CS-GY 6033 : Design and Analysis of Algorithms , Section C : Assignment-1

Khwaab Thareja — N15911999 — kt3180

October 2024

Problem 3-1. Solving Recurrences

For each of the following recurrences, use the master theorem or the recursion tree to give the tight asymptotic upper bound.

(a) $T(n) = T\left(\frac{3n}{4}\right) + n^{3/4}$

(b) $T(n) = 2T\left(\frac{n}{8}\right) + n^{1/3}$

(c) $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$

(d) $T(n) = 2T(\sqrt{n}) + \log^2 n$

(e) $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + n$

Answer 1.a

We have the equation $T(n) = T\left(\frac{3}{4}n\right) + n^{3/4}$.

Solving using the master theorem, we have:

$$a = 1, \quad b = \frac{4}{3}, \quad \text{and} \quad c = \frac{3}{4}$$

From the master theorem, we know that:

$$T(n) = O(n^{\log_b a}) \quad \text{if } c < \log_b a \tag{I}$$

$$T(n) = O(n^c \log n) \quad \text{if } c = \log_b a \tag{II}$$

$$T(n) = O(n^c) \quad \text{if } c > \log_b a \tag{III}$$

Therefore, finding $\log_b a$:

$$\log_b a = \log_{\frac{4}{3}} 1 = 0$$

Since $c = \frac{3}{4}$, this satisfies tag (III) of the master theorem. Therefore, we conclude:

$$T(n) = O(n^{3/4})$$

Answer 1.b

We have the equation $T(n) = 2T\left(\frac{n}{8}\right) + n^{1/3}$.

Solving using the master theorem, we have $a = 2$, $b = 8$, and $c = \frac{1}{3}$.

From the master theorem, we know that:

$$T(n) = O(n^{\log_b a}) \quad \text{if } c < \log_b a \tag{I}$$

$$T(n) = O(n^c \log n) \quad \text{if } c = \log_b a \tag{II}$$

$$T(n) = O(n^c) \quad \text{if } c > \log_b a \tag{III}$$

Therefore, finding $\log_b a$:

$$\log_b a = \log_8 2 = \frac{1}{\log_2 8} = \frac{1}{3}$$

Also, we have $c = \frac{1}{3}$. Therefore, $c = \log_b a$, which satisfies case II of the master theorem. Hence, we get:

$$T(n) = O(n^{1/3} \log n)$$

Answer 1.c

We have the equation $T(n) = 4T\left(\frac{n}{3}\right) + n \log n$.

In this case, we have $a = 4$, $b = 3$, and $f(n) = n \log n$.

We have to use the master theorem for dividing functions.

Here, $n \log n$ can be written in the form of $n^k (\log n)^p$ where $p = 1$ and $k = 1$.

Now finding $\log_b a$:

$$\log_b a = \log_3 4$$

Clearly, $\log_b a > k$, i.e., $\log_3 4 > 1$.

This means $n^{\log_3 4}$ will show faster growth than $n \log n$. So this makes the master theorem's condition I valid.

Hence, we get:

$$T(n) = O(n^{\log_3 4})$$

Answer 1.4

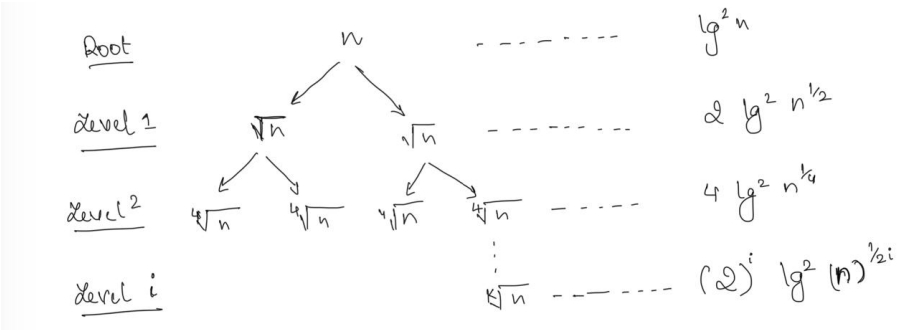
We have:

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

This problem cannot be solved using the master theorem since $b = 1$, but for the master theorem, we need $b > 1$, unless we assume T(n) as some other function

So we will solve this using the recursion tree method.

