Algorithms Example Sheet 6: Problems Solution Notes

Amortised analysis

Recommended reading:

- Jeff Errickson's notes
- CLRS Chapter 17
- Okasaki's Chapter 5 on amortised analysis

Exercise 6.P.1 [k-bit counter] Consider a k-bit binary counter. This supports a single operation, `inc()`, which adds 1. When it overflows i.e. when the bits are all 1 and `inc()` is called, the bits all reset to 0. The bits are stored in an array, $A[0]$ for the least significant bit, $A[k-1]$ for the most significant.

(a) Give pseudocode for `inc()`.

(b) Explain why the worst-case cost of `inc()` is $\mathcal{O}(k)$.

(c) Starting with the counter at 0, what is the aggregate cost of m calls to `inc()`? [**Hint:** *How many times can each bit get flipped?*]

(d) Let $\Phi(A)$ be the number of 1s in A. Use this potential function to calculate the amortized cost of `inc()`.

(e) In a binomial heap with N items, show that the amortized cost of `push()` is $\mathcal{O}(1)$.

Note: *We have seen this before, but it might be good practice to attempt this again, now that you have seen amortised analysis in more detail.*

[Exercise 2 in Lecturer's handout]

(a)
```
#include <iostream>
#include <vector>

std::vector<int> a({ 0 });

void increment() {
   bool found = false;
   for (int i = 0; i < a.size(); ++i) {
      if (a[i] == 0) {
         a[i] = 1;
         found = true;
         break;
      }
      a[i] = 0;
   }
   if (!found) a.push_back(1);
}

void print() {
   for (auto v : a) {
      std::cout << v;
   }
   std::cout << std::endl;
}

int main() {
   for (int i = 0; i < 16; ++i) {
      print();
      increment();
```

```
    }

    return 0;
}
```

(b) Because of the `break` we have, the loop goes over the prefix of 1's and turns them in 0's. The first 0 it finds, it terminates. Hence, it takes $\mathcal{O}(k)$ time.

(c) The 0-th bit gets flipped m times.
The 1-st bit gets flipped every two rounds, so at most $m/2$.
The 2-nd bit gets flipped every four rounds, so at most $m/4$.
$\vdots$
The k-th bit gets flipped every 2^k rounds, so at most $m/2^k$.

Hence, on aggregate the total number of flips is at most:

$$m + \frac{m}{2} + \frac{m}{4} + \ldots = m \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) = m \cdot \frac{1}{1 - 1/2} = 2m,$$

since it is a geometric series with ratio $1/2$.

Note: A nice animation can be found here.

(d) Each increment requires $k+1$ bit flips, where k is the number of initial 1's. So the potential Φ changes by $1-k$. Hence, the amortised cost is

$$c' = c + \Delta\Phi = k + 1 + 1 - k = 2.$$

(e) There is this isomorphism between binary addition and binomial heaps, by considering whether the binomial heap has a binomial tree of order k (or size 2^k). Each time we are adding one element to a binomial heap, the binomial trees of order k correspond to the bits that change in the binary representation of the counter. Since the amortised cost for increment is $\mathcal{O}(m)$ after m operations, the amortised cost for adding N items to the binomial heap is $\mathcal{O}(N)$.

Exercise 6.P.2 Consider a stack that, in addition to `push()` and `pop()`, supports `flush()` which repeatedly pops items until the stack is empty.

(a) Using the potential function

$$\Phi = \text{ number of items in the stack,}$$

show that the amortized cost of each of these operations is $\mathcal{O}(1)$.

[Exercise 3 in Lecturer's handout]

(b) What is the worst-case time complexity for an operation? Could this matter in practice?

(c) How would you implement the operations so that the worst-case time complexity for each operation is $\mathcal{O}(1)$?

(a) Let k_a be the time required to add an element and k_r the time required to remove an element. We define the scaled potential $\Phi' := k_r \cdot \Phi$, then

- `push`: We are adding one element, so $c'_{\texttt{push}} = k_a + \Delta\Phi' = k_a + k_r$.
- `pop`: We are removing one element, so $c'_{\texttt{pop}} = k_r + \Delta\Phi' = k_r - k_r = 0$.
- `flush:` We are removing all, say k, elements, so $c'_{\texttt{flush}} = c_{\texttt{flush}} + \Delta\Phi' = k \cdot k_r - k \cdot k_r = 0$.

Hence, any sequence of operations with N_a additions and N_r removals takes $N_a \cdot (k_a + k_r) = \mathcal{O}(N_a + N_r)$ time on aggregate.

