Homework 04 — Solutions

CS 624, 2024 Spring

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.

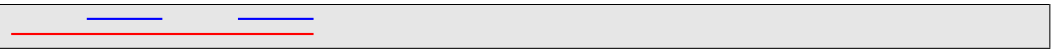
   (a) Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work.

   (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.

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 of the given points. Argue that your algorithm is correct.

   Let $X = \{x_1, x_2, \ldots, x_n\}$. A solution can be described as a set of m starting points for unit intervals—that is, as $\{a_1, \ldots, a_m\}$ such that $X \subseteq \bigcup_{i=1}^{m} [a_i, a_i + 1]$.

   Algorithm: Always take the least element ($x_{\min}$) as the starting point of an interval and recur on $X - [x_{\min}, x_{\min} + 1]$. We can do this efficiently as follows: Start by sorting the input array X. Start at $i = 1$. While $i < n$, take $a = X[i]$ as an interval starting point, and repeatedly increment i until either we hit the end of the array or $X[i] > a + 1$. The total cost is $O(n \log n + n) = O(n \log n)$, or just $O(n)$ if we are given the input array already sorted.

   Optimal substructure: Suppose that A is an optimal solution for X, and suppose that $a_k \in A$. Then $A = A_L \cup \{a_k\} \cup A_R$, where A_L is an optimal solution for $\{x \mid x \in X, x < a_k\}$ and A_R is an optimal solution for $\{x \mid x \in X, x > a_k + 1\}$.

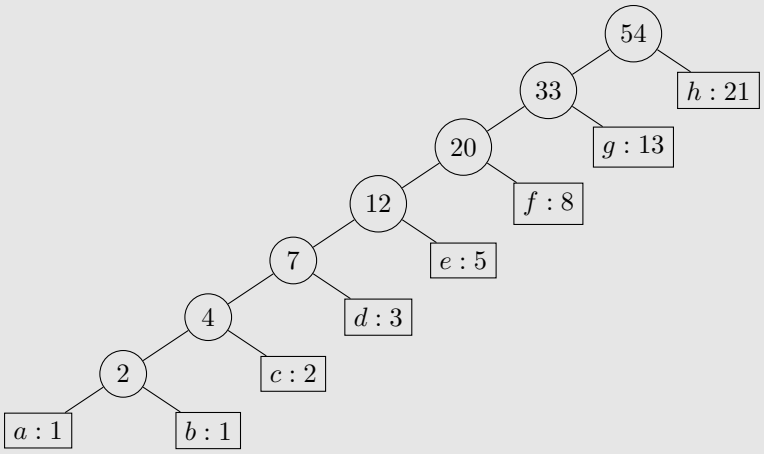
   Greedy choice property: If $x_1 = \min(X)$, then there is an optimal solution for X that contains x_1 as the starting point of an interval. *Proof:* Suppose that A is an optimal solution and $x \notin A$; then x must be covered by some interval starting at $a \in A$, and in fact $a < x$. Then the part of a's interval before x is useless (since x is the minimum element of X), and we could move it over to start at x. That interval will cover at least as much of X as before (possibly more), so the solution is at least as good as the A, which was assumed to be optimal, so the adjusted solution is also optimal.

3. 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?

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

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



(for some labeling of edges with 0 and 1)

Generalization: When we run the pairing loop, after the first iteration, the smallest two nodes will always be the paired node just created and the leaf with the next Fibonacci number. In fact:

$$\sum_{k=1}^{n} f_k = f_{n+2} - 1$$

and since $f_{n+1} \leq \sum_{k=1}^{n} f_k < f_{n+2}$, we will always pair the accumulated node and the next leaf node (f_{n+1}) together.

Theorem: $f_{n+2} = 1 + \sum_{k=1}^{n} f_k$
Proof: By induction on n.

Base case ($n = 0$):
Goal: $f_2 = 1 + \sum_{k=1}^{0} f_k = 1 + 0 = 1$.
Trivial, by definition of f_2. Done.

Inductive case:
IH: $f_{n+2} = 1 + \sum_{k=1}^{n} f_k$.
Goal: $f_{(n+1)+2} = 1 + \sum_{k=1}^{n+1} f_k$.

$$\begin{aligned} f_{(n+1)+2} &= f_{n+2} + f_{n+1} && \text{by Fibonacci recurrence} \\ &= 1 + \sum_{k=1}^{n} f_k + f_{n+1} && \text{by IH} \\ &= 1 + \sum_{k=1}^{n+1} f_k && \text{absorb term into sum} \end{aligned}$$

Done.

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.)

Add a new field called *max*, that stores the index of the high-order 1. We'll use 0-based indexing for this, so *max* is initially -1 (because the counter contains no 1 bits). We'll assume that it actually costs \$1 to check and potentially update *max*.

We'll update the invariant as follows: there is \$1 stored for every 1 bit in the counter, *plus there is \$max stored reserved for resets.*

The actual cost of INCREMENT is now \$1 per bit flipped *plus \$1 to update max*. We'll charge \$4: \$2 is spent as before, an extra \$1 to check and potentially update *max*, and \$1 if *max* increases and we need to increase the amount reserved for resets. (If *max* does not increase, we just "throw away" the last dollar instead.)

The actual cost of RESET is $\$(1 + max)$: we must zero out *max* bits, and we must also reset *max* to -1. We charge \$1 and we use the stored $\$max$ to perform the reset. (And we "throw away" the excess money saved in the data structure.)

To summarize: the amortized cost of INCREMENT is \$4 now, and the amortized cost of RESET is \$1.

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 $O(1)$.

A queue contains two stacks: a "write stack" and a "read stack".

ENQUEUE(q, *elem*) just pushes to the "write stack":

Algorithm 1 ENQUEUE(q, *elem*)

```
1: PUSH(q.writestack, elem)
```

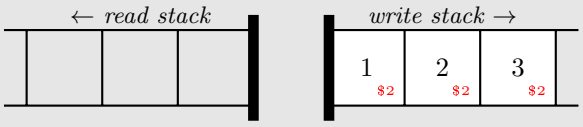
DEQUEUE(q) does the following: If the "read stack" is empty, then it first populates the read stack with the reverse of the contents of the write stack. Then it pops from the read stack.

Algorithm 2 DEQUEUE(q)

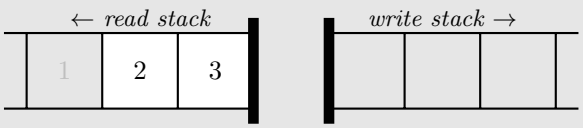
```
1: if q.readstack is empty then
2:    while q.writestack is not empty do
3:       PUSH(q.readstack, POP(p.writestack))
4:    end while
5: end if
6: return POP(q.readstack)
```

The amortized cost of ENQUEUE is \$3, and the amortized cost of DEQUEUE is \$1. The actual cost of an ENQUEUE is one PUSH (\$1), so the queue saves \$2 for each element to pay for the POP and PUSH when the write stack is reversed onto the read stack.

For example, here is the state after enqueuing 1,2, and then 3, into an empty queue:



and after dequeuing one element (1):



and after enqueuing 4 and then 5: