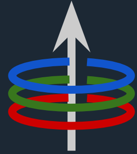


CoE 163

Computing Architectures and Algorithms

02b: Amortized Analysis

TIME COMPLEXITY

The basic asymptotic analysis method earlier gives us a good estimate of the worst-case runtime of an algorithm.

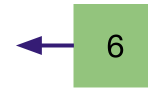
However, this analysis may sometimes be too pessimistic as we may naively sum the individual runtimes.

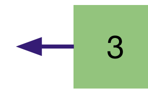
CONSIDER...

How does the “vector” data type in C++ work? It is a dynamic array!

DYNAMIC ARRAY

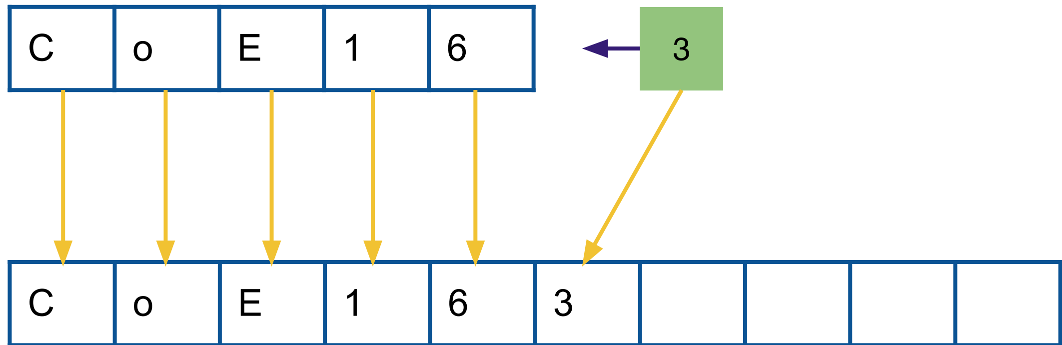
You can append an element to this array, but the array grows to twice its size if it is full to accommodate the element.





DYNAMIC ARRAY

Growing works by creating a new array of twice its size, and then copying everything to the new array.



NAIVE TIME COMPLEXITY

Assuming it takes constant time (O(1)) to append an item, the copy operation would take O(n).

The worst-case scenario is the array will grow every time an item is appended.

Append runtime is therefore

$$T(n) = O(n^2)$$

too pessimistic!

NAIVE TIME COMPLEXITY

This method of summing the worst-case run times per algorithm is not accurate because an array copy only happens when the array becomes full.

We need another method that finds the average cost of each operation in an algorithm.

AMORTIZED ANALYSIS

Amortized analysis deals with finding the worst-case run time by analyzing each operation, not each algorithm.

It is “amortized” because it aims to average out the “expensive” operations among a series of operations.

AMORTIZED ANALYSIS METHODS

- Aggregate
  - Average out the operation run time with the size of input
- Accounting
  - Assign a value to each operation and balance them like an accounting sheet
- Potential
  - Formulate a potential function that keeps track of run time similar to the accounting method

AGGREGATE METHOD

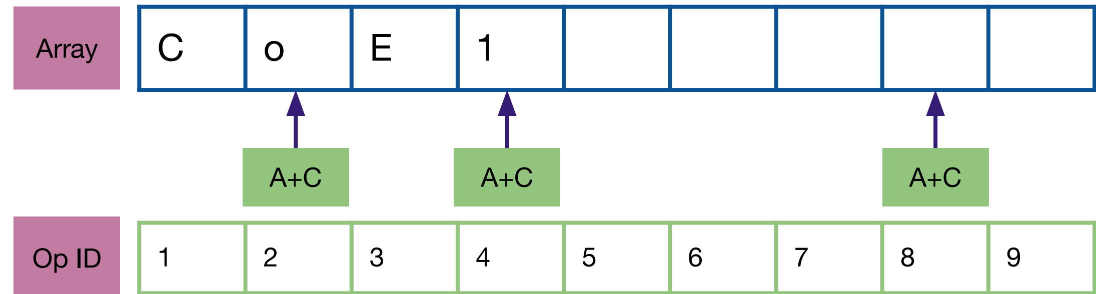
- Determine worst-case runtime of entire sequence of operations and divide this runtime with the number of operations in sequence
- Consider cost of each operation separately

AGGREGATE METHOD

Assume array has initial size of 2

Append is O(1) and append + copy is O(1) + O(n)

Array grows every iteration index with a power of two



AGGREGATE METHOD

The cost of the append operator at the ith iteration can be expressed as

$$C_i = \begin{cases} i+1 & , lg(i) \in \mathbf{Z} \\ 1 & \end{cases}$$

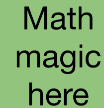
Then, we can compute for the aggregate run time

$$T(n) = \sum_{i=1}^{n} C_i$$

$$T(n) = \sum_{lg(i)\in(Z)} (i+1) + \sum_{lg(i)\notin(Z)} 1$$

AGGREGATE METHOD

$$
\begin{aligned}
T(n) &= \sum_{lg(i)\in(Z)} (i+1) + \sum_{lg(i)\notin(Z)} 1 \\
&= \sum_{lg(i)\in(Z)} i + \sum_{lg(i)\in(Z)} 1 + \sum_{lg(i)\notin(Z)} 1 \\
&= \left(\sum_{j=0}^{lg(n)} 2^j\right) + n \\
&= \frac{2^{lg(n)+1}-1}{2-1} + n \\
&= 3n-1
\end{aligned}
$$



Math magic here

The amortized worst-case runtime is apparently constant (O(1))!

