15-451/651 Algorithms, Spring 2019
Recitation #2 Worksheet

Recap of this week's lectures:

- Amortized analysis using the banker's method and the potential function method.
- Examples: Binary counter, and growing/shrinking arrays
- Union-find data structures: list-based and tree-based approaches.
- Recap of Minimum spanning tree algorithms: Kruskal, Prim, Boruvka

---

Slower resizing: Suppose we are growing a table as in lecture. Instead of doubling, we do the following: we begin with an array of size 1. The first time we resize, we add 2 to the length. The second time, we add 3. The third time we add 4. And so on. The cost for resizing the array is the size of the new array. What is (in Θ notation) the amortized cost per operation for n pushes?

Binary Counter Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the i^{th} bit costs us 2^i. (Flipping the lowest order bit `A[0]` costs $2^0 = 1$, the next higher order bit `A[1]` costs $2^1 = 2$, the next costs $2^2 = 4$, etc.) What is the amortized cost per operation for a sequence of n increments, starting from zero?

Binary Counter Re-Revisited: Suppose we are incrementing a binary counter, but instead of each bit flip costing 1, suppose flipping the i^{th} bit costs us $i+1$. (Flipping the lowest order bit `A[0]` costs $0+1 = 1$, the next higher order bit `A[1]` costs $1+1 = 2$, the next costs $2+1 = 3$, etc.) What is the amortized cost per operation for a sequence of n increments, starting from zero?

Another Dictionary Data Structure: A "dictionary" data structure supports fast insert and lookup operations into a set of items. Note that a sorted array is good for lookups (binary search takes time only $O(\log n)$) but bad for inserts (takes linear time), and a linked list is good for inserts (takes constant time) but bad for lookups (takes linear time). Here is a simple method that takes $O(\log^2 n)$ search time and $O(\log n)$ amortized cost per insert.

Here, we keep a collection of arrays, where array i has size 2^i. Each array is either empty or full, and each is in sorted order. However, there will be no relationship between the items in different arrays. The issue of which arrays are full and which are empty is based on the binary representation of the number of items we are storing. For example, if we had 11 items (where $11 = 1 + 2 + 8$), then the arrays of size 1, 2, and 8 would be full and the rest empty, and the data structure might look like this:

```
A0:  [5]
A1:  [4,8]
A2:  empty
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

Lookups. How would you do a lookup in $O(\log^2 n)$ worst-case time?

Inserts. How would you do inserts? Suppose you wanted to insert an element, you will have 12 items and $12 = 8 + 4$, you want to have two full arrays in $A2$ and $A3$ and the rest empty. Suggest a way that, if you insert an element 11 into the example above, gives:

```
A0:  empty
A1:  empty
A2:  [4, 5, 8, 11]
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

(Hint: merge arrays!)

Cost of Inserts: Suppose the cost of creating an array of length 1 costs 1, and merging two arrays of length m costs $2m$. So, the above insert had cost $1 + 2 + 4$. Inserting another element would cost 1, and the next insert would cost $1 + 2$.

What is the amortized cost of n inserts?