Homework 1 – COT5405 *

Kobee Raveendran

August 2021

1. note: all logs are implicitly base 2, but other bases may be specified explicitly. Also, I use the Master theorem template for some questions, so it is listed below.

   $T(n) = aT(\frac{n}{b}) + f(n)$, where $f(n) \in O(n^k \log^p n)$

   (a) $T(n) = 9T(\frac{n}{2}) + n^3 \longrightarrow O(n^{\log 9}) \longrightarrow O(n^{2 \log 3})$

   Steps:
   using the Master theorem, we have:
   $a = 9$
   $b = 2$
   $f(n) = n^3 \longrightarrow k = 3, p = 0$
   Since $\log_2 9 > 3$, we follow case 1 of the Master theorem ($\log_b a > k$), which means the bound is in $O(n^{\log_b a})$. After substitution, that gives us:
   $O(n^{\log 9}) = O(n^{2 \log 3})$

   (b) $T(n) = 7T(\frac{n}{2}) + n^3 \longrightarrow O(n^3)$

   Steps:
   using the Master theorem, we have:
   $a = 7$
   $b = 2$
   $f(n) = n^3 \longrightarrow k = 3, p = 0$
   We find that this recurrence falls under case 3 of the theorem, in which $\log_b a < k$ (since $(\log_2 7 < 3)$), so we arrive at a complexity of the form $O(n^k) \longrightarrow O(n^3)$.

   (c) $T(n) = T(\sqrt{n}) + \log n$
   Steps:
   using iterative substitution:
   $k = 0$: $T(n) = T(n^{\frac{1}{2}}) + \log n$
   $k = 1$: $T(n^{\frac{1}{2}}) = T(n^{\frac{1}{4}}) + \log n^{\frac{1}{2}}$

   Substituting back in, we have:
   $T(n) = T(n^{\frac{1}{4}}) + \log n^{\frac{1}{2}} + \log n = T(n^{\frac{1}{4}}) + \frac{1}{2} \log n + \log n$

   Which can be generalized to:
   $T(n) = T(n^{\frac{1}{2^{k+1}}}) + (1 + \frac{1}{2} + \frac{1}{4} + .. + \frac{1}{2^k}) \log n \approx T(n^{\frac{1}{2^{k+1}}}) + 2 \log n$

   Using a base case of $T(2) = c$, we can solve for k:
   $n^{\frac{1}{2^{k+1}}} = 2 \longrightarrow \log n^{\frac{1}{2^{k+1}}} = \log 2 = 1 \longrightarrow \frac{1}{2^{k+1}} \log n = 1 \longrightarrow \log n = 2^{k+1} \longrightarrow$
   $\log \log n = \log 2^{k+1} \longrightarrow \log \log n = k + 1 \longrightarrow k = \log \log n - 1$

   Substituting k back in, as well as $T(2) = c$, we have:
   $T(n^{\frac{1}{2^{\log \log n}}}) + 2 \log n \longrightarrow c + 2 \log n$

*For some questions, I received guidance from or collaborated with: Minh Pham, TA (office hours)

Giving us a time complexity of $O(\log n)$, which can be confirmed using the Master theorem by substituting $n = 2^m$ to get a compatible recurrence format.

(d) $T(n) = \sqrt{n}T(\sqrt{n}) + n \longrightarrow O(n \log \log n)$

Steps:

using iterative substitution, we have:

$T(n) = n^{\frac{1}{2}}T(n^{\frac{1}{2}}) + n$

$k = 1$: $T(n^{\frac{1}{2}}) = n^{\frac{1}{4}}T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}$

Substituting: $T(n) = n^{\frac{1}{2}}[n^{\frac{1}{4}}T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}] + n \longrightarrow T(n) = n^{\frac{1}{2}+\frac{1}{4}}T(n^{\frac{1}{4}}) + n + n$

