Two Pointer Technique[1]

The lecture talks about a trick colloquially called the "two pointer technique" which can speed up many algorithms from their naive versions. At an abstract level, the problems they solve are *search* problems, and, when it works, the speed-up is obtained when one realizes that large parts of the search space can be discarded by a single computation. We see a bunch of examples below, and some more examples are explored in the exercises.

- *Binary Search.* Last lecture, we saw a *recursive* algorithm for searching for an element x in a sorted array $A[1:n]$. We can reinterpret the algorithm to illustrate the "two pointer trick" of cutting out search space. We initialize two pointers $a = 1$ and $b = n$ with the semantic that x, if it exists in A, exists in $A[a:b]$. Contrapositively, it, for sure, doesn't exist in $A[1:a-1]$ or $A[b+1:n]$. With a single comparison, we will drop $b - a$ to roughly half its size, and thus get a logarithmic running time. Here is the algorithm.

```
1:  procedure BinarySearch(A[1 : n], x):
2:      ▷ A is assumed to be sorted increasing
3:      ▷ Returns NO if x ∉ A, otherwise returns i with A[i] = x
4:      a ← 1; b ← n
5:      ▷ Invariant: x ∉ A[1 : a − 1] ∪ A[b + 1 : n]
6:      while a ≤ b do:
7:          m ← ⌈(a+b)/2⌉
8:          ▷ Compare A[m] and x
9:          if x = A[m] then:
10:             return m
11:         else if x > A[m] then ▷ x cannot lie in A[1 : m]
12:             a ← m + 1
13:         else: ▷ x < A[m] ⇒ x cannot lie in A[m : n]
14:             b ← m − 1
15:     ▷ If you haven't returned yet, then x ∉ A
16:     return NO
```

The single comparison in Line 8 either terminates the algorithm, or moves one of the pointers "a lot". The correctness of this algorithm follows from the fact that $A[1:n]$ is sorted increasing: if $x > A[m]$ as in Line 11, we know $x > A[j]$ for all $j \leq m$, and so $x \notin A[1:m]$. So, moving $a \leftarrow m + 1$ maintains the invariant $x \notin A[1:a-1]$. Similarly, if $x < A[m]$, we know $x < A[j]$ for $j \geq m$, and so $x \notin A[m:n]$. So moving $b \leftarrow m - 1$ maintains the invariant $x \notin A[b+1:n]$.

The running time is of the same order as the number of while loops since each while-loop takes $O(1)$ time (only comparisons and "variable changes" are made). To argue about the number of while loops, we note that if the algorithm doesn't terminate, the quantity $b - a$ drops by at least a factor of 2. For instance, if Line 11 was true, we go from $b - a$ to $b - (\lfloor a + b \rfloor 2 + 1) \leq b - \frac{a+b}{2} \leq \frac{b-a}{2}$. The other

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 26th Jun, 2025*
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!

case is similar (please check). The algorithm starts with $b - a = n$, and so the number of loops is at most $\lceil \log_2 n \rceil$.

Exercise: *How will you alter the algorithm to return index $1 \leq i \leq n$ with $|A[i] - x|$ as small as possible?*

- *Checking for Collisions in Sorted Arrays.* In this problem, we are given two sorted increasing arrays $A[1:m]$ and $B[1:n]$. We have to decide whether there is an element in A which is also in B, that is, whether $A \cap B$ is empty or not. If not, we would like to return one such element (or maybe all elements). Note this is also a search problem: we are being asked whether there exists $1 \leq i \leq m$ and $1 \leq j \leq n$ such that $A[i] = B[j]$. There are mn many such (i, j) pairs, and so the "naive" algorithm would take $O(mn)$ time. We will now see how the two pointer trick solves this in $O(m + n)$ time.

