減価償却法

アルゴリズム論

中野

Note 7　減価償却法

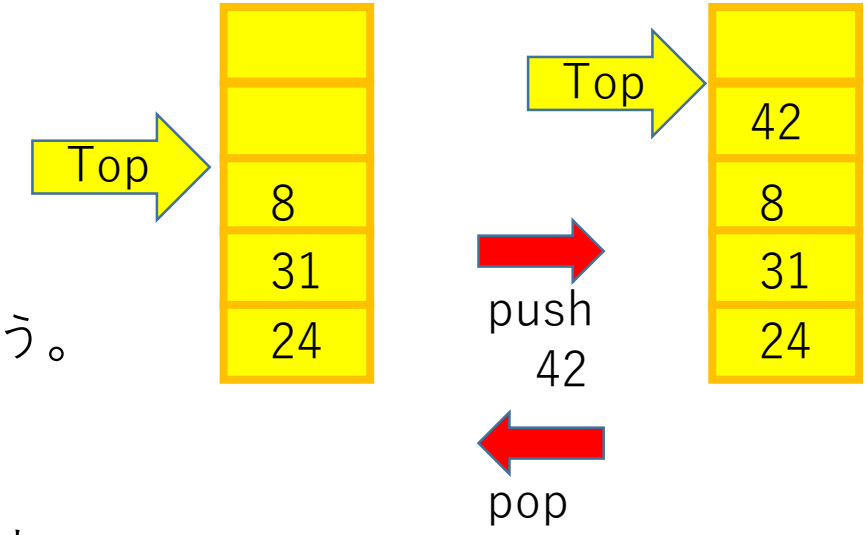
2020.5.11 作成　2021.05.24

Amortized Analysis (減価償却解析)(ならし解析)

アルゴリズムの解析の手法である。
(コードの中にはamortizedに関連することはでてこない。)

n個のoperationの列が、総計で、高々、T(n)時間かかる、
という解析である。
("高々"という制限が大切です。これに対して、確率的に平均○○時間というと大幅に平均をうわまわるものも、きわめて少ない確率でなら、存在するかもしれない。
これに対して、本手法では、最悪でも、n個のoperationで○○時間以内ということを保証します。)

スタック



下記の3つのoperationをサポートするスタックを考えよう。

PUSH(S,x)　　スタック S に　object x を積む
POP(S)　　スタック S から object をひとつ取り出す
MULTIPOP(S,k)　　スタック S から object をk個取り出す。
　　（ただし、S に objectが k-1個以下しかないときは、空になるまで取り出す。）

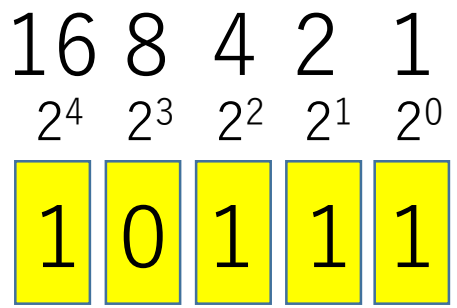
空のスタックS に n 回のoperationを実行するときを考えよう。
PUSH(S,x)はO(1)時間、
POP(S)はO(1)時間、
MULTIPOP(S,k)はO(k)時間かかる。これはO(n)時間であるともいえる。
よって、n 回のoperation では、計 $O(n^2)$時間かかる。

スタック（つづき）

しかし、各objectは、1回スタックに積まれたら、
1回取り出させるだけである。
よって、n 回のoperation において、
　スタックから取り出されるobjectの総数 ≦ スタックに積まれるobjectの総数
となるはずである。
よって、n回のoperationでは、
高々n個の object がスタックに積まれ、
高々 n 個の object がスタックから取り出されることになる。

したがって、空のスタックS に n 回のoperationを実行すると、
計算時間は、合計、O(n)時間となる。
また、各operationの、amortized 計算時間は O(1)時間である、という。