We have:



At the root level, the cost is $(\log n)^2$.

At level 1, the cost is $2(\log(n^{1/2}))^2$.

At level 2, the cost is $4(\log(n^{1/4}))^2$.

At level i, the cost is $2^i \cdot (\log(n^{1/(2^i)}))^2$.

Therefore, simplifying the i-th level, we get:

$$\frac{1}{2^i} \cdot (\log n)^2$$

Calculating the total cost:

$$T(n) = (\log n)^2 \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$$

This forms a geometric progression (GP) of constants, Which in trun results in sum as 3, which we can ignore.

Therefore, we have:

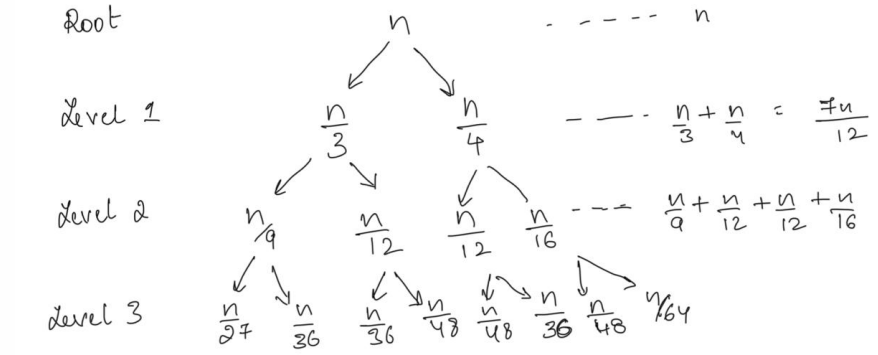
$$T(n) = O((\log n)^2)$$

Answer 1.5

We have:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + n$$

Solving this using the recursion tree method:



At the root level, we have total cost as n.

At level 1, we have total cost as:

$$\frac{n}{3} + \frac{n}{4}$$

At level 2, we have total cost as:

$$\frac{n}{9} + \frac{n}{12} + \frac{n}{12} + \frac{n}{16}$$

At level 3, we have total cost as:

$$\frac{n}{27} + \frac{n}{36} + \frac{n}{36} + \frac{n}{36} + \frac{n}{48} + \frac{n}{48} + \frac{n}{48} + \frac{n}{64}$$

At level i, we have total cost as:

$$\left(\frac{7}{12}\right)^i \cdot n$$

The total depth at level i is $\log_4(n)$, therefore the total cost is:

$$n \sum_{i=0}^{\log_4(n)} \left(\frac{7}{12}\right)^i$$

Therefore, referencing slide 50/62 from the Divide and Conquer slides, we get:

$$T(n) = O(n)$$

Problem 3-2. Double median

You are given as input two sorted length-n arrays $A[1..n]$ and $B[1..n]$, where all elements are distinct. The goal of this problem is to find the median of the union of the arrays.

Example. If $A = \langle 5, 8, 9, 10, 20\rangle$ and $B = \langle 7, 30, 31, 32, 35\rangle$, the two middle-most values are 10 and 20, so the median is 15.

Since the total number of elements is even, the answer will always be the average of the lower and upper medians.

(a) Efficient Divide-and-Conquer Algorithm

Describe an efficient divide-and-conquer algorithm for finding the median of two sorted arrays.

The naive algorithm would be to merge the two sorted arrays in $O(n)$ time, producing a size-$2n$ array of all the elements. You could then output the average of the n-th and $(n+1)$-th elements. To be considered efficient, your algorithm has to be strictly better than $\Theta(n)$.

(b) Analyze the Running Time

Analyze the running time of your algorithm.

"Analyze" means showing and explaining how you're coming up with the bound. For a divide-and-conquer problem, that means writing and solving a recurrence. Just stating a bound with no justification is not an analysis.