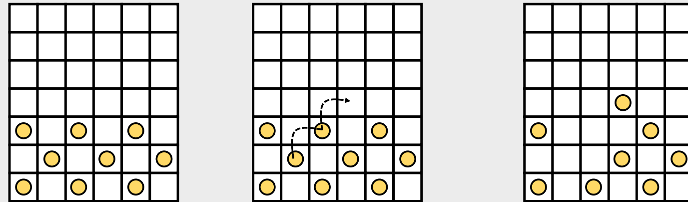
(b) The worst-case time complexity is $\Omega(N)$ where N is the total items to be insered. In real-time applications, this could be a problem as we might care about react time at each step.

(c) We represent the stack using a linked list. Whenever you do a `flush`, you can start a new linked list and not immediately discard the old one. But for each subsequent operation discard two elements from the list to be deleted. The rate of deletion is greater than the rate of insertion, so the "to-delete" list will be emptied at some point, but each step takes $\mathcal{O}(1)$ time.

Note: you can read more about real-time data structures in the handout "Further problems in lists, queues and stacks".

Exercise 6.P.3 [Functional Deque] Attempt **[2018P1Q2 (b)]**.

Exercise 6.P.4 In the game of solitaire chequers, the goal is to move one piece to the top row, via moves of the type shown. A move consists of a diagonal jump by one piece over another; the latter piece is then removed.



Define a potential function

$$\Phi(\text{board state}) = \sum_{\text{pieces } p} \phi(y\text{-coordinate of } p),$$

where you should define ϕ in such a way that any valid move leaves Φ unchanged. Use this potential function to prove that it is impossible to win the game, starting from the board configuration on the left.

[Source: Dr Thomas Sauerwald]

Let y be the initial y coordinate of the piece to be moved, then its final position is $y+1$ and the position of the piece to be removed is y, hence the change in the potential is

$$\Delta\Phi = \phi(y+2) - \phi(y+1) - \phi(y).$$

We are instructed to choose ϕ such that $\Delta\Phi = 0$, hence $\phi(y+2) = \phi(y+1) + \phi(y)$ (which is the reccurence relation for the Fibonacci numbers if we set $\phi(0) = 0$ and $\phi(1) = 1$.
The potential at the starting state is $3 \cdot \phi(1) + 3 \cdot \phi(2) + 3 \cdot \phi(3) = 3 \cdot (1+2) = 9$. The top row is at height 7, but $\phi(7) = 13 > 6$, so we cannot have any piece there.

Dynamic Array

Exercise 6.P.5 [Dynamic array with removals] Consider a stack, implemented using a dynamically-sized array. The rightmost item in the array represents the top of the stack, where we pop from and where we push new items. Suppose that the array's capacity is doubled when it becomes full, and halved when it becomes less than 25% full; and that the cost of these resizing operations is $\mathcal{O}(n)$ where n is the number of items to copy across into the new array. Using the potential method, show that the `push` and `pop` operations both have $\mathcal{O}(1)$ amortized cost. What would go wrong if we halved capacity as soon as the array became less than 50% full, rather than 25% full?

[Exercise 8 in Lecturer's handout]

Consider the potential function

$$\Phi := \begin{cases} 2 \cdot (n - \ell/2) & \text{if } n \geq \ell/2, \\ 2 \cdot (\ell/2 - n) & \text{if } n < \ell/2. \end{cases}$$

For $\ell = 1$, define $\Phi =$. Then for the `append` operation, we have the following cases:

- **Without resize:** When adding one element, the potential changes by at most 2, and the insertion cost is $\mathcal{O}(1)$, so $c' = \mathcal{O}(1)$.
- **With resize:**

$$c'_a = \mathcal{O}(n) + 2 \cdot (n + 1 - n) - 2 \cdot (n - n/2) = \mathcal{O}(n) + 2 - n = \mathcal{O}(1).$$

For the `delete`, we have the following cases:

- **Without resize:** Again, the amortised cost is $\mathcal{O}(1)$.
- **With resize:** The capacity changes from $4n$ to $2n$ and n decreases by 1. So,

$$c'_d = \mathcal{O}(n) + 2 \cdot (2n/2 - n + 1) - 2 \cdot (4n/2 - n) = \mathcal{O}(n) + 2 - 2n = \mathcal{O}(1).$$

The problem with deleting when it is 50% full is that we could be adding and deleting an element and trigerring entire copies, in just one operation. So, the amortised time complexity would be $\Omega(n)$.

Amortised analysis in search trees

Exercise 6.P.6 I have an algorithm that uses a 2-3-4 tree. I've noticed that in a typical run there are several intervals in the key space into which items get inserted, but in which the algorithm doesn't end up searching. I don't want to waste time balancing a tree with these items, but I don't want to discard them because I don't know in advance what the searches will be. I'd like to adapt the 2-3-4 tree so that insertions are lazy, $\mathcal{O}(1)$, and so that the relevant parts of the tree are tidied up on search. Suggest an appropriate design. Using the potential method, explain why your design is no worse than the original 2-3-4 tree, in the big-$\mathcal{O}$ sense.

