Design and Analysis of Algorithms- HW2

Sai Nishanth Mettu, Praneeth Kumar Thummalapalli, Mohammad Adnan Jakati

October 2, 2023

1 Solution

We will prove by mathematical induction that for all positive integers N,

$$\sum_{i=1}^{N} F(i) = F(N+2) - 1,$$

where $F(i)$ represents the i-th Fibonacci number.

Base Case: For $N = 1$,

$$\sum_{i=1}^{1} F(i) = F(1) = 1,$$

and

$$F(1+2) - 1 = F(3) - 1 = 2 - 1 = 1.$$

So, the statement is true for $N = 1$.

Inductive Hypothesis: Assume that the statement holds for some positive integer k, i.e.,

$$\sum_{i=1}^{k} F(i) = F(k+2) - 1.$$

Inductive Step: We need to prove that the statement also holds for $N = k + 1$, i.e.,

$$\sum_{i=1}^{k+1} F(i) = F((k+1)+2) - 1.$$

Consider the left side of the equation:

$$\sum_{i=1}^{k+1} F(i) = \left(\sum_{i=1}^{k} F(i)\right) + F(k+1).$$

By our inductive hypothesis, we know that

$$\sum_{i=1}^{k} F(i) = F(k+2) - 1.$$

Therefore, we can rewrite the left side as:

$$F(k+2) + F(k+1) - 1$$

Now, let's simplify this expression:

$$LHS = \sum_{i=1}^{k+1} F(i) = F(k+2) + F(k+1) - 1$$

Now, consider the right side of the equation:

$$F(k+3)-1$$

By the definition of Fibonacci Sequence we know,

$$F(k+3)=F(k+2)+F(k+1)$$

Therefore RHS upon subsitution,

$$RHS=F(k+2)+F(k+1)-1$$

Which is equal to LHS.

Since both sides of the equation are equal, we have successfully proved that if the statement holds for $N=k$, it also holds for $N=k+1$. This completes the induction step.

Conclusion: By mathematical induction, we have shown that the statement is true for $N=1$ and that if it holds for any positive integer k, it also holds for $k+1$. Therefore, by the principle of mathematical induction, the statement is true for all positive integers $N >=0$.

2 Solution

To prove that $g(n)=\Omega(E^n)$, where $E>1$ is a constant, we will use mathematical induction.

Base Case:

First, let's consider the base cases for $n=1$ and $n=2$:

1. $n=1$:
$$g(1)=A$$
2. $n=2$:
$$g(2)=B$$

For the base case, we won't assume any specific inequalities since we don't know the values of A and B. Instead, we'll proceed directly with the inductive step to prove that $g(n)\geq C\cdot E^n$ for some constants C and n_0.

Inductive Hypothesis:

Assume that for some $k\geq 2$, $g(k)\geq C\cdot E^k$ for some constants C and n_0.

Inductive Step:

We need to prove that $g(k+1)\geq C'\cdot E^{k+1}$ for some constants C' and n_0'.

Using the recursive definition of $g(n)$ when $n\geq 3$:

$$g(n)=C\cdot g(n-2)+D\cdot g(n-1)$$

Substituting $k+1$ for n in the equation:

$$g(k+1)=C\cdot g(k-1)+D\cdot g(k)$$

Now, using the inductive hypothesis, we have:

$$g(k)\geq C\cdot E^k$$
$$g(k-1)\geq C\cdot E^{k-1}$$

Multiply the first equation by C and the second equation by D and then add them together:

$$C\cdot g(k)+D\cdot g(k-1)\geq C^2\cdot E^k+C\cdot D\cdot E^{k-1}$$

From the above equation on the RHS, extract E^{k+1}

$$C \cdot g(k) + D \cdot g(k-1) \geq E^{k+1} * (C^2 \div E) + ((C \cdot D) \div E^2)$$

Now, lets assume ∃C',

$$C \cdot g(k) + D \cdot g(k-1) \geq E^{k+1} * C'$$

We know from above that,

$$C \cdot g(k) + D \cdot g(k-1) = g(k+1)$$

Therefore,

$$g(k+1) \geq C' * E^{k+1}, \exists C'$$

We have shown that $g(k+1) \geq C' \cdot E^{k+1}$ for some constants C' and n_0', which completes the inductive step.

