Amortised Analysis

- The idea of amortised analysis is *borrow from Peter to pay Paul*.
- More appropriately, save to pay for the expenses to be incurred in future.
- Some operations may require more time than necessary, but it makes other operation less time consuming. So the overall time is reduced considerably.
- So the time required can be averaged over all operations of a sequence of given operations resulting in overall reduction time.
- But it is not a probablistic analysis, no probabilities are involved.

Amortised Analysis

- Consider an n-bit binary counter $A[0 \ldots k-1]$.
- The increment operation on A could be

```
increment(A) {
   i = 0;
   while ((i < length(A)) && A[i]) {
      A[i] = 0; i = i+1;
   }
   if (i < length(A)) A[i] = 1;
}
```

- The worst case is when the `while` loop is executed O(k) times.
- So the cost of a single increment is O(k), and therefore, the cost of n increments is O(nk).

Analysis of Increment

- But if we realize that ith bit in the counter can be incremented only in intervals of 2^i, then the time

$$T(n) = \sum_{i=0}^{\log n} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- Thus the amortised time is $T(n)/n = O(1)$.

Multipop on a Stack

- Consider the execution of a sequence of O(n) `push`, `pop` and `multiPop` operations on an empty stack.
- Multipop on a stack can be written as follows:

```
multiPop(S, k) {
   while (!isEmpty(S) && k ≠ 0) {
      pop(S);
      k = k-1;
   }
}
```

- One stack operation consumes upto O(n) time.
- So, a straightforward worstcase time is O(n^2).
- But the time for the execution of `multiPop` operations can not exceed O($\#$ elements in push).
- So amortised time is O(1) per operation.

Amortised Analysis

- In amortised analysis we associate a charge to each operation with the invariant that at any step the total charge by new accounting method is an upper bound for the real cost.
- In the sequence of `push(x, S)`, `pop(S)` and `multiPop(k,S)` operations
  - **Real cost:** `push(x, S)` - 1 unit, `pop(S)` - 1 unit, and `multiPop(k, S)` - k units.
  - **New cost:** `push(x, S)` -2 units, `pop(S)` - 1 unit, and `multiPop(k, S)` - 0 units.

  Since, cost of a `multiPop(k, S)` will depend on the number of elements on the stack at the time of actually executing this operation, we can pay for this operation if we have charged each `push(x, S)` an extra unit earlier.
- So, the total cost at any time in our accounting new method exceeds total cost by earlier accounting method.

Counter Increments

$$\text{New charge: } \begin{cases} 2 \text{ units} & \text{for each flip to 1,} \\ 0 & \text{otherwise.} \end{cases}$$

- Since we start with all bits set to 0, this give an upper bound on the real cost.
- In each increment operation only one bit is flipped to 1 - O(1) amortized cost.
- Flipping from 1 to 0 can be paid from unit that was over charged to this bit at the time when it flipped from 0 to 1 earlier.
- So at each instance of time, the cost is an upper bound of the real cost.

Potential Method

- Suppose D_0 is the initial state of the data structure for an algorithm.
- We consider a set of n operations on the D_0.
- Each operation will effect the data structure.
- Let after operation i the modified data structure be D_i.
- Let $\Phi : \{D_0, D_1, \ldots, D_n\} \rightarrow R$.
- Amortised cost of operation i is

$$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- Total amortised cost is

$$\begin{aligned}\textstyle\sum_{i=1}^{n} A(c_i) &= \textstyle\sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \textstyle\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

- So if we know $\Phi(D_i) \geq \Phi(D_{i-1})$ then

$$\sum_{i=1}^{n} A(c_i) \geq \sum_{i=1}^{n} c_i$$

Multipop Operation

- Let $\Phi(D_i)$ = number of items in the stack.
- $\Phi(D_i) \geq \Phi(D_0) = 0$.
- The amortized cost of a push operation (starting with s items in the stack) is:

  $$A(c_i) = 1 + (s + 1) - s = 2$$

  The amortized cost of a `multiPop(S, k)` ($k' = \min(k, s)$)

  $$A(c_i) = k' + (s - k') - s = 0$$

- Thus, all operations have amortized cost O(1), the total cost of n operations is O(n).

Counter Increment

- Let $\Phi(D_i)$ = number of 1's in the counter.
- If counter starts with 0: $\Phi(D_i) \geq \Phi(D_0) = 0$.
- Assume that the i-th operation (increment) flipped t_i bits to 0.
- The total cost of that operation is $t_i + 1$.
- D_i has exactly $t_i - 1$ less 1 bits than D_{i-1}. So

  $$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 + 1 - t_i = 2.$$

- The total amortized cost of n operations is O(n). If the counter didn't start with 0:

  $$\begin{aligned} \textstyle\sum_{i=1}^{n} A(c_i) &= \textstyle\sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ & \textstyle\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

  or

  $$\begin{aligned} \textstyle\sum_{i=1}^{n} c_i &= \textstyle\sum_{i=1}^{n} A(c_i) + \Phi(D_0) - \Phi(D_n) \\ &\leq \textstyle\sum_{i=1}^{n} A(c_i) + k \end{aligned}$$

Analysis of UNION-FIND

- We talked about a executing a sequence of n `UNION` and `FIND` instructions in almost linear time.
- The sets were represented in the form of tree.
- The `UNION` was by size and `FIND` used path compression.
- Let us go into details of the analysis of the time complexity.
- We apply the accounting method for amortised analysis of this problem.

