AMORTIZED ANALYSIS

In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

The three most common techniques used in amortized analysis are :

1.**The Aggregate analysis method :** In this method we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The amortized cost per operation is then $T(n)/n$.

2.**The Acounting method** : In this method we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. The credit is used later in the sequence to pay for operations that are charged less than they actually cost.

3. **The Potential method** : This method is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure instead of associating the credit with individual objects within the data structure.

Two examples are given below to illustrate these three method. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

The insight into a particular data structure gained by performing an amortized analysis can help in optimizing the design.

1.The Aggregate method

In the aggregate method of amortized analysis, we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $T(n) / n$. Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.

Example 1 : Stack operations

The first example of the aggregate method, we analyze stacks that have been augmented with a new operation. The two fundamental stack operations (i.e. push and pop), each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S.

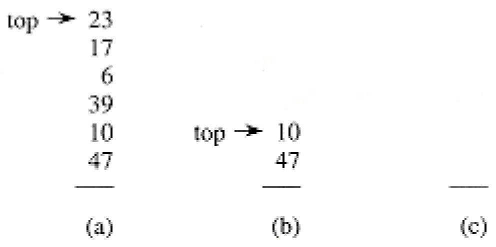
POP(S) pops the top of stack S and returns the popped object.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

The situation becomes more interesting if we add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, or pops the entire stack if it contains less than k objects. In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

```
MULTIPOP($S,k$)
1  while not STACK-EMPTY($S$) and $k \neq 0$
2        do POP($S$)
3           $k \leftarrow k - 1$
```

The actual running time of MULTIPOP(S, k) on a stack of s objects is linear in the number of POP operations actually executed, and thus it suffices to analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number min(s, k) of objects popped off the stack. For each iteration of the loop, one call is made to POP in line 2. Thus, the total cost of MULTIPOP is min(s, k), and the actual running time is a linear function of this cost.



The above figure shows the action of MULTIPOP on a stack S, shown initially in (a). The top 4 objects are popped by MULTIPOP(S, 4), whose result is shown in (b). The next operation is MULTIPOP(S, 7), which empties the stack←shown in (c)←since there were fewer than 7 objects remaining.

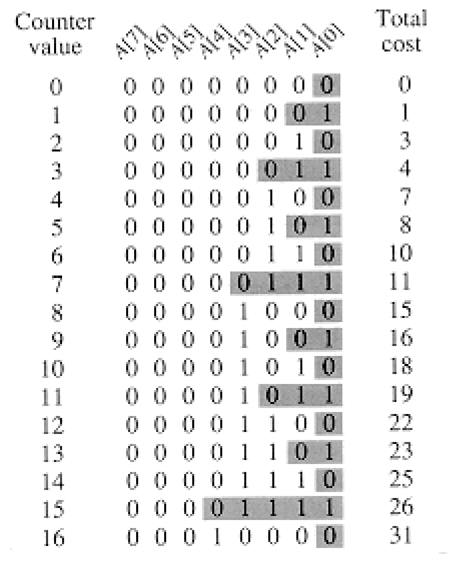
Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, obtained by considering the worst-case cost of each operation individually, is not tight.

Using the aggregate method of amortized analysis, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$. Because each object can be popped at most once for each time it is pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n. For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. The amortized cost of an operation is the average: $O(n)/n = O(1)$.

We emphasize again that although we have just shown that the average cost, and hence running time, of a stack operation is $O(1)$, no probabilistic reasoning was involved. We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Example 2 : Incrementing a binary counter

As another example of the aggregate method, consider the problem of implementing a k-bit binary counter that counts upward from 0. We use an array $A[0 \, . \, . \, k - 1]$ of bits, where *length*$[A] = k$, as the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k - 1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k - 1$. To add 1 (modulo 2^k) to the value in the counter, we use the following procedure.



