CS561 HW8

Ryan Scherbarth

October 2024

1. In a leaf-balanced binary tree, any node with 2 children has the same number of leaves in the sub-trees rooted at both children. A full binary tree is leaf-balanced. Let n be the number of nodes in the tree. Prove by induction on n that the number of leaf nodes in a leaf-balanced binary tree is always a power of 2.

   Base case: $n = 3$
   Let $f(n)$ be the number of leaf nodes in a leaf-balanced binary tree with n nodes. Then the number of leaf nodes $f(3) = 2^1 = 2$.

   Inductive Hypothesis:
   For any j s.t. $0 \leq j < n$, $f(j)$ will be some power of two.

   Inductive Step:
   Let ℓ_1 be the leaf-balanced binary tree with $n = k + 1$ nodes. By the definition of the tree, we know that every node has the same number of children.

   Given two sub-trees ℓ_1 and ℓ_2, we know that the tree is balanced, meaning that we can evenly split it such that $\ell_1 = \ell_2 = \frac{n}{2}$. Our inductive hypothesis says that any tree has a number of leaves that is a power of two, and and any new tree will be $\ell_1 + \ell_2 = 2\ell_1$, so we will have $2 * 2^n$ nodes, which will remain a power of 2.

   Therefore via proof by induction, any balanced tree with number of nodes $n \geq 3$ will be exactly a power of 2.

2. One a dark and quiet night, you find yourself climbing a creaky staircase with n stairs. Every stair, $i \in [1, n]$ has a creakiness value (or cost) c_i. In each step, you can go up either 2 or 3 stairs, and your goal is to get to the top of the staircase in a way that minimizes the sum of costs for all the stairs you visit. The bottom and top of the staircase are not creaky and so have no cost. One the last stair, either step size will take you to the top.
   Use $n = 6$ and costs $[10, 1, 5, 20, 20, 1]$.

   (a) Consider a greedy algorithm for this problem which always chooses the stair of least cost for the next move. Give an example where this greedy algorithm is not optimal.

   The greedy algorithm chooses the least cost solution at each step. Here, we would just to jump to position $i = 2$, $i = 3$, done for a total cost of 21. This is not the optimal solution.

(b) Now write a recurrence relation to solve this problem. Don't forget to first define the function in words whose solutions enable solving the big problem.

Base cases: $n = 2$, $n = 3$
When n is either 2 or 3, we're able to make the entire jump over the rest of the stairs, so the optimal case will always be to take that option and not add any cost.

Inductive Step:
For $n > 3$,

$$\begin{aligned} f(n) = min(&c_i + f(n-2) \\ &c_i + f(n-3) \\ &0 \text{ Otherwise}) \end{aligned}$$

(c) Describe a dynamic program to solve the problem using your recurrence. What are the dimensions of your table? How do you fill it in? What is the final value returned? What is the runtime of your algorithm?

We will create a 1 dimensional table DP of size $n + 1$, where each index of DP corresponds to the minimum cost to reach the top stair from that index i.

Base Case:
$DP[i] = 0$ since reaching the top stair has no cost. We fill out the table DP by working backwards through the values on the table.

Inductive Step
For each value after the base case we evaluate $DP[i] = c_i + min(c_i + f(i + 2), c_i + f(i + 3))$ and store it in the table. The final value returned is teh minimum cost to reach the top stair starting from the bottom $DP[1]$. The runtime will then be $O(n)$ because of the time it takes to fill out the table initially.

3. The ancient game of NIM is played by two players who alternate taking any positive number of stones from one of three piles. The person taking the last stone loses. An example game starting with piles of size 13, 9, and 1 is below. Here, the first move of player 1 is to take the single stone in pile 1, and player 1 eventually wins.

Player Turn	1	2	1	2	1	2
Stones left	(13,9,1)	(13,9,0)	(13,2,0)	(2,2,0)	(1,2,0)	(1,0,0)

In this problem, you will write a dynamic program to determine if Player 1 can force a win for a given input specifying the sizes of the 3 piles.

(a) Describe in words a function whose solutions for smaller problems will help you solve the big problem

We will define a function $w(x, y, z)$ where given the number of rocks in each of the three piles, it returns 1 if it is possible to force a win, and 0 otherwise.

(b) Write a recurrence relation for the dynamic program using the function you described above.

Base Case: $w(0,0,0) = 1$
When all of the piles are 0, the current player must win, since if there are no rocks left to take, the other player must have been the last one to take a rock.

