Gordon Lu, gl1589@nyu.edu
Advanced Algorithms: Homework I
September 19, 2023

Advanced Algorithms: Homework I

Question 1.

Solution:

Our in-class example with cost 1 was

Counter	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Part a:

We will now consider with cost 17. The updated table is:

Counter	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	17
2	0	0	0	0	0	0	1	0	$17 + 2(17) = 51$
3	0	0	0	0	0	0	1	1	$51 + 17 = 68$
4	0	0	0	0	0	1	0	0	$68 + 3(17) = 119$
5	0	0	0	0	0	1	0	1	$119 + 17 = 136$
6	0	0	0	0	0	1	1	0	$136 + 2(17) = 170$
7	0	0	0	0	0	1	1	1	$170 + 17 = 187$
8	0	0	0	0	1	0	0	0	$187 + 4(17) = 255$
9	0	0	0	0	1	0	0	1	$255 + 17 = 272$
10	0	0	0	0	1	0	1	0	$272 + 2(17) = 306$
11	0	0	0	0	1	0	1	1	$306 + 17 = 323$
12	0	0	0	0	1	1	0	0	$323 + 3(17) = 374$
13	0	0	0	0	1	1	0	1	$374 + 17 = 391$
14	0	0	0	0	1	1	1	0	$391 + 2(17) = 425$
15	0	0	0	0	1	1	1	1	$425 + 17 = 442$
16	0	0	0	1	0	0	0	0	$442 + 5(17) = 527$

Thus, our cost is scaled by a factor of 17 compared to the in-class example. From our discussion in class, total cost of n increment operations was $\sum_{i=0}^{k-1} \frac{n}{2^i} \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$.

With the new cost of 17, the total cost is scaled by a factor of 17: $17\sum_{i=0}^{k-1} \frac{n}{2^i} \leq 17\sum_{i=0}^{\infty} \frac{n}{2^i} = 34n \approx O(n)$.

Part b:

Similar to part a, the total cost will be scaled by a factor of i, yielding:

$\sum_{i=0}^{k-1} \frac{i}{2^i} \leq \sum_{i=0}^{\infty} \frac{i}{2^i}$

$S = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + ...$

$S = \frac{1}{2^1} + \frac{1+1}{2^2} + \frac{1+2}{2^3} + \frac{1+3}{2^4} + \frac{1+4}{2^5} + ...$

$S = \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{2}{2^3} + \frac{1}{2^4} + \frac{3}{2^4} + \frac{1}{2^5} + \frac{4}{2^5} + ...$

$S = (\sum_{i=0}^{\infty} \frac{1}{2^i}) + (\frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + ...)$

$S = (\sum_{i=0}^{\infty} \frac{1}{2^i}) + \frac{1}{2}(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + ...)$

$S = (\sum_{i=0}^{\infty} \frac{1}{2^i}) + \frac{1}{2}S$

$\frac{1}{2}S = 1$

$S = 2$

Thus, the total cost with each $b_i = i$ is: $n\sum_{i=0}^{\infty} \frac{i}{2^i} = 2n \approx O(n)$

The following proof will lend itself well in Parts (c) and (b):

Observe that for the finite sum: $\sum_{i=0} k-1$, represents the amount of bits in a k-bit representation.

However, for n INCREMENT operations, for $n \leq (k-1)$, we have n as the result for the number.

Thus, we should compute the costs in terms of n rather than k.

Assuming that we are in an unsigned binary number system, and have performed n INCREMENT operations, the number we wish to represent in binary is n. Suppose that $N \in \mathbb{N}$ requires d digits in a base 2 representation. We claim that $d = \lfloor \log_2(N) \rfloor + 1$.

Proof:

The largest value that N for a binary representation is $N = 1\ldots1$, that is d 1's in a row. But we know that: $\sum_{i=0}^{d-1} 2^i = 2^d - 1$

At a minimum, the first digit is a 1, and the rest are zeroes, as the powers start from left to right. This is $2^d - 1$.

