CS561, HW4

Ryan Scherbarth, University of New Mexico

September 2024

1. In this problem, you will prove that for a recurrence of the form $T(n) = aT(\frac{n}{b}) + f(n)$, if $af(\frac{n}{b}) \geq Kf(n)$ for some constant $K > 1$, then $T(n) = n^{log_b(a)}$.

   - (a) First, assume that for any $x \geq b$, $af(\frac{x}{b}) \geq Kf(x)$ for some fixed $K > 1$. Let the last level of the recursion tree be $\ell = \log_b(n)$. Prove by induction that for any $i \exists [0, \ell]$, $a^i f(\frac{n}{b^i}) \geq K^i f(n)$. Don't forget to say which variable you're doing induction on, and label the BC, IH, and IS.

   Base case: $i = 0$;

$$a^0 f(\frac{n}{b^0}) = f(n)$$
$$K^0 f(n) = f(n)$$

   So the base case trivially holds.

   Inductive Hypothesis:
   Assume that for all $i \leq k$, the inequality $a^i f(\frac{n}{b^i}) \geq K^i f(n)$ for all $i \leq k$.

   Inductive Step:
   Let $j > 0$, show that $a^k f(\frac{n}{b^k}) \geq K^k f(n)$ for all $i < j \leq \ell$

$$a^{j-1} f(\frac{n}{b^{j-1}}) \geq K^{j-1} f(n)$$
$$a(a^{j-1} f(\frac{n}{b^{j-1}})) \geq a(K^{j-1} f(n))$$
$$a^j f(\frac{n}{b^{j-1} * b}) \geq a(K^{j-1} f(n))$$
$$a^j f(\frac{n}{b^{j-1} * b}) \geq a(K^{j-1} f(n))$$
$$a^j f(\frac{n}{b^j}) \geq K^j f(n)$$

- (b) Using the above result, upper bound the sum of all levels of the recursion tree, i.e. upper bound $\sum_{i=0}^{\ell} a^i f(\frac{x}{b^i})$. Hint: Let $L = \Theta(a^\ell)$ be the cost of the last level and argue that the cost of the j-th level of recursion tree above ℓ is at most $\frac{L}{K^j}$. Now use what we showed in class about geometric summations. Finally, use log facts to simplify L.

2. Consider the recurrence $f(n) = 4f(\frac{n}{3}) + \sqrt{n}$

   - (a) Use the Master method to solve this recurrence

$$\begin{aligned} a &= 4 \\ b &= \frac{1}{3} \\ f(n) &= \sqrt{n} \\ \log_b(a) &= \frac{\log(4)}{\log(3)} \\ &= \log(4) - \log(3) \\ &= 2\log(2) - \log(3) \end{aligned}$$

   Since $\sqrt{n}$ grows asymptotically slower than $\log_3 4$, we fall into case 1 of the master theorem. In this case:

$$\begin{aligned} T(n) &= \Theta(n^{\log_b(a)}) \\ &= \Theta(n^{\text{some } j \geq \frac{1}{2}}) \end{aligned}$$

   Therefore, the solution to the recurrence $f(n) = 4f(\frac{n}{3}) + \sqrt{n}$ is bbd. $f(n) = \Theta(n^{\log_3(4)})$.

- (b) Now use annihilators (and a transformation) to solve the recurrence.

Using a transformation: $n = 3^k$, we can then say $n = \log_3(n)$.

$$\begin{aligned} g(k) &= f(3^k) \\ &= 4f(\frac{3^k}{3}) + \sqrt{3^k} \\ &= 4f(3^{k-1}) + 3^{\frac{k}{2}} \\ &= 4g(k-1) + 3^{\frac{k}{2}} \end{aligned}$$

We can now break it into two pieces to find the individual annihilators. Starting with $g(k) = 4g(k-1)$. This is a standard recurrence that can be annihilated with a $L - 4$.

Similarly, our remaining term makes $g(k) = 3^{\frac{k}{2}}$. This term is then annihilated by $L - 3^{\frac{1}{2}}$.

Our total annihilator for $g(k)$ is then given by $(L-4)(L-3^{\frac{1}{2}})$. We can then convert back to a general solution by plugging in $k = \log_3(n)$,

$$\begin{aligned} f(n) &= n^{\log_3(4)} + n^{\frac{1}{2}} \\ &= n^{\log_3(4)} \end{aligned}$$

So we come to the same conclusion; $f(n) = \Theta(n^{\log_3(4)})$

3. Consider the function:

```
int f (int n){
  if (n == 0) return 2;
  else if (n == 1) return 5;
  else{
    int val = 2*f(n-1);
    val = val - f(n-2);
    return val;
  }
}
```

- (a) Write a recurrence relation for the value returned by f. Solve the recurrence exactly. Don't forget to check it.

$$
\begin{aligned}
f(0) &= 2 \\
f(1) &= 5 \\
f(n) &= 2f(n-1) - f(n-2)
\end{aligned}
$$

To solve, we can choose our guess function to be r^n;

$$
\begin{aligned}
r^n &= 2r^{n-1} - r^{n-2} \\
&= r^2 - 2r + 1 \\
&= (r-1)^2
\end{aligned}
$$

So we have a root of $r = 1$. meaning

$$
\begin{aligned}
f(n) &= c_1 * 1^n + c^2 * n * 1^n \\
&= c_1 + c_2 n
\end{aligned}
$$

Now we calculate the constants;

$$
\begin{aligned}
f(0) &= c_1 + c_2 * 0 = 5 \\
c_1 &= 2 \\
f(1) &= c_1 + c_2 * 1 \\
&= 2 + c_2 \\
c_2 &= 5
\end{aligned}
$$

So our general solution is $f(n) = 2 + 3n$. We can verify by plugging in to our two base cases;

$$
\begin{aligned}
f(0) &= 2 + 3 = 0 \\
f(1) &= 2 + 3 * 1 = 5
\end{aligned}
$$

Therefore, $f(n)$ holds for all $n \geq 0$.