Recurrence Step:

$$\begin{aligned} w(x,y,z) &= 1 && \text{if} \in (x',y',z') \text{ s.t. } x' \leq x,\, y' \leq y,\, z' \leq z \text{ and } w(x',y',z') = 0 \\ &= 0 && \text{otherwise} \end{aligned}$$

We return 1 if it is possible to win on the current step and some combination of smaller steps leading up to this point, and 0 if that is not the case.

(c) Describe a dynamic program to solve the problem for any initial input consisting of x_1, x_2, x_3 stones in the piles. What are the dimensions of your table? How do you fill it in? What is the final value returned? What is the runtime of your algorithm if $x_1 = x_2 = x_3 = n$?

We will need to create a 3D table DP to store all the possible combinations of x_1, x_2, x_3. Each position on our table will correspond to some combination of x_1, x_2, x_3, and will store either a 0 or a 1 depending on if it's possible for the current player to win given the pile sizes at that index.

For each position in the table (x_1, x_2, x_3) we will evaluate our function from part (b) and store the result. The final value returned is $DP[x_i][y_i][j_i]$ which tells us that if it is player 1's turn, given piles of size x_i, y_i, z_i, 1 indicates it is possible to get a win, and 0 indicates it is not possible to force a win.

4. Walt is making a device for his friend Hector that counts how many times Hector rings a bell. the software for the device requires a binary counter data structure with INCREMENT and RESET operators.
In class we discussed an INCREMENT algorithm for incrementing a binary counter in $O(1)$ amortized time. Now we want to include a RESET algorithm that sets all the bits in the counter to 0. Below are the algorithms for INCREMENT and RESET. They use an array B of bits and an integer m giving the largest index in b set to 1.

Algorithm 1: INCREMENT(B,m)

```
i = 0
while B[i] = 1 do
  B[i] = 0
  i = i + 1
end while
B[i] = 1
if i > m then
  m = i
end if
```

Algorithm 2: RESET(B,m)

```
for i = 0 to m do
  B[i] = 0
end for
```

Let n be the number of operations on this binary counter. give the following costs as a function of n.

(a) What is the worst-case run time of INCREMENT?

INCREMENT will have a worst case runtime in the case where all digits of m are set to 1, since we will need to add a new 1 and set all the other m digits back to 0. m is the number of binary digits of our number, which means m can be at most $\log(n)$ when using a base 2 number system, so we'll end with with a worst case runtime of $O(\log(n))$

(b) What is the worst-case run time of RESET?

RESET will have a constant runtime since it will always need to check every single bit between 0 and m, therefore it will also have a runtime bounded by $O(m)$ which is equal to $O(\log(n))$

(c) Prove that in an arbitrary sequence of calls to INCREMENT and RESET, each call has amortized cost $O(1)$. Hint: Use the accounting method an save up dollars during INCREMENT for future calls to RESET.

We will assign a cost of \$2 for each INCREMENT operation flipped, \$1 for each time we flip the bit to 1, and \$1 for the time we need to loop back and replace them.

For each bit we add, we charge \$2 to set the bit, after n operations we have \$2n saved after any set of n operations. At any time We decide to do the reset, we will go back and loop through resetting every bit, which will end up costing us \$1n. After subtracting this we see that we end with an amortized cost of $O(1)$ since the cost for any n operations is \$$n$.

5. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $\frac{3}{4}$ full, we allocate a new table twice as big as our current table, insert everything into a new table, and then free the old table.
- After a deletion, if the table is less than $\frac{1}{4}$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. Hint: Do not use potential functions.

We know that each insertion or deletion will only cost $O(1)$ time when we don't need to worry about resizing, so we just need to look into the effect resizing has on our overall cost.

The cost of resizing will end up taking $O(n)$ time for either increasing or decreasing the size since we need to copy all of our elements over to a new location each time. We can use the taxation method to illustrate the amortized cost;

Lets add a tax of \$3 for each insertion or deletion. For an insertion, we will say that \$1 pays for the insertion, and then we reserve \$2 for the resizing that can happen, both for increasing and decreasing. We will use this same pricing scheme for both insertions and deletions.

After doubling the table, we require at least $\frac{n}{2}$ insertions to reach the next cutoff, so we have $(2)(\frac{n}{2}) = n$ savings in this case. Likewise after splitting the table by half we need at least $\frac{n}{2}$ deleted spaces.

Therefore, each insertion and deletion ends up having an amortized cost of $O(n)$ dollars using the \$3 taxation, which lets us conclude an amortized runtime of $O(1)$.