We now can say that: $2^{d-1} \leq N \leq 2^d - 1$

$\implies (d-1) \leq \log_2(n) \leq \log_2(2^d - 1)$

As $d-1$ is an integer, $\lfloor d-1 \rfloor = d-1$. Since $2^{d-1} \leq 2^d - 1 < 2^d$, we know that: $(d-1) = \log_2(2^{d-1}) \leq \log_2(2^d - 1) < \log_2(2^d) = d$

Thus, $\left\lfloor \log_2(2^d - 1) = d-1 \right\rfloor$ so we have: $(d-1) \leq \lfloor log_2(N) \rfloor \leq \left\lfloor \log_2(2^d - 1) \right\rfloor = d-1$

$\implies (d-1) \leq \lfloor \log_2(N) \rfloor \leq (d-1)$

So, $(d-1) = \lfloor \log_2(N) \rfloor \iff d = \lfloor \log_2(n) \rfloor + 1$

Applying this to a $n \leq 2^k - 1$, where k is the number of bits we allow, we know that the largest number we support in binary is $k-1$ 1's. In other words, $2^k - 1$. So, applying what was just proven, we know that for a number n that was the result of n INCREMENTS, it will require: $\lfloor \log_2(n) \rfloor + 1$ bits to represent. We will denote this result (*), and use it for parts c and d. We also know that $\lfloor \log_2(n) \rfloor + 1$ is an integer.

Part c:

Similar to parts a and b, the total cost will be scaled by a factor of i, yielding:

$n\sum_{i=0}^{k-1} \frac{2^i}{2^i} = n\sum_{i=0}^{k-1} 1 = k$

From (*), we know that in order to represent n in binary, we will require $\lfloor \log_2(n) \rfloor + 1$ which will always be at most k bits.

So, we can replace k with $\lfloor \log_2(n) \rfloor + 1$, as more likely than not, we will use $\lfloor \log_2(n) \rfloor + 1$ than all k bits.

Thus, we have:

$n\sum_{i=0}^{\lfloor \log_2(n) \rfloor + 1} \frac{2^i}{2^i} = n\sum_{i=0}^{\lfloor \log_2(n) \rfloor + 1} 1 = \lfloor \log_2(n) \rfloor + 1 + 1$

$= \lfloor \log_2(n) \rfloor + 2$

Thus, the total cost with each $b_i = 2^i$ is $n(\lfloor \log_2(n) \rfloor + 2) \approx O(n\log(n))$.

Part d: Similar to parts a,b and c, the total cost will be scaled by a factor of i, yielding:
$n\sum_{i=0}^{k-1}\frac{i2^i}{2^i} = n\sum_{i=0}^{k-1} i$ From (*), we know that in order to represent n in binary, we will require $\lfloor\log_2(n)\rfloor + 1$ which will always be at most k bits.

So, we can replace k with $\lfloor\log_2(n)\rfloor + 1$, as more likely than not, we will use $\lfloor\log_2(n)\rfloor + 1$ than all k bits.

Thus, we have:
$n\sum_{i=0}^{\lfloor\log_2(n)\rfloor+1}\frac{i2^i}{2^i} = n\sum_{i=0}^{\lfloor\log_2(n)\rfloor+1} i$

A simple application of Gauss' formula yields:
$n\sum_{i=0}^{k-1} i = n\frac{(\lfloor\log_2(n)\rfloor+1-1)(\lfloor\log_2(n)\rfloor+1)}{2}$
$= n\frac{(\lfloor\log_2(n)\rfloor)(\lfloor\log_2(n)\rfloor+1)}{2}$

Thus, the total cost with each $b_i = i2^i$ is $n\frac{(\lfloor\log_2(n)\rfloor)(\lfloor\log_2(n)\rfloor+1)}{2} \approx O(n\log^2(n))$.

Question 2.

Solution: Consider the decimal system, below is a very small example of 10 INCREMENT operations:

A[1]	A[0]	Total Cost
0	0	0
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9
1	0	11

Here we can see that digit 0 changed n times for n operations. We can also see that digit 1 changed $\frac{n}{10}$ times for n operations.

Extending this to base-d, we get digit i changes $\frac{n}{d^i}$ times.

