University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
P. Schneider

Algorithms and Data Structures
Winter Term 2019/2020
Sample Solution Exercise Sheet 11

Remark: This is a repetition exercise with a selection of previous topics based on your vote. Since next week will be the final lesson, there is no need to submit this exercise for feedback.

Exercise 1: Counting Bit Flips of a Binary Counter

Consider a counter represented as bit string. We increment (add 1 to) the counter n times. Show that the *amortized number of bit flips per increment operation* is $\mathcal{O}(1)$. You may assume that your counter starts with 0 and has at least $\log_2 n$ bits.

(a) Analyze the number of bit flips using the *aggregate method.* That is, count the total number of bit flips and divide it by the number of operations.

(b) Analyze the number of bit flips using the *accounting method.* Specifically, show that by paying a constant amount of coins to an account per operation, and subtracting the true cost per operation from the account, the account still stays positive all the time.

Sample Solution

First we make the following observation: Let b_i be the i^{th} bit (where $i = 0$ is the least significant bit). Then b_i gets flipped every $(2^i)^{th}$ increment operation. Assume that $n = 2^m$ is a power of two, otherwise we "round" n to the next power of two. Therefore, when we count to n the bit with number b_i gets flipped $n/2^i$ times (the higher the significance of the bit the less frequent it gets flipped).

(a) Now for the total cost C_n of counting to n we have

$$C_n = \sum_{i=0}^{m} \text{"\#flips of } b_i\text{"} = \sum_{i=0}^{m} \frac{2^m}{2^i} = \sum_{i=0}^{m} 2^i = 2^m - 1$$

Then the amortized cost is the total cost C_n divided by the number of operations. Since we "rounded" n to the next power of two, we must assume that originally, we had only $\frac{n}{2}$ operations. Then we have the following amortized cost per operation

$$\frac{C_n}{n/2} = \frac{2^m - 1}{n/2} \leq \frac{2^m}{n/2} = \frac{2 \cdot 2^m}{n} = \frac{2 \cdot 2^m}{2^m} = 2 \in \mathcal{O}(1).$$

(b) We make two more observations.

(i) Whenever we increment, there is at most one flip from zero to one (shorthand $0 \to 1$) (but there may be many flips $1 \to 0$).

(ii) For each bit-flip from one to zero (shorthand $1 \to 0$) there must have been a bit flip $0 \to 1$ before, as the counter starts with 0.

Note that the first observation is due to the fact that when we are incrementing (starting from the least significant bit b_0) we go from b_i to b_{i+1} only if b_i is flipped to 0 and a 1 is carried over to b_{i+1}. When eventually, for some i the bit b_{i+1} is 0, then b_{i+1} is flipped from 0 to 1. But then we are also done since no carryover occurs anymore.

The true cost per bit flip is always one. But we assign amortized costs in such a way that the flips $0 \to 1$ "pay more" to the account, so that there is always enough to pay for the subsequent flips $1 \to 0$. Specifically, for each flip $0 \to 1$ *we pay 2 coins*, whereas we use one coin to pay for the flip $0 \to 1$ itself and we pay the other to the account. For each flip $1 \to 0$ we *do not pay anything* but instead *subtract a coin* from the account to pay for the flip. We have to argue that there is always enough balance on the account to pay for all the flips $1 \to 0$.

On the one hand, due to observation (i), no increment operation costs more than 2 coins, since there is at most one flip $0 \to 1$ per operation that has a cost (of 2) assigned to it. Due to observation (ii) there is a coin available to pay for each flip $1 \to 0$ since the same bit must have been flipped $0 \to 1$ before, where paid a coin in the account to save for the flip back to 0. Essentially each flip $0 \to 1$ of a given bit pays for the subsequent flip $1 \to 0$.

Exercise 2: More Hashing

Let $h(s, j) := h_1(s) + j \cdot h_2(s) \bmod 13$ and let $h_1(x) = 2x + 3 \bmod 13$ and $h_2(x) = 2 + (x \bmod 12)$.

(a) Give an infinite key set (a subset of $\mathbb{N}$) that are mapped to the same table entry (for $j = 0$).

(b) Insert the keys **3,11,23,5,24,8,21,10** into the hash table of size $m = 11$ using the double hashing probing technique for collision resolution. The hash table below should show the final state.

0	1	2	3	4	5	6	7	8	9	10	11	12

Sample Solution

(a) All positive multiples of 13.

