Homework 5

Discrete Structures
Due Tuesday, February 9, 2021

Submit each question separately in .pdf format (except question 5)

---

1. (a) Trace out the values i, j, m for the binary search algorithm (Algorithm 3 on page 206) on the integer 10 and the list $1, 3, 4, 5, 8, 10, 11, 12, 15$.

i	1	1	1	6	6	6	6	6
j	–	9	9	9	9	7	7	6
m	–	–	5	5	7	7	6	6

□

(b) Trace out the list values a_i in the list $a_1 = 6, a_2 = 3, a_3 = 8, a_4 = 2, a_5 = 1, a_6 = 4, a_7 = 10$ for the bubble sort algorithm (Algorithm 4 on page 208).

a_1	6	3	3	3	3	3	3	3	2	2	1
a_2	3	6	6	6	6	2	2	2	3	1	2
a_3	8	8	2	2	2	6	1	1	1	3	3
a_4	2	2	8	1	1	1	6	4	4	4	4
a_5	1	1	1	8	4	4	4	6	6	6	6
a_6	4	4	4	4	8	8	8	8	8	8	8
a_7	10	10	10	10	10	10	10	10	10	10	10

□

(c) Trace out the values s, j for the naive string matching algorithm (Algorithm 6 on page 209) on the strings $t =$ `mississippi` and $p =$ `si`.

s	0	0	1	2	3	3	3	4	5	6	6	6	7	8	9
j	–	1	1	1	1	2	3	1	1	1	2	3	1	1	1

□

2. The *ternary search algorithm* locates an element in a list of strictly increasing integers by successively splitting the input list into three sublists of equal size, and restricting the search to the sublist in which the target integer lies.

(a) Implement the ternary search algorithm in pseudocode, writing an algorithm that locates the given element in a list or reports that it does not exist. Follow the example of the binary search algorithm on page 206.

The first call of this algorithm looks like this:

$$\text{TERNARY-SEARCH}(x : \text{integer},\ A : \text{sorted list},\ 1,\ n).$$

We search item x in the list (already sorted in increasing order: $a_1 < a_2 < \ldots < a_n$), and also the left and the right endpoint for the search (initially the left endpoint is $\ell = 1$ and the right endpoint is $r = n$).
In the first step and all the subsequent steps do the following: If $\ell = r$, the search interval is reduced to a single number and we finish. Otherwise, compute two "mid-points" m_1 and m_2 that subdivide the list $[a_\ell, \ldots, a_r]$ in approximately 3 equal parts. Compare the searchable x with one of the endpoints m_1 (and if it is not smaller, then also with m_2).

TERNARY-SEARCH(x : int, A : sortedList, ℓ : int, r : int)
1 **while** $\ell < r$ *(For the initial call $\ell = 1$, $r = n$)*
2   $m_1 = \ell + \lfloor (r - \ell)/3 \rfloor$
3   $m_2 = \ell + \lfloor (2(r - \ell) + 1)/3 \rfloor$
4   **if** $x \leq A[m_1]$
5   TERNARY-SEARCH(x, A, ℓ, m_1)
6   **else if** $x \leq A[m_2]$
7   TERNARY-SEARCH$(x, A, m_1 + 1, m_2)$
8   **else**
9   TERNARY-SEARCH$(x, A, m_2 + 1, r)$
10 **if** $x \leq A[\ell]$
11   *location* $:= \ell$
12 **else**
13   *location* := NOT FOUND
14 **return** *location*

Here are some examples how the initial interval $[1; n]$ is split into 3 (almost) equal parts for some small values of n. (m_1 and m_2 are computed on Lines 2,3 of the pseudocode.)

ℓ	r	m_1	m_2	$[\ell, r] \to [\ell, m_1], [m_1 + 1, m_2], [m_2 + 1, r]$
1	2	1	2	$[1; 2] \to [1; 1], [2; 2], -$
1	3	1	2	$[1; 3] \to [1; 1], [2; 2], [3; 3]$
1	4	2	3	$[1; 4] \to [1; 2], [3; 3], [4; 4]$
1	5	2	4	$[1; 5] \to [1; 2], [3; 4], [5; 5]$
1	6	2	4	$[1; 6] \to [1; 2], [3; 4], [5; 6]$
1	7	3	5	$[1; 7] \to [1; 3], [4; 5], [6; 7]$
1	8	3	6	$[1; 8] \to [1; 3], [4; 6], [7; 8]$
1	9	3	6	$[1; 9] \to [1; 3], [4; 6], [7; 9]$

Note that interval of length 9 ($[1; 9]$) splits into three intervals of length 3 ($[1; 3]$, $[4; 6]$, $[7; 9]$), any interval of length 3 is split into three intervals of length 1 (where the **while** loop stops).

$\square$

(b) What is the worst case for this algorithm? Give an example input.

