**Divide and Conquer**

**Table of Contents**

[Interval Scheduling 4](#_Toc63000979)

[Greedy Approach 4](#_Toc63000980)

[Interval Scheduling with Weights 8](#_Toc63000981)

[Interval Scheduling with Multiple Resources 11](#_Toc63000982)

[Convex Hull 12](#_Toc63000983)

[Problem Definition 12](#_Toc63000984)

[Brute Force Approach 13](#_Toc63000985)

[Divide and Conquer Approach 14](#_Toc63000986)

[Matrix Multiplication 19](#_Toc63000987)

[Divide and Conquer Approach 20](#_Toc63000988)

[Merge Sort 23](#_Toc63000989)

[Sorting 23](#_Toc63000990)

[Merge Sort Algorithm 24](#_Toc63000991)

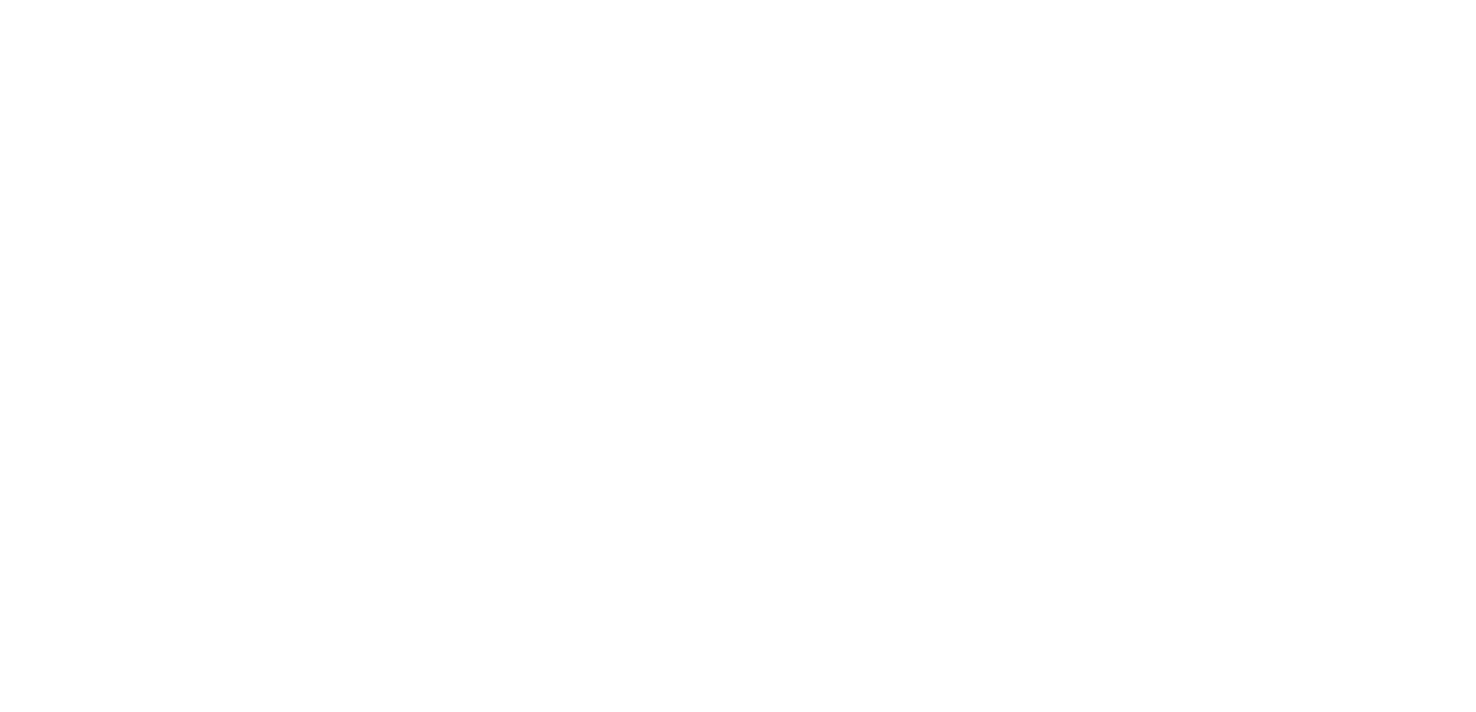
[Time Complexity 25](#_Toc63000992)

We have already seen a few Divide and Conquer algorithms, such as merge sort, even if we did not recognize them as such. The first topic we will cover under this module is Interval Scheduling.

## Interval Scheduling

Consider that we have a single resource, such as a printer at an office, and we have requests to use that resource.

Consider this diagram:



Each request has a start time and an end time , and obviously . Two requests can be considered compatible if their times do not overlap, i.e. .

Our goal is to find the compatible subset of requests that has the maximum size.

### Greedy Approach

We can solve this perform using a greedy algorithm. A greedy solution is a myopic, or short-sighted, solution. It does not look ahead, but instead process the input one piece at a time, without considering future cases. This is in direct contrast to dynamic programming. This leads to some efficient solutions, since we are not having to consider all possible solutions. We can just work off of the current solution to get the best overall solution.

We will follow a rule to choose a request . Then we will reject all requests that are incompatible with . Finally, we will repeat this process until all the requests are processed. This is the general idea behind every greedy solution.

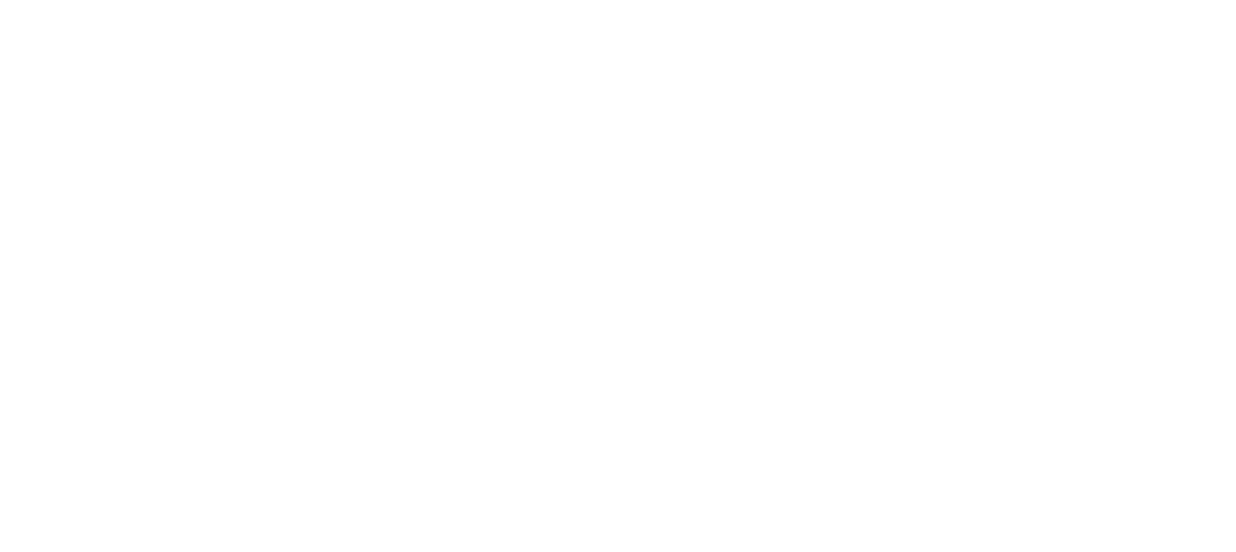
#### Proof

Before going to the actual algorithm, let’s prove that a greedy approach works. Our claim is that this approach will give us a certain list of intervals, such that . Essentially, every task will be compatible with the one after it.

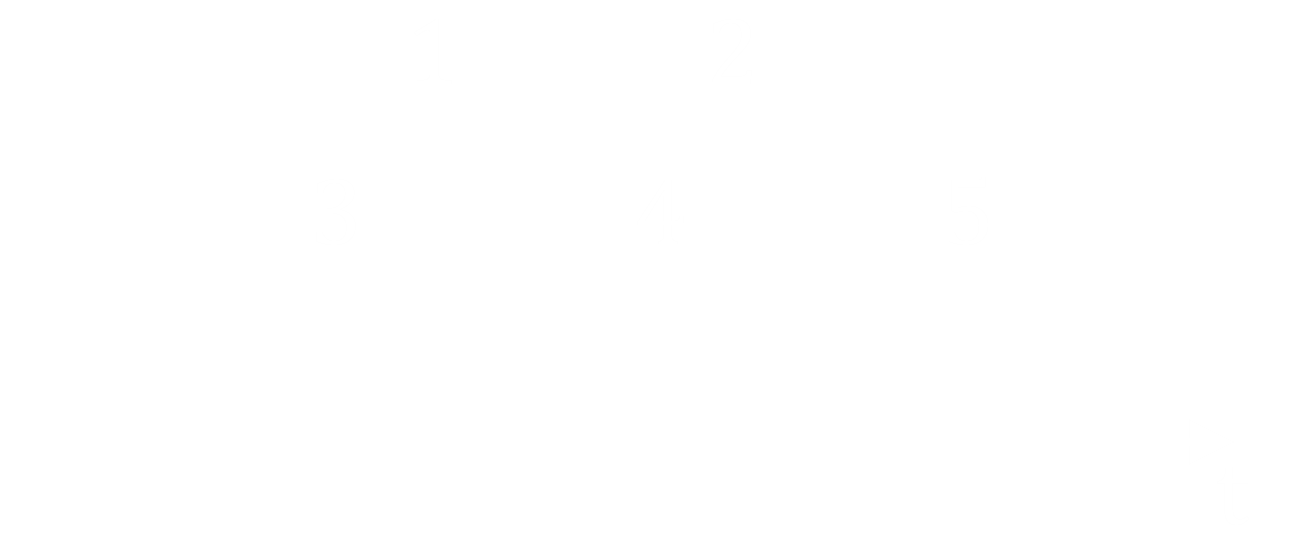
We shall prove this is true using a contradiction. If the given statement is not true, then there is some task for which . Thus, the intervals for and must overlap. However, selecting would reject all items that are incompatible with it due to the way our algorithm works, so this case cannot be true. We have a contradiction. Since it cannot be true, the given statement has to be true.

#### Rules

☒ Selecting the request with the earliest start time – Consider the diagram below. Using this rule would cause us to pick the first level instead of the second.



☒ Selecting the request with the smallest interval – Consider the diagram below. Using this rule would cause us to pick the first level instead of the second.



☒ Selecting the request which causes the minimum number of requests to become incompatible – Consider the diagram below. Using this rule would cause us to pick the second level, instead of the first.



☑ Selecting the request with the earliest finish time – Using this rule always works. It also leaves the maximum amount of time left for the rest of the tasks.

Under this last rule, take a look at the first diagram again. The item that finishes first is , so we pick it. finishes next, but it is incompatible with , so we ignore it. finishes next, and since it is compatible, we pick it. and finish after this, but they are both incompatible and hence ignored. Finally, finishes and is compatible, so it is picked.

We can prove that the last rule works. Our claim is that, given a list of intervals, the greedy algorithm proposed, which takes the items with the earliest finish times, removes the incompatible requests, and keeps processing until it has run out of requests, would produce intervals, where is maximum.

Say the greedy approach gives us a set of intervals, and that the optimal solution is a set . The optimal solution has to have at least as many items in it that does, since otherwise, it would not be the optimal solution. Thus, . Our goal is to prove that in fact, , i.e. is one of the optimal solutions.

Say for some arbitrary point , all items up to the point from both sets and are the same. This is just to make the image a little clearer. could in fact be the first item or even the last item. Thus, and . Thus, is the first point at which there is a difference in the sets.

We know that . We know this because the greedy algorithm always picks the item with the least finish time first. If had an earlier finish time than , it would have been picked instead, since every item before this point is the same. Thus, we can replace in with . This will not cause any conflicts and it will not change the size of .

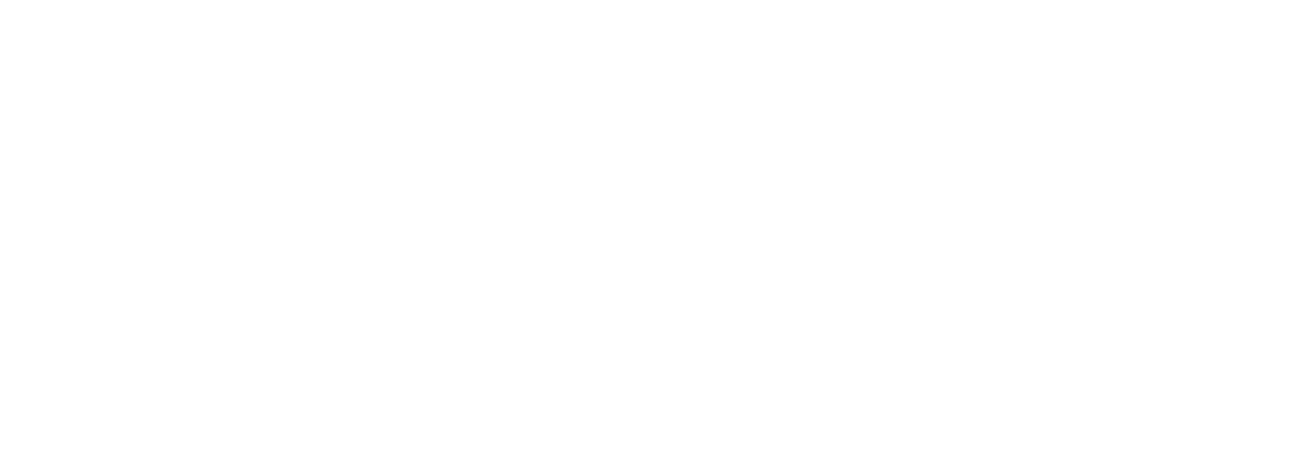
After this, we can continue to make the same argument for every item until . Thus, . Now for the next part. If is smaller than , i.e. is not an optimal solution, then there are more items left in , represented by the here. However, the greedy algorithm would keep going if there were more non-conflicting items left. As such, it is not possible for any more unexchanged items to be left in . Thus, .

This proves that is one of the optimal solutions. Of course, there may be items that have the same interval length as some item in and can thus be used instead, meaning there can be other optimal sets, but regardless, is still one of the optimal sets.

### Interval Scheduling with Weights

Let’s make a small change to the interval scheduling problem. Say every request has a weight attached to it, which denotes its priority. Now our goal is to select the subset of compatible requests that maximizes the total weight.

The greedy algorithm we created will no longer work here. Consider the diagram below:



The algorithm we created would pick the three requests on the lower level, whereas the correct answer is the two requests on the top level.

In fact, the greedy approach cannot be used at all. We cannot devise a simple rule which does not require us to look towards what might happen in the future in this case. As such, we will need to rely on dynamic programming to solve this.

#### Subproblem Definition

In this case, the subproblem can be defined as

.

#### Relating Subproblems

Here, . Essentially, we are adding the weight of the current task to the maximum value obtained from the subproblem of all other tasks that start after this task finishes, i.e. there is no overlap. Since every subproblem relies on subproblems that come after it, the dependency graph will by acyclic.

Note that we are having to take the maximum value obtained by starting from all other possible starting points, i.e. every possible task, not just the task that comes immediately after the current one. Each subproblem relies on every single subproblem that comes after it.

#### Base Case

If is the last task, then .

#### Complete Solution

The complete solution requires us to check the maximum possible value by starting from each of the tasks, i.e. . This follows from the same explanation given while discussing the relation between subproblems.

#### Time Complexity

We have subproblems, and each subproblem will potentially have to loop over every single subproblem other than itself, which is approximately . Thus, the time complexity is .

#### Better Approach

There is another approach that can be used with dynamic programming that gives even better results.

Say we sort the tasks by start time. Using a fast sorting algorithm like merge sort to do this would ensure that the sorting has a time complexity of .

Next, we can loop over this sorted list. At each task, we can decide between either keeping it or not keeping it. One of these two choices will give us the best results. If we keep it, we need to keep looping over items and skipping them until we find the next non-conflicting item. If we do not keep it, we can just move onto the next item. Thus,

As a result of using this algorithm, we will only be looping over the list a single time. Thus, the total time complexity, including the initial sorting, is .