Conclusion:
By mathematical induction, we have shown that for all $n \geq n_0$, where n_0 is the maximum of the base case values (1 and 2), $g(n) \geq C \cdot E^n$ for some constants C and E. This means $g(n) = \Omega(E^n)$.

3 Solution

(a) In this scenario, we have an array A of length n containing words from a dictionary of m words, where each word can appear multiple times. We want to calculate the expected number of times the for-loop is carried out to determine if a given word e is in the array A.
Let's define the variables:

n is the length of array A.

m is the number of words in the dictionary.

p is the probability of e being in array A.

k is the expected number of times the for-loop is carried out when e is in A.

First, let's calculate p: Since all words are equally likely to be elements in A and e, the probability of a specific word being e is $1/m$. Since there are n elements in A, the probability of e being in A is n/m.

Therefore, $p = \frac{n}{m}$.

Now, let's calculate k: - If e is present in A, the expected number of iterations would be the position of e in A, which is $n + 1/2$ on average since e is equally likely to be anywhere in the array. It is $\frac{n+1}{2}$ because The average is (n+1)/2 Since there is equal possibility of finding the elem at any position. Let us assume all cases of finding the elements and consider its average case.
No of loops for each possible location: 1 + 2 + 3 + ... + n.

Total no of loops for all n positions considered one at a time :

$$\frac{n * (n+1)}{2}.$$

Average no of loops: $\frac{n*(n+1)}{2*n}$

If e is not present in A, the expected number of iterations would be $n + 1$ because the for-loop iterates n times and then one additional time for the "not found" case.

Now, let's calculate $E(X)$, which represents the expected number of times the for-loop is carried out:

$$E(X) = (1 - p) \cdot (n + 1) + p \cdot k$$

Substitute the values of n, m, p, and k:

$n = 10^4$ (since $n = 10,000 = 10^4$)

$m = 5 \times 10^4$ (since $m = 50,000 = 5 \times 10^4$)

$p = \dfrac{n}{m} = \dfrac{10^4}{5 \times 10^4} = 1/5$

k would be $\dfrac{n+1}{2}$ on average if e is in A, and $n + 1$ if e is not in A.

Now, calculate $E(X)$:

$$E(X) = \left(1 - \frac{1}{5}\right) \cdot \frac{10^4 + 1}{2} + \frac{1}{5} \cdot \frac{10^4}{2}$$

Now, calculate the above expression:

$$E(X) = \left(\frac{5-1}{5}\right) \cdot \frac{10^4 + 1}{2} + \frac{1}{10} \cdot (10^4)$$

$$E(X) = \frac{8}{10} \cdot \frac{10^4 + 1}{2} + 10^3$$

Now, calculate the final value:

$$8000 + 0.8 + 1000 = 9000.8$$

$$E(X) = 9000.8$$

Rounded to the nearest integer, the expected number of times the for-loop is carried out is approximately 9001.

So, for $n = 10,000$ and $m = 50,000$, the expected number of iterations of the for-loop is approximately 9001.

Case (b): When e is in the array A and each element in A is a unique word in a dictionary of m words, where $n \leq m$.

We will use the Law of Total Expectation as suggested:

$$E(X) = P(I \in F) \cdot E(X|I \in F) + P(I \notin F) \cdot E(X|I \notin F)$$

Where: - $E(X)$ is the expected number of iterations. - $P(I \in F)$ is the probability that e is in the array A. - $P(I \notin F)$ is the probability that e is not in the array A. - $E(X|I \in F)$ is the expected number of iterations when e is in A. - $E(X|I \notin F)$ is the expected number of iterations when e is not in A.

