CS 3510 – Assignment 1

Due `Friday, May 27, 2022` at `11:59pm` on Canvas

Instructor: Shahrokh Shahi

- Please type your answers (LaTeX is highly recommended) and upload a single PDF file named `<Your-GT-Account>.pdf`, e.g., GBurdell3.pdf, including all your answers. You can submit multiple times. Canvas keeps track of the submissions and append a version number when you re-submit. We always grade your most recent submissions.
- Please read the policies, and do not forget to acknowledge your collaborators and cite your references.
- If you do not understand a question, please ask on Piazza or come to office hours well ahead of the due date.

Problem 1 — Asymptotic Notations (10 pts)

1. (5 pts) For each pair of functions f and g, write whether f is in $\mathbb{O}(g)$, $\Omega(g)$, or $\Theta(g)$, whichever is most accurate. Just write the asymptotic notation; no explanation is required.

   (a) $f = (n + 1000)^4$, $g = 1000n^4 - 2n^3 + 1$

   (b) $f = \log_{1000} n$, $g = \log_2 n$

   (c) $f = n^{1000}$, $g = n^2$

   (d) $f = 2^n$, $g = n!$

   (e) $f = (n + 1)^3$, $g = 4^{\log_2 n}$ (Hint: $a^{\log_b c} = c^{\log_b a}$)

2. (5 pts) Use the mathematical definition of big-O notation to prove the following additivity properties: f, g and h are functions of input size n. Prove that if $f \in \mathbb{O}(h)$ and $g \in \mathbb{O}(h)$, then $f + g \in \mathbb{O}(h)$.

Solution

1. (a) $f = (n + 1000)^4$, $g = 1000n^4 - 2n^3 + 1$ $\quad$ $\mathbf{f \in \Theta(g)}$

   (b) $f = \log_{1000} n$, $g = \log_2 n$ $\quad$ $\mathbf{f \in \Theta(g)}$

   (c) $f = n^{1000}$, $g = n^2$ $\quad$ $\mathbf{f \in \Omega(g)}$

   (d) $f = 2^n$, $g = n!$ $\quad$ $\mathbf{f \in O(g)}$

   (e) $f = (n + 1)^3$, $g = 4^{\log_2 n}$ $\quad$ $\mathbf{f \in \Omega(g)}$ $\quad$ Note: $4^{\log_2 n} = n^{\log_2 4} = n^2$

2. Proof:

   - $f \in \mathbb{O}(h) \Rightarrow$ there exist a constant $c_1 > 0$ and $n_1 \geq 0$, such that for any $n \geq n_1$, $f(n) \leq c_1 h(n)$.
   - $g \in \mathbb{O}(h) \Rightarrow$ there exist a constant $c_2 > 0$ and $n_2 \geq 0$, such that for any $n \geq n_2$, $g(n) \leq c_2 h(n)$.
   - Let $n_3 = \max(n_1, n_2)$, then for any $n \geq n_3$, we can write:

     $$f(n) \leq c_1 h(n) \tag{1}$$

     and

     $$g(n) \leq c_2 h(n) \tag{2}$$

   - From 1 and 2, we have:
     There exist a constant $c = c_1 + c_2 > 0$ and $n_3 = \max(n_1, n_2)$, such that for any $n \geq n_3$, $f(n) + g(n) \leq (c_1 + c_2)h(n)$. Thus, $f + g \in \mathbb{O}(h)$.

Problem 2 — Divide and Conquer (20 pts)

You are given a sorted array $S = [s_1, s_2, \ldots, s_n]$ with n distinct integers, i.e., $s_i < s_{i+1}$, for all $1 \leq i < n$. Design a divide-and-conquer algorithm to decide whether there exists an index k such that $S[k] = k$. If such an element exists return the index, otherwise return -1. Your algorithm should run in $O(\log n)$ time.

- Provide a description of your algorithm (in words and pseudocode), and justify its correctness.
- Discuss the running time by providing the recurrence relation and applying the Master Theorem.

Solution

- **Design and Correctness:**
  The algorithm can be designed as a binary search. We start with comparing $S[n/2]$ with $n/2$. If $S[n/2] = n/2$, then we return $n/2$ as the answer. Otherwise, if $S[n/2] > n/2$, then we claim that for any $k > n/2$, we must have $S[k] > k$ because all integers in S are distinct and sorted in ascending order. Thus, for any $k > n/2$, we have $S[k] \geq S[n/2] + (k - n/2) > n/2 + k - n/2 = k$. Therefore, when we have $S[n/2] > n/2$, we can discard the right half of the array and limit our search to the left sub-array. Similarly, if $S[n/2] < n/2$, then we can discard the left sub-array and limit our search to the right sub-array. This procedure has been demonstrated in the following pseudo-code:

