Amortized Analysis

Objectives

By the end of this module, you will be able to:

- Explain what is amortized analysis
- Define amortized cost
- List three methods of analyzing algorithm complexity more efficiently
  - Aggregate analysis, the accounting method, and the potential method
- Examine the dynamic tables example
- Examine the stack multi-pop example
- Examine the binary counter example

Reading Assignment

- Cormen, Chapter 17
- Module Jupyter Notebook: Amortized analysis

Introduction

Generally algorithms perform several operations. The naive method of assessing the complexity is by examining the loops in an algorithm, examine the worst case cost and assigning complexities. A more efficient way of examining the complexity is the time per operation, such as counting the number of operations.

The motivation for **amortized analysis** is that looking at the worst-case time per operation can be too pessimistic. For example, consider the case when an algorithm performs many cheap operations beforehand and then execute a large operation.

Examples:

1. Washing individual dishes (fast per operation) *versus* washing them in a dishwasher (fast in aggregate)
2. Building a dam. Initial cost is huge but in the long term each operation will be very cheap

Amortized Analysis Methods ¶

1. Aggregate analysis
2. The accounting method
3. The potential method

Aggregate Analysis

For a sequence of n operations, the worst case cost is T(n). Then, in the worst case, the amortized cost (average cost) per operation is T(n)/n.

Definition: The **amortized cost** of n operations is the total cost of the operations divided by n.

Example: Array Stack (Dynamic Tables)

Implement a stack as an array as in the following.

```
def push(x):
    A[top] = x; top += 1

def pop():
    top -= 1; return A[top]
```

Question: What happens when `A` array is full (e.g. implemented as a `numpy` array)?

Answer: Resize the array, copy the content from the previous array.

- Case 1: Resize one by one, i.e. add 1 more space to the array
  - After n pushes, the resizing cost, $T(n) = \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2}$
- Case 2: Resize by doubling the capacity, when needed
  - After n pushes, the resizing cost, $T(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} 2^i = 1 + 2 + 4 + \ldots + 2^i = 2n - 1$

The Accounting Method

Assign different costs to different operations which might differ by an order of magnitude. Use the amortized cost as the amount we charge and when amortized cost is higher than the actual cost the the difference is stored in a credit object. The cumulative credit should not be negative.

Specifically, $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, where $\hat{c}_i$ and c_i are the amortized and regular costs of the operation i, respectively.

Total credit stored = $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \underset{required}{\geq} 0$

Example: Multi-pop

Define the multi-pop `mpop` stack function to pop and return `k` elements from a stack, and assume `k < top`.

```
def mpop(k):
    return [pop() for _ in range(k)]
```

A. **Aggregate analysis:** cost of `push` = $\mathrm{O}(1)$; cost of `pop` = $\mathrm{O}(1)$; cost of `mpop` = $\mathrm{O}(k)$

However the cost of `mpop` can be at most $\mathrm{O}(n)$ because there are n many elements in the stack. Therefore, `mpop` cost = $\frac{\mathrm{O}(n)}{n}$

B. The **Accounting method:** assign cost of `push` = 2, and cost of `pop` = cost of `mpop` = 0. Total amortized cost for n pushes is $\mathrm{O}(2n)$.

Example: Binary Counter

Consider binary counters and binary number incrementing process as in the following cell. Note that this is a representation of a similar large scale problems and each bit manipulation is the atomic operation that will go to the cumulative cost.

Assume that the length of the counter is k and with `A[k-1]` as the low order and `A[0]` as the high order bits, like a string. Notice that each increment operation involves different number of bits. The operation cost is defined as flipping the number of bits. Example, if 3 bits are flipped as in `011` $\rightarrow$ `100` then the cost is 3. (Note that setting is `0` $\rightarrow$ `1` and resetting is `1` $\rightarrow$ `0`)

A. Aggregate analysis can naively report that for a k-bit counter, every bit has to be examined so the complexity would be $\mathcal{O}(nk)$ for n increments. However as can be seen in the below cell, the bit `i` is flipped every $\left\lfloor \frac{n}{2^i} \right\rfloor$ times. Thus, using aggregate analysis, the cost is $\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$

B. The accounting method sets a **credit** or pay ahead when setting a bit to `1` in the amount of 2:

- Setting a bit to 1: cost 2
- Resetting a bit to 0: cost 0 (paid before)

And this scheme results in 2n operations for n increments which agree with the aggregate analysis method.

In [1]:

```
from IPython.display import Markdown

span1 = '<span style=\"font-family:Courier; font-size:12pt; white-spa
ce:pre; \">'
span2 = '<span style=\"font-family:Courier; font-size:12pt; white-spa
ce:pre; font-weight:bold; \">'

# bold shows the bits that change or flipped
def get_binwork(_i0, _i1):
    dbg = ''
    for bit0, bit1 in zip(f'{_i0:08b}', f'{_i1:08b}'):
        dbg += (span2 if bit0!=bit1 else span1) + bit1 + '</span>'
    return dbg

dbg = f"{span2}{'Count':5s}  {'Array A[]':9s}  {'Cost':4s}  {'Tot Cos
t':8s} </span><br>"
totcost, i0 = 0, 0
for i1 in range(17):
    cost = (i0 ^ i1).bit_length()
    totcost += cost
    dbg += f"{span1}{i1:^5d}   </span>" + f"{get_binwork(i0, i1)}{spa
n1}   {cost:^4d}  {totcost:^8d}</span><br>"
    i0 = i1

display (Markdown(dbg))
```

Count	Array A[]	Cost	Tot Cost
0	00000000	0	0
1	0000000**1**	1	1
2	000000**10**	2	3
3	0000001**1**	1	4
4	00000**100**	3	7
5	0000010**1**	1	8
6	000001**10**	2	10
7	0000011**1**	1	11
8	0000**1000**	4	15
9	0000100**1**	1	16
10	000010**10**	2	18
11	0000101**1**	1	19
12	00001**100**	3	22
13	00001101	1	23
14	00001**10**	2	25
15	00001111	1	26
16	000**10000**	5	31

The Potential Method

The potential method is similar to the accounting method and it requires operations to pay ahead. However, the actual interpretation of the potential method is similar to managing energy in a system. Specifically, as some operations are performed, potential energy is introduced into the system. Then, other operations can release this potential energy. As long as the potential energy remains non-negative, the model works.

In general, the potential method starts with an initial data structure, D_0. Let c_i represent the cost of the i^{th} of n operations. Let D_i correspond to the data structure after the i^{th} operation has been performed, and assume the operation was performed on data structure D_{i-1}. Then define a potential function $\Phi()$ that maps a data structure D_i to a real number $\Phi(D_i)$. Or, $\Phi : D \rightarrow \mathbb{R}$.

Define the amortized cost of operation i as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ ..... (Eq.1)

Thus, $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0)$

As long as we define $\Phi()$ such that $\Phi(D_n) \geq \Phi(D_0)$, then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

Multi-pop

$\Phi(D_0) = 0$ and $\Phi(D_i) = s_i \geq 0$, i.e. there are s items in the stack.

Given $c_i = 1$ and using (Eq.1) in the previous cell, `push` amortized cost =
$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + s + 1 - s = 2$.

Similarly, `pop` amortized cost = $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + (s-1) - s = 0$.

`mpop` amortized cost = $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + (s-k) - s = k + (s-k) - s = 0$.

Array Stack

Let num_i denote the number of elements in T, and let $size_i$ denote the allocated size of T (i.e. the capacity) following the i^{th} operation. As usual, Φ_i is the potential after the i^{th} operation. Initially, $num_0 = size_0 = \Phi_0 = 0$. On the i^{th} operation, *if there is no expansion,* $size_i = size_{i-1}$.

Load factor $\alpha = \frac{num}{size}$

Set potential, $\Phi(T) = 2num[T] - size[T]$

- Just before the expansion, $size = num \implies \Phi = num \implies$ have enough potential to pay for moving all items
- Just after the expansion (doubling the previous size, i.e.$\alpha = 0.5$), $size = 2num \implies \Phi = 0$
- $\Phi \geq 0$ is required

Case 1. No expansion: Then, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$
$\hat{c}_i = 1 + 2num_i - size_i - (2(num_i - 1) - size_i) = 1 + 2 = 3$.

Case 2. Expansion: Then, $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1})$
$\hat{c}_i = num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3$

In [2]:

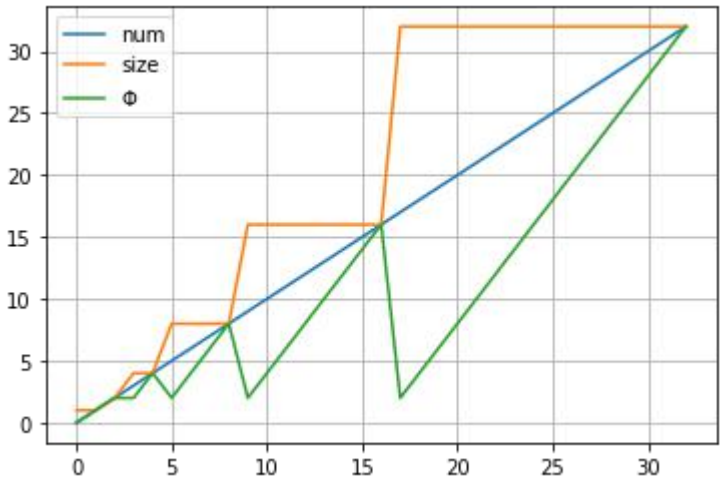
```
%matplotlib inline
import matplotlib.pyplot as plt

def get_potential(n):
    num, size, phi = [0]*n, [1]*n, [0]*n
    for i in range(1,n):
        # expand when necessary
        size[i] = 2*size[i-1] if num[i-1]/size[i-1] >= 1 else size[i-
1]

        num[i] = i
        phi[i] = 2*num[i]-size[i]
    return num, size, phi

num, size, phi = get_potential(33)

plt.plot(num, label='num')
plt.plot(size, label='size')
plt.plot(phi, label='$\Phi$')
plt.legend()
plt.grid()
plt.show()
```



Binary Counter

Naturally, $\Phi = b_i =$ the number of `1`'s after i^{th} increment.

Assume i^{th} operation resets t_i bits to `0`. Then, $c_i \leq t_i + 1$ (resets t_i bits, and sets $\leq$ `1` many bits)

Case 1. If $b_i = 0$, then the i^{th} operation reset all k bits and did not set any, so that
$b_{i-1} = t_i = k \implies b_i = b_{i-1} - t_i$

Case 2. If $b_i > 0$, then the i^{th} operation reset t_i bits and set 1, so that $b_i = b_{i-1} - t_i + 1$

Both cases combined, $b_i = b_{i-1} - t_i + 1$

Thus, $\Delta\Phi(D_i) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$. Then, $\hat{c}_i = c_i + \Delta\Phi(D_i) \leq (t_i + 1) + (1 - t_i) = 2$.
Amortized cost of n operations is $\mathrm{O}(2n)$.