$k = 2$: $T(n^{\frac{1}{4}}) = n^{\frac{1}{8}}T(n^{\frac{1}{8}}) + n^{\frac{1}{4}}$

Substituting: $T(n) = n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}T(n^{\frac{1}{8}}) + n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}} + n + n$

Simplifying the fractional sums (which converge to 1), we have the general form:

$T(n) = nT(n^{\frac{1}{2^{k+1}}}) + (k+1)n$

Using base case $T(2) = c$ and solving for k, we have:

$n^{\frac{1}{2^{k+1}}} = 2 \longrightarrow \frac{1}{2^{k+1}} \log n = 1 \longrightarrow \log \log n = \log 2^{k+1} \longrightarrow \log \log n - 1 = k$

Substituting k back in and using the base case, we have:

$cn + (\log \log n)n \longrightarrow O(n \log \log n)$

(e) $T(n) = 3T(\frac{n}{3}) + \frac{n}{3}$

Steps:

using the Master theorem, we have:

$a = 3$

$b = 3$

$f(n) = \frac{n}{3} \longrightarrow k = 1, p = 0$

This recurrence falls under case 2 of the theorem, in which $\log_b a = k$ (since $\log_3 3 = 1$), so our complexity is of the form $O(n^k \log n) \longrightarrow O(n \log_3 n)$ (base changes since we divide the work done each recurrent step by 3 rather than the usual 2).

(f) $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n \log n$

Steps:

we can rationally assume that $T(\frac{n}{4}) \leq T(\frac{n}{2})$, and try to narrow down the runtime of $T(n)$ since it is sandwiched between the two. Both recurrences would still incur the $n \log n$ work in each call, so our inequality could look like this:

$$2T(\tfrac{n}{4}) + n \log n \leq T(n) \leq 2T(\tfrac{n}{2}) + n \log n$$

Finding the asymptotic bounds of either side lets us narrow down the bounds for $T(n)$:

(using the Master theorem for both)

LHS:

$a = 2$

$b = 4$

$f(n) = n \log n \longrightarrow k = 1, p = 1$

$\log_b a = \log_4 2 = \frac{1}{2} \longrightarrow \frac{1}{2} < 1 \longrightarrow$ case 3 of Master theorem $\longrightarrow O(n \log n)$

RHS:

$a = 2$

$b = 2$

$f(n) = n \log n \longrightarrow k = 1, p = 1$

$\log_b a = \log_2 2 = 1 = k \longrightarrow$ case 2 of Master theorem $\longrightarrow O(n \log n)$

Since $T(n)$ is thus bounded on both sides by $O(n \log n)$, it is safe to claim that $T(n) \in O(n \log n)$.

2. idea: we can compute the k^{th} smallest number in $A[1, n]$ in $O(n)$ via the median of medians selection algorithm, but cannot do so for all $k = \log_2 n$ elements as that would result in a total runtime of $O(n \log n)$. This naive approach, however, is only needed if we had to make use of the entire array after finding each k^{th}-smallest element. In this case though, after we find a k^{th}-smallest element, everything that was less than it is not useful in our search for the $(k+1)^{st}$-smallest element. So instead, we can try reducing the median of medians search space after each k^{th}-smallest element is found, by ignoring all elements that cannot be valid candidates for the next smallest element (anything smaller than the k^{th}-smallest). Since earlier powers of 2 are closer together, I instead opt to start from the last power of 2 and work backwards, pruning elements *larger* than the k^{th}-smallest, to reduce more of the search space earlier. And since median of medians yields an array partitioned around the k^{th}-smallest element, we can just continually recurse on the half of the array to find the $(k+1)^{st}$-smallest element.

   In short, the algorithm would operate as follows:

   (a) initialize i such that $k = 2^i$ is the greatest power of 2 that is $\leq n$: $i = \lfloor \log n \rfloor$

   (b) run median of medians to find the 2^i-th smallest element; the array will now be partitioned around the 2^i-th smallest element, with the left half being less than it and the right half being greater than it

   (c) decrement i and repeat steps (b) and (c) by calling median of medians on the left partition; repeat while $i \geq 0$.