Algorithm 1: `SearchIndex`

Input: $S = \{s_1, s_2, ..., s_n\}, lo = 1, hi = n$
Result: index k such that $S[k] = k$, otherwise -1

```
   // base case
 1 if (lo == hi and S[lo] = lo) then
 2 |   return lo
 3 end
 4 if (lo == hi and S[lo] ≠ lo) then
 5 |   return −1
 6 end

   // recurrence relation
 7 mid ← [(lo + hi)/2]
 8 if (S_mid == mid) then
 9 |   return mid
10 else if (S_mid < mid) then
11 |   return SearchIndex(S, mid+1, hi)
12 else
13 |   return SearchIndex(S, lo, mid-1)
14 end
```

Alternative solution: We can solve this problem by a binary search approach. For this purpose, we can define a new array $B[i] = A[i] - i$. It can be shown that the new array is also sorted in increasing order, i.e., $B[i] \leq B[i+1]$:

$$B[i] = A[i] - i \leq (A[i+1] - 1) - i = A[i+1] - (i+1) = B[i+1]$$

It is obvious that $A[i] = i$ iff B[i] = 0. Therefore, our search problem transforms to finding an index i such that B[i] = 0. Thus, the binary search algorithm can be employed to find the index of the element which is equal to the target value 0.

- **Running time analysis:**
  Similar to the binary-search algorithm explained in the lectures, at each step of the recursion, the search domain is divided by a factor of 2, and the recurrence relation is

  $$T(n) = T(n/2) + \Theta(1)$$

  Therefore, using Master Theorem, we have $a = 1$, $b = 2$, and $d = 0$, which is the second case $a = b^d$. Thus, $T(n) \in \Theta(\log n)$.

Problem 3 — Divide and Conquer (10 pts)

You are given a rotated sorted array S of size n. Design a binary search algorithm to find the minimum element of this array. Your algorithm should run in $O(\log n)$ time. Provide a description of your algorithm. Runtime analysis is not required.

Def. Rotated sorted array of size n is a sorted array, where its elements are shifted k times ($0 \leq k < n$) to the right. For instance, let $S = [0, 1, 2, 3, 4, 5, 8]$ be a sorted array before rotation, then

- *After $k = 3$ rotations: $S = [4, 5, 8, 0, 1, 2, 3]$*
- *After $k = 6$ rotations: $S = [1, 2, 3, 4, 5, 8, 0]$*

Note for both examples, your algorithm should return 0 as the minimum of the array.

Solution

We can apply a modified version of `Binary-search` algorithm discussed in the lectures to find the minimum of a rotated sorted array. If a sorted array is not rotated then we have $a_1 < \ldots < a_n$, thus $a_1 < a_n$ and a_1 is the min of the array. For instance, if $A = [2, 3, 4, 5, 8, 10, 11]$ then we have $a_1 = 2 < a_n = 11$ and 2 is the minimum value of the array. Otherwise, If $a_1 > a_n$ it means the sorted array is rotated (Ex. $A = [5, 8, 10, 11, 2, 3, 4]$, where $a_1 = 5 > a_n = 4$). That implies that there exists an element a_i such that $a_{i-1} > a_i$ for $1 < i \leq n$, where a_i is the min of the array, also known as the rotation (or pivot) point. We can modify the `Binary-search` to find this element:

`while` $lo < hi$:

1. Find the mid element $mid \leftarrow [\frac{lo+hi}{2}]$
2. If $a_{mid} < a_n$, it means that the right half is sorted; therefore, the rotation point is in the left sub-array. Thus, set $hi \leftarrow mid$ to continue the search in the left sub-array.
3. If $a_{mid} > a_n$, it means that the right half is not sorted; therefore, the rotation point is in the right sub-array. Thus, set $lo \leftarrow mid + 1$ to continue the search in the right sub-array.
4. The search stops when two pointers meet each other at the position of the minimum element, i.e., $lo = hi$, where $min = a_{lo} = a_{hi}$

Note alternatively, you can compare the mid element with a_1 at each iteration.

Algorithm 2: `BinarySearchRotated`

Input: $A = \{a_1, a_2, ..., a_n\}, lo = 1, hi = n$
Result: $\min(A)$

```
1 while (lo < hi) do
2 |   mid ← [(lo + hi)/2]
3 |   if (S_mid < S_hi) then
4 |   |   r ← mid
5 |   else if (S_mid > S_hi) then
6 |   |   l ← mid + 1
7 end
8 return A[lo]
```

A running time analysis is not required in this problem. But with a reasoning similar to the previous problem, we can obtain $T(n) \in \mathbf{\Theta}(\log n)$.

Notes:

- While an iterative approach is presented in this solution, any other correct algorithm, e.g., recursive approach, is also accepted.
- The $\mathbb{O}(\log n)$ solution is guaranteed only if the numbers in this array are distinct. Otherwise, the time complexity is on average $\mathbb{O}(\log n)$, but in the worst case, where all elements are identical, the binary search will iterate over each element, and thus, it gives $\mathbb{O}(n)$ running time.