

Heaps - Recitation 4

Correctness, Loop Invariants And Heaps.

November 5, 2022

Apart of quantify the resource requirement of our algorithms we are also interested to prove that they indeed work. In this Recitation we will demonstrate how to prove correctness via the notation of loop invariant. In addition we will present the first (non-trivial) data structure in course, and prove that it allows us to compute the maximum efficiently.

Correctness And Loop Invariant.

In this course, any algorithm is defined relative to a task/problem/function, And it will be said correct if for any input it computes desirable output. For example, suppose that our task is to extract the maximum element from a given array. So the input space are all the arrays of numbers, and proving that a given algorithm is correct, requires to prove that for an arbitrary array the algorithm's output is the maximal number. Formally:

Correctness.

We will say that an algorithm \mathcal{A} (an ordered set of operations) computes $f : D_1 \rightarrow D_2$ if for every $x \in D_1 \Rightarrow f(x) = \mathcal{A}(x)$. Sometimes when it's obvious what is the goal function f , we will abbreviate and say that \mathcal{A} is correct.

Other Examples of functions f might be including any computation tasks: file saving, summing numbers, posting a message in the forum, etc. Let's dive back into the maximum extraction problem and see how correctness should be proved in practice.

Task: Maximum Finding. Given $n \in \mathbb{N}$ numbers $a_1, a_2, \dots, a_n \in \mathbb{R}$ write an Algorithm which returns their maximum.

Consider the following suggestion. How would you prove it correct?

Maximum finding.

Result: returns the maximum of $a_1 \dots a_n \in \mathbb{R}^n$

```
1 let  $b \leftarrow a_1$ 
2 for  $i \in [2, n]$  do
3    $b \leftarrow \max(b, a_i)$ 
4 return  $b$ 
```

Usually it will be convenient to divide the algorithms into subsections and then characterize, and prove correctness for each of them separately. One main technique is using the notation of Loop Invariants. Loop Invariant is a property that characterizes a loop segment code and satisfies the following conditions:

Loop Invariant.

1. Initialization. The property holds (even) before the first iteration of the loop.
2. Conservation. As long as one performs the loop iterations, the property still holds.
3. (optional) Termination. Exiting from the loop carrying information.

What is the Loop Invariant here? "at the i -th iteration, $b = \max\{a_1 \dots a_{i-1}\}$ ". The proof is almost identical to the naive case.

Claim. Consider the while loop. The property: "for every $j' < j \leq n + 1 \Rightarrow a_{j'} \leq a_i$ " is a loop invariant that is associated with it.

Proof: first, the initialization condition holds, as at the first iteration $j = 1$ and therefore the property is trivial. Assume by induction, that for every $m < j$ the property is correct, and consider the j -th iteration. If back again to line (5), then it means that $(j - 1) < n$ and $a_{j-1} \leq a_i$. Combining the above with the induction assumption yields that $a_i \geq a_{j-1}, a_{j-2}, \dots, a_1$.

Correctness Proof. Split into cases, First if the algorithm return result at line (9), then due to the loop invariant, combining the fact that $j = n + 1$, it holds that for every $j' \leq n < j \Rightarrow a_i \geq a_{j'}$ i.e a_i is the maximum of a_1, \dots, a_n . The second case, in which the algorithm returns Δ at line number (10) contradicts the fact that n is finite, and left as an exercise. the running time is $O(n^2)$ and the space consumption is $O(n)$.

Task: Element finding. Given $n \in \mathbb{N}$ numbers $a_1, a_2, \dots, a_n \in \mathbb{R}$ and additional number $x \in \mathbb{R}$ write an Algorithm that returns i s.t $a_i = x$ if there exists such i and False otherwise.

Element finding.

Result: returns the maximum of $a_1 \dots a_n \in \mathbb{R}^n$

```
1 for  $i \in [n]$  do
2   if  $a_i = x$  then
3     return  $i, a_i$ 
4 return  $\Delta$ 
```

Loop Invariant In The Cleverer Alg. Consider now the linear time algorithm:

Heaps.

We have seen in the Lecture that no Algorithm can compute the max function with less than $n - 1$ comparisons. So our solution above is indeed the best we could expect for. The same is true for the searching problem, and yet we saw that if we are interested in storing the numbers then, by storing them according to sorted order, we could compute each query in logarithmic time via binary search. That arises the question, is it possible to have a similar result regarding the max problem? Heaps are structures that in addition to supporting adding and removing elements are also able to compute the maximum efficiently.

Heapify.

Result: returns the maximum of $a_1 \dots a_n \in \mathbb{R}^n$

```
1 next  $\leftarrow$  i
2 if  $2i < n$  and  $H_{next} \leq H_{2i}$  then
3   | next  $\leftarrow 2i$ 
4 if  $2i + 1 < n$  and  $H_{next} \leq H_{2i+1}$  then
5   | next  $\leftarrow 2i + 1$ 
6 if  $i \neq next$  then
7   |  $H_i \leftarrow H_{next}$  Heapify(next)
```

Heappush.

Result: returns the maximum of $a_1 \dots a_n \in \mathbb{R}^n$

```
1
2 for  $i \in [n]$  do
3   |
4     | if  $a_i = x$  then
5       |   | return  $i, a_i$ 
6 return  $\Delta$ 
```

Heappop.

Result: returns the maximum of $a_1 \dots a_n \in \mathbb{R}^n$

```
1
2 for  $i \in [n]$  do
3   |
4     | if  $a_i = x$  then
5       |   | return  $i, a_i$ 
6 return  $\Delta$ 
```