   In general, this approach will find all such k^{th} smallest elements in $O(n)$ time, because during each step, we are processing approximately half the array of the previous iteration (in the general case). At any given iteration we will do $\frac{n}{2^i}$ work to find the k^{th}-smallest element, then call the function again for half the current subarray. Generalized over all iterations, we will do $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + \frac{n}{2^i} \approx 2n$ steps of total work. Depending on whether n is a power of 2, the coefficient will change, but the runtime will remain $O(n)$.

3. (a) for a set of integers in the interval $[0, n]$, roughly half of them will have a most-signifcant bit of 1, while the other half will have a most significant bit of 0. Since John's array is sorted, we know that the most significant bit of the numbers in $A[0, \frac{n}{2}]$, is 0, while the most significant bit of the numbers in $A[\frac{n}{2}, n]$ is 1. So, the process for finding the missing integer could be as follows:

      i. ask John for the most significant bit of $A[\frac{n}{2}]$; this gives us the element at the inflection point, where the most significant bit would have changed from 0 to 1. Since one element is missing however, we know that the number of integers with leading bit 0 and leading bit 1 may not be equal, so one "half" is shifted up or down by one index (if n is odd, there should be an equal number of leading 0's as 1's, and if even, there should normally be 1 more leading 0 than 1; removing a number upsets this balance). This means the element in the middle must belong to the half that *isn't* missing the integer.

      ii. Thus, if the j^{th} bit of the middle element is a 1, we know the missing integer's j^{th} bit is a 0 (and so must belong in the bottom half of the interval), and vice versa. Once we know which half of the interval to search in, we can eliminate the other half.

      iii. Once we determine the correct "half" of the interval, we can increment j (so now, we will ask John about the 2^{nd} most significant bit), and this time ask for this j^{th} bit at $A[\frac{n}{4}]$ and $A[\frac{n}{4} + 1]$ or $A[\frac{3n}{4}]$ and $A[\frac{3n}{4} + 1]$, depending on what we found in step ii.

      iv. From here, we can repeat steps i., ii., and iii. in a manner similar to binary search, by repeatedly cutting the search space in half each time and asking John about the midpoints. Each time we limit the search space, we can "fill in" the j^{th} bit of the missing number. This process will continue recursively until the $\log n^{th}$ bit is reached and its value is filled in, giving us the complete binary representation of the missing number.

      v. since the search space is cut in half after each question and the array is sorted, we only need to ask 1 question per bit to find the missing number's j^{th} bit. We need $\lceil \log n \rceil$ bits to represent numbers in the range $[1, n]$, so we only need to ask John $\lceil \log n \rceil$ total questions.

Below is a toy example of this, with numbers in the range of 0 to 6, with 1 being the missing integer. I use $\frac{n}{2}$ as the midpoint, where n is the length of the subarray. Note that all arrays here are 0-indexed, and bits are indexed from the most- to least-significant bit for simplicity. For example, $A[i][j]$ signifies the j^{th} bit (from the left) of $A[i]$.

i. John's sorted array in binary form:
A[5]: 110
A[4]: 101
A[3]: 100
A[2]: 011
A[1]: 010
A[0]: 000

ii. ask John for $A[3][0]$; it's a 1, so the missing number should be in the lower half. Thus, our currently "built" representation of the missing number is "0". The subarray to search in now is:

A[2]: 011
A[1]: 010
A[0]: 000

iii. Ask John for $A[1][1]$; it's 1, so again the missing number is in the lower half. So, we add another 0 to our missing number's representation, which is now "00". The subarray is now:

A[0]: 000

iv. We have only 1 number left, and we ask John for $A[0][2]$, which is 0, so we know the missing number's rightmost bit must be 1 (if there was an upper half, we would go there, but we've already filled in all bits after this iteration). Appending that to our current representation, we find that the missing integer is "001" = 1.