Answer 2.a

We have two sorted arrays A and B. This problem seems to be a variation of the selection problem, where in this case, instead of one, we have two sorted arrays.

We will find the middle element of both sorted arrays. Whichever array's median is greater, we will keep it as array B for our algorithm to work. Then we will discard half of the array whose median is smaller and will keep on doing it until we reach the median.

For this, we will have two separate functions: one will find the median, and the other will find the i-th element, which is the position of the median.

This recursion will continue until we reach the median or the base case.

The base case will be if, in some situation, the length of either of the arrays is 0, then we would simply return the middle element or the average of the two middle elements of the other array since the array is already sorted.

Another base case to end the recursion would be when $i = 1$ from continuous reduction, then we would simply return the minimum element from the first element of both arrays.

For finding the median function, we would have to take the total combined length of both arrays and then find the i-th element depending on whether the length is odd or even.

The element we would like to find is the middle element (for an odd combined length of the two arrays) or the two middle elements (for an even combined length of the two arrays), i.e.,

$$i\text{-th element} = \frac{n_1 + n_2}{2} \quad \text{(if odd)}$$

$$i\text{-th element} = \frac{n_1 + n_2}{2} \quad \text{and} \quad (i+1)\text{-th element} \quad \text{(if even)}$$

Algorithm: selection_two_arrays(A, B, i)

Algorithm 1 select_ith_element

```
1: procedure SELECT_ITH_ELEMENT(A, B, i)
2:     if size(A) > size(B) then
3:         return SELECT_ITH_ELEMENT(B, A, i)      ▷ Ensure A is the smaller array
4:     end if
5:     if size(A) = 0 then
6:         return B[i] ▷ Case 1: If A is empty, return the i-th element from B
7:     end if
8:     if i = 1 then
9:         return min(A[1], B[1])       ▷ Case 2: If looking for the 1st smallest element, return the smaller between A and B
10:    end if
11:    part_A ← min(i//2, size(A))                ▷ Divide A into part_A elements
12:    part_B ← min(i//2, size(B))                ▷ Divide B into part_B elements
13:    if A[part_A − 1] < B[part_B − 1] then
14:        return SELECT_ITH_ELEMENT(A[part_A..], B, i - part_A)  ▷ Discard first part_A elements of A
15:    else
16:        return SELECT_ITH_ELEMENT(A, B[part_B..], i - part_B)  ▷ Discard first part_B elements of B
17:    end if
18: end procedure
```

Algorithm 2 find_median

procedure FIND_MEDIAN(A, B)
    total_size ← size(A) + size(B) ▷ Compute the total number of elements
    if total_size is odd **then**
        return SELECT_ITH_ELEMENT(A, B, (total_size // 2) + 1) ▷ If odd, return the middle element
    else
        median_lower ← SELECT_ITH_ELEMENT(A, B, total_size // 2) ▷ Find the smaller middle element
        median_upper ← SELECT_ITH_ELEMENT(A, B, (total_size // 2) + 1) ▷ Find the larger middle element
        return (median_lower + median_upper) / 2 ▷ Return the average of both middle elements for an even count
    end if
end procedure

```python
def select_ith_element(A, B, i):



    if len(A) > len(B):
        return select_ith_element(B, A, i)

    if len(A) == 0:
        return B[i - 1]

    if i == 1:
        return min(A[0], B[0])

    part_A = min(i // 2, len(A))
    part_B = min(i // 2, len(B))

    if A[part_A - 1] < B[part_B - 1]:
        return select_ith_element(A[part_A:], B, i - part_A)
    else:
        return select_ith_element(A, B[part_B:], i - part_B)

def find_median(A, B):
    total_size = len(A) + len(B)

    if total_size % 2 == 1:
```

```
25          return select_ith_element(A, B, (total_size // 2) +
                1)
26      else:
27          median_lower = select_ith_element(A, B, total_size
                // 2)
28          median_upper = select_ith_element(A, B, (total_size
                // 2) + 1)
29          return (median_lower + median_upper) / 2
```

Answer 2.b

The algorithm works by:

- Picking a middle point in both arrays.
- Comparing the elements at those points.
- Reducing the problem size by removing part of one array, based on which element is smaller.

