Communicating Algorithms[1]

One of the goals of this course is to teach how to *communicate* algorithms to other humans. This is not as trivial as it sounds. To illustrate this, let me take the example mentioned at the end of Lecture 1's notes.

MAX&MIN
Input: An array $A[1:n]$
Output: A maximum element of A *and* a minimum element of A.
Size: The number of elements in the array, that is, n.

I want to tell you the algorithm which can solve the above problem making at most $3n/2$ comparisons. This is better than the $2n-3$ comparison algorithm of first finding the maximum, and then finding the minimum. Before I show the pseudocode of the algorithm, I want to first describe the idea in plain English. The reason is that we are trying to communicate to humans (this is important to remember). The issue with English, however, is that it is often vague, and therefore writing ideas down requires some skill.

The English Description. Given the array, we first pair up[2] neighboring elements. For now, let us consider the case when the length n is even; we will take care of the case of odd n subsequently. So, for instance[3], if the array was $A = [13, 4, -3, 5, 90, 34]$, we are pairing up the elements[4] as $(13, 4)$, $(-3, 5)$, and $(90, 34)$. We compare the paired elements, and we put the *larger* number into a list B and the *smaller* number into a list C. At this point, note[5] that the maximum element of A must lie in B, and the minimum element of A must lie in C. Why[6]? Because, the maximum element by definition will be larger than its partner and therefore be sent to B. A similar argument shows that the minimum element of A will be in C. Therefore, to complete the algorithm, we simply return the maximum element of B and the minimum element of C. This completes the description[7] of the algorithm when n is even.

To take care of the case of *odd* n, we pair the first $(n-1)$ elements, that is, $A[1:n-1]$, to obtain the lists B and C as above. Then, we simply append $A[n]$ to *both* B and C. Note that the maximum of A is still in B (even if $A[n]$ were the maximum), and the minimum of A is still in C (even if $A[n]$ were the minimum). The algorithm returns the maximum element of B and minimum element of C. This completes the description of the algorithm in all cases[8].

Analysis. Let us see how many comparisons we make. For the case of even n, we make $\frac{n}{2}$ comparisons to get the lists B and C. Note that both B and C have length $n/2$ each. Finding the min and max in each takes $(\frac{n}{2}-1)$, and thus the total number of comparisons is $\frac{n}{2} + \left(\frac{n}{2}-1\right) + \left(\frac{n}{2}-1\right) = \frac{3n}{2} - 2$.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] This is a vague statement. But hopefully expressive enough to understand

[3] Always a good idea to give examples, especially, when you feel the description is vague (as is probably here)

[4] Note that at this point I am not caring what data-structure I will use to pair up. That is not important for the idea.

[5] This is where we are making the **key** observation that imply the algorithm's correctness. And this will help the human reader immensely to understand what is going on.

[6] One shouldn't be so colloquial, but I pepper my proofs in lecture notes to sort of earmark parts of the argument which I feel are not completely trivial. You should do the same, and then probably erase the "Why?"s

[7] At this point, most readers should be (a) have a basic idea of what the algorithm does, (b) should also know why the algorithm is correct, and (c) should be reasonably confident that they can implement it.

[8] Do you see how the understanding of the odd case was made easier because the even case was explained earlier? The even case contains the ***essence*** of this algorithm. Always, try to describe the essence of your algorithms first.

For the case of odd n, we make $\frac{n-1}{2}$ comparisons to get B and C. Then we append $A[n]$ resulting in the lengths of B and C being $\frac{n-1}{2}+1 = \frac{n+1}{2}$ each. Their mins and maxes can be found in $\frac{n+1}{2} - 1 = \frac{n-1}{2}$ time each. Thus, the total time is $\frac{3(n-1)}{2}$.

Pseudocode. Now, let us write the *pseudocode* of the above algorithm. It is really *closer* to the English version than the code. It is used to iron out vagueness (in this case, the pairing).

1: **procedure** MAXMIN($A[1:n]$):
2: ▷ *Returns a maximum and a minimum element of A.*
3: Initialize two empty lists B and C.
4: **for** $i = 1$ to $\lfloor n/2 \rfloor$ **do**:
5: Compare $A[2i-1]$ and $A[2i]$. ▷ *Compare neighboring pairs.*
6: Send the larger number to B and the smaller to C.
7: If n is odd, then send $A[n]$ to both B and C.
8: **return** $\max(B)$ and $\min(C)$.

How should you never describe an algorithm. I cannot stress this strongly enough

*Pseudo-code is **not** code with "types and semicolons*[9]*" removed.*

Describing an algorithm as code is an exasperating way to communicate[10] an algorithm. To illustrate this, forget all the description of the algorithm above, and imagine you are hearing the problem for the first time. You are amazed to hear that the max and min can be found using $\frac{3n}{2}$ comparisons...and then you see this[11]:

```
def MAXMIN(A):
  n = len(A)
  B = list()
  C = list()
  for j in range(n//2):
    if (A[2*j] < A[2*j+1]):
      B.append(A[2*j+1])
      C.append(A[2*j])
    else:
      C.append(A[2*j+1])
      B.append(A[2*j])
  if(n % 2 == 1):
    B.append(A[n-1])
    C.append(A[n-1])
  r = 0
  for j in range(1, len(B)):
    if(B[j] > B[r]):
      r = j
  s = 0
  for j in range(1, len(C)):
    if(C[j] < C[s]):
      s = j
  return (B[r],C[s])

```

[9] Python doesn't even have these
[10] Imagine being told a story, one letter at a time
[11] This code is written by me and not by any of you. It is especially beastly for it is (deliberately) not commented at all.

What would your reaction be? You would have to first parse the above code to see what it is doing. However, even after parsing it, you don't immediately *understand* it. Why is the code doing what it is doing? It can be pretty time-consuming. So please, **never describe any algorithm to a human using code**[12].

[12] But try coding up all your algorithms.