Thus, the total cost is for k total digits in base-d is: $\sum_{i=0}^{k-1}\frac{n}{d^i} \leq n\sum_{i=0}^{\infty}\frac{1}{d^i}$

Since $d > 1$, we can evaluate the geometric series $\sum_{i=0}^{\infty}\frac{1}{d^i}$ as:
$\sum_{i=0}^{\infty}\frac{1}{d^i} = \frac{1}{1-\frac{1}{d}}$
$= \frac{d}{d-1}$

Therefore, the total cost of n INCREMENT operations in base-d is: $\sum_{i=0}^{k-1}\frac{n}{d^i} \leq n\frac{d}{d-1}$

Technically, (*) should be used, but since we find the cost by evaluating the infinite series, we will arrive at the same result, even if we replace $(k-1)$ with $\lfloor\log_2(n)\rfloor + 1)$, but this will not impact our analysis, as seen below.

The worst-case time for a sequence of n INCREMENT operations on an initially zero counter in base-d is therefore $O(n\frac{d}{d-1})$. The amortized cost per operation is therefore $\frac{O(n\frac{d}{d-1})}{n} = O(\frac{d}{d-1})$.

So, the amortized cost per operation depends on the number of digits. Intuitively, this makes sense. The more digits that a number system has, the likelihood of a digit flip decreases, as we must exhaust all possibilities before reaching a flip.

Question 3.

Solution: In the array doubling example, we set the potential function $\varphi = 2n - s$, where s was the array size (capacity) and n was the actual number of elements in the array.

In order to guarantee that the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$, we must find φ such that $\varphi(D_n) \geq \varphi(D_0)$, and that $\forall \varphi$, $\varphi \geq 0$.

If we consider $\varphi = cn - s$, where c is a positive constant, to guarantee that the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$, we must show that $cn - s \geq 0, \forall \varphi(D_i)$.

We must now show under what conditions $cn - s \geq 0$.

When we resize, prior to inserting the new element, we have that $s = 2s$, and $n = s$. In other words, $2n - s \rightarrow 2(\frac{1}{2}s) - s \approx 0$, as the addition of the new element makes this slightly larger than 0.

We now consider two possibilities:

Case I: c < 2

In this case, we have $cn - s \rightarrow c(\frac{1}{2}s) - s$. Since,

$0 < c < 2 \implies 0 < c\frac{1}{2} < 1$

$\implies 0 < c\frac{1}{2}s < s$

Thus, in the case that $c < 2$, $cn - s$ is negative and results in a loss of potential and cannot be used to provide a bound on the amortized complexity of an insert.

We now consider the other case:

Case I: $c \geq 2$

In this case, we have $(cn - s) \rightarrow c(\frac{1}{2}s) - s$. Since,

$2\ c \geq 2 \implies 1 \leq c(\frac{1}{2})$

$\implies s \leq c(\frac{1}{2}s)$

So in the case that $c \geq 2$, $(cn - s) \geq 0$.

Therefore, when $c \geq 2$, the amortized cost of an insert is an upper bound for the actual cost of an insert.

The amortized complexity, $\hat{c}_i$ can be computed like so:

In the simple case, where we do not resize, we have:

$\hat{c}_i = c_i + \Delta\varphi$

$= 1 + [(c(n+1) - s) - (cn - s)]$

$= 1 + [(cn + c) - s - cn + s]$

$= 1 + [c]$

In the case where we need to resize, the cost is $1 + n$. 1 to insert the new element and n to create the new array and move the elements:

$\hat{c}_i = c_i + \Delta\varphi$

$= (n+1) + [(c(n+1) - 2s) - (cn - s)]$

$= (n+1) + [(cn + c - 2s) - cn + s]$

$= (n+1) + [c - s]$

$= s + 1 + c - s = 1 + c$

In either case, we see that the amortized complexity of an insert is computed as: $(1 + c)$ where $c \geq 2$. If $c < 2$, $cn - s < 0$, and the amortized cost is not an upper bound on the actual cost, and we cannot use the potential method in such a case.

Question 4.

Solution: In analyzing the complexity of the Multipop array stack with doubling, let's consider using the Accounting method:

The actual costs of the operations are:

MULTIPOP	k
PUSH	1
DOUBLE	n

where $k \leq n$, and n is the total number of elements in the Multipop array stack.

Let's define the amortized cost as so:

PUSH:

When we call PUSH, it will have a cost of 1, and credit of 2. We save 1 for **MULTIPOP(k)** and 1 for when we need to call **DOUBLE**.

MULTIPOP:

When we call MULTIPOP(k), we have multiple cases.

Case 1: $k < n$

When $k < n$, we know that $n < (2n - k) < 2n$, so we will always have enough credit to cover the costs when $k < n$.

