CS 624 - ANALYSIS OF ALGORITHMS

Assignment 4

Aravind Haridas – 02071139

1. Problem 16.1-3 on p422.

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities.

(a) Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work.

(b) Do the same for the approach of always selecting the compatible activity that overlaps the fewest other remaining activities.

(c) Do the same for the approach of always selecting the compatible remaining activity with the earliest start time.

To illustrate why certain greedy strategies do not work for the activity-selection problem, let's consider the following examples:

(a) Selecting the activity of the least duration:

Scenario: Activity A: Start at 8:00 AM, Finish at 12:00 PM (4 hours)

Activity B: Start at 9:00 AM, Finish at 10:00 AM (1 hour)

Activity C: Start at 11:00 AM, Finish at 5:00 PM (6 hours)

Greedy Strategy Outcome: Choosing Activity B because it has the shortest duration prevents the selection of Activity A and Activity C, which overlap with B. However, choosing A and then C covers more time throughout the day even though each individual activity is longer.

(b) Selecting the compatible activity that overlaps the fewest other activities:

Scenario: Activity D: Start at 7:00 AM, Finish at 9:00 AM

Activity E: Start at 8:00 AM, Finish at 11:00 AM

Activity F: Start at 9:00 AM, Finish at 12:00 PM

Activity G: Start at 10:00 AM, Finish at 1:00 PM

Greedy Strategy Outcome: If we choose Activity D, because it overlaps only with E, we are left unable to select E, F, or G which overlap with each other to a greater extent but form a longer chain of activities. Selecting E, F, and G sequentially, each overlapping the fewest others sequentially, would allow for three activities rather than just one.

(c) Selecting the compatible activity with the earliest start time:

Scenario: Activity H: Start at 6:00 AM, Finish at 10:00 AM

Activity I: Start at 7:00 AM, Finish at 8:00 AM

Activity J: Start at 8:00 AM, Finish at 12:00 PM

Greedy Strategy Outcome:

Selecting Activity H because it starts the earliest allows only one activity in the schedule, whereas selecting Activity I and then Activity J maximizes the number of activities to two, even though H starts earlier than I and J.

CS 624 - ANALYSIS OF ALGORITHMS

2. Problem 16.2-5 on p428. Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all the given points. Argue that your algorithm is correct.

An efficient algorithm for determining the smallest set of unit-length closed intervals that contains all given points on the real line. This problem is typically solved using a greedy approach that leverages sorting.

Algorithm:

Sort the Points: Begin by sorting the set of points $\{x_1, x_2, \ldots, x_n\}$ in non-decreasing order based on their value. Let's denote the sorted points by $\{y_1,y_2,\ldots,y_n\}$ where $\{y_1 \leq y_2 \leq y_3, \ldots, \leq y_n\}$.

Initialize Intervals: Start with an empty list of intervals.

Cover Points with Intervals:

- Initialize $i = 1$.

While $i \leq n$:

Let $a = y_i$ be the starting point of the current interval.

Set the interval to start at a and end at $a+1$ (since the interval length is 1).

Add this interval to the list.

Move i to the first point that is not covered by the current interval $[a,a+1]$.

Repeat until all points are covered pseudo code is here :

```
function findUnitIntervals(points):
    sort points
    intervals = []
    i = 0
    while i < points.length:
        start = points[i]
        end = start + 1
        intervals.append([start, end])
        i = i + 1
        while i < points.length and points[i] <= end:
            i = i + 1
    return intervals
```

Correctness Argument:

Greedy Choice: By always starting the interval from the current point y_i and extending it by 1, we ensure maximum coverage without unnecessary extension beyond y_i+1.

Optimality: Each interval covers as many points as possible from the leftmost uncovered point, minimizing the total number of intervals. Deviating from this would imply suboptimal coverage, contradicting our strategy.

Efficiency:

Sorting: Sorting the points takes $O(n\log n)$ time.

Interval Placement: Placing intervals with a linear scan is $O(n)$.

Overall Time Complexity: $O(n\log n)$.

This approach guarantees that we find the minimal number of unit-length intervals needed to cover all points, by making the optimal greedy choice at each step and covering the maximum number of points with each interval without overlap.

CS 624 - ANALYSIS OF ALGORITHMS

Problem 16.3-3 on p436.

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a : 1 b : 1 c : 2 d : 3 e : 5 f : 8 g : 13 h : 21

Can you generalize your answer to find the optimal code when the frequencies are the first *n* Fibonacci numbers?

