Weekly Assignment 2: Solution
Total: 100

CS 2500: Algorithms

Due Date: September 10, 2024 at 11.59 PM

Solution

1. We want to prove the following formula for the sum of a finite number of terms of a geometric progression:

$$S_n = \sum_{j=0}^{n} ar^j = a + ar + ar^2 + \cdots + ar^n = \frac{ar^{n+1} - a}{r - 1}, \quad \text{when } r \neq 1$$

where n is a nonnegative integer, a is the initial term, and r is the common ratio.

We will use mathematical induction on n.

Base Case: $n = 0$

For $n = 0$, the sum is:

$$S_0 = a$$

The formula for the sum also gives:

$$S_0 = \frac{ar^{0+1} - a}{r - 1} = \frac{ar - a}{r - 1} = a$$

Thus, the base case holds.

Inductive Hypothesis: Assume the formula holds for some arbitrary nonnegative integer $n = k$. That is, assume:

$$S_k = \sum_{j=0}^{k} ar^j = \frac{ar^{k+1} - a}{r - 1}$$

Inductive Step: We need to show that the formula also holds for $n = k + 1$. That is, we need to prove:

$$S_{k+1} = \sum_{j=0}^{k+1} ar^j = \frac{ar^{(k+1)+1} - a}{r - 1} = \frac{ar^{k+2} - a}{r - 1}$$

Start with the sum S_{k+1}:

$$S_{k+1} = S_k + ar^{k+1}$$

By the inductive hypothesis, we substitute S_k:

$$S_{k+1} = \frac{ar^{k+1} - a}{r-1} + ar^{k+1}$$

Factor out ar^{k+1} from the two terms:

$$S_{k+1} = ar^{k+1}\left(\frac{1}{r-1} + 1\right) - \frac{a}{r-1}$$

Simplify the expression inside the parentheses:

$$\frac{1}{r-1} + 1 = \frac{1 + (r-1)}{r-1} = \frac{r}{r-1}$$

Thus, the sum becomes:

$$S_{k+1} = ar^{k+1} \cdot \frac{r}{r-1} - \frac{a}{r-1}$$

Now factor the result:

$$S_{k+1} = \frac{ar^{k+2} - a}{r-1}$$

This matches the desired formula for $n = k + 1$.

Conclusion: By the principle of mathematical induction, the formula holds for all nonnegative integers n.

2. We are given the harmonic numbers H_j, which are defined as:

$$H_j = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{j}$$

We are tasked with proving the inequality:

$$H_{2^n} \geq 1 + \frac{n}{2}$$

for all nonnegative integers n, using mathematical induction.

Base Case: $n = 0$

For $n = 0$, we have:

$$H_{2^0} = H_1 = 1$$

The inequality becomes:

$$H_1 \geq 1 + \frac{0}{2} = 1$$

Since $H_1 = 1$, the base case holds.

Inductive Hypothesis: Assume that the inequality holds for some arbitrary nonnegative integer $n = k$. That is, assume:

$$H_{2^k} \geq 1 + \frac{k}{2}$$

Inductive Step: We need to show that the inequality holds for $n = k+1$. That is, we need to prove:

$$H_{2^{k+1}} \geq 1 + \frac{k+1}{2}$$

Step 1: Express $H_{2^{k+1}}$ in terms of H_{2^k}

We can express $H_{2^{k+1}}$ as the sum of the harmonic number H_{2^k} and the sum of the terms from $2^k + 1$ to 2^{k+1}. This is because the harmonic number $H_{2^{k+1}}$ includes all the terms up to 2^{k+1}, and we can split this into the sum up to 2^k (which is H_{2^k}) and the remaining terms from $2^k + 1$ to 2^{k+1}:

$$H_{2^{k+1}} = \sum_{i=1}^{2^{k+1}} \frac{1}{i} = \sum_{i=1}^{2^k} \frac{1}{i} + \sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i}.$$

This simplifies to:

$$H_{2^{k+1}} = H_{2^k} + \sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i}.$$

Step 2: Estimate the sum $\sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i}$

We now estimate the sum $\sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i}$. Notice that for $i \geq 2^k + 1$, each term in the sum is at least $\frac{1}{2^{k+1}}$. There are 2^k terms in this sum, because the range from $2^k + 1$ to 2^{k+1} contains exactly 2^k numbers. Therefore:

$$\sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i} \geq 2^k \cdot \frac{1}{2^{k+1}} = \frac{1}{2}.$$

Thus, we can say that:

$$H_{2^{k+1}} = H_{2^k} + \sum_{i=2^k+1}^{2^{k+1}} \frac{1}{i} \geq H_{2^k} + \frac{1}{2}.$$

Step 3: Apply the Inductive Hypothesis

By the inductive hypothesis, we know that:

$$H_{2^k} \geq 1 + \frac{k}{2}.$$

Substituting this into the inequality for $H_{2^{k+1}}$, we get:

$$H_{2^{k+1}} \geq 1 + \frac{k}{2} + \frac{1}{2}.$$

Simplifying the right-hand side:

$$H_{2^{k+1}} \geq 1 + \frac{k+1}{2}.$$

Conclusion: We have shown that if the inequality holds for $n = k$, it also holds for $n = k + 1$. Therefore, by the principle of mathematical induction, the inequality $H_{2^n} \geq 1 + \frac{n}{2}$ holds for all nonnegative integers n.