(b) Since the array is no longer sorted, we can't *automatically* eliminate half the array after each question like we did previously. This prevents us from "knowing" how many integers have a j^{th} bit of 1 or 0 unless we count each one. So, that's what we'll do: we can take the same approach as in part (a), but we would be responsible for pruning out the invalid integers manually. We can do this by first narrowing down the j^{th} bit of the missing integer with n questions, then eliminating integers that do not have the same j^{th} bit, which should be half of them. This would require $2n$ operations per iteration: n questions to find the missing number's j^{th} bit, and another n questions to find and keep any integers that have the same j^{th} bit as the missing integer. In the next iteration, we'd repeat the process on $\frac{n}{2}$ integers, then the next on $\frac{n}{4}$ integers, etc. until we are left with 1 integer.

The steps could go like this:

i. initialize an (unordered) set with all numbers from 0 to n (exclusive); this will serve to track which indices i in A will be valid.
ii. ask John for the leftmost (0^{th}) bit of all numbers in the set, counting the number of 1's and 0's. If the number of 0's is $\leq$ the number of 1's, the missing number's leftmost (0^{th}) bit is 0; else it's a 1.
iii. once again ask John for the j^{th} bit (where j is now 1) of all available numbers in the set, but remove numbers whose j^{th} bit does not match the missing number's j^{th} bit. This leaves us with half of the search space from the previous iteration.
iv. repeat steps ii. and iii. until the set contains 1 index. That index's element's rightmost bit must be the inverse of the missing number's rightmost bit, so fill it in and end the loop.

This approach involves doing $2k$ operations per iteration, where k is the current cardinality of the valid set. Expanded out, the algorithm does $2n + 2 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \ldots 2\frac{n}{2^i} \approx 4n$ total work, which is $O(n)$.

4. (a) divide and conquer ($O(n \log n)$): idea is to find max partial sums of all subarrays in the overall array, but rather than exhaustively test all subarrays, we can notice that the

max partial sum of two subarrays is either the max of each individual subarray's sum, or their combination (which crosses the midpoint). So, a simple solution is to recursively call this function on the left and right halves of the array, and the max partial sum will be the $max(f(left), f(right), f(left + right))$. The left and right half sums are computed recursively, while the midpoint sum requires iterating from the midpoint to its left and right and computing the max sum manually. See the below example using the array given in the homework problem:

$$[3, -5, 6, 7, 8, -10, 7]$$
$$[3, -5, 6, 7] \text{ — } [8, -10, 7]$$
$$[3, -5] \text{ — } [6, 7] \text{ — } [8, -10] \text{ — } [7]$$
$$[3] \text{ — } [-5] \text{ — } [6] \text{ — } [7] \text{ — } [8] \text{ — } [-10] \text{ — } [7]$$

On the way up from the base cases (single elements), the max sums (between the left, right, and across midpoint) are computed:

$$[3, -5, 6, 7, 8, -10, 7](21- \text{ from crossing midpoint})$$
$$[3, -5, 6, 7](13) \text{ — } [8, -10, 7](7)$$
$$[3, -5](-2) \text{ — } [6, 7](13) \text{ — } [8, -10](-2) \text{ — } [7](7)$$
$$[3] \text{ — } [-5] \text{ — } [6] \text{ — } [7] \text{ — } [8] \text{ — } [-10] \text{ — } [7]$$

(b) dynamic programming ($O(n)$): overall idea is to store maximum partial sums that we've encountered as we traverse through the array. For each index, we'd store the max partial sum if the ending index, k, were to be at that index, and also track the starting index, j, of each max partial sum's subarray. At each index, we will decide whether to "continue" the subarray if the current element's addition will increase the sum, or start a new subarray if the current element is already greater than whatever sum we have (i.e. the current sum was negative). If the current max partial sum is ever greater than the overall max partial sum, we update it and also update the start and end points of the subarray, $final_j$ and $final_k$.

