CS 161 Winter 2020 Section 2

January 16-17, 2020

Recurrence Relations

Master Theorem

Recall the Master theorem from lecture:

Theorem 0.1 *Given a recurrence* $T(n) = aT(\frac{n}{b}) + O(n^d)$ *with* $a \geq 1$, *and* $b > 1$ *and* $T(1) = \Theta(1)$, *then*

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

What is the Big-Oh runtime for algorithms with the following recurrence relations?

1. $T(n) = 3T(\frac{n}{2}) + \Theta(n^2)$
2. $T(n) = 4T(\frac{n}{2}) + \Theta(n)$
3. $T(n) = 2T(\sqrt{n}) + O(\log n)$

1. Using the Master Theorem, $a = 3, b = 2,$ and $d = 2$. Since $a = 3 < b^d = 4$, we fall into the second case. So, the runtime is $O(n^d) = O(n^2)$.
2. Using the Master Theorem, $a = 4, b = 2,$ and $d = 1$. Since $a = 4 > b^d = 2$, we fall into the third case. So the runtime is $O(n^{\log_b a}) = O(n^{log_2 4}) = O(n^2)$.
3. In order to solve this question, we must use a substitution trick. Define $k = \log n$, so $n = 2^k$, and $\sqrt{n} = 2^{\frac{k}{2}}$. So the recurrence relation is:

   $$T(2^k) = 2T(2^{\frac{k}{2}}) + O(k)$$

   Next, let $S(k) = T(2^k)$ so $S(\frac{k}{2}) = T(2^{\frac{k}{2}})$:

   $$S(k) = 2S(\frac{k}{2}) + O(k)$$

   Using master theorem, we get

   $$S(k) = O(k^d \log k) = O(k \log k) = O(\log n \log(\log n))$$

Substitution Method

Use the Substitution Method to find the Big-Oh runtime for algorithms with the following recurrence relation:

$$T(n) = T\left(\frac{n}{3}\right) + n$$

$$T(1) = 1$$

You may assume n is a multiple of 3, and use the fact that $\sum_{i=0}^{log_3(n)} 3^i = \frac{3n-1}{2}$, from the finite geometric series sum. Please prove your result via induction.

First, we unravel the recurrence relation...

$$\begin{aligned}
T(n) &= T(n/3) + n \\
&= T(n/9) + n/3 + n \\
&= T(n/27) + n/9 + n/3 + n \\
&\vdots \\
&= 1 + 3 + 9 + ... + n/9 + n/3 + n \\
&= \sum_{i=0}^{log_3(n)} 3^i \\
&= \frac{3n-1}{2} \\
&= O(n).
\end{aligned}$$

Note: We used the finite geometric series formula $S_n = \frac{a_1(1-r^n)}{1-r}$.
For the curious: https://www.varsitytutors.com/hotmath/hotmath_help/topics/geometric-series.

Now for the proof! Using the definition of Big-Oh, we choose $c = 2$ (since its larger than 3/2) and $n_0 = 1$, and prove $T(n) \leq 2n$ for all $n \geq 1$.

- **Inductive Hypothesis.** $T(n) \leq 2n$.
- **Base Case.** We let $n = 1$ and notice $T(n) = 1 \leq 2 * 1$. Thus the inductive hypothesis holds for $n = 1$.
- **Inductive Step.** Let $k > 1$, and suppose that the inductive hypothesis holds for all $n < k$, namely T(k/3) $\leq 2k/3$. We will prove the inductive hypothesis holds for k. We have

$$\begin{aligned}
T(k) &= T(k/3) + k \\
&\leq 2k/3 + k \\
&= 5k/3 \\
&\leq 2k.
\end{aligned}$$

  This establishes the inductive hypothesis for n=k.
- **Conclusion.** By strong induction, we have established the inductive hypothesis for all $n > 0$; thus, $T(n) \leq 2n$ for all $n \geq 1$, and (choosing $c = 2$ and $n_0 = 1$ in the definition of Big-Oh) we have established that $T(n) = O(n)$, as desired.

Divide and Conquer

Penguins in a Line

You arrive on an island of n penguins. All n penguins are standing in a line, and each penguin has a distinct height (i.e. no 2 penguins have the same height). A *local minimum* is a penguin that is shorter than both its neighbors (or one neighbor for the first and last penguin).

Design an efficient algorithm that takes as input an array of penguin heights, and finds a local minimum. Please give a clear English description, pseudocode, explanation of runtime, and a formal proof of correctness.

English description

We use a variant of **binary search**. Check whether the middle penguin is a local minimum. Otherwise at least one of its neighbors must be shorter; recurse on a side with a shorter neighbor. Base case: if we have only one penguin, it's a local minimum!

Pseudocode (Note: this pseudocode isn't exact about ceilings/floors)

```
def LocalMin(Penguins):
    n = len(Penguins)
    if n == 1                                // base case
        return Penguins[0]

    if Penguins[n/2] is a local minimum:     // compare neighbors
        return Penguins[n/2]
    else if Penguins[n/2 - 1] < Penguins[n/2]:
        return LocalMin(Penguins[:n/2]       // excludes mid
    else:
        return LocalMin(Penguins[n/2 + 1:])  // excludes mid
```

Running Time

- **Running time at level t:** $O(1)$ (only one comparison).
- **Number of sub-problems at level t:** 1 (notice that we only recurse on one half of the array!).
- **Depth of recursion (number of levels):** $O(\log(n))$ (since number of penguins halves in each iteration).
- **Recurrence relationship for running time:** $T(n) = T(n/2) + O(1)$
- **Total running time:** $\boxed{\text{T(n) = O(log(}n\text{))}}$ (Either by using Master method or by multiplying # of levels times running time at each level.)

Proof of Correctness:

We prove by induction on the number of penguins.

- **Inductive hypothesis:** Given a line of n penguins, the algorithm returns a local minimum.
- **Base case:** We show the inductive hypothesis holds when $n = 1$. When $n = 1$, the only penguin is trivially a local minimum since it has no neighbors.

- **Inductive step:** Assume the inductive hypothesis holds for n where $1 \leq n < k$, and k is an integer greater than 1. In particular, this means that the inductive hypothesis holds for $\frac{k}{2}$, i.e. the algorithm returns a local minimum when given a line of $\frac{k}{2}$ penguins.

  Now, we need to show that the inductive hypothesis holds for $n = k$. If the middle element is indeed less than its neighbors, we would return it right away. Otherwise, we choose the side with a shorter neighbor and return the element p that our algorithm finds as the local minimum of that half-size sub-array, which we know to be a correct local minimum by our inductive hypothesis. We just need to check that p is still a true local minimum in the context of the entire length k array.

  If p has all of its neighbors (of which there may be one or two) in the half-size sub-array we recursed on(i.e. p is not adjacent to the midpoint of the size-k array), then p is clearly also a local minimum in the context of the larger array. This is because the validity of p as a local minimum is not affected if all of p's neighbors lived in the half-array that we had recursed on.

  Suppose instead that p is at the end of its sub-array and is adjacent to the midpoint of the size-k array (i.e. Penguins[n/2] is p's neighbor). We know by our inductive hypothesis that p is less than any neighbor within its own sub-array. We also know that we originally chose to recurse on the half-array that p lives in because p was less than Penguins[n/2], its neighbor. Therefore p is also a local minimum in the context of the original size-k array since it is less than all its neighbors. Thus, we have shown that the inductive hypothesis holds for $n = k$.
- **Conclusion:** By strong induction, we have shown that our inductive hypothesis holds for all natural numbers n, i.e. our algorithm returns a local minimum for a line of any number of penguins.