The reason we ended up back at dynamic programming while studying divide and conquer is because dynamic programming is actually a divide and conquer approach. The only difference is that general divide and conquer solutions are not as smart as dynamic programming. In dynamic programming, we are dividing the main problem into subproblems and solving the subproblems, just like divide and conquer, but we are also saving the results of each subproblem, so that we do not have to repeat anything. This last step is missing from general divide and conquer solutions.

### Interval Scheduling with Multiple Resources

Let’s make one more change to the interval scheduling problem. Up till now, we had one resource that all the tasks were competing for. Now, let’s say we have multiple resources. If all the resources are identical, for the unweighted case, the greedy approach will still work and for the weighted case, the dynamic programming approach will still work. Thus, for this problem, we will consider that the resources are not identical.

Say we have a set of resources , and that all our tasks have the same weight. Along with a start and end time, the tasks will also have a set of demands, , which tells us which of the elements of are required by the task.

Say we are given a number and asked if requests can be served or not. This problem has no polynomial time solution. The solution is an exponential time solution, and it is considered an NP Complete solution. What an NP Complete solution is, is a topic for another day.

The question of the maximum number of requests that can be scheduled is considered an NP-Hard problem, which is again, a topic for another day. However, this solution is also an exponential or worse solution, with no polynomial time solution possible.

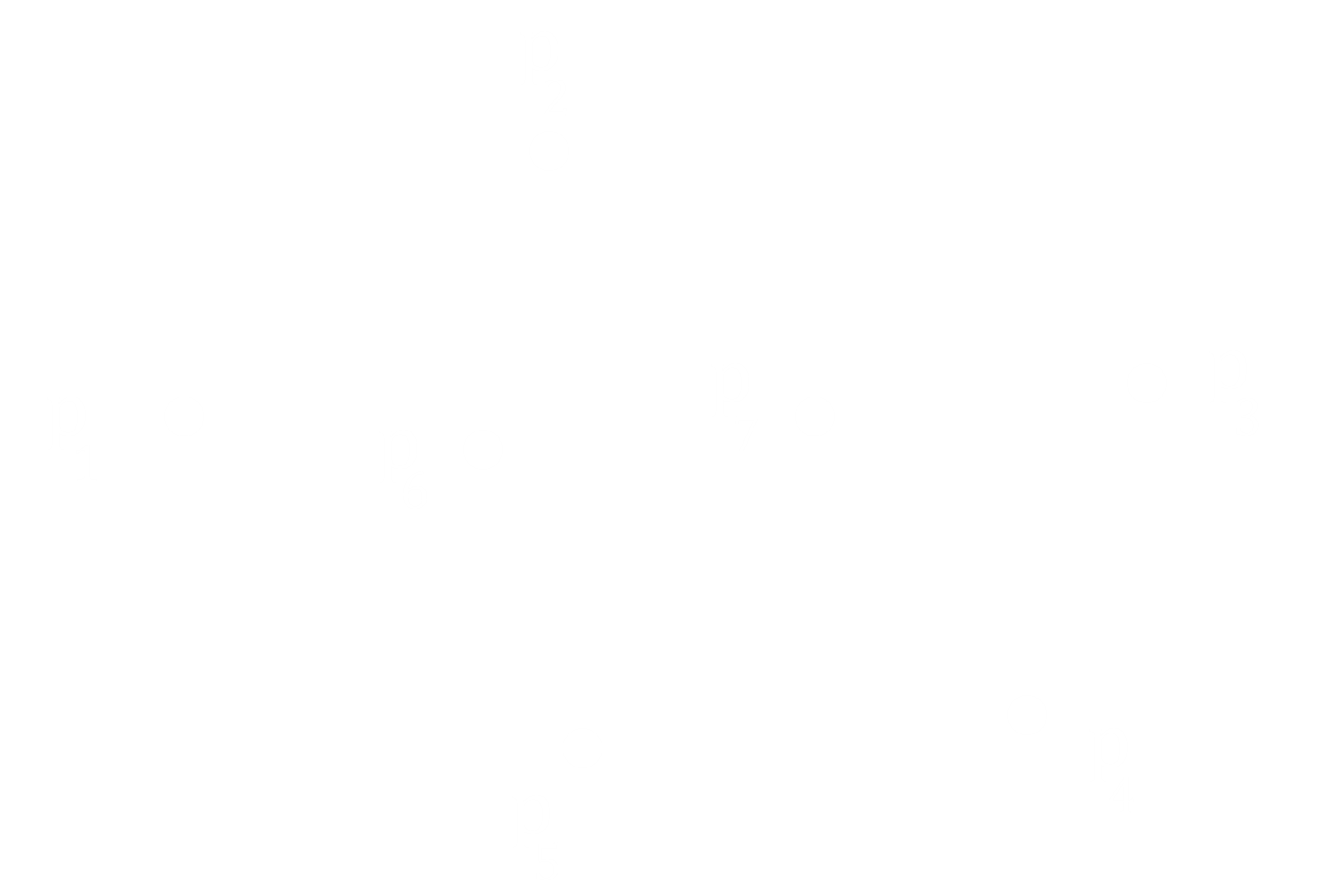
## Convex Hull

In divide and conquer, we are given a problem of size , and we divide it into subproblems, each with size , where and . We want to solve each subproblem recursively, and then combine the solutions of the subproblems to get the original solution. Thus, .

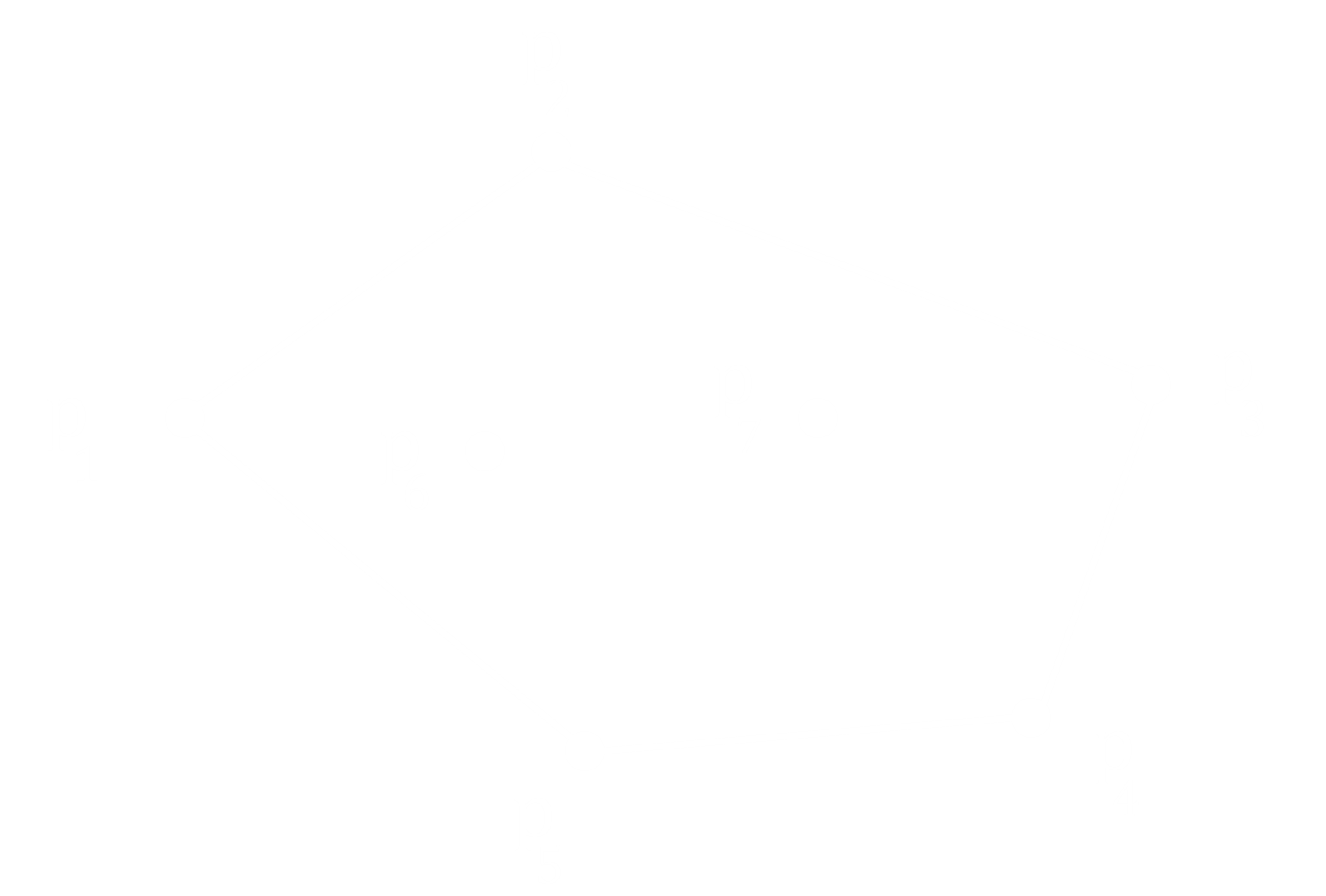
In general, we devise a brute force solution, say with a time complexity of . Then we divide the problem into smaller parts recursively until we get subproblems that are very small, such as or . At such a small scale, the brute force solution we created will not be a problem. Alternatively, we will keep dividing until we have a trivial case on our hands from which we can return the answer for the subproblem directly.