Base Case: If one array is empty, the algorithm simply returns the i-th smallest element from the other array. This takes constant time, or $O(1)$.

Recursive Step: In each step, the algorithm looks at the middle element from both arrays (around the $i//2$-th element). Then, it compares them and discards half of one of the arrays. This means that with each step, the size of at least one array gets cut in half.

Recurrence Relation: At each recursive step, one of the arrays is halved. The recurrence relation for the time it takes can be written as:

$$T(i) = T\left(\frac{i}{2}\right) + O(1)$$

This is similar to how binary search works, where the problem size is reduced by half each time. The $O(1)$ here means the time spent comparing and selecting elements.

When we solve this recurrence, we find that the algorithm runs in:

$$T(i) = O(\log \min(m, n))$$

where m and n are the sizes of the two arrays. The time complexity depends on the smaller array, because that array is the one that gets halved with each recursive step.

In simple terms, the algorithm runs faster than a linear scan and works in logarithmic time based on the size of the smaller array.

Problem 3-3. Local Minimum

Suppose we are given an array $A[1...n]$ of distinct numbers (i.e., no two of the numbers are the same). We say that cell i is a local minimum if its value is strictly smaller than the values of its neighbors, i.e., if $A[i] < A[i-1]$ and $A[i] < A[i+1]$.

For example, there are four local minima (circled) in the following array:

2 1 15 18 9 12 10 8 7 13 17 21 19 10 6

It is easy to find all local minima in $\Theta(n)$-time by scanning through the array and comparing each cell against its neighbors. However, if we only need to output a single local minimum, instead of all, we can do better. This problem asks you to produce sublinear algorithms that do not require reading the entire input.

(a)

Describe an $O(\log n)$-time divide-and-conquer algorithm for returning a local minimum. Justify that your algorithm achieves the desired running time.

Now let's generalize the problem to a 2-dimensional array:

Given a 2-dimensional array $A[1...n, 1...C]$, we now have an $n \times C$ table of distinct numbers. To design your algorithm, focus primarily on the n-dimension,

treating C as a constant.

The definition of a local minimum in this 2-dimensional table remains the same: a cell (i, j) is a local minimum if its value is smaller than the neighboring cells:

$$(i-1, j), \quad (i+1, j), \quad (i, j-1), \quad (i, j+1)$$

Here is an example of such a table for $n = 7$ and $C = 4$, where all local minima are circled:

6	14	11	2
22	1	5	53
15	27	91	35
36	18	31	39
13	21	9	25
17	24	12	33
19	10	16	41

(b)

Suppose you are provided with an array $M[1...n]$ where $M[i]$ is the global minimum element in the i-th row of A, i.e.,

$$M[i] = \min_{1 \le j \le C} \{A[i, j]\}$$

For example, for the table above, M would be:

2
1
15
18
9
12
10

Given both A and M, describe how to efficiently find a local minimum in A. What is the running time of your algorithm? How many cells of M does your algorithm read? *Hint: you should leverage your previous algorithm.*

(c)

Now drop the assumption that you are given M. Describe an $O(C \log n)$-time algorithm for finding a local minimum in A. *Hint: you cannot afford to precompute M since that would take $\Theta(nC)$ time. However, consider how much of M you actually need and when you need it. Keep your answer concise.*

Answer 3.a

We can solve the problem using a divide-and-conquer approach.

The idea is to first find the middle element of the array. Then, we compare the middle element with its neighbors. If the middle element is smaller than both of its neighbors, then it is a local minimum.

If

$$A[\text{mid}] > A[\text{mid} - 1],$$

then there must be a local minimum in the left half of the array. Therefore, we continue the search in the left half.

If

$$A[\text{mid}] > A[\text{mid} + 1],$$

then there must be a local minimum in the right half of the array. Therefore, we continue the search in the right half.

Next, we create a recursive function for either the left or the right half of the array, depending on the comparison of the middle element as described above.