Remark: *You may already see an $O(\min(m \log n, n \log m))$ time algorithm: just take every element of the smaller array and binary search for it in the larger sorted array.*

The algorithm starts with $a = 1$ and $b = 1$ where a is searching in A and b is searching in B. We maintain that there are no collisions in $A[1:a]$ and $B[1:b]$ except possibly $A[a]$ could be $B[b]$. And so, we compare $A[a]$ and $B[b]$. If equal, we are done. Otherwise, we make the following observation. If $A[a] < B[b]$, then since B is sorted increasing, we know $A[a] < B[j]$ for all $j \geq b$. By the invariant maintained, we know $A[a]$ is not equal to anything in $B[1:b]$. And so we *can forget* $A[a]$; we don't need to compare it to anything else. So we move the pointer $a \leftarrow a + 1$. We will never go back. Analogously, if $B[b] < A[a]$, then we increment $b \leftarrow b + 1$. We stop when a becomes $m + 1$ or b becomes $n + 1$ (saying NO).

```
1:  procedure DETECT COLLISION(A[1 : m], B[1 : n]):
2:      ▷ Detect if there is a 1 ≤ i ≤ m, 1 ≤ j ≤ n such that A[i] = B[j].
3:      ▷ Returns NO if no such pair.
4:      a ← 1; b ← 1
5:      ▷ Invariant: A[1 : a − 1] ∩ B = ∅ and A ∩ B[1 : b − 1] = ∅.
6:      while a ≤ m and b ≤ n do:
7:          ▷ Compare A[a] and B[b]
8:          if A[a] = B[b] then
9:              return (a, b) ▷ Found collision
10:         else if A[a] < B[b] then: ▷ A[a] isn't there in B since B is sorted.
11:             a ← a + 1 ▷ Invariant satisfied
12:         else:▷ B[b] isn't there in A since A is sorted.
13:             b ← b + 1 ▷ Invariant satisfied
14:     return No Duplicates ▷ ..if you reach here
```

The running time, as in the previous example, is of the same order of the number of while loops. In each while loop $a + b$ increases by 1. We begin with $a + b = 2$ and the largest $a + b$ can be is $m + n$. So, the number of while loops is at most $m + n - 2$, implying the $O(m + n)$ running time as promised.

Exercise: *How will you alter the algorithm to count the number of pairs (i, j) with $A[i] = B[j]$?*

- *Combining Sorted Arrays.* Given two sorted arrays $A[1 : m]$ and $B[1 : n]$ which are sorted increasing, we want to return an array $C[1 : m+n]$ which is sorted and contains the elements of both arrays (with duplicates, if any). If you have understood the previous algorithm, then this one would, hopefully, be immediate. The idea is almost the same: we keep two pointers reading the arrays, and when we compare $A[a]$ and $B[b]$, we put the smaller one at the "end" of the array C which we keep building.

```
1:  procedure COMBINE(A[1 : m], B[1 : n]):
2:      ▷ A and B are sorted increasing
3:      ▷ Outputs C[1 : m + n], a sorted array of elements in A and B.
4:      a ← 1; b ← 1; c ← 0
5:      ▷ Invariant: C[1 : c] is sorted version of A[1 : a − 1] ∪ B[1 : b − 1]
6:      ▷ Invariant: A[a] and B[b] are ≥ C[c] if c > 0
7:      while a ≤ m and b ≤ n do:
8:          c ← c + 1
9:          if (A[a] ≤ B[b]) then: ▷ A[a] ≤ B[j] for j ≥ b
10:             C[c] ← A[a]
11:             a ← a + 1
12:             ▷ Check all invariants hold
13:         else: ▷ B[b] ≤ A[j] for j ≥ a
14:             C[c] ← B[b]
15:             b ← b + 1
16:             ▷ Check all invariants hold
17:     ▷ We have now sorted array C which contains one array completely and part of another
        array
18:     ▷ By invariant, the remaining elements should be appended at the end of C.
19:     if a > m then:
20:         Append rest B[b : n] to R
21:     else:
22:         Append rest of A[a : m] to R
23:     return R.
```

To prove correctness it suffices to show the invariants hold. For instance, let's check Line 12. By invariant, $A[a]$ is larger than $C[c]$, so appending $A[a]$ to the end keeps C sorted increasing. Since $A[a] \leq B[b]$, the invariant $B[b]$ at least last element of C still holds, Since A is sorted increasing, we have $A[a + 1]$ is at least last element of C (which is $A[a]$). So all invariants hold. The other check is analogous, and so the algorithm is correct.

