Name: Sakina Ghafoor
COSC 40403 - Analysis of Algorithms: Fall 2024: Homework 1
Due: 23:59 on September 1, 2024

1. (5 points) Practice your LaTeX. Write out the definitions of $O(f(n))$, $\Omega(f(n))$, and $\Theta(f(n))$ in equation format using LaTeX.

Answer:

$O(f(n)) = \{f(n)$: $\exists$ positive constants c, n_0 such that $0 \leq f(n) \leq c * g(n), \forall n \geq (n_0)\}$.

$\Omega(f(n)) = \{f(n)$: $\exists$ positive constants c, n_0 such that $0 \leq c * g(n) \leq f(n), \forall n \geq (n_0)\}$.

$\Theta(f(n)) = \{f(n)$: $\exists$ positive constants c, n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n), \forall n \geq (n_0)\}$.

2. (5 points) Arrange the following functions in order of increasing growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$.

   (a) $2^{\lg n}$

   (b) $2^{2^{\lg n}}$

   (c) $n^{5/2}$

   (d) 2^{n^2}

   (e) $n^2 \lg n$

Answer:

A) $2^{\lg n} = O(n)$

E) $n^2 \lg n = O(n^2 log n)$

C) $n^{5/2} = O(n^{5/2})$

B) $2^{2^{\lg n}} = O(2^n)$

D) $2^{n^2} = O(2^{n^2})$

3. (15 points total) **(True or False) and Justify** the correctness of the following statements assuming that $f(n)$ and $g(n)$ are asymptotically positive functions.
   (a) (5 points) $n^3 \in O(n^2)$
   (b) (5 points) $f(n) + g(n) \in O(\max(f(n), g(n)))$
   (c) (5 points) $2^{n+1} \in O(2^n)$

Answer:

(a) The statement shows that n^2 is the upper bound for n^3. This is false because n^3 grows at a faster rate than n^2, so it can not be the upper bound on a faster growth rate. False.

(b) The statement shows the the sum of $f(n)$ and $g(n)$ is upper bounded by the maximum of both or $O(\max(f(n), g(n)))$. This is true because for any value of n for either function, even if $f(n)$ and $g(n)$ have an exponent, whichever larger value of n will be the upper bound taken by big O with the max function. True.

(c) The statement shows that 2^{n+1} is bounded above by 2^n. This is true because when breaking up the function 2^{n+1}, we are left with $2^n * 2^1$. Constants and lower order terms are suppressed in asymptotic notation so the growth rate of the function $f(n)$ or 2^n is less than or equal to $c * g(n)$ or $O(2^n)$. True.

4. (10 points) Similar to the peak finding problem, the "valley finding" problem finds a valley in an array. A valley in an array is an array value at index v, such that, $A_{v-1} \geq A_v \leq A_{v+1}$. Given an array A of distinct integers, write a Python function and test program that finds a valley in an array. You must use the divide-and-conquer approach as discussed in class.

Answer:

```
def valley(a):
    first = 0
    last = len(a)-1
    valley_index = find(a, first, last)
    return valley_index

def find(a, first, last):
    if first==last:
        return first

    # first thing to do in divide and conquer algorithm is to compute
    # the middle index
    middle = (first+last)//2

    # then test if A_mid is a valley (if true then return it)
    if (middle==0 or a[middle-1]>=a[middle]) and (middle==last or
                                                  a[middle+1]>=a[middle]):
        return middle

    # if not then valley should be to the left or right,
    # neighbor less than A_mid

    #if A_mid-1 < A_mid then the peak must be to the left,
    # recurs on A[1 : mid-1]
    elif middle>0 and a[middle-1] < a[middle]:
        return find(a, first, middle-1)

    # if A_mid+1 < A_mid then the peak must be to the right,
    # recurs on A[mid+1 : n]
    elif middle>0 and a[middle+1] < a[middle]:
        return find(a, middle+1, last)


if __name__ == "__main__":
    array = [10, 7, 8, 5, 3, 12, 6]
    valley_index = valley(array)
    print(f"Valley found at index: {valley_index}")
    print(f"Value: {array[valley_index]}")
```

5. (15 points total) Checking for duplicates. Our textbook on page 31 has an example involving nested loops to check if array A has a duplicate entry. The following Python program implements two solutions to this problem.

   (a) (10 points) Using the analysis technique presented in class, what is this asymptotic upper-bound of `has_duplicate_1`? Show your work to receive full credit.