### Problem Definition

For the Convex Hull problem, we are given a set of points on a plane, where , and are the coordinates of the points. We can assume that none of the points will have the same coordinates or coordinates and none of the points will sit on a line.



Our goal is to find the smallest convex polygon that contains all the points in , .

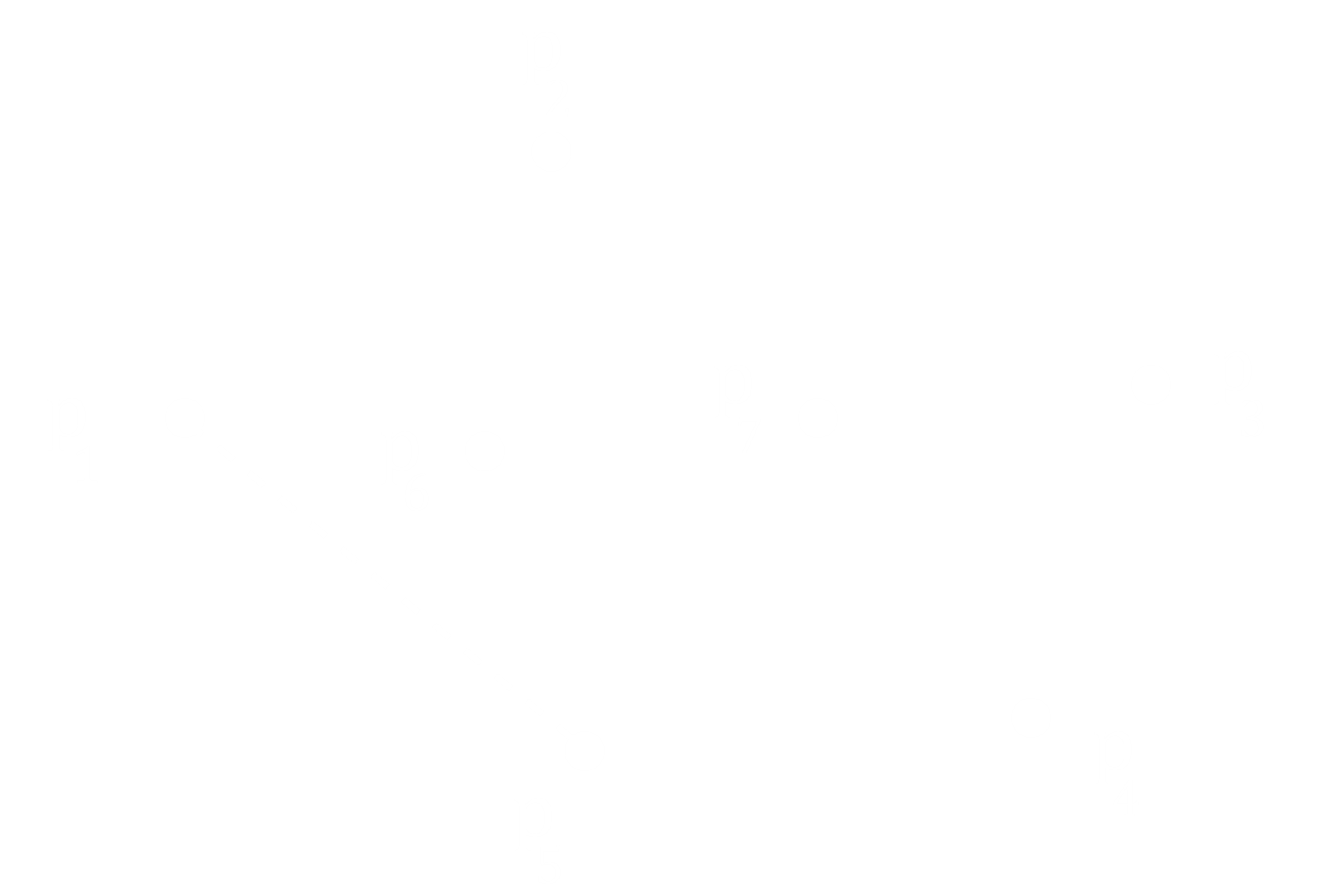


The fact that the polygon is convex tells us that the internal angles will all be less than .

Specifically, we want to find the sequence of boundary points. For the above diagram, this sequence is , , , , . The points should be in clockwise order, and the list should be considered a doubly linked list.

### Brute Force Approach

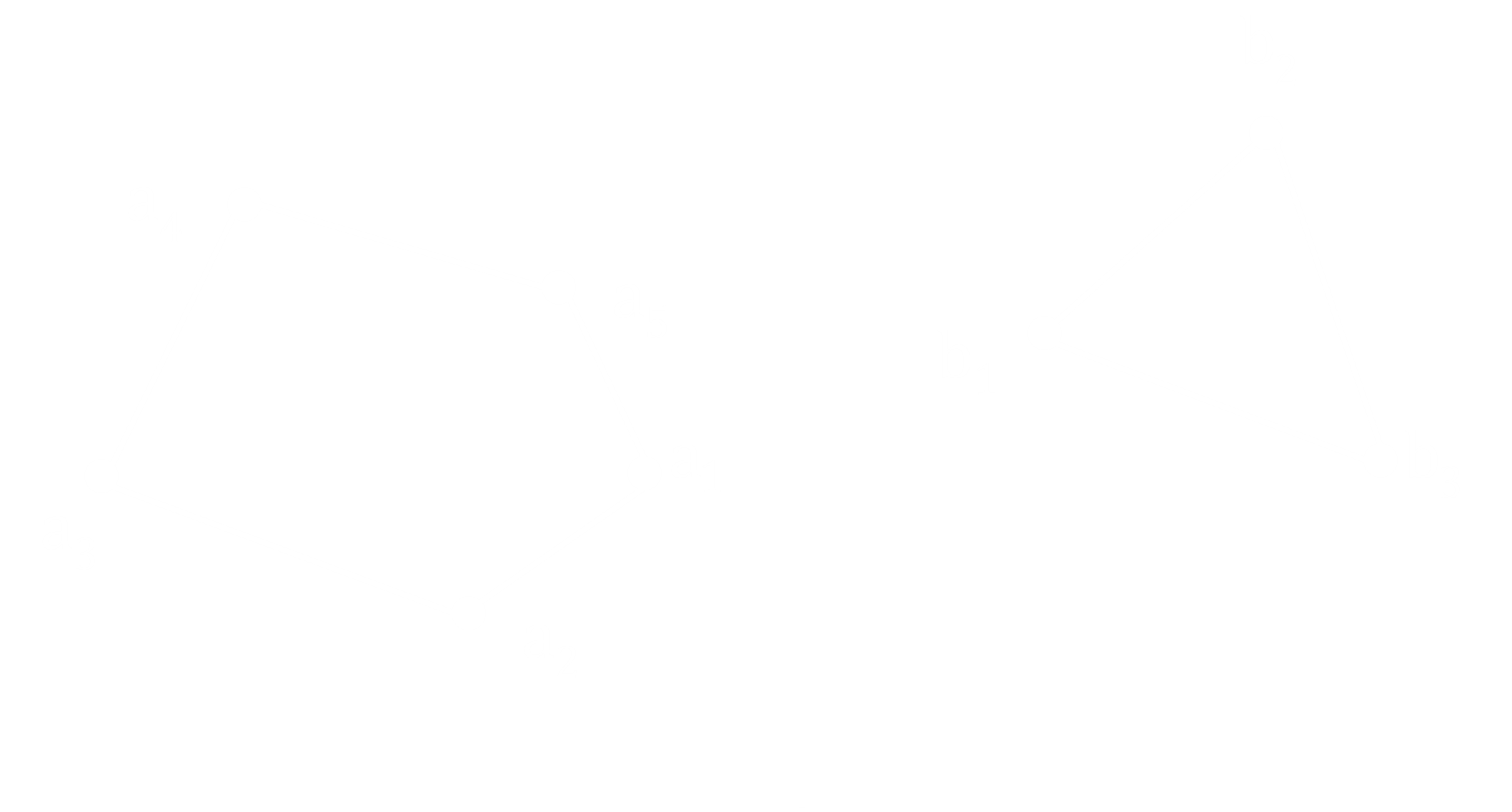
One possible solution is to create line segments. We can create a line segment between two points, and if all the other points are on one side of the line segment, then this line segment is part of the convex hull. If there are points on both sides of the line segment, it cannot be part of the convex hull.