The recursion continues until a single element remains, which is trivially a local minimum since it has no neighbors.

The algorithm is similar to binary search because it reduces the problem size by half at each step. Depending on the comparisons, the search continues either in the left half or the right half of the array.

This results in the following recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Solving this recurrence gives:

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

Algorithm 3 find_local_minimum

```
1: procedure FIND_LOCAL_MINIMUM(A, low, high)
2:     if low == high then
3:         return A[low]                                          ▷ Base case
4:     end if
5:     mid ← (low + high) // 2
6:     if A[mid] ¡ A[mid-1] and A[mid] ¡ A[mid+1] then
7:         return A[mid]                                          ▷ local minimum
8:     else if A[mid] ¿ A[mid-1] then
9:         return FIND_LOCAL_MINIMUM(A, low, mid - 1)  ▷ Search in the left
   half
10:    else
11:        return FIND_LOCAL_MINIMUM(A, mid + 1, high)     ▷ Search in the
   right half
12:    end if
13: end procedure
```

which is the time complexity of the algorithm.

Answer 3.b

LocalMin2D(A, M, n, C):

- Initiates the binary search across rows using `RowSearch`.

RowSearch(A, M, low, high, C):

- **Binary Search on Rows:** Finds the middle row (`mid_row`).
- **Use M:** Retrieves the column index (`min_col`) of the local minimum in `mid_row` from `M`.
- **Check Neighbors:** Compares `A[mid_row][min_col]` with its above, below, left, and right neighbors to determine if it's a global local minimum.
- **Move Search Space:** If not a local minimum:
  - If the above neighbor is smaller, move to the upper half (`high = mid_row - 1`).
  - Otherwise, move to the lower half (`low = mid_row + 1`).
- Returns the index (`mid_row, min_col`) if a local minimum is found.

Time Complexity:

- Binary search across rows takes $O(\log n)$.
- Neighbor comparisons take $O(C)$.
- **Total Complexity:** $O(\log n + C)$.

This approach efficiently finds a global local minimum by leveraging precomputed local minima from `M` and using binary search to reduce the number of rows being checked.

Algorithm: Local Minimum in 2D Array

```
def LocalMin2D(A, M, n, C):
    # Perform binary search over the rows of A using the indices provided in M
    return RowSearch(A, M, 0, n - 1, C)

def RowSearch(A, M, low, high, C):
    while low <= high:
        mid_row = (low + high) // 2

        # Get the column index of the local minimum in the middle row from M
        min_col = M[mid_row]

        # Check if the element at (mid_row, min_col) is a global local minimum
        if (mid_row == 0 or A[mid_row][min_col] < A[mid_row - 1][min_col]) and \
           (mid_row == n - 1 or A[mid_row][min_col] < A[mid_row + 1][min_col]) and \
           (min_col == 0 or A[mid_row][min_col] < A[mid_row][min_col - 1]) and \
           (min_col == C - 1 or A[mid_row][min_col] < A[mid_row][min_col + 1]):
            return (mid_row, min_col)  # Return the index of the global local minimum

        # Move to the part of the array that contains a smaller neighbor
```

```
        if mid_row > 0 and A[mid_row - 1][min_col] < A[
            mid_row][min_col]:
            high = mid_row - 1  # Move to the upper half
        else:
            low = mid_row + 1  # Move to the lower half

    return "No local minimum found"
```

Answer 3.c

Our solution uses **binary search on each column** to efficiently find a local minimum in the 2D array $A[1...n, 1...C]$.

The algorithm goes through each column j (from 1 to C) and applies binary search on the rows of that column to find a local minimum. This reduces the number of rows checked.

For each column, the middle row mid is calculated, and the element $A[\text{mid}][j]$ is compared with the elements above and below it. If $A[\text{mid}][j]$ is smaller than both its neighbors, it is considered a **local minimum**, and the value is returned.

