# Heaps - Recitation 4

Correctness, Loop Invariant And Heaps.

November 8, 2022

Apart from quantifying the resource requirement of our algorithms, we are also interested in proving 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 the 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 correctly 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 is all the arrays of numbers, and proving that a given algorithm is correct requires proving that the algorithm's output is the maximal number for an arbitrary array. Formally:

**Correctness.**

We will say that an algorithm $\mathcal{A}$ (an ordered set of operations) computes $f : D_1 \to D_2$ 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 functions $f$ might be including any computation task: 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 proven in practice.

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

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

**Maximum finding.**

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

```
1 let b ← a_1
2 for i ∈ [2, n] do
3     b ← max(b, a_i)
4 return b
```

Usually, it will be convenient to divide the algorithms into subsections and then characterize and prove their correctness separately. One primary technique is using the notation of Loop Invariant. Loop Invariant is a property that is characteristic of 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. Termination. Exiting from the loop carrying information.

Let's denote by $b^{(i)}$ the value of $b$ at line number 2 at the $i$th iteration for $i \geq 2$ and define $b^{(1)}$ to be it's value in the it's initialization. What is the Loop Invariant here? **Claim.** *"at the i-th iteration,* $b^{(i)} = \max\{a_1...a_i\}$*"*.

**Proof.** Initialization, clearly, $b^{(1)} = a_1 = \max\{a_1\}$. Conservation,by induction, we have the base case from the initialization part, assume the corretness of the clain for any $i' < i$ and consider the $i$th iteration (ofcourse, assume that $i < n$). Then:

$$b^{(i)} = \max\{b^{(i-1)}, a_i\} = \max\max\{a_1, ..a_{i-2}, a_{i-1}\}, a_i\} = \max a_1, ..a_i\}$$

And that compleate the Conservation part. Termination, followes by the conservation, at the $n$ iteration, $b^{(i)}$ is seted to $\max\{a_1, a_2..a_n\}$.

**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 the 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 $a_i \geq a_{j-1}, a_{j-2}, ...a_1$.

**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 $a_1, ....a_n$. The second case, in which the algorithm returns $\Delta$ 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* $a_1, a_2, \cdots a_n \in \mathbb{R}$ *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 $a_1...a_n \in \mathbb{R}^n$

```
1 for i ∈ [n] do
2     if a_i = x then
3         return i, a_i
4 return Δ
```

**Task: The Superpharm Problem.** *You are requested to maintain a pharmacy line. In each turn, you get one of the following queries, either a new customer enters the shop, or the pharmacist requests the next person to stand in front. In addition, different customers have different priorities, So you are asked to guarantee that in each turn, the person with the height priority will be at the front.*

Before we consider a sophisticated solution, What is the running time for the naive solution?(maintaining the line as a linear array) $(\sim O\left(n^2\right))$.

## Heaps.

Heaps are structures that enable computing the maximum efficiency in addition to supporting adding and removing elements.

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 search problem. Yet, we saw that if we are interested in storing the numbers, then, by keeping them according to sorted order, we could compute each query in logarithmic time via binary search. That raises the question, is it possible to have a similar result regarding the max problem?

**Heap**

Let $n \in \mathbb{N}$ and consider the sequence $H = H_1, H_2 \cdots H_n \in \mathbb{R}(*)$. we will say that $H$ is a Heap if for every $i \in [n]$ we have that: $H_i \leq H_{2i}, H_{2i+1}$ when we think of the value at indices greater than $n$ as $H_{i>n} = -\infty$.

**Checking vital signs.** Are the following sequences are heaps?

1. 1,2,3,4,5,6,7,8,9,10 (Y)
2. 1,1,1,1,1,1,1,1,1,1 (Y)
3. 1,4,3,2,7,8,9,10 (N)
4. 1,4,2,5,6,3 (Y)

How mach is cost (running time) to compute the the min of $H$? (without change the heap). ($O(1)$). Assume that option 4 is our Superpharm Line, let's try to imagine how should we maintain the line. After serving the customer at top, what can be said on $\{H_2, H_3\}$? or $\{H_{i>3}\}$? (the second heighset value is in $\{H_2, H_3\}$.)

**Subtask: Extracting Heap's Minimum.** *Let $H$ be an Heap at size $n$, Write algorithm which return $H_1$, erase it and returns $H'$, an Heap which contain all the remain elements.* **Solution:**

**Heappop.**

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

```
1 ret ← H_1
2 H_1 ← ∞
3 Heapify-down(1)
4 return ret
```

**Heapify-down.**

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

```
1 next ← i
2 if 2i < n and H_next ≤ H_2i then
3     next ← 2i
4 if 2i + 1 < n and H_next ≤ H_2i+1 then
5     next ← 2i + 1
6 if i ≠ next then
7     H_i ↔ H_next
8     Heapify-down(next)
```

**Claim.** Assume that $H$ satisfies the Heap inequality for all the elements except $H_j$. Namely for any $i \neq j$ we have that $H_i \leq H_{2i}, H_{2i+1}$. Then applying Heapify-down on $H$ at index $j$ returns an heap.

**Proof.** Induction on $n - j$. Base case, $n - j < n/2 \Rightarrow j > n/2$, Hence $2j, 2j + 1 \notin [n]$ and we have that the Heap property holds for any $i$.

Assume the correctness of the claim for any $j'$ satisfy $j' > j$ and consider $j$. Denote by $H'$ the state of the heap after the swapping at line number 7 and let $i \in [n]$. First Notice that if $j \neq 2i, 2i+1$ and $i \neq 2j, 2j+1$ then $H'_i = H_i \leq H_{2i} = H'_{2i}$ (and in similar to $2i + 1$). So it left to consider the case where $i = \lfloor j/2 \rfloor$ and $i = j$.

**Insertion.** blabla

**Heapify-up.**

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

```
1 parent ← ⌊i/2⌋
2 if parent > 0 and H_perent ≤ H_i then
3 |   H_i ↔ H_perent
4 |   Heapify-up(perent)
```

**Heappush.**

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

```
1 H_n ← v
2 Heapify-up(n)
3 n ← n + 1
```