Solution :The two smallest frequencies are combined at each step, we get,

let's create the Huffman coding binary tree:

a : 1 b : 1 c : 2 d : 3 e : 5 f : 8 g : 13 h : 21

now we need to combine the two smallest frequencies together and then next two frequencies and go on until the last frequencies, we get below

Combine 'a' and 'b' (smallest frequencies): 'ab': 2

Combine 'ab' with 'c': 'abc': 4

Combine 'abc' with 'd': 'abcd': 7

Combine 'abcd' with 'e': 'abcde': 12

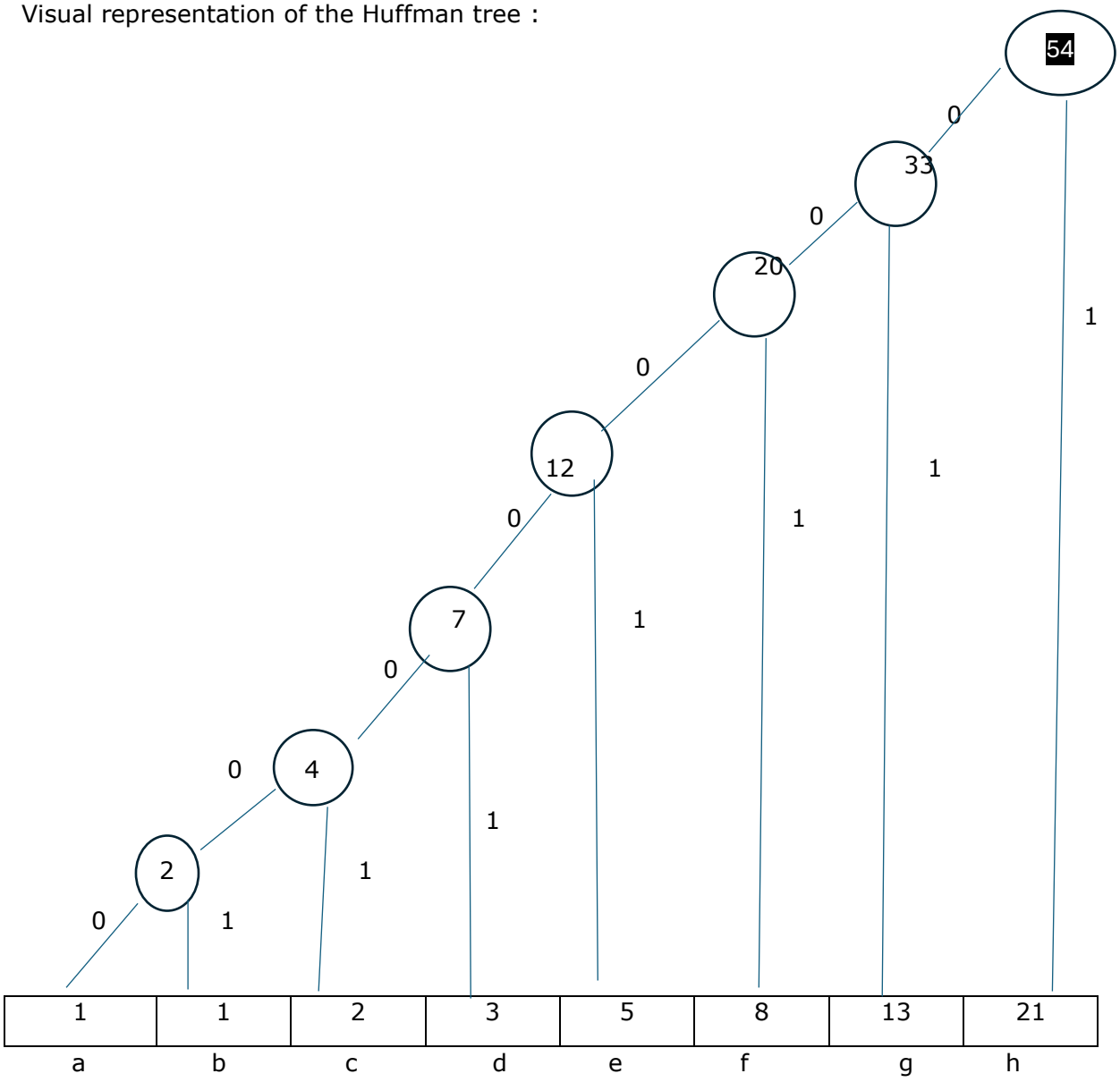
Combine 'abcde' with 'f': 'abcdef': 20

Combine 'abcdef' with 'g': 'abcdefg': 33

Combine 'abcdefg' with 'h': 'abcdefgh': 54 (This is the total and final combination)

CS 624 - ANALYSIS OF ALGORITHMS

Visual representation of the Huffman tree :



CS 624 - ANALYSIS OF ALGORITHMS

CHAR	FREQUEINCIES	CODE
a	1	0000000
b	1	0000001
c	2	000001
d	3	00001
e	5	0001
f	8	001
g	13	01
h	21	1

Generalization for first n Fibonacci numbers are :

The optimal Huffman code for a set of frequencies based on the first n Fibonacci numbers will follow a similar pattern, where each character is added progressively following the Fibonacci sequence. Each character gets combined in pairs, reflecting the Fibonacci property where the next number is the sum of the two preceding ones.