If the element above the middle is smaller, the search continues in the upper half. If the element below is smaller, the search continues in the lower half, reducing the search space.

```

def LocalMin2D():
    for j in range(1, C+1):  # Iterating through columns
        low = 1
        high = n

        while low <= high:
            mid = (low + high) // 2

            if (mid == 1 or A[mid][j] < A[mid - 1][j]) and \
               (mid == n or A[mid][j] < A[mid + 1][j]):
                return A[mid][j]                    # Local
                    minimum found

            if mid > 1 and A[mid - 1][j] < A[mid][j]:
                high = mid - 1                      # Search
                    in the upper half
```

```
            else:
                low = mid + 1                          # Search
                    in the lower half

    return "No local minimum found"
```

Listing 1: Python code for finding local minimum in a 2D array

For each column, binary search on the rows takes $O(\log n)$. Since this is done for C columns, the total time complexity is $O(C \log n)$.

Problem 3-4. Popular Testing

Widget Co. produces many types of electronic widgets. You've decided to get into the widget business as well, so you'd like to reverse engineer their designs. Since your resources are limited, you'd like to focus your energy on the most popular widget types. The good news is that you located a shipment of n widgets that "fell off a truck." The bad news is that the widgets all look the same, so identifying the popular widget types is going to be a challenge. You don't know up front whether all the widgets are different, or whether there are a few prevailing popular types.

We say that a widget type is popular if at least a third of the widgets in the shipment are of that type. You'd like to identify one representative widget from each of the popular types. (There can be anywhere from 0 to 3 popular types in the shipment.) Of course, this would be impossible without some way of comparing widgets. You have at your disposal an equivalence tester, which takes two widgets as input and determines whether they are of the same type.

Your widgets are provided as input $w_1, w_2, \ldots, w_n$ (or as an array $w[1...n]$ if you prefer that notation). The equivalence tester is the function `EQ`, where `EQ`(w_i, w_j) returns `TRUE` if w_i and w_j are of the same type.

(a)

Describe a simple $\Theta(n)$-time iterative algorithm for testing whether a particular candidate widget w_i is popular.

(b)

Describe a simple $O(n^2)$-time iterative algorithm for identifying exactly one representative widget from each of the popular types. Note that your algorithm should return at most 3 widgets.

(c)

Describe an efficient divide-and-conquer algorithm that returns a representative widget from each of the (up to 3) popular widget types. Analyze the running time of your algorithm.

Hint: Can a widget be popular overall without being popular in at least one subproblem?

Answer 4.a

We need to check whether a specific widget w_i is "popular," meaning it appears at least $\frac{n}{3}$ times in a collection of n widgets.

We can form the algorithm by the following steps:

- **Counting Matches:**
  - Start by initializing a counter to zero.
  - Compare the widget w_i with each other widget in the collection using the 'EQ' function. If they are of the same type, increment the counter.
- **Checking If It's Popular:**
  - After going through all the widgets, check if the counter is greater than or equal to $\frac{n}{3}$.
  - If it is, w_i is considered popular. Otherwise, it is not.

Algorithm 4 check_popularity

```
1: procedure CHECK_POPULARITY(w, i, n)
2:     count ← 0                                   ▷ Start by setting the counter to 0
3:     for j = 1 to n do                           ▷ Loop through every widget in the shipment
4:         if EQ(w[i], w[j]) then                  ▷ If w[i] is the same type as w[j]
5:             count ← count + 1                   ▷ Increment the counter
6:         end if
7:     end for
8:     if count ≥ n/3 then                         ▷ Check if the count meets the popularity condition
9:         return TRUE                             ▷ The widget is popular
10:    else
11:        return FALSE                            ▷ The widget is not popular
12:    end if
13: end procedure
```

Time Complexity:

The algorithm compares the widget w_i with all n widgets, so it operates in linear time, which is $O(n)$.

Answer 4.b

Your solution identifies **up to 3 popular widget types** in a shipment, where a type is considered popular if it appears at least $\frac{n}{3}$ times. The algorithm runs in $O(n^2)$ time.

