CS 161 (Stanford, Winter 2026) Section 2

1 Divide and Conquer

1.1 Finding a fixed point of an array

Given a 1-indexed sorted array A of n integers (repeats allowed) such that $A[1] \geq 1$ and $A[n] \leq n$, a (very) special case of Tarski's fixed point theorem says that there is some i such that $A[i] = i$. Design an algorithm for finding such an i and provide its runtime.

Solution

Description: We use a version of binary search. At each iteration, there are three cases to handle:

- Case 1: $A[\text{mid}] = \text{mid}$, we found the fixed point.
- Case 2: $A[\text{mid}] < \text{mid}$. Recurse on the left half since that produces a smaller instance of the original problem. Note that all three conditions are met: the subarray is sorted, the first element $A[1] \geq 1$, and the last element of the subarray $A[\text{mid} - 1] \leq A[\text{mid}] < \text{mid}$.
- Case 3: $A[\text{mid}] > \text{mid}$. Similar reasoning as Case 2, but flipped.

Time Complexity: At each iteration we perform $O(1)$ work and divide the size of the array in half. This takes $\underline{O(\log(n))}$ time in total.

Code:

```
def fixedPoint(A):
# Assume A is 1-indexed
  left = 1
  right = len(A)
  mid = 1
  while A[mid] != mid:
    mid = int((left + right) // 2)
    if A[mid] == mid:
      break
    elif A[mid] < mid:
      right = mid
    elif A[mid] > mid:
      left = 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) Describe your algorithm in words.
   (b) What is the time 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 among 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. **Description:** 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 $n/2$. To arrive at the final answer we return the max among these three types.
   Time Complexity: We can compute the third case in O(n) time. This gives us a recurrence relation of the form $T(n) = 2T(n/2) + O(n)$ which solves to $T(n) = O(n \log n)$.
   Code (not required, included for clarity):

```
def maxSubArray(A, l, r):
  if l == r:
    return l, r, A[l]

  mid = (l + r) // 2

  # first case:  entirely in left half
  l1, r1, V1 = maxSubArray(A, l, mid)

  # second case:  entirely in right half
  l2, r2, V2 = maxSubArray(A, mid + 1, r)

  # third case:  crossing the midpoint
  maxL = -∞
  sumL = 0
  left = mid
```

```
for i from mid down to l:
  sumL = sumL + A[i]
  if sumL > maxL:
    maxL = sumL
    left = i

maxR = -∞
sumR = 0
right = mid + 1
for i from mid + 1 to r:
  sumR = sumR + A[i]
  if sumR > maxR:
    maxR = sumR
    right = i

V3 = maxL + maxR

if V1 ≥ V2 and V1 ≥ V3:
  return l1, r1, V1
else if V2 ≥ V1 and V2 ≥ V3:
  return l2, r2, V2
else:
  return left, right, V3
```

2 Space Complexity

Given an array of size $n-1$ containing all the integers between 1 and n except for one (not necessarily sorted), design an algorithm to find the missing number using $O(\log(n))$ extra space.

Solution

One way is to loop through the array and sum up all of the numbers. Then we can use the formula for the sum of all the numbers from 1 to n (which equals $n(n+1)/2$) and subtract our sum from that value to find the missing number. This runs in time $O(n)$ because we just read through the array once and then do one calculation.

3 Recurrence Relations

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-O 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. 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. Here, we assume that T(n) is non-decreasing and we only consider values of n that are power of 2. 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

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

4 Select algorithm

In lecture, we encountered the Select algorithm to find the kth smallest element in an array. We saw that the time complexity of this algorithm depends on how close our pivot element is to the median of the array. This problem explores this dependence in more detail.

1. If we are guaranteed to select exactly the median as our pivot (i.e. we get a 50-50 split), what is the runtime of Select? What if we have no guarantees on our pivot?
2. If we are guaranteed to select a pivot that gives us at worst a c-$(1-c)$ split, where $\frac{1}{2} \leq c < 1$ is some constant that doesn't depend on n, what is the runtime of Select?
3. Assume that in the worst case, it takes more than a constant number of splits to cut the size of our array in half. (This means that the number of splits required is $\Omega(1)$

but not $\Theta(1)$, i.e. $\omega(1)$.) Show that the runtime of Select in this case must be larger than $O(n)$.

Solution

1. If we select exactly the median, then we divide the array in half each time, so the maximum depth of our recursion tree is $\log n$. Since the amount of work at each level of the tree is equal to the current length of the subarray, the total amount of work is
$$\sum_{i=1}^{\log n} 2^i = O(n).$$
If we select the worst case element each time, we only decrease the length of A by 1 in each step. Therefore the total runtime is
$$\sum_{i=1}^{n} i = O(n^2).$$

2. In this case, we are guaranteed to at worst multiply the length of A by $c < 1$. Therefore after t levels of recursion, the size of our subarray is at most $n \cdot c^t$. Solving for the value of t that makes this equal to 1, we find that the maximum depth of our recursive tree is $t = \log n + \log(1/c) = O(\log n)$. The runtime for each level of the tree is proportional to the current length of the subarray, which is $O(c^i)$ at the ith level from the smallest case. Therefore the total runtime is
$$\sum_{i=1}^{O(\log n)} O(nc^i) = \frac{1}{1-c} O(n) = O(n).$$

3. Say it takes $f(n)$ splits to cut the size of the array in half, where $f(n) = \omega(1)$. The runtime at each of these splits before reaching half the original array size is at least $n/2$. Therefore the total runtime is at least $f(n) \cdot n/2 = \omega(n)$.