Since the splitting of intervals (if n is not divisible by 3) creates three unequal parts, we need to work backwards to build the worst-case example for any given number of iterations of the **while** loop. Denote the number of iterations of **while** by k.

- $k = 1$ iteration is first achieved for $n_1 = 2 = 3^0 + 1$. (Each iteration needs at most two lookups in the array $A[i]$ to compare with x).
- $k = 2$ iterations are first achieved for $n_2 = 4 = 3^1 + 1$.
- $k = 3$ iterations are first achieved for $n_3 = 10 = 3^2 + 1$.
- $k = 4$ iterations are first achieved for $n_4 = 28 = 3^3 + 1$.

In general, k iterations are first achieved for $n_k = 3^{k-1} + 1$. But for these values n_k that are only slightly larger than 3^{k-1} we would only use one comparison: We would make a recursive call on Line 5 of the algorithm (and avoid comparison on Line 6). Since we will need the worst case for the number of comparisons (not the iterations of the **while** loop), we need to ensure that the Line 6 is evaluated for every recursive call. In this case the worst-case numbers will be different.

- $k = 1$ iteration is first achieved for $n'_1 = 2$. Two comparisons are needed, if we search $A[2]$ in $A[1..2]$.
- $k = 2$ iterations (2 comparisons each) are first achieved for $n'_2 = 5$. The worst case happens, if we search $A[4]$ in $A[1..5]$. (First iteration splits $[1;5]$ into $[1;2]$, $[3;4]$, $[5;5]$ and picks $[3;4]$. The next iteration uses two more comparisons and finds $A[4]$ in $A[3..4]$.)
- $k = 3$ iterations (2 comparisons each) are first achieved for $n'_3 = 3 \cdot n'_2 - 1 = 14$. Maximum number of comparisons happens if we search $A[9]$ in $A[1;14]$.

We get the following sequence: $n'_1 = 2$, $n'_{k+1} = 3n'_k - 1$ for all $k \geq 1$. The first members look like this:

$$2, 5, 14, 41, 122, 365, 1094, \ldots$$

We can also express this sequence by a closed formula:

$$n'_k = 1 + \left(1 + 3 + 3^2 + \ldots + 3^{k-1}\right) = 1 + \frac{3^k + 1}{2}. \tag{1}$$

Note. If your ternary search algorithm does different rounding and splits the interval in slightly different places m_1 and m_2, your worst case could look different. The important part is that it is approximately a geometric progression with common ratio 3.

□

(c) How many comparisons does this algorithm need in the worst case?

For any given list length n, express the k from the worst-case equation (1). We get at least k iterations where

$$k = \lceil \log_3(2(n-1)) \rceil + 1.$$

Each iteration uses at most two comparisons, so the total number of comparisons is

$$2\lceil \log_3(2(n-1)) \rceil + 2.$$

□

3. For each function $f(n)$ defined below, find the optimal $g(n)$ such that $f(n)$ is $O(g(n))$, and find C, n_0, such that $|f(n)| < C \cdot |g(n)|$ as long as $n > n_0$.

(a) $f(n) = 3n^4 + \log_2(n^8)$

Can take $g(n) = n^4$, $C = 11$, $n_0 = 1$. $\square$

(b) $f(n) = \sum_{k=1}^{n}(k^3 + k)$

Can take $g(n) = n^4$, $C = 2$, $n_0 = 1$. $\square$

(c) $f(n) = (n + 2)\log_2(n^2 + 1) + \log_2(n^3 + 1)$

Can take $g(n) = n \log n$, $C = 8$, $n_0 = 2$. $\square$

(d) $f(n) = n^3 + \sin(n^7)$

Can take $g(n) = n^3$, $C = 2$, $n_0 = 1$. $\square$

4. Assume that you have n coins; it is known that $n-1$ of these coins have equal weight, but one of them is heavier than the others. The input to the algorithm is a list of n integer variables representing the weights of the coins.

Note. An algorithm to find the maximum coin in a list of $a_1, a_2, \ldots, a_n$ is given in the textbook (Algorithm 1 on page 203); it needs $n - 1$ comparisons between individual numbers/coins.

You have a generalized comparison function that behaves like two-sided balance scales:

$$\text{compare}(\text{list}_1, \text{list}_2) = \begin{cases} -1, & \text{if } S_1 < S_2, \\ 0, & \text{if } S_1 = S_2, \\ 1, & \text{if } S_1 > S_2, \end{cases} \quad \text{where } S_1 = \sum_{a_i \in \text{list}_1} a_i, \quad S_2 = \sum_{a_j \in \text{list}_2} a_j.$$

Namely, you are allowed to compare any two groups of coins (of sizes $1, 2, \ldots, \lfloor n/2 \rfloor$ each); and the scales will tell you, if first group is lighter, same or heavier than the other group.

(a) Describe an algorithm that shows how to find the heaviest coin among n coins, if all the others have the same weight. You can write pseudocode or just explain precise steps in English.

