Recurrence Relations

Master Theorem

Recall the Master Theorem from lecture:
Theorem (Master Theorem). *Given a recurrence* $T(n) = aT(\frac{n}{b}) + O(n^d)$ *with* $a \geq 1$, $b > 1$ *and* $T(1) = \Theta(1)$, *then*

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

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. $T(n) = 2T(\sqrt{n}) + O(\log n)$

Solution.

1. We can apply Master's Theorem with $a = 3$, $b = 2$, and $d = 2$. Since $a = 3 < b^4 = 4$, we fall into the second case. Therefore, the runtime is $O(n^d) = O(n^2)$. □

2. We can apply Master's Theorem with $a = 4$, $b = 2$, and $d = 1$. Since $a = 4 > b^4 = 2$, we fall into the third case. Therefore, the runtime is $O(n^{\log_d a}) = O(n^{\log_2 4}) = O(n^2)$. □

3. To solve this question, we must first use a substitution trick. Let $k = \log n$, so that $n = 2^k$ and $\sqrt{n} = 2^{\frac{k}{2}}$. The recurrence relation is now

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

Now, let $S(k) = T(2^k)$, so that $S(\frac{k}{2}) = T(2^{\frac{k}{2}})$. We get the following recurrence relation:

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

Finally, we can apply Master's Theorem with $a = 2$, $b = 2$, $d = 1$. Since $a = 2 = b^d$, we fall into the first case. Therefore, the runtime is

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

□

Problem Solving Notes:

1. *Read and Interpret:* The first two recurrence relations are of the form $T(n) = aT(n/b) + O(n^d)$. Perfect for Master's Theorem! The last one looks a bit different. Is there a way for you to rewrite it so that it is of the same form as needed for Master's Theorem?

2. *Information Needed:* What information do you need in order apply Master's Theorem? It's helpful to explicitly write down what the values of a, b and d are.
3. *Solution Plan:* Once you've written the problem in the form $T(n) = aT(n/b) + O(n^d)$, you're good to directly apply Master's Theorem!.

Substitution Method

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

$$T(n) = T\left(\frac{n}{3}\right) + n; \quad T(1) = 1$$

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 sum. Please prove your result via induction.

Solution. First, we unravel the recurrence relation:

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

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/hotmathhelp/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 $T(n) \le 2n$ for all $n \ge 1$.

- **Inductive hypothesis:** $T(n) \le 2n$.
- **Base case:** We let $n = 1$ and notice $T(n) = 1 \le 2 \cdot 1$. 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 $T(k/3) \le 2k/3$. We will prove the inductive hypothesis holds for k. We have

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

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. □

Problem Solving Notes:

1. *Read and Interpret:* Here the problem tells you what method you should use: substitution method. However, could you also have use Master's Theorem here? Answer: yes, give it a try!
2. *Information Needed:* Before you can prove by induction what the Big-Oh runtime is, you need to come up with the Big-Oh runtime you want to prove. To do this, you need to apply the substitution method similarly to how you come up with an *explicit* formula for a sequence ($a_n = f(n)$) from a recurrence relation ($a_{n+1} = g(a_n)$). Specifically, repetitively substitute in the recurrence relation into itself until you find a trend, and can generalize an explicit formula for $T(n)$.
3. *Solution Plan:* First, come up with a "guess" for the runtime by using the substitution method, then formally prove it by induction.

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.

Solution.

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
```

Runtime.

- 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 the number of penguins halves in each iteration).
- Recurrence relationship for running time: $T(n) = T(n/2) + O(1)$

Therefore, the total running time is $T(n) = O(\log(n))$. You can find this either by using Master method or by calculating the number 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.

Problem Solving Notes:

1. *Read and Interpret:* Think about the definition given to you: *local minimum.* Specifically, if a certain penguin is *not* a local minimum, what does that mean? That at least one of the neighbors is shorter than it. If there is only one penguin, what does it mean to be a local minimum?
2. *Work out examples:* Try to visualize the smaller cases - start with 3-5 penguins with arbitrary heights (but all distinct). How would you find a local minimum?
3. *Information Needed:* What do you need in order to break this down into a smaller problem? How do you define a ”smaller problem”?

Maximum Sum Subarray

Given an array of integers $A[1..n]$, find a contiguous subarray $A[i, ..j]$ with the maximum possible sum. The entries of the array might be positive or negative.

1. What is the complexity of a brute force solution?
2. The maximum sum subarray may lie entirely in the first half of the array or entirely in the second half. What is the third and only other possible case?
3. Using the above, apply divide and conquer to arrive at a more efficient algorithm.
   (a) Prove that your algorithm works.
   (b) What is the complexity of your solution?
4. Advanced (Take Home) - Can you do even better using other non-recursive methods? ($O(n)$ is possible)

Solution.

1. The brute force approach involves summing up all possible $O(n^2)$ subarrays and finding the max amongst them for a total run time of $O(n^3)$. We can optimize this by pre-computing the running sums for the array so that we can find the sum of each subarray in $O(1)$ giving us a total run time of $O(n^2)$. $\square$
2. The maximum sum subarray can also overlap both halves; in other words it passes through the middle element. $\square$
3. We divide the array into two and recurse to find the maximum sub array in the two segments. The best subarray of the third type consists of the best subarray that ends at $n/2$ and the best subarray that starts at $n/2$. We can compute these in O(n) time. To arrive at the final answer we return the max amongst these three types. This gives us a recurrence relation of the form $T(n) = 2T(n/2) + O(n)$ which solves to $T(n) = O(n \log n)$.