3. The normal recursive algorithm is shwon in Algorithm 1. The divide-and-conquer algorithm is shown in Algorithm 2.

Algorithm 1: Find Mode Recursively in a List

```
Function ModeRecursive(lst, freq_dict=None, idx=0):
    if freq_dict == None then
        freq_dict ← {}
    if idx == len(lst) then
        max_freq ← -1
        mode_element ← None
        foreach (element, frequency) in freq_dict do
            if frequency > max_freq then
                max_freq ← frequency
                mode_element ← element
        return mode_element
    current_element ← lst[idx]
    if current_element in freq_dict then
        freq_dict[current_element] ← freq_dict[current_element] + 1
    else
        freq_dict[current_element] ← 1
    return ModeRecursive(lst, freq_dict, idx+1)
```

Algorithm 2: Find Mode Recursively in a List

```
Function find_mode(list, start, end):
    if start == end then
        return list[start]
        Base case: only one element, it is the mode.
    mid ← ⌊(start+end)/2⌋ left_mode← find_mode(list, start, mid)
    right_mode ← find_mode(list, mid+1, end)
    left_count ← count(list, start, end, left_mode)
    right_count ← count(list, start, end, right_mode)
    if left_count ≥ right_count then
        return left_mode
    else
        return right_mode

Function count(list, start, end, value):
    count ← 0
    for i ← start to end do
        if list[i] == value then
            count ← count + 1
    return count
```

Generating Valid Binary Strings Using Recursion

To generate valid binary strings of length n, where every substring of length 2 contains at least one "1", we can use a recursive approach. The function will build the string character by character, ensuring that each substring of length 2 satisfies the given condition.

```python
# Python implementation of generate_valid_strings function

def generate_valid_strings(n):
    def backtrack(curr_str):
        # Base case: If the current string is of length n, add it to result
        if len(curr_str) == n:
            result.append(curr_str)
            return

        # Recursive case: Try appending '0' and '1' if valid
        if len(curr_str) < 1 or curr_str[-1] == '1':
            backtrack(curr_str + '0')
        backtrack(curr_str + '1')

    result = []
    backtrack("")
    return result
```

The recursive function `backtrack(curr_str)` constructs a valid binary string by appending either "0" or "1" to the current string. It checks that the last two characters of the string do not violate the rule that every substring of length 2 contains at least one "1". If the current string reaches length n, it is added to the result list. The function starts with an empty string and recursively builds valid strings. For each position, the function tries to append "1" or "0" if it doesn't result in two consecutive "0"s.

Part 2: Analysis

1. **Time Complexity:**
   The time complexity is $O(2^n)$ because the recursion explores all possible binary strings of length n. In each recursive call, the function either appends "1" or "0", and thus explores all possible combinations.

2. **Space Complexity:**
   The space complexity is $O(n)$ due to the recursion stack. Each recursive call appends one character to the current string, so the maximum depth of the stack is n, which corresponds to the length of the strings being generated.

3. **Ensuring Unique Strings:**
   The function ensures that each valid binary string is generated exactly once by appending characters to the current string in a systematic way, ensuring no duplicates are generated. By avoiding appending "0" after another "0", the function naturally prevents invalid strings from being generated.

4. **Handling Edge Cases:**

   (a) $n = 1$**:**
   For $n = 1$, the valid strings are "0" and "1", as there are no substrings of length 2 to check.

   (b) $n = 2$**:**
   For $n = 2$, the valid strings are "01", "10", and "11" The string "00" is invalid because it contains two consecutive "0"s.

   (c) **Large values of** n**, such as** $n = 20$**:**
   As n increases, the recursive function explores more possibilities, but since it avoids generating invalid strings early on, it remains reasonably efficient for large values of n. However, the number of valid strings still grows exponentially.

5. **Performance for Large** n**:**
   As n increases, the number of recursive calls grows exponentially, leading to slower performance. The main bottleneck is the time complexity, which is $O(2^n)$. One way to address this is to use dynamic programming to memoize results for subproblems that have already been solved.

6. **Extension for Substrings of Length 3:**
   If the problem required that every substring of length 3 contains at least one "1", we would need to modify the recursive function to check the last two characters before appending the next one. Specifically, before appending a new character, the function would need to check that appending the character would not result in three consecutive "0"s. This change would slightly increase the complexity of the function, but the overall approach would remain similar.