In this problem, it is mentioned that $I = (A, e)$ is chosen uniformly at random, which means that there are n possible values of e in A and $m - n$ possible values of

e not in A.

$P(I \in F) = \frac{1}{n}$

$P(I \notin F) = \frac{m-n}{m}$

Now, we can calculate $E(X)$ using these probabilities and the conditional expectations we computed earlier for each case:

$$E(X) = \frac{1}{n} \cdot \frac{n+1}{2} + \left(1 - \frac{n}{m}\right) \cdot (n+1)$$

Now, let's express this in terms of n and m:
- $n = 10^4$ (since $n = 10,000 = 10^4$) - $m = 5 \times 10^4$ (since $m = 50,000 = 5 \times 10^4$)

$$E(X) = \frac{1}{10^4} \cdot \frac{10^4+1}{2} + \frac{4 * 10^4}{5 * 10^4} \cdot (10^4+1)$$

Now, simplify:

$$E(X) = \frac{1}{2} + \frac{4}{5} \cdot (10^4+1)$$

Now, calculate the expression:

$$E(X) = \frac{1}{2} + \frac{8}{10} \cdot (10^4+1)$$

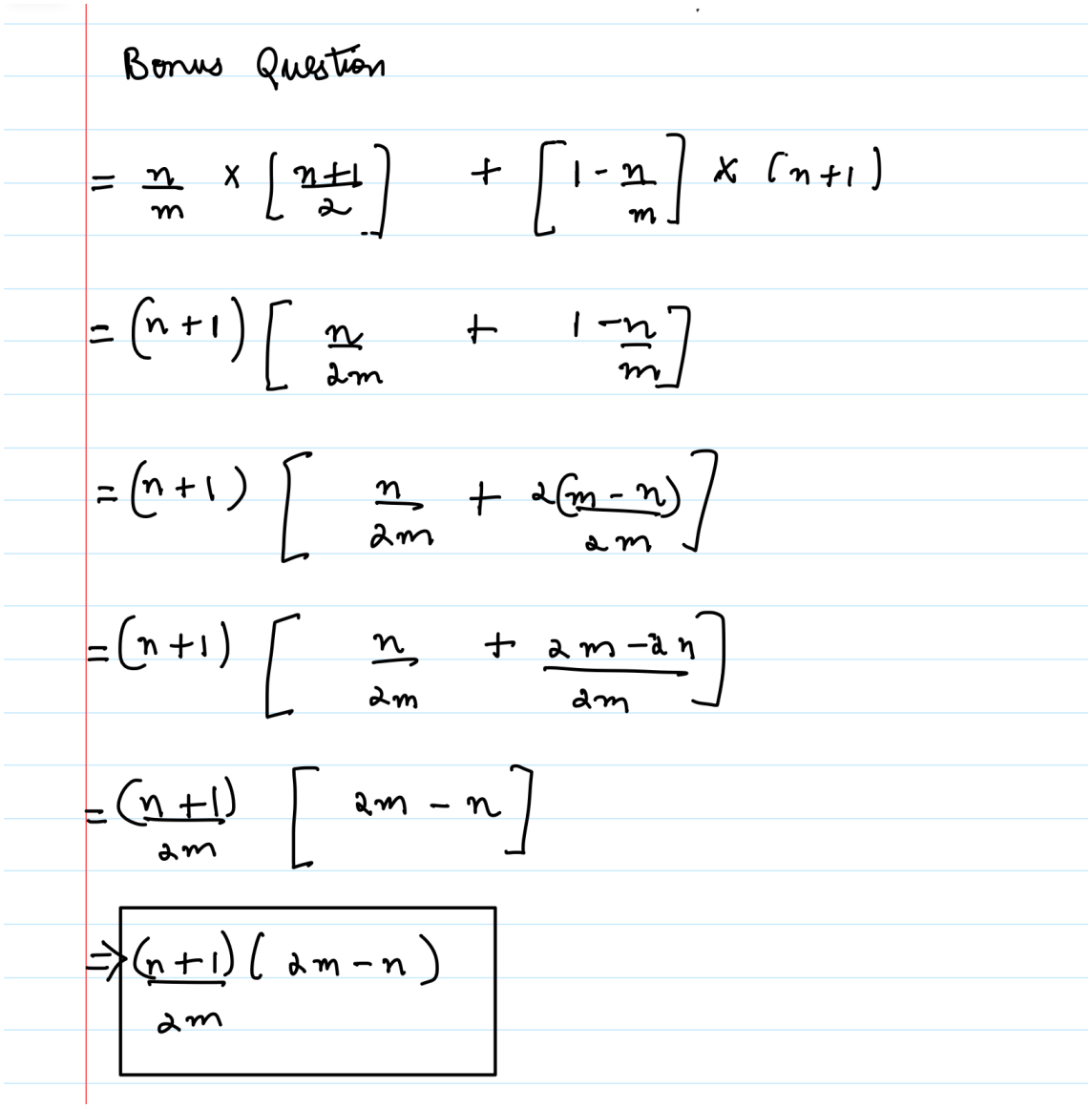
$$E(X) = \frac{1}{2} + 8000 + 0.8$$

Now, add these values:

$$E(X) = 8001.3$$

Rounded to the nearest integer, the expected number of times the for-loop is carried out is approximately $8,001$.

So, for $n = 10,000$ and $m = 50,000$, the expected number of iterations of the for-loop is approximately $8,001$.



4 Solution

To determine the values of k for which insertion sort has an asymptotically better worst-case running time than merge sort for increasingly k-almost sorted arrays, we need to analyze the time complexities of both algorithms for these scenarios.

Insertion Sort:

In the worst-case scenario, insertion sort has a time complexity of $O(n^2)$ for standard unsorted arrays. However, for nearly sorted arrays, including increasingly k-almost sorted arrays, insertion sort performs significantly better.

For an increasingly k-almost sorted array, the number of inversions (pairs of elements out of order) is bounded by a constant multiple of k. Therefore, insertion sort's worst-case time complexity for such an array is $O(nk)$.