Algorithm 1 Dynamic programming approach ($O(n)$ time with constant space)

```
j, k, final_j, final_k ← 0
max_total ← −∞
max_curr ← 0
N ← length(A)
for i ← 0 to N do
    if A[i] > max_curr + A[i] then
        j ← i
        k ← i + 1
        max_curr ← A[i]
    else
        k ← i
        max_curr ← max_curr + A[i]
        if max_curr > max_total then
            final_j ← j
            final_k ← k
            max_total ← max_curr
        end if
    end if
end for
```

5. idea: If the two sums X and Y are unequal but can be made equal by swapping their addends, it should be the case that one sum has an addend that "over-contributes" to it while the other sum has an addend that "under-contributes", causing the difference between the sums. If so, their sums can only be made equal if the two are swapped, and the difference between the sums must be an even number (if it is odd, swapping any of the numbers would still result in an unequal sum). So, our goal is to find the pair of numbers that contribute "evenly" to the difference in the two sums; that is, their difference must be equal to $\frac{1}{2}$ the difference between X and Y. Thus, when they're swapped, the lesser sum between X and Y increases by $\frac{d}{2}$ and the greater sum decreases by $\frac{d}{2}$ (where $d = |X - Y|$ and d is even), thus making them equal.

Analysis: My approach uses a hash map to quickly retrieve elements in X that satisfy $X_i + Y_j = \frac{d}{2}$ for some i, j in constant rather than linear time, allowing us to find suitable pairs in amortized $O(max(n, m))$ time instead of $O(n \cdot m)$. The total runtime is amortized $O(max(n, m))$ since the addends of both X and Y are iterated over (either to populate the hashmap or to find a suitable Y_j). Of course, worst case insertion into a hashmap is $O(n)$, taking the total worst-case runtime to $O(n^2)$, which is only seen in the event of many hash collisions caused by a poor hash function. The total space complexity is $O(n)$, since every addend in X is mapped to its index via the hashmap.

Algorithm 2 Hash map pair checking $O(max(n, m))$ time with $O(n)$ space

```
diff ← X − Y
if diff %2 ≠ 0 then
    return null
end if
diff ← diff/2
diffMap ← HashMap{x_i → i | i ∈ 1...n}

for j ← 0 to m do
    if diff + y_j ∈ diffMap then
        i ← diffMap[diff + y_j]
        return (i, j)
    end if
end for
```

6. idea: since the rows and columns are sorted, we can eliminate entire sections of the grid if we find an element is greater than or lesser than the target to be found. In the case where an element of the matrix $A[i][j]$ is greater than the target, we can eliminate cells with $row > i$ and $col > j$, since all of those elements will also be greater than the target. We'd then increment the row i and try again. The inverse applies if the current element is less than the target; we'd instead eliminate all cells with $row < i$ and $col < j$, since the target is larger than them. We'd then decrement the column j and try again. If $A[i][j]$ is ever equal to the target, we've found it; else, if the loop ends, the target must not exist in the matrix. We can start by initializing the current cell to the top right corner of the matrix. After each iteration, the current cell will move either down or to the left, eliminating rows and columns as we go along. Thus, at worst, we will have to cover $n + n$ cells, giving us a runtime of $O(n)$.

   This algorithm is demonstrated more explicitly in algorithm 3.

Algorithm 3 Iterative elimination approach ($O(n)$ time with constant space)

```
N ← length(A)
i ← 0
j ← N − 1
while i < N and j ≤ 0 do
    if target = A[i][j] then
        return (i, j)
    else
        if target > A[i][j] then
            j ← j − 1
        else
            i ← i + 1
        end if
    end if
end while
```

   To prove that we cannot achieve an $o(n)$ runtime, we can disprove the assumption that an $o(n)$ algorithm exists for this problem. To do that, we can observe that we also cannot solve a subset of this problem in $o(n)$ time.