(b) Resulting table:

5	24			21		8		10	3	23		11
0	1	2	3	4	5	6	7	8	9	10	11	12

Exercise 3: Frequent Numbers

You are given an Array $A[0 \ldots n-1]$ of n integers and the goal is to determine frequent numbers which occur at least $n/3$ times in A. There can be at most three such numbers, if any exist at all.

(a) Give an algorithm with runtime $\mathcal{O}(n \log n)$ based on the divide and conquer principle that outputs the frequent numbers (if any exist).

(b) Argue why your algorithm is correct, give a recurrence relation for the runtime and use it to prove the runtime.

Sample Solution

(a) **Algorithm 1** `Frequent-Numbers`(A, ℓ, r)

```
if ℓ = r then return {A[ℓ]}                                      ▷ base case
C ← Frequent-Numbers(A, ℓ, ⌈(ℓ+r)/2⌉−1)     ▷ candidates are the frequent numbers of left ...
C ← C∪ Frequent-Numbers(A, ⌈(ℓ+r)/2⌉, r)                      ▷ ... and right sub-array
for c ∈ C do
    count the number of occurrences of c in A[ℓ...r]
    if c occurs less than (r−ℓ)/3 times in A[ℓ...r] then C ← C \ {c}
return C
```

A call of `Frequent-Numbers`$(A, 0, n-1)$ solves the problem.

(b) We split the given array A into two parts of (almost) equal size. A frequent number of that array must be a frequent number in the left half or the right half (or both). Thus it suffices to first find the frequent numbers of the left sub-array and then the ones of the right (if they exist). We do this by applying the procedure recursively and then check whether some of these are also frequent in A, by simply counting the number of occurrences of the candidates.

In each iteration we make recursive calls on two sub-arrays of half the size of A. Afterwards we count elements in the current array which takes at most $\mathcal{O}(n)$ read operations if n is the current size of the array (note that the set C has size at most 6). We obtain the recurrence relation $T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + 6n$ with base case $T(1) = \mathcal{O}(1)$, which solves to $T(n) \in \mathcal{O}(n \log n)$ with the Master Theorem.

Exercise 4: Analysing an Algorithm

Algorithm 2 `algorithm`(A) ▷ **integer array** $A[0 \ldots n-1]$

```
for i ← 1 to n−1 do
    for j ← 0 to i−1 do
        for k ← 0 to n−1 do
            if |A[i]−A[j]| = A[k] then
                return true
return false
```

(a) What does the above algorithm compute?

(b) Give the asymptotic running time of the above algorithm and a short explanation for that.

(c) Describe an algorithm that computes the same output but asymptotically faster.

Sample Solution

(a) It checks whether there is a pair of numbers (from different positions) in the array such that the absolute difference of those two numbers equals tha value of another array entry.

(b) We have three nested loops. We show that the second loop makes $\Theta(n^2)$ iterations.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$$

Then we have a third inner loop which makes $\Theta(n)$ iterations for each iteration of the second loop. Hence the total asymptotic runtime is $\Theta(n^3)$.

(c) We do the following. First we use two nested loops to store all values of the form $|A[i]-A[j]|$ for all $(i,j) \in \{0,\ldots,n-1\}^2$ in a seperate array B of size n^2. This takes $\Theta(n^2)$ time. Second, we sort B using, e.g., the merge sort algorithm that we saw in a previous exercise, which takes $\Theta(n^2 \log(n^2)) = \Theta(2n^2 \log n) = \Theta(n^2 \log n)$ time (merge sort takes $\Theta(N \log N)$ time to sort an array of size N).

Third, for each $k \in \{0,\ldots,n-1\}$ we start a binary search for $A[k]$ on B (we have also seen this algorithm on a previous exercise sheet). This takes $\Theta(n \log(n^2)) = \Theta(n \log n)$ time as binary search has runtime just $\Theta(\log N)$ to search for a value in an array of size N. The total runtime is dominated by the second step, which takes $\Theta(n^2 \log n)$.

Alternative solution: We can aloe use a hash table H of size $\Theta(n^2)$ and insert all values $|A[i]-A[j]|$ for all $(i,j) \in \{0,\ldots,n-1\}^2$, which takes $\Theta(n^2)$ time. Then we can lookup $A[k]$ for each $k \in \{0,\ldots,n-1\}$ in $\Theta(n)$ time. The total runtime is dominated by the first step of hashing the values into table H, with runtime $\Theta(n^2)$. Note that the runtime is only in expectation for a randomly selected hashfuntion from a universal family.