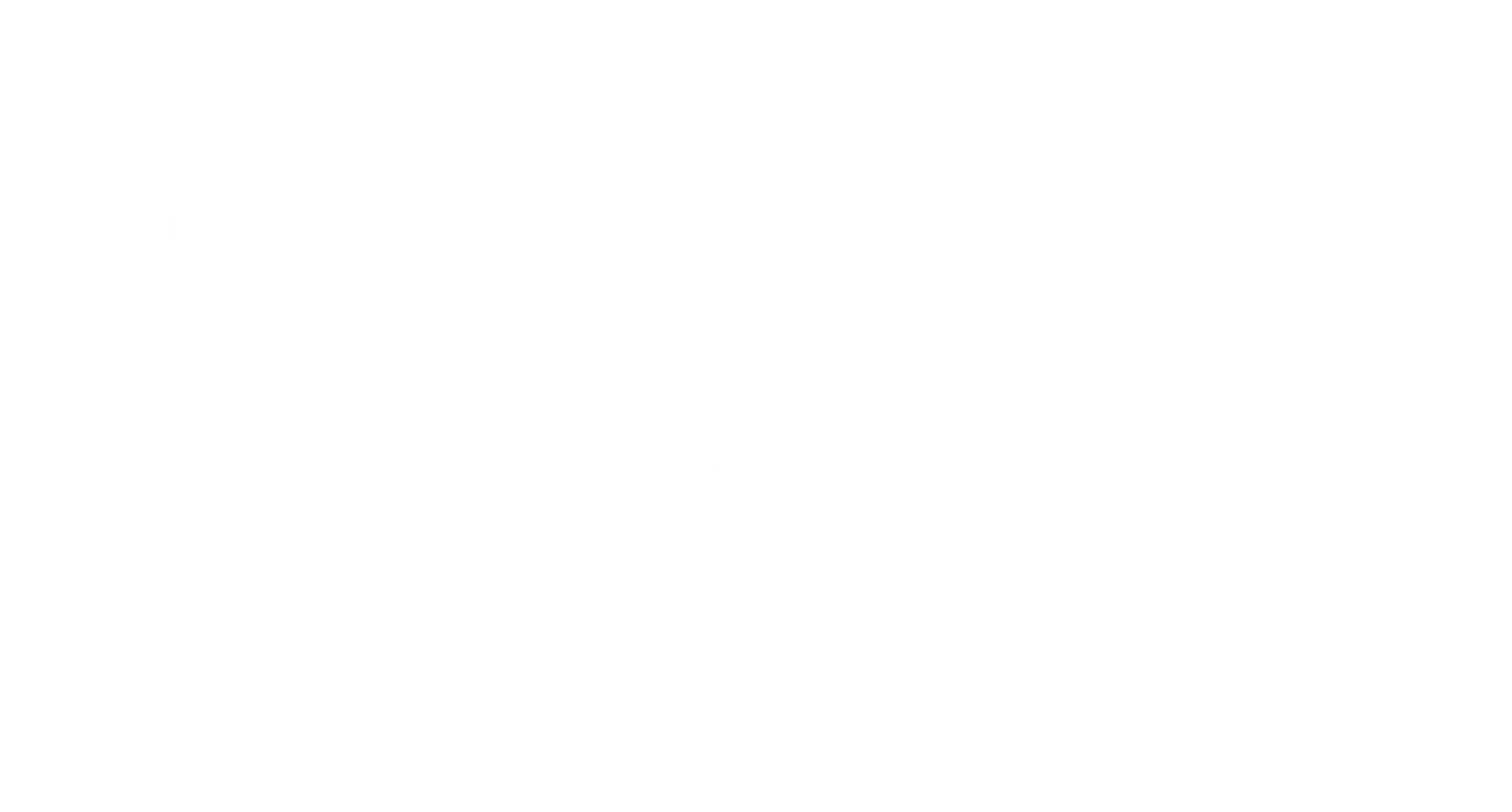
For this approach, if we have points, there are possible pairs. For each pair, we have to check all the other points. Thus, the time complexity is .

### Divide and Conquer Approach

The first thing we will do is sort the given set based on the -coordinates of the points. Next, we will divide the points into two equal halves, a left half and a right half based on those -coordinates, and calculate and . Finally, we will combine the two results. We keep dividing the points recursively until we reach a small number of points, say or or even . At that point, we will use the brute force approach. For or points, we will have a trivial case.

