$T(n) \geq \frac{n(n-1)}{2}$$

$$T(n) \in \Omega(n^2)$$

already shown $T(n) \in \mathcal{O}(n^2)$

so, $T(n) \in \Theta(n^2)$

$$T(n) \in \Theta(n^2)$$

What other sorting algorithms have **n²** worst-case running time?

Bubble Sort, Insertion Sort, …

Is QuickSort really "quick" ?

Yes, in **average-case.**

Average-case Analysis of QuickSort

Input distribution all permutations of array [1, 2, …, n]

array is chosen uniformly at random from among these permutations

One can prove that the **expected number of comparisons** is **O(n log n).**

Read CLRS Chapter 7.4

Summary Quicksort

The worst-case runtime is $\Theta(n^2)$.

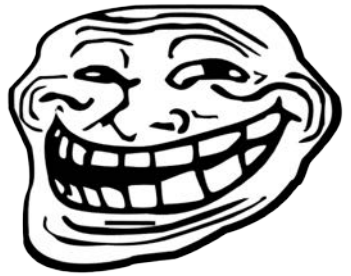
The average-case runtime of QuickSort is $O(n \log n)$, given that the inputs are uniformly random permutations of an array.

However, in Real Life...

The assumption of uniform randomness is NOT really true, because it is often impossible for us to know what the input distribution really is.

QuickSort(A)

Even worse, if the person who provides the inputs is **malicious**, they can totally only provide worst-inputs and guarantee worst-case runtime.



The theoretical average case O(nlog n) is no way guaranteed in real life…

How can we get guaranteed performance in real life? Use Randomisation!

- We **shuffle** the **input array** "uniformly randomly", so that after shuffling the array looks like drawn from a uniform distribution
- Even the malicious person's always-worst inputs will be shuffled to be like uniformly distributed
- This makes the assumption of the average-case analysis true
- So we can guarantee **O(n log n) expected runtime**

Randomised-QuickSort(A):

```
permute A uniformly randomly
QuickSort(A)
```

How exactly do we perform the permutation so that we can prove that it's going to be like uniform distribution? (Read CLRS Chapter 5.3)

Randomized Algorithms

Use Randomization to Guarantee Expected Performance

We do it everyday.

Two Types of Randomized Algorithms

Las Vegas Algorithms

Deterministic answer, random runtime

Monte Carlo Algorithms

Deterministic runtime, random answer

Randomized-QuickSort is a … Las Vegas algorithm

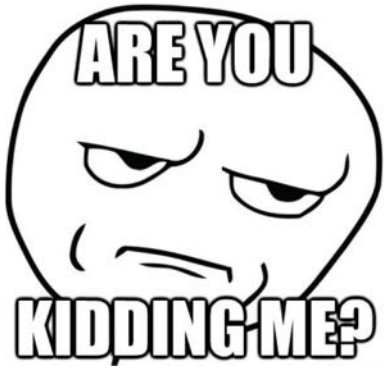
An Example Monte Carlo Algorithm

"Equality Testing"

The Problem

Given two binary numbers x and y, decide whether x = y.

```
def equal(x, y):
    return x == y
```



The Problem

Given two binary numbers x and y, decide whether **x = y**.

No kidding, what if the **size** of **x** and **y** are **10TB** each?

The above code needs to compare ~10^{14} bits.

Even worse, what if x and y are stored on two separate computers which are connected through a slow network connection?

Can we do better?

Why assuming x and y are of the same length?

Let n = len(x) = len(y) be the length of x and y.

```
def equal_monte_carlo(x, y):
    Randomly choose a prime number p ≤ n²
    # len(p) ≤ log₂(n²) = 2log₂(n)
    return (x mod p) == (y mod p)
```

Need to compare **at most 2log(n)** bits.

But, does it give the correct answer?

$\log_2(10^{14}) \approx 46.5$

Huge improvement on runtime!

Does it give the Correct Answer?

If **(x mod p) ≠ (y mod p)**, then…

Must be **x ≠ y**, our answer is correct **for sure**.

If **(x mod p) = (y mod p)**, then…

Could happen that **x ≠ y but (x mod p) = (y mod p)**, so our answer **might be** wrong, **if choosing a "bad" p**.

So, what's the probability of a wrong answer?

It's upper-bounded by the probability of choosing a "bad" p.

Prime Number Theorem

Theorem

In range **[1, m]**, there are roughly **m/ln(m)** prime numbers.

So in range **[1, n²]**, there are $n^2/\ln(n^2)$ = **$n^2/2\ln(n)$** prime numbers.

Given x and y, how many (**bad**) primes in **[1, n²]** could satisfy **(x mod p) = (y mod p)** even if **x ≠ y** ?

At most n

Proof: At most n Bad Primes

There are at most **n** primes **p**, such that **(x mod p) = (y mod p)** while **x ≠ y**.

Proof

(x mod p) = (y mod p) ⇔ |x - y| is a multiple of p, i.e., p is a prime divisor of |x - y|.

$|x - y| < 2^n$ (as they are n-bit binary numbers)

so it has no more than n prime divisors

(otherwise if it has more than n prime divisors, the product of them will be larger than 2^n, since all primes >= 2).

So...

```
def equal_monte_carlo(x, y):
    Randomly choose a prime number p ≤ n²
    # len(p) ≤ log2(n²) = 2log2(n)
    return (x mod p) == (y mod p)
```

Out of the **$n^2/2\ln(n)$** prime numbers we choose from, at most **n** of them could cause the algorithm to return a **wrong answer** (bad primes).

If we choose a **good prime**, the algorithm gives **correct** answer for sure.

If we choose a **bad prime**, the algorithm **may** give a **wrong** answer.

So the probability of wrong answer is upper-bounded by

$$\frac{n}{n^2/(2\ln n)} = \frac{2\ln n}{n}$$

Error Probability of our Monte Carlo Algorithm

$$\Pr(\text{error}) \leq \frac{2 \ln n}{n}$$

When n = 10^{14} (10TB)

Pr(error) ≤ 0.00000000000644

Performance Comparison (n = 10TB)

The Regular Algorithm

x == y

- Perform 10^{14} comparisons
- Error probability: 0

The Monte Carlo Algorithm

(x mod p) == (y mod p)

- Perform < 100 comparisons
- Error probability: 0.000000000000644

If you feel like: "This error probability is too high!"

- Run Monte Carlo Algorithm twice
- Perform < 200 comparisons
- Error probability squared: 0.000000000000000000000415

Summary

Randomized Algorithms

Guarantees expected performance

Make algorithm less vulnerable to malicious inputs

Monte Carlo Algorithms

Gain time efficiency by sacrificing a tiny bit of correctness.

In 2 weeks

Graphs