Counter value	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

The above figure shows an 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is never more than twice the total number of INCREMENT operations.

```
INCREMENT(A)
1  i ← 0
2  while i< length[A] and A[i] = 1
3         do A[i] ← 0
4            i ← i + 1
5  if i < length[A]
6       then A[i] ← 1
```

This algorithm is essentially the same one implemented in hardware by a ripple-carry counter. The above figure shows what happens to a binary counter as it is incremented 16 times, starting with the initial value 0 and ending with the value 16. At the start of each iteration of the **while** loop in lines 2-4, we wish to add a 1 into position i. If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that adding a 1 into position i, flipping the 0 to a 1, is taken care of in line 6. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array A contains all 1's. Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

We can tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of n INCREMENT'S by observing that not all bits flip each time INCREMENT is called. As the last figure shows, $A[0]$ does flip each time INCREMENT is called. The next-highest-order bit, $A[1]$, flips only every other time: a sequence of n INCREMENT operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT'S. In general, for $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i > \lfloor \lg n \rfloor$, bit $A[i]$ never flips at all. The total number of flips in the sequence is thus

$$\begin{aligned}\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n ,\end{aligned}$$

by equation. The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$, so the amortized cost of each operation is $O(n)/n = O(1)$.

2 .The accounting method

In the accounting method of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its amortized cost. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as credit. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. This is very different from the aggregate method, in which all operations have the same amortized cost.

One must choose the amortized costs of operations carefully. If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. Moreover, as in the aggregate method, this relationship must hold for all sequences of operations. Thus, the total credit associated with the data structure must be nonnegative at all times, since it represents the amount by which the total amortized costs incurred exceed the total actual costs incurred. If the total credit were ever allowed to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Example 1. Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min(k,s)$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are $O(1)$, although in general the amortized costs of the operations under consideration may differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. a. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we put on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate is prepayment for the cost of popping it from the stack. When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we needn't charge the POP operation anything.

Moreover, we needn't charge MULTIPOP operations anything either. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged at least enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Example 2.Incrementing a binary counter

As another illustration of the accounting method, we analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we needn't charge anything to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.

The amortized cost of INCREMENT can now be determined. The cost of resetting the bits within the **while** loop is paid for by the dollars on the bits that are reset. At most one bit is set, in line 6 of INCREMENT, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars. The number of l's in the counter is never negative, and thus the amount of credit is always nonnegative. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

3. The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the potential method of amortized analysis represents the prepaid work as "potential energy,"or just "potential," that can be released to pay for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

The potential method works as follows. We start with an initial data structure D_0 on which n operations are performed. For each i = 1, 2, . . . , n, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A potential function Φmaps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i. The amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$
....eq. (1)

The amortized cost of each operation is therefore its actual cost plus the increase in potential due to the operation. By first equation(eq. 1), the total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \ .$$
....eq. (2)

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost. In practice, we do not always know how many operations might be performed. Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we guarantee, as in the accounting method, that we pay in advance. It is often convenient to define $\Phi(D_0)$ to be 0 and then to show that $\Phi(D_i) \geq 0$ for all i.Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the actual cost of the operation is paid by the decrease in the potential.

The amortized costs defined by equations (eq.1) and (eq.2) depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. There are often trade-offs that can be made in choosing a potential function; the best potential function to use depends on the desired time bounds.

Example 1.Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\Phi(D_i) \geq 0$$
$$= \Phi(D_0).$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s$$
$$= 1 .$$

By equation (eq.1), the amortized cost of this PUSH operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Suppose that the ith operation on the stack is MULTIPOP(S,k) and that $k' = min($k,s$)$ *objects are popped off the stack. The actual cost of the operation is* k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Example 2.Incrementing a binary counter

As another example of the potential method, we again look at incrementing a binary counter. This time, we define the potential of the counter after the ith INCREMENT operation to be b_i, the number of 1's in the counter after the ith operation.

Let us compute the amortized cost of an INCREMENT operation. Suppose that the ith INCREMENT operation resets t_i bits. The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to a 1. The number of 1's in the counter after the ith operation is therefore $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1}$$
$$= 1 - t_i.$$

The amortized cost is therefore

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is O(n).

The potential method gives us an easy way to analyze the counter even when it does not start at zero. There are initially b_0 1's, and after n INCREMENT operations there are b_n 1's, where $0 \le b_0$, $b_n \le k$. We can rewrite equation (2) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

....eq.3

We have $\widehat{c}_i \le 2$ for all $1 \le i \le n$. Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 .\end{aligned}$$

Note in particular that since $b_0 \le k$, if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Application : Dynamic tables

In some applications, we do not know in advance how many objects will be stored in a table. We might allocate space for a table, only to find out later that it is not enough. The table must then be reallocated with a larger size, and all objects stored in the original table must be copied over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. This problem of dynamically expanding and contracting a table is discussed here. Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, we shall see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single slot, that is, a space for one item. Likewise, TABLE-DELETE can be thought of as removing an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant; we might use a stack, a heap, or a hash table. We might also use an array or collection of arrays to implement object storage.