   (b) (5 points) You were talking about the `has_duplicate_1` function to a friend at another university. They mention say *"Oh yeah. I can write a Python function to solve this problem using only one loop. Mine's faster"*. You see their solution listed below (`has_duplicate_2`). What is the asymptotic upper bound of their solution? Show your work to receive full credit.

```python
#=====================================================================
# Problem: Given a list of integers, return True if the list has a duplicate
#          and False otherwise.
# Example:
#   Input: [1,2,3,4,5,6,7,8,9,5]
#   Output: True
#   Input: [1,2,3,4,5,6,7,8,9]
#   Output: False
#=====================================================================
#=====================================================================
# Solution 1:
#   - Use two nested loops to compare each element with every other element
#   - If the same element is found, return True
#   - If no duplicate is found, return False
#=====================================================================
def has_duplicate_1(A:list) -> bool:
    i = 0
    while i < len(A):
        j = i + 1
        while j < len(A):
            if A[i] == A[j]:
                return True
            j = j + 1
        i = i + 1
    return False


#=====================================================================
# Solution 2:
#   - Use a single loop to iterate through the list
#   - Check if the current element is in the rest of the list
#   - If the element is found, return True
#   - If no duplicate is found, return False
#=====================================================================
def has_duplicate_2(A:list) -> bool:
    i = 0
```

```
37      while i < len(A):
38          if A[i] in A[i+1:]:
39              return True
40          i = i + 1
41      return False
42
43
44  #====================================================================
45  # Main program
46  #====================================================================
47  # if __name__ == "__main__":
48  a  = [1,2,3,4,5,6,7,8,9,5]
49  print(has_duplicate_1(a))
50  print(has_duplicate_2(a))
51
52  b = [1,2,3,4,5,6,7,8,9]
53  print(has_duplicate_1(b))
54  print(has_duplicate_2(b))
```

Answer:

(a) The asymptotic upper bound of `has_duplicate_1` is $O(n^2)$.
Worst Case Analysis
line 16 = c1 = 1 time
line 17 = c2 = 1 time
line 18 = c3 = n+1 times
line 19 = c4 = n times
line 20 = c5 = n(n+1)
line 21 = c6 = n*n times
line 22 = c7 = 0 times
line 23 = c8 = n*n times
line 24 = c9 = n times
line 25 = c10 = 1 time
$T(n) = c_1+c_2+c_3*(n+1)+c_4*n+c_5*[n(n+1)]+c_6*(n*n)+c_7*0+c_8*(n*n)+c_9*n+c_{10}*1$
$T(n) = c_1 + c_2 + c_3n + c_3 + c_4n + c_5n^2 + c_5n + c_6n^2 + c_8n^2 + c_9n + c_{10}$
$T(n) = (c_5 + c_6 + c_8)n^2 + (c_3 + c_4 + c_5 + c_9)n + (c_1 + c_2 + c_3 + c_{10})$
$T(n) = (a)n^2 + (b)n + (c)$
$O(n^2)$

(b) The asymptotic upper bound of `has_duplicate_2` is $O(n^2)$.
Worst Case Analysis
line 35 = c1 = 1 time
line 36 = c2 = 1 time
line 37 = c3 = n+1 times
line 38 = c4 = n*n times
line 39 = c5 = 0 times

line 40 = c6 = n times
line 41 = c7 = 1 time
$T(n) = c_1 * 1 + c_2 * 1 + c_3 * (n + 1) + c_4 * (n * n) + c_5 * 0 + c_6 * n + c_7 * 1$
$T(n) = c_1 + c_2 + c_3 n + c_3 + c_4 n^2 + c_6 n + c_7$
$T(n) = (c_4)n^2 + (c_3 + c_6)n + (c_1 + c_2 + c_3 + c_7)$
$T(n) = (a)n^2(b)n + (c)$
$O(n^2)$

6. (15 points total) Consider the ***Searching problem***:
Input: A sequence of n numbers $A = \langle a_1, a_2, a_3, \ldots, a_n \rangle$ and a value v.
Output: An index i such that $v = A[i]$. If v is not in A, then the algorithm will return -1.

Observe that if a sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The BINARY-SEARCH algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time.

(a) (5 points) Write pseudocode (not Python code), either iterative or recursive, for BINARY-SEARCH.

(b) (5 points) Develop the cost-model equation for $T(n)$.

(c) (5 points) Show that the worst-case running time is $\Theta(\lg n)$.

Answer:

(a) Recursive Binary Search

```
binarySearch(v, A, lower, upper)
  i ← (lower+upper)/2
  if lower>upper
    return -1
  if A[i] == v
    return i
  else if A[i] < v
    return binarySearch(v, A, lower, i-1)
  else
    return binarySearch(v, A, i+1, upper)
```

(b) $T(n) = T(n/2) + (c)$
The array or sequence is split into 2 and each part is analyzed to find a value and if it is not found further split the halves. This is represented by n/2. The constant represents the comparison of the middle element i and v. Doing the recursion a second time yields $T(n) = T(n/4) + c + c$. Need to determine how many times the array is split. In this case, the length of the array is important. We can insert the length of the input array, i, into the cost model equation. $T(n) = T(n/2^i) + i * c$
Since the algorithm is recursive, it could keep going until the base case and fail to find v. The base case is a length of 0 or 1. To reach this, each sequence or part is split logn times. This can be substituted for i in the cost model equation since that is the amount of steps it takes to reach the base case.
$T(n) = T(n/2^{logn}) + c * logn$
$T(n) = T(n/n) + c * logn$
$T(n) = T(1) + c * logn$
T(1) is a constant.
$T(n) = c_0 + c * logn = O(logn)$

(c) We have already established $O(logn)$ for the growth rate. To find the worst case running time, we know in each step the problem sized is halved. So we know there is

an upper bound of logn since this is the maximum number of steps that can be taken to search through the array. In the worse case, the value is not found even in the last step in the base case, the minimum number of steps is also logn. To determine that the value is not in the sequence, no fewer or more logn steps can be taken. Thus, the worst case running time is $\Theta(logn)$.
Size of search space $= n/2 + n/4 + n/8 + ... = n/2^k$
$n/2^k = 1$
$n = 2^k$
$k = log_2 n$