Merge Sort:

Merge sort has a consistent time complexity of $O(n \log n)$ regardless of the input data's order.

To determine the values of k for which insertion sort is asymptotically better than merge sort, we need to find the crossover point where $O(nk)$ for insertion sort is better than $O(n \log n)$ for merge sort.

Set $O(nk) < O(n \log n)$ and solve for k:

$$nk < n \log n$$

Divide both sides by n:

$$k < \log n$$

Since k needs to be a positive integer, we can express this as:

$$k \leq \log n$$

So, for increasingly k-almost sorted arrays, insertion sort is asymptotically better than merge sort when:

$$k \leq \log n$$

For k values greater than $\log n$, merge sort will have a better worst-case time complexity for sorting increasingly k-almost sorted arrays.

5 Solution

Algorithm 1 Check if Sets in Arrays A and B are Equal

Input: Two arrays A and B, each of size n containing integers.
Output: `true` if the sets of numbers in A and B are the same, `false` otherwise.
if length(A) $\neq$ length(B) **then**
  return `false` {Arrays have different sizes; sets cannot be equal}
end if
{Sort both arrays using Merge Sort, which has $O(n \log n)$ complexity}
MergeSort(A)
MergeSort(B)
for i from 0 to length(A) $-$ 1 **do**
  if $A[i] \neq B[i]$ **then**
    return `false` {Sets are not equal; found differing elements}
  end if
end for
return `true` {All elements match; sets are equal} =0

The overall worst-case time complexity of the algorithm is $O(n \log n)$, primarily determined by the sorting step, specifically the Merge Sort algorithm. Here's the breakdown:

1. **Sorting Step (Merge Sort):** Merge Sort is a well-known comparison-based sorting algorithm with a worst-case time complexity of $O(n \log n)$. In the algorithm, we apply Merge Sort independently to both arrays A and B, each of size n. Therefore, the sorting step takes $2 \cdot O(n \log n)$ time, which simplifies to $O(n \log n)$.