- **Initialize a List**: You create an empty list ‘cd‘ to store the popular widget types.
- **Iterate Over All Widgets**: For each widget w_i, a ‘count‘ is initialized to 0, and you compare it with every other widget w_j using ‘EQ(wi, wj)‘ to count how many widgets are of the same type as w_i.
- **Check for Popularity**: If the count of w_i’s type is $\geq \frac{n}{3}$, you check if it is already in the ‘cd‘ list.
- **Avoid Duplicates**: If w_i is not already in ‘cd‘, it is added to the list.
- **Stop After Finding 3 Types**: Once you find 3 popular types, the search stops, and the list ‘cd‘ is returned.

```
def popularWidget(w, n):
    cd = []
    for i in range(1, n + 1):
        count = 0
        for j in range(1, n + 1):
            if EQ(w[i], w[j]):
                count += 1
        if count >= n / 3:
            exists = False
            for k in cd:
                if EQ(w[i], k):
                    exists = True
                    break
            if not exists:
                cd.append(w[i])
        if len(cd) == 3:
            break
    return cd
```

Listing 2: Algorithm to find popular widget types

- You check if a widget type is popular by comparing each widget with every other one.
- The algorithm ensures no duplicate types are added to the list.
- The time complexity is $O(n^2)$ due to the nested loops.

The algorithm finds up to 3 popular widget types by counting how often each type appears. Once 3 types are identified, the process stops.

Answer 4.c

Divide-and-Conquer Algorithm for Finding Popular Widget Types

Split the list of widgets $w[1...n]$ into two halves: the left half and the right half. Recursively apply the same logic to both the left and right halves to find potential popular widget types in each half.

Combine the results from the two halves. The candidate popular widgets from the left and right halves are merged into a single list.

Count how many times each of the candidate widgets appears in the entire array (both halves).

Keep only those candidates that appear at least $\frac{n}{3}$ times across the entire list.

Return the final list of representative widgets from each of the popular types (up to 3 types).

The recurrence relation for the running time of the algorithm is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Time complexity is:

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

Algorithm 5 DivideConquerWidget

```
1: procedure DivideConquerWidget(w, n)
2:     if n == 1 then
3:         return [w[0]]                          ▷ Base case: Return the single widget
4:     end if
5:     if n == 2 then
6:         if w[0] == w[1] then
7:             return [w[0]]                      ▷ Both widgets are the same type
8:         else
9:             return [w[0], w[1]]                ▷ Both widgets are different
10:        end if
11:    end if
12:    mid ← n//2                                 ▷ Divide the array into two halves
13:    A ← w[0 : mid]                             ▷ First half of the array
14:    B ← w[mid : n]                             ▷ Second half of the array
15:    C_A ← DivideConquerWidget(A, mid)          ▷ Recursively find popular
       candidates from A
16:    C_B ← DivideConquerWidget(B, n − mid)      ▷ Recursively find
       popular candidates from B
17:    C ← C_A + C_B                              ▷ Combine candidates from both halves
18:    finalList ← []                             ▷ Initialize the list of verified popular widgets
19:    for all k ∈ C do
20:        count ← 0
21:        for all x ∈ w do
22:            if x == k then
23:                count ← count + 1
24:            end if
25:        end for
26:        if count ≥ n/3 and k ∉ finalList then
27:            append k to finalList
28:        end if
29:    end for
30:    return finalList                           ▷ Return up to 3 popular widgets
31: end procedure
```

Question 6 - Problem Statement:

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine `Merge` used by merge sort. In this problem, we will prove a lower bound of $2n-1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

First, we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

(a): Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

(b): Using your answer to part (a), show that any decision-tree algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Note: This is a little oh, not a big oh. So you should show that the number of comparisons is very close to $2n$, but with a lower-order term subtracted. You can use the formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Now, we will show a slightly tighter $2n-1$ bound using an entirely different analysis, i.e., not based on a decision tree. This second analysis is actually easier (or at least it involves far less algebra), but it is also very specific to this problem. In contrast, the decision-tree approach generalizes more readily to other problems.