Step 1. If $n = 1$, we know that the only coin is the heaviest one.

Step 2. If $n = 2$, compare two coins using one weighing. Return the heaviest one.

Step 3. If $n \geq 3$, divide all n coins into three lists of equal sizes (if n is not divisible by 3, create nearly equal lists of sizes that differ by at most 1). There will always be two lists of the same size. For example, $3 = 1 + 1 + 1$; $4 = 1 + 1 + 2$; $5 = 1 + 2 + 2$; $6 = 2 + 2 + 2$.

Step 4. Take two lists of equal sizes list_1, list_2 with $|\text{list}_1| = |\text{list}_2|$. The third group is G_3 (could have one more or one less coin than the other two lists). Compare list_1 and list_2 on scales.

Step 5. If $\text{compare}(\text{list}_1, \text{list}_2) = -1$, the heaviest coin is in list_2.
If $\text{compare}(\text{list}_1, \text{list}_2) = 1$, the heaviest coin is in list_1.

If compare$(\text{list}_1, \text{list}_2) = 0$, the heaviest coin is in list_3.
In any of these three cases repeat the steps 1–5 again, but replace the original list with one of the three sublists.

$\square$

(b) Find the times you call "compare$(\text{list}_1, \text{list}_2)$". Express the number of calls as a function of n (the worst-case estimate).

Given the number of coins n, denote by a_n the number of comparisons before we find the heaviest coin. The sequence a_n looks like this:

$$a_1 = 0,\ a_2 = 1,\ a_3 = 1,\ a_4 = 2,\ a_5 = 2,\ a_6 = 2,\ a_7 = 2,\ a_8 = 2,\ a_9 = 2,\ a_{10} = 3, \ldots.$$

The closed formula for the n-th member is this:

$$a_n = \lceil \log_3 n \rceil .$$

$\square$

(c) Show that you used as few calls to "compare$(\text{list}_1, \text{list}_2)$" as possible.

Claim. It is not possible to find the heaviest coin with less than $\lceil \log_3 n \rceil$ comparisons (which the algorithm in (a) ensures).
To prove this, note that there are n different ways how the heaviest coin can be located in a list of n coins (it can be the 1st coin, the 2nd coin, and so on). Every comparison generates one of three possible outcomes (the scales can return values $+1$, 0 or -1). The number of comparisons k should be such that $3^k \geq n$ (otherwise there are two possible locations for the heaviest coin that are not distinguishable).
Take the logarithm of the both sides:

$$k \geq \log_3 n \ \rightarrow\ k \geq \lceil \log_3 n \rceil .$$

The first inequality implies the other one, because k must be an integer number. So the algorithm in (a) which uses exactly $\lceil \log_3 n \rceil$ is optimal and cannot be improved (it is impossible to do it with less comparisons). $\square$

5. Complete the proofs in Coq. You may use the non-constructive `classic` and `NNPP` axioms if needed, but try to minimize their use. Submit your file as plain-text `hw5_question5.v`.

Full answer is available in the course Webpage under *Discrete 2021: Assignments*. See `https://bit.ly/3aipOmx`.

$\square$

```
Section Predicate_Logic_Examples.

(* A is a nonempty set (containing element 'something' *)
Variables A : Set.
Variables something: A.
(* Assume that P,Q are 1-argument predicates defined on A *)
```

```
Variables P Q : A->Prop.

(* Can distribute 'exists' quantifier over a disjunction *)
Lemma sample5_1:
    (exists (y:A), (P y)) \/ (exists (y:A), (Q y)) <->
    exists (x:A), (P x) \/ (Q x).
Proof.
  (* Insert a proof; then replace 'Admitted' by 'Qed' *)
  Admitted.

(* A variant of De Morgans law *)
Lemma sample5_2:
    (exists (x:A), ~~(P x)) <-> ~(forall (y:A), ~(P y)).
Proof.
  Admitted.

(* If (P x) always implies (Q x), then the existence
   of some (P x0) leads to existence of some (Q x1) *)
Lemma sample5_3:
    (forall (x:A), P x -> Q x) ->
        ((exists (x:A), (P x)) -> exists (x:A), (Q x)).
Proof.
  Admitted.

(* If P being true sometimes implies that also Q is true sometimes,
   then there is some x0 for which (P x) implies (Q x) *)
Lemma sample5_4: ((exists (x:A), (P x)) -> (exists (x:A), (Q x))) ->
    (exists (x:A), ((P x) -> (Q x))).
Proof.
  Admitted.

(* If P(x) always implies Q(x), and P(x) is always true,
   then Q(x) is always true. *)
Lemma sample5_5: (forall (x:A), ((P x) -> (Q x))) ->
    ((forall (x:A), (P x)) -> forall (x:A), (Q x)).
Proof.
  Admitted.

End Predicate_Logic_Examples.
```