```
def MaxSubArray(A):
    if len(A) == 1                          // Base case
        return A, A[0]
    mid = floor(len(A)/2)
    A1, V1 = MaxSubArray(A[:mid])           // Case 1: max in left half
    A2, V2 = MaxSubArray(A[mid+1:])         // Case 2: max in right half
    maxL, sumL = 0                          // Now we'll compute the third case:
    l = mid+1
    for i in range(0, mid):
        sumL += A[mid-i]
        if sumL > maxL:
            maxL = sumL
            l = mid - i
    maxR, sumR = 0
    r = mid
    for i in range(mid+1, len(A)-1):
        sumR += A[i]
        if sumR > maxR:
            maxR = sumR
            r = i
    V3 = maxL+ maxR                         // Case 3: max goes through middle
    if V1 > V2,V3:
        return A1, V1
    else if V2 > V3:
        return A2, V2
    else:
        return A[l:r], V3
```

4. Here is a way to do this in linear time:

```
def MaxSubArray(A):
    maximumValue = 0
    currentValue = 0
    for i in range(len(A)):
        currentValue = max(0, currentValue + A[i])
        maximumValue = max(maximumValue, currentValue)
    return maximumValue
```

Problem Solving Notes: (excluding part 4)

1. *Read and Interpret:* This problem is already broken down for you! :)
2. *Information Needed:* The main thing to notice is in question 2: how can I break down my problem into *strategical* subproblems?
3. *Solution Plan:* Though this problem alreaady gave you a solution plan, this is actually a general good strategy for most divide and conquer problems: 1) To help yourself get familiarized with the problem,

briefly think of a brute force solution. 2) Now, is there a way for me to simplify the solution by instead working in subproblems? If so, what would those subproblems be? 3) Come up with an algorithm that leverages these subproblems. 4) Formally prove that it works and calculate the complexity of your solution. How much better than the brute force solution is it?

Light Bulbs and Sockets

You are given a collection of n differently sized light bulbs that have to be fit into n sockets in a dark room. You are guaranteed that there is exactly one appropriately-sized socket for each light bulb and vice versa; however, there is no way to compare two bulbs together or two sockets together as you are in the dark and can barely see! (You are, however, able to see where the sockets and light bulbs are.) You can try and fit a light bulb into a chosen socket, from which you can determine whether the light bulb's base is too large, too small, or is an exact fit for the socket. If the bulb fits exactly, it will flash once, in which case you have a correct match. (Note that the flashing light does not allow you to visually compare bulbs/sockets to other bulbs/sockets.) Suggest a (possibly randomized) algorithm to match each light bulb to its matching socket. Your algorithm should run strictly faster than quadratic time in expectation. Give an upper bound on the worst-case runtime, then prove your algorithm's correctness and expected runtime.

Solution. Consider the following procedure for matching lightbulbs with their corresponding sockets. If the cardinalities of L and S are equal to 1, then we know that $\ell \in L$ matches $s \in S$, so we can match and return. Otherwise, we run the following recursive procedure, $\texttt{Match}(L, S)$:

- Select a lightbulb $\ell \in L$ uniformly at random.
- For every socket $s \in S$: test whether ℓ is too small, too big, or just right (call the matching socket $s^\star$).
- For every other lightbulb ℓ': test whether ℓ' is too big or too small to fit into $s^\star$.
- S_{big}, $S_{\text{small}} \leftarrow$ sockets too big and too small for ℓ, respectively.
- L_{big}, $L_{\text{small}} \leftarrow$ lightbulbs too big and too small for $s^\star$ respectively.
- $\texttt{Match}(L_{\texttt{big}}, S_{\texttt{big}})$
- $\texttt{Match}(L_{\texttt{small}}, S_{\texttt{small}})$

Correctness. Consider a given call to `Match`. For the lightbulb we pick at random, ℓ, we go through all the sockets in S, so we are guaranteed to find its unique matching socket, $s^\star$. Note that if a bulb is too big to fit in $s^\star$, then it must fit in a socket that was too big for ℓ; likewise, if a bulb is too small to fit in $s^\star$, then it must fit in a socket that was too small for ℓ. Thus, we can partition the bulbs and sockets simultaneously, such that we only have to compare "small" bulbs to "small" sockets and "big" bulbs to "big" sockets. Thus, our recursive calls will correctly match the remaining bulbs to their corresponding sockets.

Runtime. Note that at each level, we perform a linear amount of work: we go through each socket and each bulb and then partition the bulbs and sockets accordingly. Then, we recurse on the big and small groups. Thus, our runtime will be

$$T(n) \leq T(|L_{\text{big}}|) + T(|L_{\text{small}}|) + cn$$

Note that because we pick ℓ uniformly at random, and the bulbs/sockes are distinct sizes, this recurrence is exactly the same as the Quicksort recurrence. Thus, our algorithm has expected $O(n \log n)$ runtime.

(**Note:** Using randomness allows us to get a runtime which has expected run time of $O(n \log n)$ on EVERY input. Otherwise, there might be "bad" inputs which run in $\Omega(n^2)$ time.)

Problem Solving Notes:

1. *Read and Interpret:* The prompt already gives you a hint as to how you can break down the problem into different categories: given a light bulb ℓ, you have exactly one socket that fits just right, while the rest are either too big or too small. Is there a way for you to leverage the fact that you know that a socket is either too small or big for ℓ?

2. *Work out examples:* Try to visualize a small example - start with 3-5 light bulbs and sockets and assign them arbitrary sizes (but all distinct as described in the prompt). How do you match them with each other?

3. *Information Needed:* If a socket is too big for ℓ, which socket is it more likely to fit? A lightbul bigger than ℓ. Similar reasoning applies for sockets that are too small for ℓ. How can you leverage this in your algorithm?