Algorithm 2 Merge Sort

Input: An array arr.
Output: The sorted array arr.
if length(arr) ≤ 1 **then**
  return arr {Array is already sorted or empty}
end if
{Split the array into two halves}
$middle = \frac{\text{length}(arr)}{2}$
$left = arr$[0 to middle $- 1$]
$right = arr$[middle to end]
{Recursively sort both halves}
$left =$ **MergeSort**($left$)
$right =$ **MergeSort**($right$)
return **Merge**($left$, $right$) =0

Algorithm 3 Merge

Input: Two sorted arrays $left$ and $right$.
Output: The merged sorted array.
$result = []$
while left is not empty and right is not empty **do**
  if left[0] $\leq$ right[0] **then**
    $result$.append(left[0])
    left = left[1 :]
  else
    $result$.append(right[0])
    right = right[1 :]
  end if
end while
{Append any remaining elements (if any)}
$result$.extend(left)
$result$.extend(right)
return $result$ =0

2. **Merging Step:** Within the Merge Sort algorithm, the merging step combines two sorted arrays into a single sorted array. This merging step is linear and has a time complexity of $O(n)$. Since we perform this merging step twice (once for each array), the total time complexity for merging both arrays is $2 \cdot O(n)$, which simplifies to $O(n)$.

3. **Comparison Step:** After sorting and merging both arrays, we iterate through both sorted arrays once in a loop to compare the elements. The comparison step involves examining each element exactly once and has a time complexity of $O(n)$.

While the merging and comparison steps are linear in nature, they do not significantly impact the overall time complexity because the sorting step (Merge Sort) dominates in terms of time complexity. As a result, the overall worst-case time complexity of the entire algorithm is $O(n \log n)$ due to the dominant sorting step.

6 Solution

Algorithm 4 findSmallestNonOverlapElement(A, B)

```
procedure FindSmallestNonOverlapElement(A, B)
    m ← length(A)
    n ← length(B)
    if m > n then
        swap(A, B)
        swap(m, n)
    end if
    left ← 0
    right ← m − 1
    while left ≤ right do
        mid ← (left+right)/2
        if A[mid] not in B then
            if mid = 0 or A[mid] < B[0] then
                return A[mid]
            else
                right ← mid − 1
            end if
        else
            left ← mid + 1
        end if
    end while
    return B[0]
end procedure
```

Algorithm: findSmallestNonOverlapElement

The *findSmallestNonOverlapElement* algorithm is designed to locate the smallest element in array A that either does not exist in array B or is smaller than the elements in array B.

Algorithm 5 binarySearch(arr, target)

```
0: procedure BinarySearch(arr, target)
0:     left ← 0
0:     right ← length(arr) - 1
0:     while left ≤ right do
0:         mid ← (left+right)/2
0:         if arr[mid] = target then
0:             return True
0:         else if arr[mid] < target then
0:             left ← mid + 1
0:         else
0:             right ← mid − 1
0:         end if
0:     end while
0:     return False
0: end procedure=0
```

The algorithm begins by identifying the smaller array between A and B and assigns it to A to optimize the search process.

It employs binary search to efficiently locate the smallest qualifying element in array A.

Algorithm: binarySearch

The *binarySearch* algorithm serves as a helper function, enabling binary search within an array to determine the presence of a target element.

Example Usage

An illustrative example showcases how to apply the *findSmallestNonOverlapElement* function with sample arrays A and B, ultimately displaying the result.

This algorithm capitalizes on binary search and the sorted order of the arrays, ensuring efficient identification of the desired element with a time complexity of $O(\log n)$.

7 AUTHORS

Net ID: sm11326
NYU ID: N19867031
Student Name: Sai Nishanth Mettu.

Net ID: pt2427
NYU ID: N11088414
Student Name: Praneeth Kumar Thummalapalli.

NetID: mj3184
NYU ID: N19881140
Student Name: Mohammed Adnan Jakati,