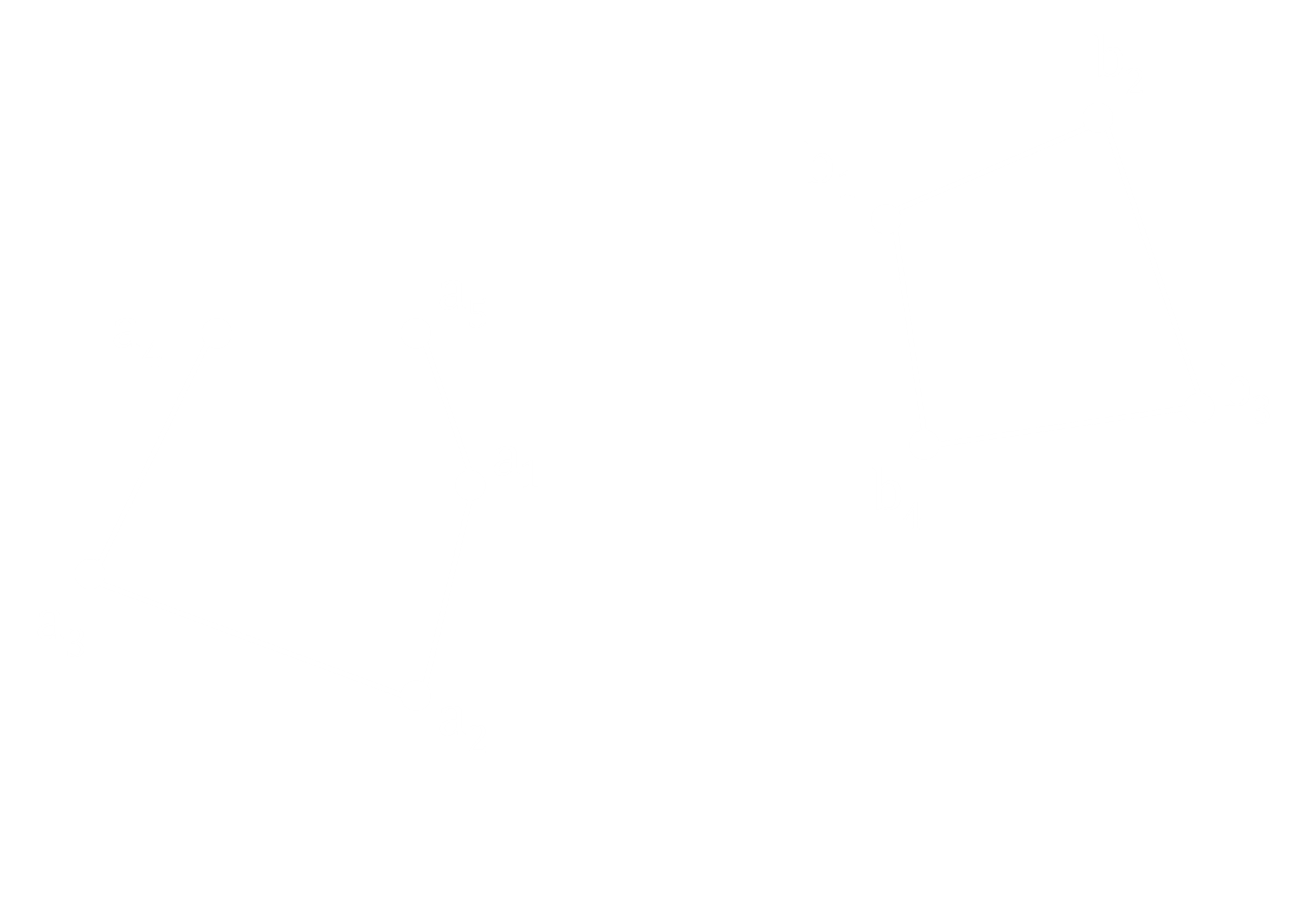


The interesting part is how we will combine two results. To do this, we can generate pairwise line segments and check if that line segment is on the combined convex hull or not. For the above diagram, we would be checking , , and so on. We need to find just two line segments to connect the two parts. For this diagram, the lines we get are , the upper tangent, and , the lower tangent. The upper tangent will cross the intersection line at the highest point, , and the lower tangent will cross the intersection line at the lowest point, .



Since we are checking every possible pair, the time complexity is . However, it is possible to reduce this.

At first sight, it may seem like finding the upper or lower tangent is as simple as joining the points with the maximum or minimum values on each half. However, this is not always the case. Consider this diagram:



If we were to join and here, we would be entirely wrong.

Instead, we use the Two-Finger Algorithm.

#### Two-Finger Algorithm

The two-finger algorithm essentially attempts to maximize , the point of intersection between a point from the left half and a point from the right half, and the partitioning line. This partitioning line is not special, and is arbitrarily set between the two halves.

For the left half, we arrange the points such that the rightmost points come first, and for the right half, we arrange the points such that the leftmost points come first. After that, we begin to join points and try to find the maximum . For the left half, we move counter clockwise, and for the right half, we move clockwise.

The first line we create is between and . Next, we try and , but we find that the intersection point we get is worse, i.e. lower. As such, we stop travelling clockwise for the right half. The next attempts go in the following order:

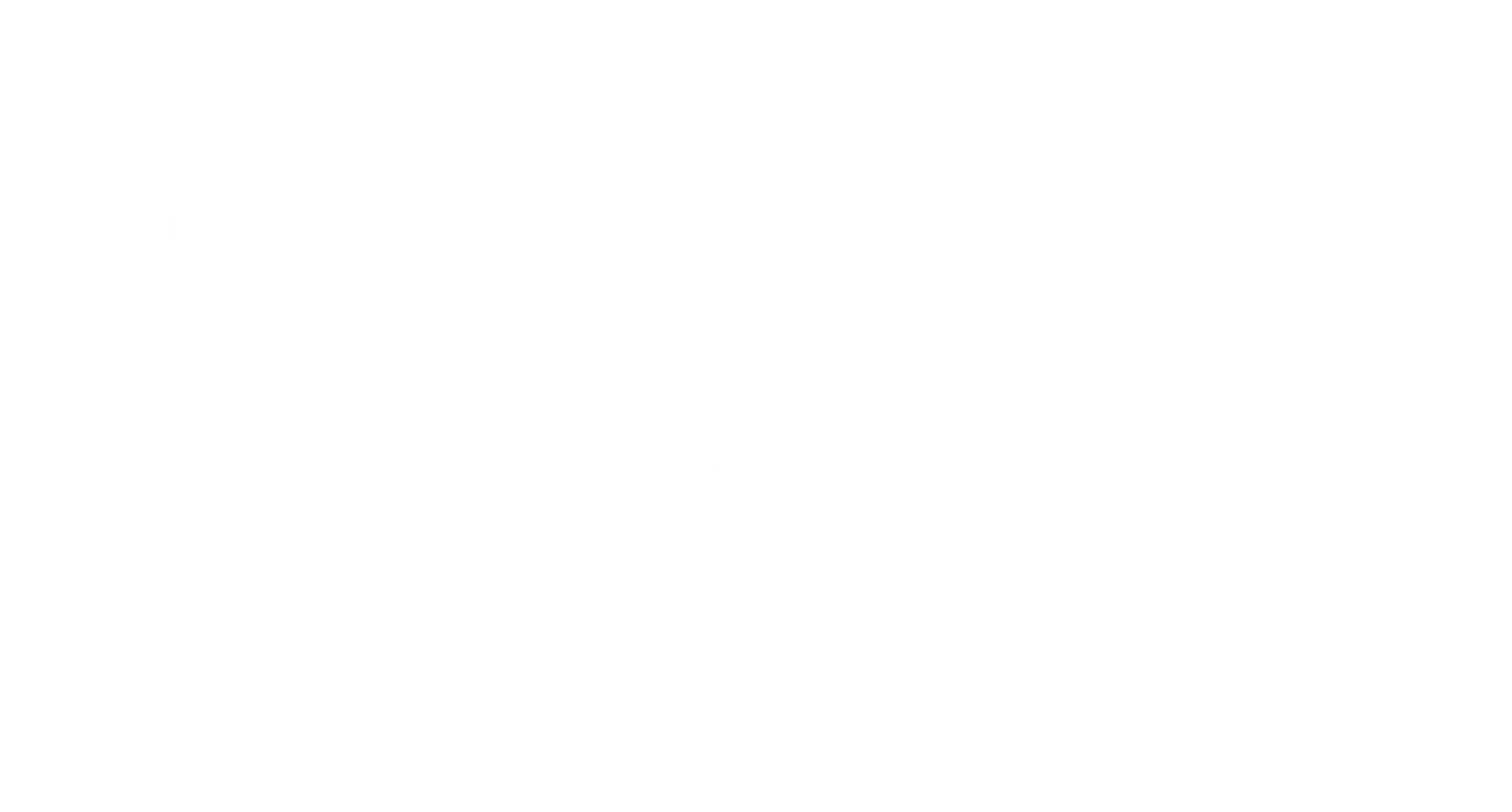
* and – improvement
* and – worse, so stop
* and – slight improvement
* and – worse, so stop
* and – worse, so stop

We then repeat similar steps for the lower tangent.

We have moved along the points on either half in a nested iteration, albeit in a smart manner, by stopping as soon as things started to get worse. As such, we will never have to loop over all possible pairs. In the worst case, we will have a time complexity of .

The proof of this algorithm is beyond the scope of this course.

#### Merging



Once we have found the upper and lower tangents, we need to merge the two lists. This is done with the Cut and Paste method. Essentially, we start at the left side of the upper tangent and go to the right side, i.e. , . After this, we keep moving clockwise along the points of the right side until we reach the right side of the lower tangent, . Finally, we go the left side of the lower tangent, , and travel clockwise along the points of the left side, which just means visiting here, until we reach the left side of the upper tangent again.

Thus, for the list of points and , we start at , go to , travel till , go back to , and travel till . The final resulting list is .

The complexity of merging the two lists is , since we only traverse the points.

#### Time Complexity

The first thing we need to acknowledge is all the sorting we’ve been doing. If we use the merge sort algorithm, that would give us a complexity of , and if we use something even faster, like radix sort, we can have a complexity of . These two sorting algorithms will be covered shortly.

Next, we divide the given list by choosing a partition at the centre. The time complexity of the set with elements depends on the time complexity of the two halves, i.e. .