Incrementing a binary counter



0から順に、1,2,3,....を数える、k bitの2進カウンタをプログラムで実現するとしよう。
bit の配列 A[k] を用意し、
A[0] を最下位bitに対応させ、A[k-1] を最上位bitに対応させるとしよう。
値は $A[0]\cdot 2^0 + A[1]\cdot 2^1 + \ldots + A[k-1]\cdot 2^{k-1}$で計算できることになる。
はじめは $A[0] = A[1] = A[2] = \ldots = A[k-1] = 0$ とする。

```
INCREMENT(A)
i = 0
While    i < k    and    A[i] = 1 do
                 A[i] = 0
                 i = i + 1
if i < k
                 then A[i] = 1
```

1を0に

0を1に

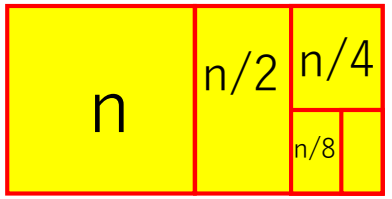
.

010100~~111111~~
010101000000

1回の INCREMENT(A) の計算時間は O(k)時間であるから、
n回の INCREMENT(A) の計算時間は、合計、O(kn)時間である。

Incrementing a binary counter (つづき)

しかし、
A[0]は毎回反転するが、
A[1]は2回に1回反転するだけであるし、
A[2]は4回に1回反転するだけである。
A[3]は8回に1回反転するだけである。

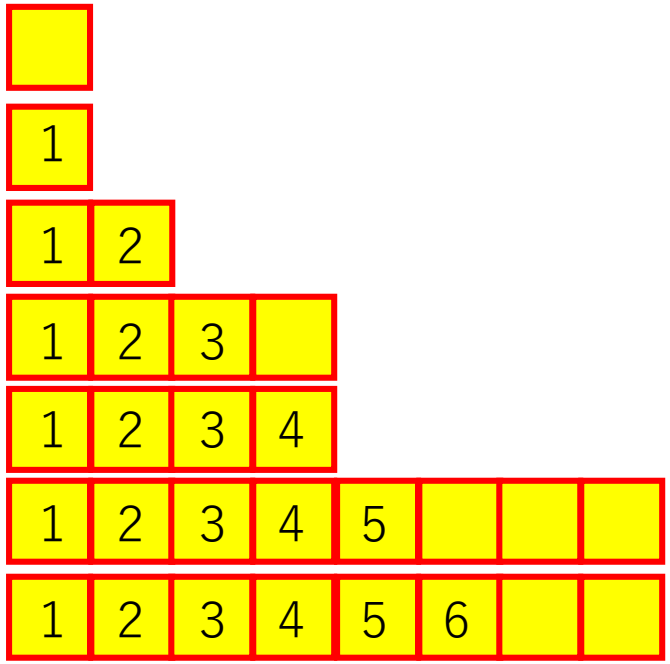


よって、
$\Sigma_{i=0,1,2,\ldots,\log n} n/2^i < 2n$

したがって、n回の INCREMENT(A)の計算時間は、
(最悪で、)合計、O(n)時間となる。
また、1回の INCREMENT(A)の amortized 計算時間は
O(1)時間である、という