We shall find it convenient to use a concept introduced in our analysis of hashing .We define the load factor $\alpha(T)$ of a nonempty table T to be the number of items stored in the table divided by the size (number of slots) of the table. We assign an empty table (one with no items) size 0, and we define its load factor to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table in which only insertions are performed. We then consider the more general case in which both insertions and deletions are allowed.

.

A Familiar Definition :

Load factor α = *num/size*, where *num* = items stored and *size* = the allocated size of the table.

For the boundary condition of *size* = *num* = 0, we will define $\alpha = 1$.

We never allow $\alpha > 1$ (no chaining).

Insertion Algorithm :

We'll assume the following about our tables.

When the table becomes full, we double its size and reinsert all existing items. This guarantees that $\alpha \geq 1/2$, so we are not wasting a lot of space.

```
Table-Insert (T,x)
  1  if T.size == 0
  2        allocate T.table with 1 slot
  3        T.size = 1
  4  if T.num == T.size
  5        allocate newTable with 2*T.size slots
  6        insert all items in T.table into newTable
  7        free T.table
  8        T.table = newTable
  9        T.size = 2*T.size
  10  insert x into T.table
  11  T.num = T.num + 1
```

Each *elementary insertion* has unit actual cost. Initially *T.num* = *T.size* = 0.

Aggregate Analysis of Dynamic Table Expansion

Charge 1 per elementary insertion (including copying items into a new table). Count only these insertions, since all other costs are constant per call.

c_i = actual cost of ith operation (number of items inserted).

- If the table is not full, $c_i = 1$ (for lines 1, 4, 10, 11).
- If full, there are i − 1 items in the table at the start of the ith operation. Must copy all of them (line 6), and then insert the ith item. Therefore $c_i = i - 1 + 1 = i$.

A sloppy analysis: In a sequence of n operations where any operation can be O(n), the sequence of n operations is O(n^2).

This is "correct", but inprecise: we rarely expand the table! A more precise account of c_i is based on the fact that if we start with a table of size 1 and double its size every time we expand it, we do the expansions when the number of items in the table is a power of 2:

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

Then we can sum the total cost of all c_i for a sequence of n operations:

$$\begin{aligned} \text{Total cost} &= \sum_{i=1}^{n} c_i \\ &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2-1} \\ &< n + 2n \\ &= 3n \end{aligned}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

Accounting Method Analysis of Dynamic Table Expansion

We charge CY$3 for each insertion into the table, i.e., the amortized cost of the i-th insertion as $\hat{c}_i = 3$. The actual cost of the i-th insertion is as before:

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

We must show that we never run out of credit

Informal Explanation

In case it helps to understand why this works:

We charge CY$3 for inserting some item x into the table. Obviously, we are overcharging for each simple insertion which costs only CY$1. However, the overcharging will provide enough credit to pay for future copying of items when the table becomes full:

- The first CY$1 pays for the actual cost of inserting x.
- The second CY$1 will pay for the cost of copying x into a new table the next time table becomes full.

But the table might need to be expanded more than once after x is inserted, so x might need to be copied more than once. Who will pay for future copying of x? That's where the third CY$1 comes in:

- The third CY$1 will pay for the cost of copying some other item currently in the table that had been copied at least once before.

Let's see why CY$3 is enough to cover all possible future exansions and copying associated with them.

- Suppose the capacity of the table is m immediately after an expansion. Then it holds m/2 items and no item in the table contains any credits.
- For any insertion of an item x we charge CY$3. CY$1 pays for the actual insertion of x; we place CY$1 credit on x to pay for the cost of copying it in the future, and we place CY$1 credit on some other item in the table that does not have any credit yet.
- We will have to insert m/2 items before the next expansion of the table. Therefore, by the time the table will get expanded next time (and, consequently, items need to be copied), every item of the table will have CY$1 credit associated with it and this credit will pay for copying that item.