The running time is $O(m + n)$ as in previous example.

- *Container with most water.* Now we look at a more interesting problem. This is an example from Leetcode.

To state their problem, we are given n lines of different heights placed vertically with one endpoint at $(i, 0)$ and the other at $(i, H[i])$ where $H[1 : n]$ is the array of heights. The goal is to pick two lines

which along with the x-axis "traps the maximum amount of water" (see their website for an example and illustration). Note that the "water trapped" is the area enclosed by distance between the segments horizontally, and the *shorter* line vertically.

If you think a bit about it, then you will see that what it is asking is to find

$$\max_{1 \le i < j \le n} F(i,j) \quad \text{where} \quad F(i,j) := (j-i) \cdot \min(H[i], H[j])$$

and as usual, we want a pair (i,j) which attains this maximum value as well. The reason I have abstracted out it as $F(i,j)$ is that the method we describe will use a property which will hold for the above definition. It also makes presentation easier.

The above problem is an example of an *optimization* (in this case maximization) problem. But it can also be thought of as a "search" problem, we are trying to search for the (i,j) which achieves the maximum. To this end, we keep a "running maximum" rmax which will be updated as we find a better solution; this can be initialized to 0. We now start our two pointer search process, and initialize then with $a \leftarrow 1$ and $b \leftarrow n$ (as in binary-search). The idea of the pointers is that at any point of time we have "searched" pairs involving $A[1:a-1] \cup B[b+1:n]$. More precisely, we will maintain that our running max rmax will satisfy the constraint rmax $\ge F(i,j)$ for $1 \le i \le a-1$ and *any* $j > i$, or $b+1 \le j \le n$ and *any* $i < j$.

We begin by computing $F(a,b)$ and comparing with rmax updating it if needed; note this makes rmax only bigger and thus invariants remain satisfied. In any case, after this, we have rmax $\ge F(a,b)$. Now we compare $H[a]$ and $H[b]$ and make the *crucial* observation: if $H[a] \le H[b]$, then $F(a,j) \le F(a,b)$ for all $a < j \le b$. This is because $(j-a) \le (b-a)$ and $\min(H[a], H[j]) \le H[a] = \min(H[a], H[b])$.

Why is this useful? Because this allows us to increment $a \leftarrow a+1$ since the above shows rmax $\ge F(a,b) \ge F(a,j)$ for $a < j \le b$, and rmax $\ge F(a,j)$ for $j \ge b+1$ was implied by the invariant. So we can increment the pointer of a. Similarly, if $H[b] < H[a]$, we can decrement the pointer of b. We stop when $a = b$, and return the rmax. The pseudocode is below; hopefully you see the runtime is $O(n)$.

```
1:  procedure MaxWaterContainer(H[1 : n]):
2:      ▷ Find 1 ≤ i < j ≤ n maximizing F(i, j) = (j − i) min(H[i], H[j]).
3:      a ← 1; b ← n; rmax ← 0
4:      ▷ Invariant: rmax ≥ F(i, j) for 1 ≤ i ≤ a − 1, i < j ≤ n
5:      ▷ Invariant: rmax ≥ F(i, j) for b + 1 ≤ j ≤ n, 1 leqi < j
6:      while a < b do:
7:          Compute F(a, b) and if F(a, b) > rmax, rmax ← F(a, b) ▷ If you want indices, then
    you'd need to remember this here
8:          if H[a] ≤ H[b] then:
9:              a ← a + 1 ▷ Since F(a, j) ≤ F(a, b) for a < j ≤ b
10:         else: ▷ H[b] < H[a]...
11:             b ← b − 1 ▷ ...implying F(i, b) ≤ F(a, b) for a ≤ i < b
12:     return rmax
```