00000
00001
00010
00011
00100
00101
00110
00111
01000
01001
01010
01011
01100
…

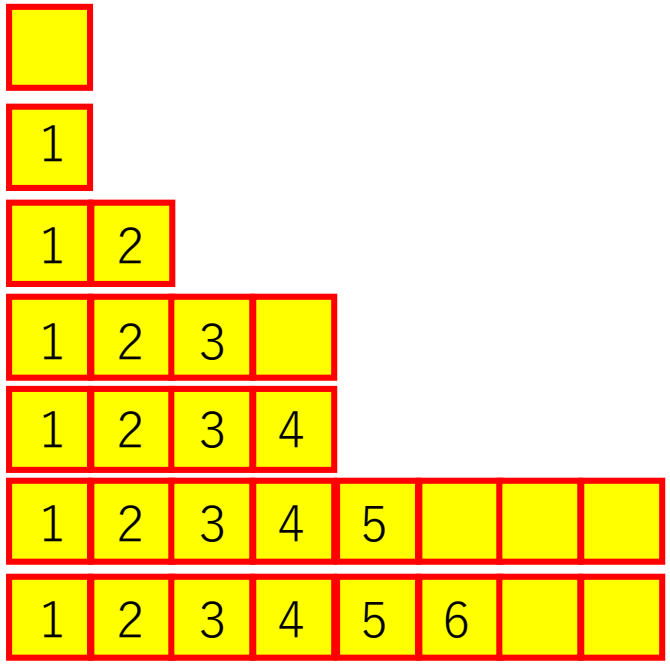
Dynamic Table



テーブル(配列・表としてよい)に、将来、
どのくらいの個数のobjectがはいるのかが予想できないとしよう。
それでも、テーブルの未使用率が
定数(たとえば50%)以下であるようにしたい。
objectは追加されるだけとしよう。(削除はされない。)

方針は、以下である。
はじめは、大きさが1のテーブルを用意しておく。はじめは、このテーブルは空である。
順次objectを追加するとき、
(1)もし,そのテーブルにまだ空きがあれば、そこにobjectを追加する。
(2)もし,そのテーブルにもう空きがなければ、倍のサイズのテーブルを
　準備し、旧テーブルをすべてコピーし、新たにできた空きにobjectを追加する。
としよう。

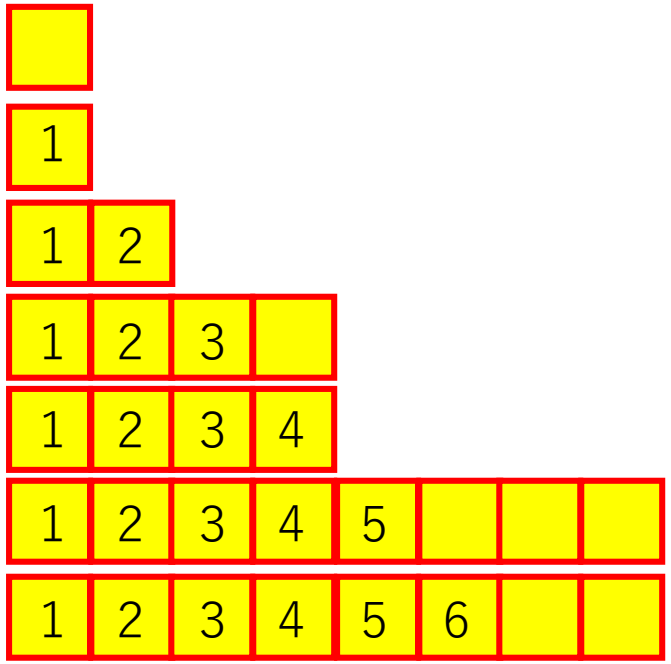
Dynamic Table （つづき）



つまり、
2回目の追加のときに、大きさ2のテーブルを作成し、
3回目の追加のときに、大きさ4のテーブルを作成し、
5回目の追加のときに、大きさ8のテーブルを作成し、
9回目の追加のときに、大きさ16のテーブルを作成し、…
のように進む。

n個のobjectを追加するとしよう。
i回目の追加を考えよう。
iが2のべき乗+1のときは、O(i)時間かかる!!!!!
(旧テーブルのコピーに i-1 回の演算が必要。
i番目のobject の新テーブルへの追加に1回の演算が必要。計i回の演算が必要。)
よって、n回のobjectの追加の計算時間は $O(n^2)$時間である。

Dynamic Table （つづき）



しかし、(amortized analysisを使えば。。。)
i 回目のobject追加のときに必要な演算の個数をc_iとすると、

$$\Sigma_{i=0,1,2,\ldots,n} c_i < n + \Sigma_{i=0,1,2,\ldots,\log n} 2^i < n + 2n = 3n$$

である。
よって、n回のobjectの追加の計算時間は合計 O(n)時間である。
よって、1回のobjectの追加のamortized計算時間は O(1)時間である。

objectが追加および削除されるときも同様に解析できる。
削除したとき、テーブルの使用率が50%以下になったら、
より小さいテーブルにコピーしなおす。