In terms of generalization, each node in the Huffman tree from the first n Fibonacci numbers will be formed by combining the frequencies of the two preceding nodes, exactly reflecting the Fibonacci sequence formation. Thus, the tree will grow in a manner where each level combines nodes in a way that mirrors the progression of the Fibonacci sequence.

CS 624 - ANALYSIS OF ALGORITHMS

4. Problem 17.2-3 (p459) Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n Increment and Reset operations takes time $O(n)$ on an initially zero counter. (Hint: Keep a pointer to the high-order 1.)

To implement a counter using an array of bits and ensure that both increment and reset operations on the counter are efficient, we can use the following strategy. This approach utilizes a pointer to track the high-order bit that is set to 1, facilitating quick resets and efficient increments.

Counter Implementation:

Data Structure:

counter: An array of bits representing the binary number.

high: A pointer to the highest order bit set to 1 in the counter.

Initialization: Initialize all bits in the counter array to 0.

Set high to -1 indicating no bits are set.

Increment Operation (Increment): Start from the lowest order bit.

Flip each bit. If a bit is flipped from 0 to 1, update high to this bit's index and stop.

If a bit is flipped from 1 to 0, continue to the next higher bit.

If high is the last bit and it flips to 0 (overflow), reset high to -1 (optional for bounded counters).

Reset Operation (Reset):Directly use the high pointer to identify the highest order bit that is 1.

Set all bits from the 0th index to the high index to 0.

Reset the high pointer to -1.

Efficiency Analysis: Increment: Each increment operation, in the worst case, might flip every bit until it finds a 0, which becomes 1. However, each bit flip happens only once per transition from 0 to 1 across multiple increments until a reset.

Reset: Reset operation is linear with respect to the highest order 1-bit because it directly accesses the bits up to high and resets them in a single pass.

Amortized Analysis: Incrementing operations may involve flipping multiple bits, but each specific bit changes from 0 to 1 once between resets. Thus, the total cost of flipping bits can be distributed across all the increment operations that occurred before a reset.

Resetting directly uses the high pointer to clear bits, taking time proportional to the number of 1-bits, which is at most the number of increments since the last reset.

Conclusion:

The overall cost for n operations i.e increments and resets is $O(n)$, since each bit operation (flip or set to 0) is effectively counted once per operation cycle from 0 to full 1s and back to 0. This method ensures that both operations are managed in linear time with respect to the number of operations, satisfying the requirement for an $O(n)$ time complexity for a sequence of operations.

CS 624 - ANALYSIS OF ALGORITHMS

5. Problem 17.3.6 (p463) Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each Enqueue and each Dequeue operation is *O(1).*

To implement a queue using two stacks such that each operation (Enqueue and Dequeue) has an amortized cost of *O(1),* we utilize the properties of stacks—specifically, their LIFO (Last In, First Out) nature. By cleverly distributing the work between two stacks, we can ensure that elements are enqueued and dequeued in a FIFO (First In, First Out) manner typical of queues.

Stack1: This stack is used for enqueue operations.

Stack2: This stack is used for dequeue operations.

Operations: Enqueue(x):

Push element x onto Stack1.

Amortized cost: O(1) since a single operation is involved (pushing onto Stack1).

Dequeue(): If Stack2 is empty, transfer all elements from Stack1 to Stack2 by repeatedly popping from Stack1 and pushing onto Stack2 until Stack1 is empty. This reverses the order of the elements, making the bottom element of Stack1 the top element of Stack2.

Pop the top element from Stack2 and return it.

If Stack2 is not empty, simply pop and return the top element from Stack2.

Amortized cost: *O(1).* Although moving all elements from Stack1 to Stack2 seems costly, each element is moved exactly twice (once to Stack1 and once to Stack2) between two Dequeue operations. This cost is spread over all the operations, making the amortized cost constant.

Explanation of Amortized Cost:

The key to achieving amortized *O(1)* cost lies in the fact that each element is only transferred once from Stack1 to Stack2 between two Dequeue operations. Here's a breakdown:

Each Enqueue operation involves a single push to Stack1, which is *O(1).*

Each Dequeue operation either involves a single pop from Stack2, which is *O(1),* or in cases where Stack2 is empty, several pops from Stack1 and pushes to Stack2. However, each element is only transferred once after being enqueued, and the cost of these operations can be amortized over multiple Dequeue operations.

Conclusion: By using two stacks, we efficiently implement a queue where the FIFO order is maintained through the stack reversal process. The amortized analysis shows that despite the occasional higher cost for certain Dequeue operations (when **Stack2** is empty and elements need to be transferred), the average cost per operation, when spread out over a sequence of operations, remains constant, i.e., O(1). This setup effectively leverages the strengths of stacks to mimic queue operations while maintaining optimal efficiency in an amortized sense.