In the merging section, the Two Finger algorithm has a time complexity of , as does the Cut and Paste method. We can ignore the brute force cases, since we only use that for very small lists.

Combining all of this, we get:

There is something called the master method, which we have not covered, which essentially proves that the given time complexity is the same as the time complexity for merge sort. Thus, .

## Matrix Multiplication

When studying dynamic programming, we saw that the order in which we multiply matrices makes a difference to the overall number of smaller multiplications we have to perform. We also saw an approach to finding the optimal order. Now, we will study a method to reducing the cost of multiplications for the multiplication of two matrices. So, we already know how to find the best order in which to multiply a bunch of matrices, and now we want to find the fastest way to multiply two matrices.

Say we want to multiply two matrices, and . For simplicity, let’s consider them both to be square matrices with elements each. Thus, for a single entry in the resultant matrix, , we need to perform multiplications, and sum the results of each i.e. .

Since both and are matrices, will also be an matrix, meaning there will be a total of entries. Thus, the total complexity is .

That’s … well, shit. We need a better method. The reason we’re getting so uppity about this stuff is because matrix multiplications are super useful in linear algebra and deep learning, which is involved in everything these days.

### Divide and Conquer Approach

Let’s assume that . Even if it is not, then we can append s to the matrices to make it so, such as adding columns and rows to a matrix to make it a matrix. This will not affect the multiplications.

Next, we divide the matrix intro four matrices. We continue dividing the matrices in this manner into smaller and smaller parts until we get single elements. Next, we will use brute force to multiply those single elements and then combine the results.

Say and . To find , we need to multiply with and with . Of course, each of these elements could themselves be matrices, which we would need to divide into even smaller parts. The four formulae we need here are:

Notice that to find , we need multiplications of . We also need additions. Each addition makes us matrices, so there are a total of additions.

Thus, the total run time comes to . Again, the details of how this is correct has not been covered as of yet, but if you’re reading this a few months down the line I’m sure you agree with me.

#### Stassen’s Method

In computer science, multiplications are always more costly than additions. We can try to reduce the time complexity we just found by performing more additions and fewer multiplications.

Stassen found a method to perform one less multiplication by performing a bunch of more additions. He suggested the seven following multiplications:

There are lot of subtractions here, but that doesn’t matter because that’s the same as negating the matrix and adding it.

In each matrix, there is only one multiplication. Finally, we can find the original answer in this manner:

Each of these are correct, but as proof, consider the second one.

Thus, we are getting the exact same results we were before, but by performing fewer multiplications.

The time complexity for all of this is:

Again, the exact method that proves how this equation is correct will be taught later.

This tiny improvement may feel like it is not worth all the work we had to do achieve it, but for extremely large values of , even this small change can make a huge difference.

## Merge Sort

### Sorting

Given an array of elements, we want to find a permutation of the elements such that . We have assumed that the order is ascending, but it could be descending too.

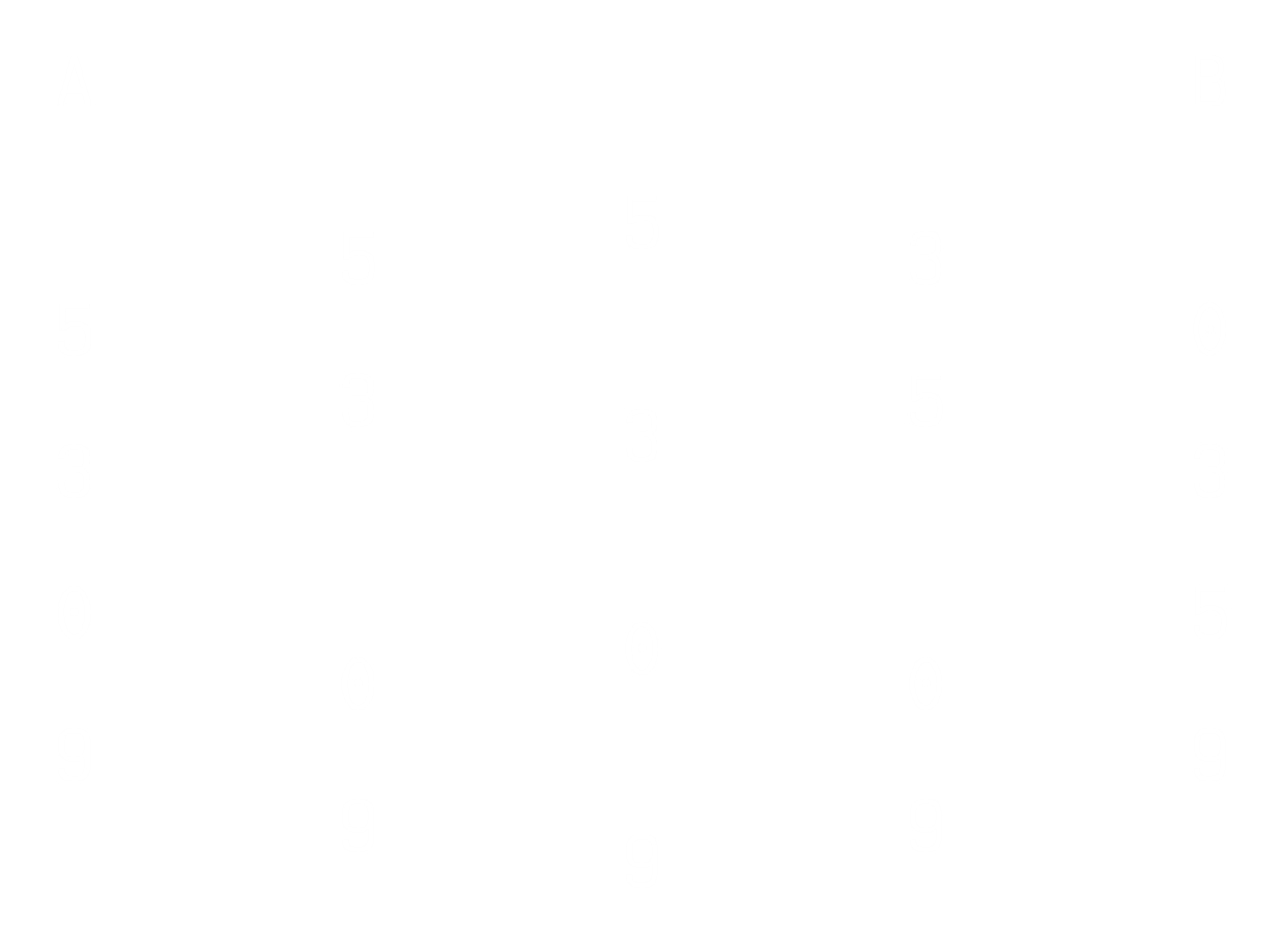
Some applications of sorting include organizing files in a folder, which can be sorted based on specific criteria such as file names, creation dates etc. We can even sort based on multiple criteria, such as first by creation date and then file names. Another application could be to organize contacts in a phone.

There are some less obvious applications of sorting that we can use to make our lives easier. For example, if we want to find the median of an array, we can just sort the array and take the middle element. Another application is when we want to find a specific element. If we sort the array, then we can use binary search (which is a divide and conquer algorithm if you think about it).

Sorting can even be used with data compression. If we have a bunch of values and we sort them, then we can easily find repeated values and remove repetitions, storing only one copy of the value and the number of times it occurs instead of storing it multiple times. A similar thinking can be followed to save data on computer graphics generation, where graphical elements can be sorted such that elements on top are shown first, thus eliminating the need to generate the elements that are beneath them, since they cannot be seen anyways.