ACCOUNTING METHOD

- We come up with an initial “cost” of running the algorithm
- We balance the contents of our “bank” by assigning appropriate “costs” to each operation
- The general goal is to save “cost” from the cheap operations to pay for the expensive operations without going broke (negative “cost”)
- One approach is trial and error

ACCOUNTING METHOD

Append is ₱1 and append + copy is ₱(i + 1)

Let's try having ₱1 in our account every iteration

Negative balance!

Array	C	o	E	1					
			A+C		A+C				A+C
Op ID	0	1	2	3	4	5	6	7	8
Old Bal	1	1	1	-1	-1	...			
New Bal	0	0	-2	-2	-6	...			

ACCOUNTING METHOD

Append is ₱1 and append + copy is ₱(i + 1)

Let's try having ₱2 in our account every iteration

Still negative balance!

Array	C	o	E	1					
			A+C		A+C				A+C
Op ID	0	1	2	3	4	5	6	7	8
Old Bal	2	3	3	2	3	0	...		
New Bal	1	1	0	1	-2	-1	...		

ACCOUNTING METHOD

Append is ₱1 and append + copy is ₱(i + 1)

Let's try having ₱3 in our account every iteration

Looks promising!

Array	C	o	E	1					
			A+C		A+C				A+C
Op ID	0	1	2	3	4	5	6	7	8
Old Bal	3	5	7	7	9	7	9	11	13
New Bal	2	4	4	6	4	6	8	10	4

ACCOUNTING METHOD

An inductive proof is needed to show that the amortized worst-case runtime is actually 3.

Prove that

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$

… for C(hat)_i = 3, the amortized runtime.

ACCOUNTING METHOD

<u>Base case</u>

If array is of size 0 initially, then the “cost” of ₱3 can pay for the operation (copy + append), which costs ₱1. $B_0 = 2$

<u>Inductive step</u>

Since there are two operations, we need to split the proof into two. For this to work, we want to use the following relation:

$$B_i = B_{i-1} + \hat{C}_i - C_i$$

We must prove that, for $\hat{C}_i = 3$

$$B_i \geq 0, i > 0$$

ACCOUNTING METHOD

Inductive step

Copy

For this operation, $C_i = 1$. Assuming that $B_{i-1} \geq 0$

$$\begin{aligned} B_i &= B_{i-1} + \hat{C}_i - C_i \\ &= B_{i-1} + 3 - 1 \end{aligned}$$

Computing for the change in budget yields

$$B_i - B_{i-1} = 2$$

This shows that you always have an additional ₱2 in your balance every time this operation takes place, so it definitely shows what we wanted to prove.

ACCOUNTING METHOD

<u>Inductive step</u>

Append + Copy

Assuming that the array has just completed a copy step. The array will be half-empty, and we need to encounter (n / 2) - 1 appends (with at least $B_{i-1} = n + 2$ saved) before an append + copy operation takes place, which costs ₱(n + 1).

For $C_i = i + 1$, similar to the proof for the copy operation,

$$\begin{aligned} B_i &= B_{i-1} + \hat{C}_i - C_i \\ &\geq i + 2 + 3 - (i + 1) \\ &\geq 4 \end{aligned}$$

This definitely shows what we wanted to prove.

POTENTIAL METHOD

- Similar to the accounting method but models the cost (potential) as a function of the data structure variables
- Most flexible of all methods as it depends only on the current state of the data structure used

POTENTIAL METHOD

Define a potential function with a base and general value

$$\Phi(0) = 0, \Phi(d_i) \geq 0$$

The amortized worst-case runtime is defined as

$$\hat{C}_i = C_i + (\Phi(d_i) - \Phi(d_{i-1})$$

… where d is the state of data at the ith iteration of the algorithm

POTENTIAL METHOD

<u>Cost analysis</u>

We can treat our potential function like a bank account. We define zero potential if the array has just doubled its size. Potential increases as the number of elements reach the maximum size of the array.

<u>Function formulation</u>

We formulate the function in terms of the size of the array and the number of elements in it. The maximum potential of the array is its length n_{arr}

$$\Phi(d_i) = 2n_{elms} - n_{arr}$$

POTENTIAL METHOD

Computation

We split the analysis into two since there are two operations associated with inserting an element.

Append only

$$
\begin{aligned}
\hat{C}_i &= C_i + (\Phi(d_i) - \Phi(d_{i-1})) \\
&= 1 + ((2n_{1,elms} - n_{1,arr}) - (2n_{0,elms} - n_{0,arr})) \\
&= 1 + ((2(n_{0,elms} + 1) - n_{0,arr}) - (2n_{0,elms} - n_{0,arr})) \\
&= 1 + 2(1) \\
&= 3
\end{aligned}
$$

POTENTIAL METHOD

Computation

We split the analysis into two since there are two operations associated with inserting an element.

Copy + Append

$$\begin{aligned}\hat{C}_i &= C_i + (\Phi(d_i) - \Phi(d_{i-1})) \\ &= (n_{0,arr} + 1) + ((2n_{1,elms} - n_{1,arr}) - (2n_{0,elms} - n_{0,arr})) \\ &= (n_{0,arr} + 1) + ((2(n_{0,elms} + 1) - 2n_{0,arr}) - (2n_{0,elms} - n_{0,arr}) \\ &= n_{0,arr} + 1 + 2n_{0,elms} + 2 - 2n_{0,arr} - 2n_{0,elms} + n_{0,arr} \\ &= 3\end{aligned}$$

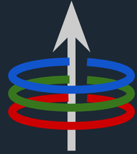
We can formulate other potential functions, but the amortized time may be different in each case.

TIPS

- Practice how to use induction to prove runtimes
- Practice how to balance spreadsheets and accounts
- Jog your abstract thinking, educated guessing, and trial-and-error skills

RESOURCES

- Resource from the University of Hawaii
- Resource from the Cornell University
- Resource from MIT



CoE 163

Computing Architectures and Algorithms

02b: Amortized Analysis