Some Definitions

- $F(.)$ is a fast growing function:

$$F(i) = \begin{cases} 1 & \text{for } i = 0 \\ 2^{F(i-1)} & \text{for } i > 0. \end{cases}$$

n	0	1	2	3	4	5
$F(n)$	1	2	4	16	65536	2^{65536}

- Define $G(.)$ as follows:

$$G(n) = \min\{k | F(k) \geq n\}.$$

- Since $F(5) = 2^{65536}$, and for almost all cases the value of $n \leq 2^{65536}$ we have $G(n) \leq 5$ for all practical purposes.

- Our aim to show that a sequence of cn `UNION`s and `FIND`s takes time of at most $c'G(n)$ time where c, and c' are constants, c' depends on c.

Definition of Ranks

- Delete all `FIND`s from the sequence σ, call the new sequence as σ'.
- Execute all `UNION`s of σ'.
- Rank of a vertex is the height of v in the resulting forest.
- There are at most $n/2^r$ vertices of rank r
  - In executing `UNION` the root of the tree with fewer vertices made the child of the root of the larger tree, therefore, by an easy induction it follows that no tree of the resulting forest will have a height h unless it has at least 2^h vertices.
  - Since, a vertex of rank r must have 2^r descendants.
  - And as there are only n vertices, there cannot be more $n/2^r$ vertices of rank r.

UNION and FIND

- We partition ranks into groups as follows.

Ranks	Rank groups
0, 1	0
2	1
3, 4	2
5, . . ., 16	3

- For $n \geq 1$ the largest rank group possible is $G(\lfloor \log n \rfloor) = G(n) - 1$.

UNION and FIND

- Since each `UNION` is executed in O(1) time the sequence σ' which consists of $O(n)$ `UNION` requires time of $O(n)$.
- For `FIND` instruction we develop the following accounting method.
  - **Rule 1** If v is the root or if v and `Parent(v)` are in different rank groups then charge 1 to `FIND`
  - **Rule 2** If v and `Parent(v)` are in the same rank group then charge one unit to v.

UNION and FIND

- Unit kept with v will be used to pay path compression of the vertices which have its parent in the same rank group.
- If a vertex acquires a parent in the higher rank group, then every time there is path compression involving this vertex the charge due to movement up will be billed by rule 1.
- Inside a rank group $g > 0$ the maximum depth of a vertex can be no more than $F(g) - F(g-1)$.
- This means no vertex can be charged more than $F(g) - F(g-1)$ times by rule 2.

UNION and FIND

- Since there are total $G(n)$ rank groups, no vertex can be billed more than $G(n)$ times.
- Now we require to prove the following
  - If at sometime during execution of σ a vertex w becomes descendant of v then the rank of w is less than that of v.
  - There are $n/F(g)$ vertices in the rank group $g > 0$.

UNION and FIND

- Note that if a vertex w becomes a descendant of v during the execution of σ, then it must also be descendant of v during execution of σ'.
- This follows directly from definition of σ' which is derived from σ.
- Then by definition of ranks, the rank of w must be greater than that of v. In other words, the ranks are monotonically increasing along the path going up.

UNION and FIND

- We have already proved that there are $n/2^r$ vertices of rank r.
- So,

$$
\begin{aligned}
N(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r} \\
&\leq \tfrac{n}{2^{F(g-1)+1}}[1 + \tfrac{1}{2} + \tfrac{1}{4} + \ldots] \\
&\leq \frac{n}{2^{F(g-1)}} \\
&\leq \frac{n}{F(g)}
\end{aligned}
$$

UNION and FIND

- Now putting everything together, the charge is
  - Total charge due to rule 1 is: $cnG(n)$, assuming σ consists of cn `FIND` instructions.
  - Charge due to rule 2 for all vertices in a particular rank group $g > 0$ is

$$N(g) \times (F(g) - F(g-1)) \leq n$$

  - Charge due to all vertices in all rank groups is thus $nG(n)$.
  - Overall charge is obtained by summing the above two charges:

$$cnG(n) + nG(n) = c'nG(n)$$

Summary of Amortised Analysis

- The amortized running time of an operation within a sequence of operations is the worst case running time of the entire sequence of operations divided by the number of operations.
- So, it gives the average running time per operation in executing a sequence of selected operations.

Summary of Amortised Analysis

- There are two methods for analysis of amortised time.
  - Accounting method: borrow from Peter to pay Paul
  - Potential method: assumes that the requisite data structure is in an initial energy state, and due to execution of the sequence of operations it gains energy to attain the final state. The sum of the differential energy between the two states and the net effort spent in executing all the operations gives the total amortised cost.
- Multipop and binary counter increment examples are discussed.
- Finally `UNION-FIND` algorithm is analysed for amortised running time of each operation.

