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).

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)

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.

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

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.

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.