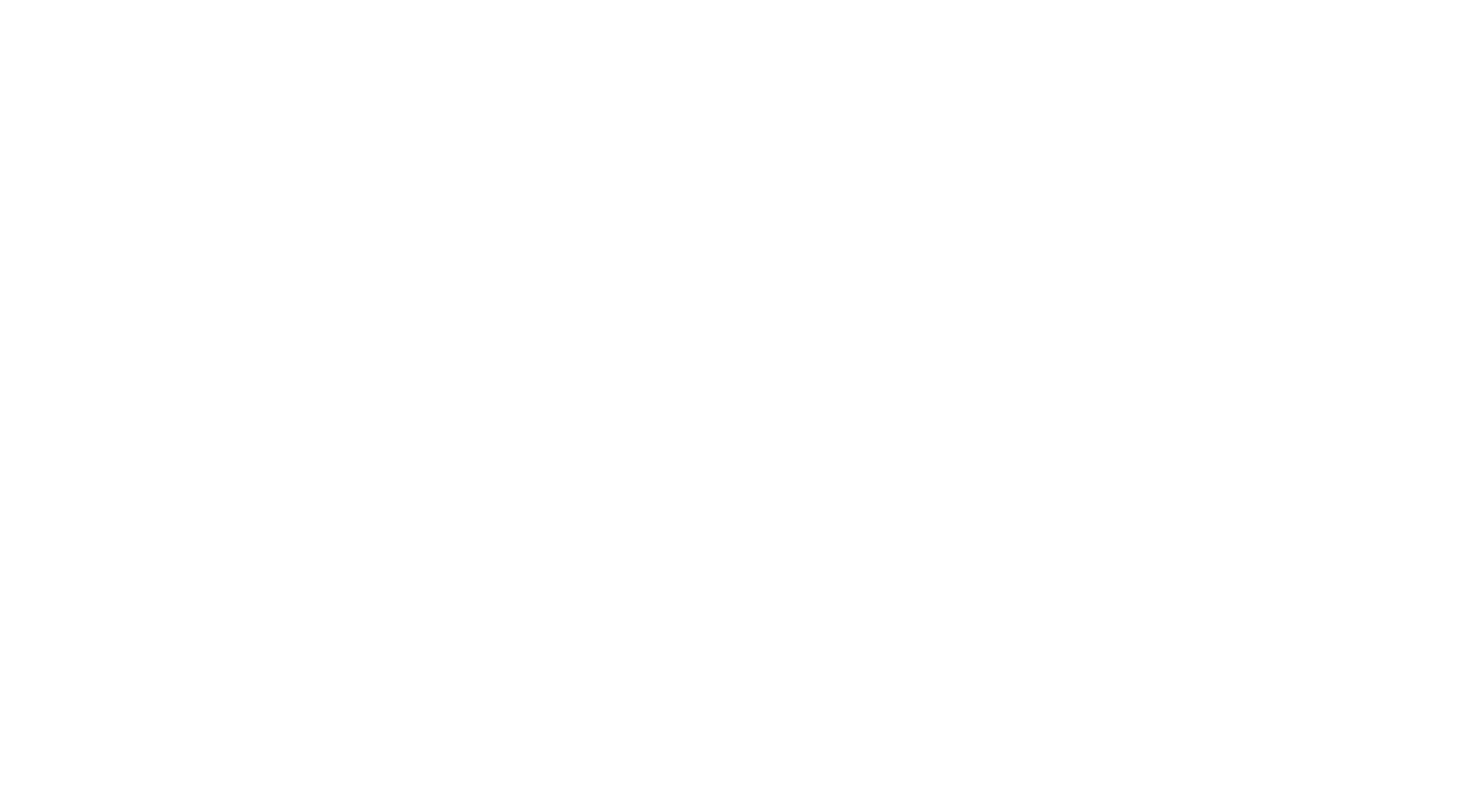
### Merge Sort Algorithm

In merge sort, our steps are to divide the array into two halves, and keep dividing all the resulting arrays until we are left with single elements. At this point, all the single element ‘arrays’ are said to be trivially sorted. Next, we merge the elements back together in a sorted order.



The divisions are simple enough, since we can just recursively call the same function on two halves. The merging step is of interest here. The algorithm that is used in the merging step is, confusingly, called the two-finger algorithm here as well.

The way this works is, the first elements of each sorted half are compared and the smallest element is added to the result. Next, the array from which the element was taken is incremented, but the array that was not chosen is not incremented. This process is repeated until we have run out of elements in one array. The remaining elements in the other array are then simply added to the result.



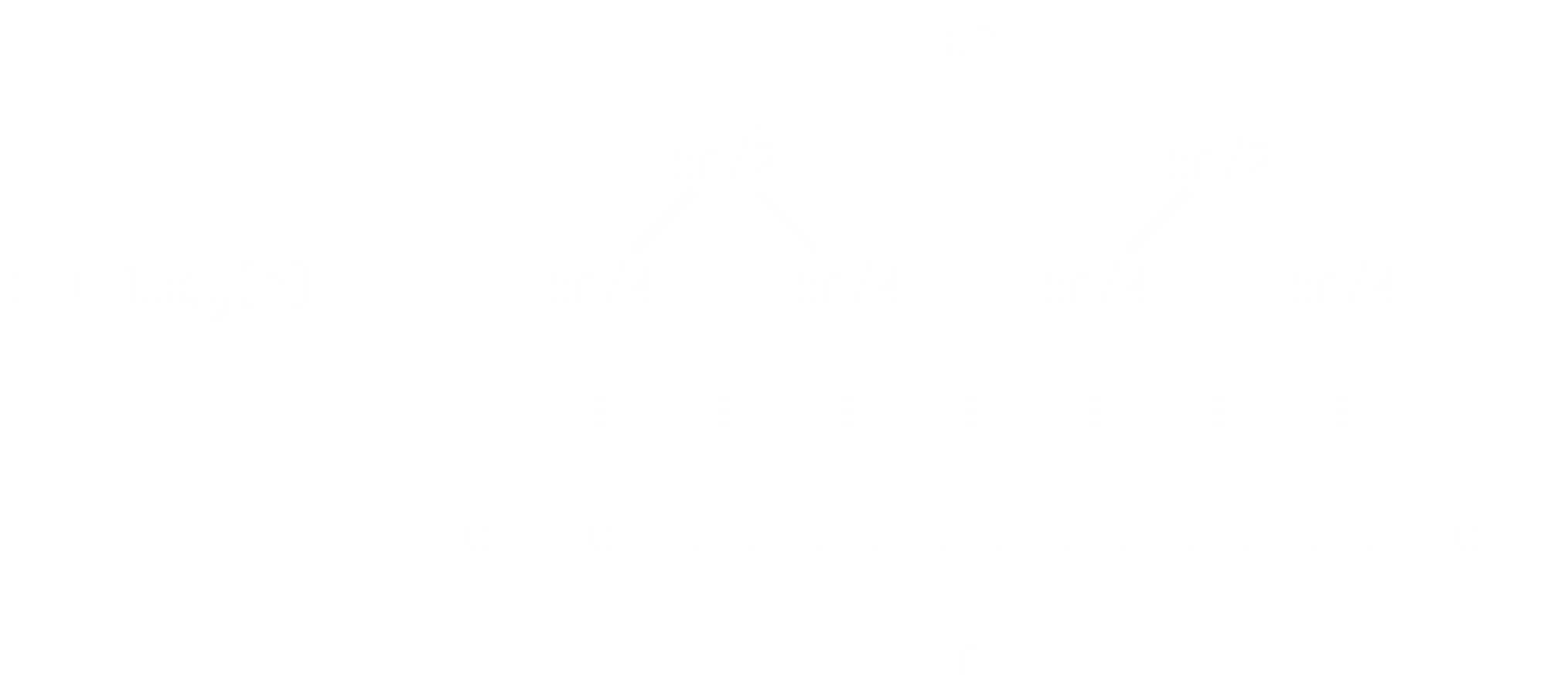
### Time Complexity

For an array with elements, this will be divided into two arrays of elements. These will individually be sorted and merged back together. During the merging stage, we have to loop over all items, so the merge stage has a time complexity of .

Notice the equation that has come up. We saw equations like this in previous parts of this module and simply wrote down the answers without going into the mathematics behind it. It is time to tackle that problem now.

We previously saw a method where we replaced with and replaced that with and so on. That is called the substitution method. Instead of using that, we will use the recursion tree method in this case.

Consider the tree below:



This tree will keep going until we reach single elements. We can divide elements a total of times before reaching those single elements. As such, there are levels to any given tree.

To find the total time complexity, we need to sum up the total time complexities on each level. On the first level, this is , on the second it is , on the fourth level it is and so on. We are doing an equal amount of work per level. Thus, the total time complexity is .