(c): Argue that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

(d): Use your answer to the previous part to show a lower bound of $2n-1$ comparisons for merging two sorted lists. *Hint: provide a specific input for which you can leverage the previous part many times.*

Answer 6.a [Bonus]

The number of ways to pick n numbers from a set of $2n$ numbers is given by the binomial coefficient $C(2n, n)$.

The binomial coefficient $C(2n, n)$ tells us how many different ways we can choose n elements from $2n$ elements. It is calculated as:

$$C(2n, n) = \frac{(2n)!}{n! \cdot n!}$$

So, the total number of ways to divide the $2n$ numbers into two sorted lists of size n is:

$$\frac{(2n)!}{(n!)^2}$$

This represents all the possible ways to split the $2n$ numbers into two groups of n numbers, which can then be sorted.

Answer 6.b [Bonus]

In a decision tree for merging two sorted lists, each node represents a comparison between two elements from the lists. The tree must show all possible ways the elements can be ordered in the merged list. The number of leaves in the tree tells us how many different ways the lists can be merged, and the depth of the tree shows the maximum number of comparisons needed.

We need to prove that the decision tree has at least $2n - o(n)$ leaves, meaning the number of comparisons is nearly $2n$.

From part (a), the number of ways to divide 2n elements into two sorted lists of n elements each is given by the binomial coefficient:

$$C(2n, n) = \frac{(2n)!}{n! \cdot n!}$$

This means there are $C(2n, n)$ leaves in the decision tree, representing all possible orderings of the merged elements.

To estimate $C(2n, n)$, we can use Stirling's approximation for factorials:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Using this, we approximate $C(2n, n)$ as:

$$C(2n, n) \approx \frac{(2n)!}{(n!)^2} \approx \frac{(2n)^2}{n\sqrt{n}} \cdot e^{-n}$$

Since e^{-n} becomes very small for large n, we simplify it to:

$$C(2n, n) \approx \frac{(2n)^2}{n\sqrt{n}}$$

We can further simplify this to:

$$C(2n, n) \approx \frac{2n^2}{n\sqrt{n}} = 2n\sqrt{n}$$

This shows that the decision tree must have at least $2n\sqrt{n}$ leaves, meaning the number of comparisons required is at least $2n\sqrt{n}$.

The lower bound for the number of comparisons needed to merge two sorted lists is $2n - o(n)$, which means the number of comparisons is very close to $2n$ with a small term subtracted. This gives a strong lower bound on how many comparisons are required to merge two sorted lists in the worst case.

Answer 6.c

If two elements are next to each other in the final sorted merged list and come from different input lists, they must be compared. This is because we can't know their order without comparing them.

When merging two sorted lists, the elements within each list are already in order, but we don't know how elements from one list compare to elements from the other. If two consecutive elements in the merged list come from different lists, a comparison is needed to figure out which comes first.

For example, take two lists:

$$L_1 = [1, 3, 5] \quad \text{and} \quad L_2 = [2, 4, 6]$$

To merge them correctly, we need to compare 3 (from L_1) and 2 (from L_2) to see which comes first. Without comparing them, we can't know if 3 or 2 should be placed first in the merged list.

So, whenever consecutive elements in the merged list are from different lists, we need to compare them to get the correct order.

When two consecutive elements in the merged list come from different lists, a comparison is necessary to determine their correct order. This makes such comparisons essential for merging two sorted lists.

Answer 6.d

From part (c), we know that when merging two sorted lists with n elements each, there are $n - 1$ consecutive pairs of elements from different lists that need to be compared.

To merge two sorted lists correctly, we need to compare consecutive elements from different lists to determine their proper order in the merged list. Since each list has n elements, there are $n - 1$ consecutive pairs that must be compared.

Each of these comparisons takes one operation. Therefore, at least $n - 1$ comparisons are required for the consecutive pairs, plus another n comparisons to fully merge all the elements.

This gives a total of $2n - 1$ comparisons to merge the two lists.

The minimum number of comparisons required to merge two sorted lists with n elements is $2n - 1$. This is because $n - 1$ comparisons are needed for consecutive elements from different lists, and n more are needed to complete the merge. Hence, the lower bound for merging the two lists is $2n - 1$ comparisons.