   In our grid, the bottom-to-top diagonal may not necessarily be sorted. This means that the diagonal can be treated as an unsorted 1-D array, B. We know that to search for a target value in B, an optimal algorithm would need to traverse all n elements in B since B is unsorted, taking $\Omega(n)$ time.

The contradictory claim we make would be as follows: assume an algorithm S exists that can find a target element in the 2-D array (as described in the problem statement) in $o(n)$ time. Now construct a matrix A' such that its diagonal would be an unsorted list, like B. Assume the search target x is in the diagonal, and thus all cells above the diagonal would be strictly less than x, while all cells below the diagonal would be strictly greater than x.

If S were to find x in A' (the matrix with B as its diagonal), it follows that S would also find x in B (the unsorted 1-D array) as well, and if x is not found in A', it must also not be in B. This transforms the 2-D search into an indirect 1-D search. This means that, if S were to find x in A' in $o(n)$ time, it must also have found x in B in $o(n)$ time. In other words, it has found an element in an unsorted array without needing to process all n elements in the array (in the worst case), which is impossible. Since this is contradictory, our assumption that S exists and can find the target in $o(n)$ time is also false.

7. (a) From viewing the blocks in a k-sorted array, it's evident that there is a partition-like structure in the blocks, since the elements in each block are greater than those of previous blocks and less than those of later blocks. We can thus try partitioning the array recursively until we reach depth $\log k$, at which point the combined subarrays will be k-sorted. We know it will be k-sorted because the pivots and subarray bounds used in each partitioning call would serve as the "boundaries" between blocks as seen in the example, and by definition these $k-1$ pivots will have lesser elements to their left and greater elements to their right. The recursion tree would resemble a binary search tree, and our k-sorted array would simply be a concatenation of the "leaves" (which are subarrays) at the $\log k$-th level.

   Here's how this might work with an example. The goal is to 4-sort the array [7, 4, 12, 15, 9, 1, 6, 11, 2, 10, 5, 3, 14, 13, 16]. It should be done in $\log 4 = 2$ steps of partitioning:

   i. original array ($k = 0$): [7, 4, 12, 15, 9, 1, 6, 11, 2, 10, 5, 3, 14, 13, 16]

   ii. ($k = 1$) 1st step of median of medians (MoM) partitioning: ($M = [9, 6, 13]$, MoM is 9). After partitioning, we have:

   [7, 4, 1, 6, 2, 5, 3, 9, 12, 15, 11, 10, 14, 13, 16]

   iii. recursively call on left and right halves of partition (of course, include the partition in one of the halves)

   iv. ($k = 2$) 2nd step of MoM partitioning on the halves:

   left half: [7, 4, 1, 6, 2, 5, 3], $M = [4, 5]$, MoM is 5. After partitioning, we have:

   [4, 1, 2, 3, 5, 7, 6]

   right half: [9, 12, 15, 11, 10, 14, 13, 16], $M = [11, 14]$, MoM is 14. After partitioning, we have:

   [9, 12, 11, 10, 13, 14, 15, 16]

   v. when combined (returning up the call stack), the array is now k-sorted:

   [4, 1, 2, 3 |5, 7, 6 |9, 12, 11, 10 |13, 14, 15, 16]

   We are continually dividing the array in half around the MoM pivot, and the MoM partitioning happens in $O(m)$ time, where m is the number of elements in the subarray. We arrive at the k-sorted array in $\log k$ steps, and each step incurs $O(n)$ total work for partitioning (since we are technically processing every element in the array at some point in the recursive level), so our final runtime is $O(n \log k)$.

   (b) We can prove this using a proof by contradiction. Our claim will be that we can k-sort any array with less than $n \log k$ comparisons. In the worst case, $k = n$ which means the array is fully sorted (each block is of size 1). And, to fully sort any array via comparison, we know that optimal algorithms like mergesort require $n \log n$ comparisons. This is a contradiction, since we claimed to be able to k-sort with less than $n \log k$ comparisons (and $k = n$).