Case 2: $k = n$

When $k = n$, we know that $(2n - k) = n$, so we will always have enough credit to cover the costs when $k = n$

DOUBLE:

First, we let N be the maximum capacity of the array. When we need to double, n, the total number of items will be N.

When we double, we know that we have done n PUSHES, with cost n and credit $2n$. We must consider multiple cases in our analysis of the **DOUBLE** operation.

Case 1: MULTIPOP before DOUBLE:

We first perform n **PUSH** operations, then perform **MULTIPOP(k)** where $k < n$, so at this point the cost for the n **PUSH** operations was n with credit $2n$, As $k < n$, we know that $n < (2n - k) < 2n$, so we will always have enough credit to cover the costs when $k < n$. In order to trigger a **DOUBLE** operation, we must get the array to full capacity, call this N. We know with the n operations, it is $< N$, and $(n - k) < n$. Therefore, we require $(n - k) + (k + c) = N$ **PUSH** operations to trigger a **DOUBLE** operation, where c is the number of **PUSH** operations to trigger a **DOUBLE** operation from an array of n elements.

The intuition behind adding $(k + c)$ is that we must perform k pushes to bring the array back to n elements before the **MULTIPOP(k)** was performed. We then add on c to get to full capacity and trigger a **DOUBLE** operation. The cost of performing the $(k + c)$ **PUSH** operations will be $(k + c)$ with credit $2(k + c)$.

The total credit before the *DOUBLE* operation for the $(2n - k) + (2k + 2c)$, since we know that when $k < n$:

$n < (2n - k) < 2n$

$\implies n < (2n - k) + (2k + 2c)$

Since the **DOUBLE** operation costs n, we know that we will always have enough credit to cover the costs when $k < n$.

If $k = n$, then we know that $(2n - k) = n$, and we must perform $(n + c)$ **PUSH** operations in order to get to full capacity and trigger a **DOUBLE** operation. The cost of performing the $(k + c)$ **PUSH** operations will be $(k + c)$ with credit $2(n + c)$.

The total credit before the *DOUBLE* operation for the $(2n - k) + (2k + 2c)$, since we know that when $k = n$ is therefore: $n + 2n + 2c = 3n + c$, and since the cost of performing **DOUBLE** is N, where $N = (n + c)$, we know that we will always have enough credit to cover the costs.

Case 2: MULTIPOP AFTER DOUBLE: We first perform $n = N$ **PUSH** operations, in order to trigger the **DOUBLE** operation. At the point of calling the **DOUBLE** operation, we know that the N **PUSH** operations cost N with credit $2N$. Since the actual cost of the **DOUBLE** operation is N, we will be able to cover the **DOUBLE**. There are two cases that arise when we call **MULTIPOP** after **DOUBLE**.

$\mathbf{k < n}$: When $k < n$, we know that $(n - k) < n$, so we will always have enough credit to cover the costs when $k < n$, as the credit is n.

$\mathbf{k = n}$: When $k = n$, we know that $(2n - k) = n$, so we will always have enough credit to cover the costs when $k = n$, in fact it will be able to exactly match it.

Observe that if we were to do **PUSH** operations to trigger **DOUBLE**, it would fall under Case 2.

Case 3: Normal Doubling: We first perform $n = N$ operations to trigger a **DOUBLE** operation. We know that the credit of N **PUSH** operations will be $2N$. When a **DOUBLE** operation is performed, the real cost N. Since we have at least $2N$ in credit, we will always have enough to cover the costs.

Thus, we have proved in every case of the **DOUBLE** operation, it will always have enough credit to cover the costs. Note that if we were to do a sequence of m **PUSH**, **MULTIPOP**, and **DOUBLE** operations, we would always have enough credit to cover the costs. The costs and credit may look a bit different, but algebraically, it would still work out.

In particular, since we will always have credit that is more than n, , we will always be able to cover the cost of doubling in any case.

Therefore, all three operations will have a constant amortized cost, $O(1)$. Specifically,

MULTIPOP	3
PUSH	0
DOUBLE	0

It will in fact cost us 0 for **MULTIPOP** and **DOUBLEs** as we have paid for it in advance with **PUSH**. The the total amortized cost is an upper bound on the total actual cost, and for a sequence of n operations, it is $O(n)$.