CS 161 (Stanford, Fall 2025) Section 2

1 Divide and Conquer

1.1 Single-dimensional Tarski's fixed point theorem

We say that a function f from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$ is monotone if $f(i) \geq f(j)$ whenever $i \geq j$. A (very) special case of Tarski's fixed point theorem says that for any monotone f, there exists an i such that $f(i) = i$.

Suppose that f is given as array A of n integers such that $f(i) = A[i]$. Notice that in a single dimension, monotonicity of f simply means that A is sorted. Design an algorithm for finding i such that $A[i] = i$ (as is guaranteed to exist by Tarski's fixed point theorem).

Solution.

English description: We'll use a binary search algorithm, i.e. a divide-and-conquer algorithm where at each iteration we'll see if for the middle element, $A[i]$ is bigger or smaller than i, and then recurse on the right or left half of A.

Running time: At each iteration we peform $O(1)$ work and half the size of the array. This takes $\underline{O(\log(n))}$ time.

```
def fixedPoint(A):
    lower = 0
    upper = len(A)-1
    mid = 0
    while A[mid] != mid:
        mid = int((upper + lower)//2)
        if A[mid] == mid:
          break
        elif A[mid] < mid:
            upper = mid
        elif A[mid] > mid:
            lower = mid + 1
    return A[mid]
```

1.2 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 $O(n^2)$. □
2. The maximum sum subarray can also overlap both halves; in other words it passes through the middle element. □
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
    A2, V2 = MaxSubArray(A[mid+1:])        // Case 2: max in right
    maxL, sumL = 0                         // Computing 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
```

2 More Divide and Conquer

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 idea. Use a binary-search-like approach. Check the middle penguin. If it is a local minimum, return it. Otherwise at least one of its neighbors is shorter; recurse on the half that contains a shorter neighbor. If the array size becomes 1, that penguin is a local minimum.

Pseudocode. (Here indices are 1-based.)

```
function LocalMin(A[1..n]):
    if n == 1:
        return 1              # index of the single penguin
    mid := floor(n/2)
    if (mid == 1 or A[mid] < A[mid-1]) and (mid == n or A[mid] < A[mid+1]):
        return mid           # A[mid] is a local minimum
    else if mid > 1 and A[mid-1] < A[mid]:
        return LocalMin(A[1..mid-1])      # recurse on left half
    else:
        return LocalMin(A[mid+1..n])      # recurse on right half
```

Runtime. Each call does $O(1)$ comparisons and recurses on one half of the array. The recurrence is

$$T(n) = T(\lfloor n/2 \rfloor) + O(1),$$

so $T(n) = O(\log n)$.

Proof of correctness. Prove by strong induction on n.

Base case. $n = 1$: the single penguin is trivially a local minimum.

Inductive step. Assume the algorithm returns a local minimum for all arrays of size $< k$. Consider an array A of size k. - If the middle element $A[m]$ is a local minimum, the algorithm returns it, so it is correct. - Otherwise, at least one neighbor of $A[m]$ is smaller than $A[m]$. If $A[m-1] < A[m]$, then there exists a local minimum in the left half $A[1..m-1]$: intuitively the values cannot strictly decrease forever without creating a local minimum at an endpoint, and more formally the recursive argument applies. The algorithm recurses on $A[1..m-1]$, which has size $< k$, so by the inductive hypothesis it returns a local minimum of that half;

that index is also a local minimum in the full array because it is less than all its neighbors within the half and (by the choice of half) also less than $A[m]$. - The symmetric argument holds if $A[m+1] < A[m]$ and we recurse on the right half.

Thus in every case the algorithm returns a true local minimum. By strong induction, it is correct for all n.

3 Solving Recurrences

3.1 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*

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

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

1. $T(n) = 3T(\frac{n}{2}) + O(n^2)$
2. $T(n) = 4T(\frac{n}{2}) + O(n)$
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 $O(n^d) = O(n^2)$. $\square$

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 $O(n^{\log_d a}) = O(n^{\log_2 4}) = O(n^2)$. $\square$

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

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

$\square$

3.2 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...

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

Note: We used the finite geometric series formula $S_n = \frac{a_1(1-r^n)}{1-r}$.

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 $T(n) = 1 \leq 2 \times 1$. 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, $T(n) \le 2n$ for all $n \ge 1$, and (choosing $c = 2$ and $n_0 = 1$ in the definition of Big-Oh) we have established that $T(n) = O(n)$, as desired.

3.3 Tree Method

Consider the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n; \quad T(1) = 1$$

.

Use the Tree Method to find the Big-Theta runtime for algorithms with the above recurrence relation.

Solution.

We use the tree method to prove the claim.

$$\begin{tikzcd}
\text{Level 0:} & & & & n\log n \arrow[ll] \arrow[rr] & & & \\
\text{Level 1:} & & \frac{n}{2}\log\left(\frac{n}{2}\right) \arrow[dl] \arrow[dr] & & & & \frac{n}{2}\log\left(\frac{n}{2}\right) \arrow[dl] \arrow[dr] & \\
\text{Level 2:} & \frac{n}{4}\log\left(\frac{n}{4}\right) & & \frac{n}{4}\log\left(\frac{n}{4}\right) & & \frac{n}{4}\log\left(\frac{n}{4}\right) & & \frac{n}{4}\log\left(\frac{n}{4}\right) \\
& & & & \vdots & & & \\
& & {} \arrow[dl] \arrow[d] \arrow[dr] & & \cdots & & & \\
\text{Level } \log n\text{:} & 1 & 1 & 1 & & & &
\end{tikzcd}$$

The recursion tree drawn above has $\log n + 1$ levels. For each level, we analyze the total runtime of the level and finally, we sum over all levels to solve the recursion. Consider level i. At level i, there are 2^i subproblems, each of size $(n/2^i) \cdot \log(n/2^i)$. Therefore, the total runtime of level i is

$$2^i \times \frac{n}{2^i} \times \log\left(\frac{n}{2^i}\right) = n \log\left(\frac{n}{2^i}\right).$$

So the total runtime for all levels is

$$\sum_{i=0}^{\log n} n \log\left(\frac{n}{2^i}\right) = n \cdot \sum_{i=0}^{\log n} \log\left(\frac{n}{2^i}\right) \leq n \cdot \sum_{i=0}^{\log n} \log n = n \cdot (\log n + 1) \cdot \log n,$$

which is $O(n \log^2 n)$ if we choose $c = 2$ and $n_0 = 2$. On the other hand, we have that

$$\begin{aligned}
n \cdot \sum_{i=0}^{\log n} \log\left(\frac{n}{2^i}\right) \geq n \cdot \sum_{i=0}^{(\log n)/2} \log\left(\frac{n}{2^i}\right) &\geq n \cdot \sum_{i=0}^{(\log n)/2} \log\left(\frac{n}{2^{(\log n)/2}}\right) \\
&\geq n \cdot \frac{\log n}{2} \cdot \log\left(\frac{n}{2^{(\log n)/2}}\right) \\
&= n \cdot \frac{\log^2 n}{4}
\end{aligned}$$

which is $\Omega(n \log^2 n)$ if we choose $c = 1/4$ and $n_0 = 2$.