   (c) Since the array is k-sorted, we know that the elements in each block are almost where they should be when fully-sorted. For instance, in the problem statement's example, the 1 in the first block can be *at most* $\frac{n}{k} - 1$ indices away from its true index in the fully sorted array. We can take advantage of this fact, since it means we do not need

to sort the entire array as a whole; we can just sort each block individually, and the combined sorted blocks would be sorted, since each block is already in the right place relative to the other blocks by definition of k-sorting.

We can use an optimal comparison-based sorting algorithm that runs in $O(n \log n)$ time to do the sorting within the blocks. To sort a single block, we would thus take $O(\frac{n}{k} \log (\frac{n}{k}))$ time. And since we must run this for all k blocks, our total complexity would be $O(k \cdot \frac{n}{k} \log (\frac{n}{k})) = O(n \log(\frac{n}{k}))$.

8. As mentioned, the naïve implementation will always run in $O(n^2)$ time, since every bolt is exhaustively compared with every nut. However, that approach doesn't take advantage of the fact that after we do a series of n comparisons for a bolt, we have a relative idea of which nuts are smaller than that bolt's nut, and which nuts are larger. This allows us to eliminate smaller nuts as we search for the largest, and likewise for the bolts. The algorithm would go something like this:

   (a) pick a random bolt or nut from the pile as a reference piece, and test it against the nuts or bolts, respectively (I'll henceforth refer to the reference piece as the main piece, and the other as the complementing piece; for example, if the main piece is a bolt, the counterpiece would be the nut)

   (b) if you encounter a counterpiece that is larger than the main piece, discard the main piece and make the counterpiece the new main piece. If the counterpiece is smaller, discard it and continue. If they match, keep the counterpiece to the side (as you may have found the largest matching pair), and continue comparing with the main piece

   (c) repeat (b) until there are no remaining nuts or bolts; the main piece and it's counterpiece should now be the largest nut and bolt pair

   As a small example, consider a pile of 4 nuts and 4 bolts, with sizes $[1, 4]$ (the bolt number will correspond to its size for simplicity). Let B be the set of available bolts, N be the set of available nuts, and S be the bolts/nuts left to the side or in hand. Walking through the algorithm:

   (a) pick up a bolt, say bolt 2; our two sets are now $B = \{1, 3, 4\}, N = \{1, 2, 3, 4\}, S = \{B_2\}$

   (b) pick up a nut, i.e. nut 1. nut 1 is smaller than the nut that would fit bolt 2, so discard nut 1; $B = \{1, 3, 4\}, N = \{2, 3, 4\}, S = \{B_2\}$

   (c) pick up another nut, i.e. nut 3. After comparing, we see nut 3 is larger than bolt 2, so discard bolt 2; $B = \{1, 3, 4\}, N = \{2, 4\}, S = \{N_3\}$

   (d) pick up a bolt, say bolt 3. It matches nut 3, so keep nut 3 to the side and continue comparing with bolt 3; $B = \{1, 4\}, N = \{2, 4\}, S = \{B_3, N_3\}$

   (e) pick up a nut, i.e. nut 4. It's larger than bolt 3 so discard bolt 3 and nut 3 and continue. $B = \{1, 4\}, N = \{2\}, S = \{N_4\}$

   (f) pick up a bolt, i.e. bolt 4. It matches so keep nut 4 to the side and continue; $B = \{1\}, N = \{2\}, S = \{B_4, N_4\}$

   (g) pick up a nut, i.e. nut 2. It's smaller than bolt 4 so discard it. $B = \{1\}, N = \{\}, S = \{B_4, N_4\}$

   (h) we have run out of nuts to compare, so the nut we have in side must be the largest nut. The bolt we have in side matches it, so it must also be the largest bolt.