[Exercise 9 in Lecturer's handout]

Exercise 6.P.7 [Restructuring RBTs] (optional) Attempt Problem 7.2 from this problem sheet.

Disjoint sets

Exercise 6.P.8 [Extensions to Disjoint Sets] Sketch out how you might implement the lazy forest disjoint set, so as to efficiently support:
(a) "Given an item, print out all the other items in the same set".

[Exercise 12 in Lecturer's handout]

(b) "Given an item, print the smallest index in the same set".
Explain carefully how `get_set_with` and `merge` are implemented.

(a) For each vertex, maintain a list to all its direct children. This requires updating `merge` such that when we make u a parent of v, we also append v to u's list. We also keep a pointer of v in its parent list, so that if we update its parent via `get_set_with`, we have an efficient (namely $\mathcal{O}(1)$ way of removing the node). To find all vertices in the same set as v, go to $p =$ `get_set_with`(v) and perform a DFS search through its children.

(b) For each root vertex v of a set, we keep the value of the smallest index in the set. Whenever we `merge` two vertices, we update the parent's min to be the minimum of the two. To find the minimum of the group containing v, we do `get_set_with`(v)`.min`. The implementation of `get_set_with` does not change.

Exercise 6.P.9 Consider an undirected graph with n vertices, where the edges can be coloured blue or white, and which starts with no edges.

(a) The graph can be modified using these operations:

- `insert_white_edge(u,v)` inserts a white edge between vertices u and v
- `colour_edges_of(v)` colours blue all the white edges that touch v
- `colour_edge(u,v)` colours the edge $u \leftrightarrow v$ blue
- `is_blue(u,v)` returns True if and only if the edge $u \leftrightarrow v$ is blue

Give an efficient algorithm that supports these operations, and analyse its amortized cost.

(b) Extend your algorithm to also support the following operation, and analyse its amortized cost:

- `are_blue_connected(u,v)` returns `True` if and only if u and v are connected by a blue path.

(c) Extend your algorithm to also support the following operation, and analyse its amortized cost:

- `remove_blue_from_component(v)` deletes all blue edges between pairs of nodes in the blue-connected component containing v.

*[**Note:** It's easy to gloss over difficulties, so be sure to be explicit about all operations. If you change your data structure to answer a later part, make sure your earlier answers are still complete. This is the sort of question you might be asked in a Google interview; the interviewer will be looking for you to take ideas that you have been taught and to apply them to novel situations.]*

[Source: This question is from Alstrup and Rauhe, via Inge Li Gortz]

(a) **(Solution 1):** For each node u keep the two lists, ℓ_b one containing edges that are blue and ℓ_w containing edges that are white. Then, support the operations as follows:

- `insert_white_edge(u, v)`: Set $t_{(u,v)} = T$ and $c_{(u,v)} = w$.
- `colour_edges_of(v)`: Set $t_v = T$ and $c_v = b$.
- `colour_edge(u, v)`: Set $t_{(u,v)} = T$ and $c_{(u,v)} = b$.
- `is_blue(u, v)`:

(Solution 2): It is possible to support these four operations using an $\mathcal{O}(1)$ worst-case time complexity for each operation. For each edge e, we keep track the last time t_e that we coloured the edge and the colour c_e that we used. Also, for each vertex $v \in V$ we keep track of the last time t_v we updated the colouring of its incident edges and the colour c_v that we used. So, keep a timer T and support the operations, in the following way:

- `insert_white_edge(u, v)`: Set $t_{(u,v)} = T$ and $c_{(u,v)} = w$.
- `colour_edges_of(v)`: Set $t_v = T$ and $c_v = b$.
- `colour_edge(u, v)`: Set $t_{(u,v)} = T$ and $c_{(u,v)} = b$.
- `is_blue(u, v)`: We consider three cases:
  - If $t_{(u,v)} \geq \max(t_u, t_v)$, then answer is $c_{(u,v)}$==b.
  - If $t_u \geq \max(t_u, t_{(u,v)})$, then answer is c_u==b.
  - Same as in **??**, by with v instead of u.

Note: Actually this approach allows us to support colouring edges either blue or white.

Exercise 6.P.10 [Disjoint set problems] (optional) Think how you would solve some of the following problems:

(a) **[CSES Road Construction]**

(b) **[Usaco Closing the farm]**

(c) **[Usaco MooTube]**