- (b) Write a recurrence relation for the running time of f. Get a tight upper bound (i.e. big-O) on the solution to this recurrence.

The recurrence $T(n) = T(n-1) + T(n-2) + O(1)$ since the two base cases take a constant amount of time.

Using the Master theorem, we can characterize $T(n)$ as $r^2 - r - 1 = 0$. Solving here we come to $r = \frac{1 \pm \sqrt{5}}{2}$ using the quadratic formula.

The dominant root $\phi = \frac{1 \pm \sqrt{5}}{2}$ will be our upper bound on running time. We can now conclude our running time will be bounded by $T(n) = O(\phi^n)$. Since $1 < \phi < 2$ we will round up to 2 and say $T(n) = O(2^n)$.

4. A bakery sells donuts in boxes of three different quantities, x_1, x_2, and x_3. In the Donuts Buying problem, you are given the numbers x_1, x_2, and x_3, and an integer n and you should return either 1) the minimum number of boxes needed to obtain exactly n donuts if this is possible, along with a set of boxes that obtains this minimum; or 2) "DON!" if it is not possible to obtain exactly n donuts.
For example if $x_1 = 4$, $x_2 = 6$, $x_3 = 9$ and $n = 17$, then you should return that 3 boxes suffices, with 2 boxes of size 4 and 1 box of size 9. However, if $n = 11$ you should return "DOH!" since it is not possible to buy exactly 11 donuts with these boxes.

   - (a) For any positive x, let $m(x)$ be the minimum number of boxes needed to buy x donuts if this is possible, or INFINITY otherwise. Write a recurrence relation fr the value of $m(x)$. Don't forget the base case(s)!

   Base cases:
   Case 1: $T(0) = 0$.
   Case 2: for any $x < 0$, $T(x) = INFINITY$.

   The number of boxes we will use is given by the recurrence $T(n) = 1+T(n-x_1)+T(n-x_2)+T(n-x_3)$.

   - (b) Give an efficient algorithm for solving the problem. How does its running time depend on x_1, x_2, x_3 and n? Is it an algorithm that runs in polynomial time in the input sizes?

   We can create a greedy algorithm that will work backwards taking the largest possible boxes, and moving to smaller box sizes as we try to find a valid combination.

   Assume that x_1, x_2, and x_3 are in sorted order. We start with dividing n by x_3. We then take the remainder and divide that by x_2, then x_1.

   If we get down to x_1 and our remainder is not evenly divisible by x_1, we will subtract one from the larger box, in this case x_2, and then try dividing by x_1 again. We'll continue this until we bubble all the way back up.

   In the best case, our running time is bounded by n. This is when the number of donuts is evenly divisible by the largest box size, so we never need to consider x_1 or x_3.

   Furthermore, our worst case is when when there is no possible solution. In this case, we end up testing every possible combination of dividing the donuts between the three sizes before we can conclude it is not possible.

   This algorithm does run in polynomial time, because we have a constant number of box sizes which is what is ultimately bounding our running time.

5. A thief repeatedly robs the same bank. To avoid capture, he decides to never rob the bank fewer than 10 days after the last robbery. He has obtained information, for the next n days, on the amount of money b_i that is held at the bank on day i.

   - (a) Let $r(i)$ be the maximum amount of revenue the thief can safely obtain from day 1 through day i. Give a recurrence relation for $r(i)$.

   Case 1: thief robs the bank on day i.
   $r(0) = 0$

   Case 2: thief does not rob the bank on day i.
   $r(1) = b_1$

   Given the cool down of 10 days, we can make the function for the maximum that can be robbed from a given day as $r(i) = b_i + r(i - 10)$. On a day where the bank is not robbed, the maximum amount of money that can be stolen up to day i is $r(i - 1)$.

   So our final equation will be given by $r(i) = r(i - 1) + b_i + r(i - 10)$. Here, $r(i - 1)$ gives the maximum up to day $i - 1$ if the bank is not robbed on day i. $b_i + r(i - 10)$ gives the maximum if a thief does rob the bank on day i.

   - (b) Describe a dynamic program based on this recurrence. What is the runtime of your algorithm?

   Similar to our dynamic implementation of solving the Fibonacci sequence, we will create this dynamic program by creating an array r of size $n + 1$. This array will store the maximum revenue for each day starting from say 0 to n.

   We will just simply plug in the base case $r[0] = 0$, and calculate indexes [1..9] using our formula. Then, for each day without breaking our 10 day rule, the maximum which can be gained will be given by $r[i]$.

   This gives us the opportunity to run in $O(n)$ time, but notably also has a space complexity of $O(n)$ in memory.