Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Teachers' Guide
Grade 9/10 Session 6

Session 6: Divide-and-Conquer

Egg Drop

Suppose you have a building with 1000 floors, and some eggs. How strong are eggshells? Let's do some science!

- If I drop an egg from floor 0 (the ground floor), it stays intact.
- If I drop an egg from floor 1000 (the roof), it definitely breaks.
- There exists a special "breakpoint" k, the highest floor from which we can drop an egg and still have it survive the fall. The egg remains intact if dropped from floor k or lower, and the egg breaks if dropped from floor $k+1$ or higher.
- How can we find k?

Our first idea might be complete search:

- Try floor 1. If it breaks, $k = 1$. If not...
- Try floor 2. If it breaks, $k = 2$. If not...
- Try floor 3. If it breaks, $k = 3$. If not...

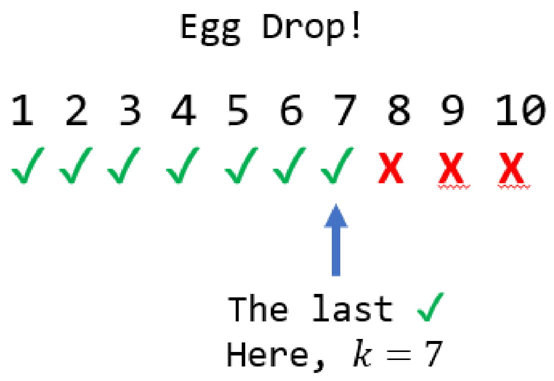
...

Unfortunately, this procedure is rather slow—and wasteful! Imagine if the egg only breaks at floor $k = 1000$. Then, you would have to do 1000 iterations to reach that conclusion!

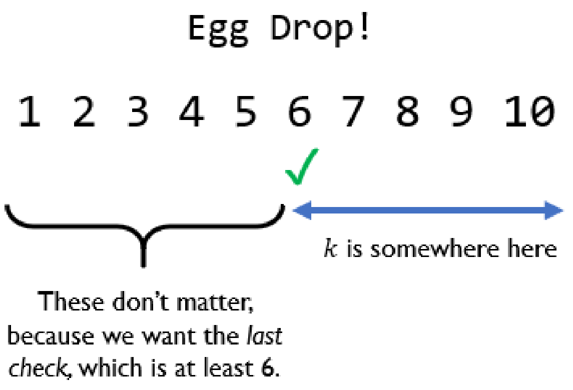
Is there a better solution? Yes, actually, and we have to take advantage of one of the defining properties of the setup—the egg remains intact if dropped from all floors $\leq k$, and breaks if dropped from all floors $> k$.

Reaching the breakpoint is like flicking on a light switch. Imagine we did hypothetically query all the floors. What would the responses look like?

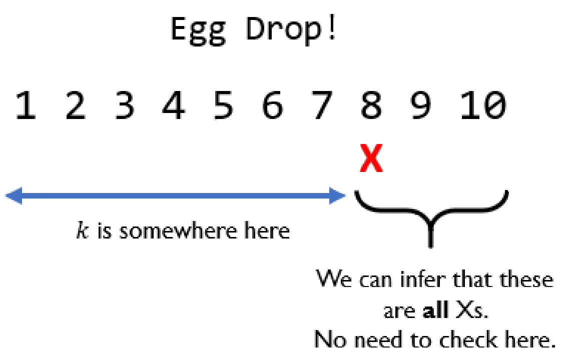
The answer is always `YES, YES, YES, YES`… until it reaches k, after which it becomes `NO, NO, NO, NO` from that point onwards. Importantly, **there is only one such time when the result changes**.



Let's drop the egg at some floor m. If the egg *remains intact* at some at floor m, then we know that $m \leq k$. Why? We want the *last* floor for which the egg remains intact. So, if it remains intact at m, then we know the answer is *at least* m. No need to check anything less than m.


Egg Drop!
1 2 3 4 5 6 7 8 9 10
✓
k is somewhere here
These don't matter, because we want the *last check*, which is at least 6.

If the egg *breaks* at floor m, then we know that $k < m$. Why? By the setup of the problem, if it breaks at m, then it'll also break if we dropped it from anywhere higher than m. Remember we want the last floor for which it *doesn*'t break. So, no need to check those floors anymore.


Egg Drop!
1 2 3 4 5 6 7 8 9 10
X
k is somewhere here
We can infer that these are **all** Xs. No need to check here.

In general, let's suppose that we know that k lies somewhere in the range $[L, R)$ – that's L inclusive and R *exclusive*. Let's call that range our *search space*. Now, choose some m in that range and drop the egg at floor m. If the egg breaks, then k must be in the range $[L, m)$. If the egg survives, then k must be in the range $[m, R)$.

Essentially, we are *partitioning* the search space into $[L, m)$ and $[m, R)$. [1] Then, we are able to *narrow down* our search space by figuring out which half k belongs to.

Class Discussion. What should our optimal choice of m be? In other words, how should we best split the search space?

Answer. Since k could be in either of the halves (and we don't know which), it would be for the best to keep things *balanced* by splitting the search space into equal partitions, i.e., cutting it in half. So, we can choose an m close to the midpoint, such as $m = \frac{L+R}{2}$ (rounded down, if not an integer).

What next? Well, do it again! If we keep repeating this process, then the search space will keep narrowing and narrowing and narrowing, until it consists of just a single value. Once the search space has been reduced to a single value, we already know what the answer is—that value!

To summarize, our strategy is as follows.

- What I know: I can find the answer in the range $[L, R)$
- If $L + 1 = R$ (so $[L, R)$ consists of just L), then we know that the answer is L.
- Otherwise, take $m = \frac{L+R}{2}$ (rounded down).
    - o If the egg breaks, then we know that k must be in the range $[L, m)$
    - o If the egg remains intact, then we know k must be in the range $[m, R)$

And since the last two points are in the same form as our original problem (but smaller), we can express this in code with recursion!

```python
def egg_drop(L, R):
    if L + 1 == R:
        return L

    m = (L + R) // 2
    if drop_egg(m) == 'BREAK':
        return egg_drop(L, m)
    else:
        return egg_drop(m, R)
```

[1] Note that the weird inclusive-exclusive choice was made so that the search space would be partitioned neatly, as seen here.

How efficient is this solution? Note that since m is (roughly) the midpoint of L and R, it (roughly) partitions the search space into **two equally sized halves**. Thus, if we find out which half contains k and throw away the other one, that means we essentially **halve** the size of our search space at each iteration.

```
   Search Space, at each step
_________________________________
  0) 1 2 3 4 5 6 7 8
  1)       5 6 7 8
  2)       5 6
  3)         6
```

So, how many iterations will this take? Suppose the initial search space is size n. Then, the number of iterations that we need should be roughly equal to the number of times we can divide n by 2 until we get 1. In other words,

$$n \div 2 \div 2 \div 2 \div \ldots \div 2 = 1$$

can have at most how many division signs?

We have a fancy function that's defined as the answer to this question, and we denote it by $\lg n$, or $\log_2 n$ (read as "base-2 logarithm of n). So, for example, $\lg 128 = 7$, because we can divide 128 by 2 up to 7 times before hitting 1. Note that if you plug this function into the calculator with a non-power-of-2, you won't get a clean integer. But that's okay. For our purposes, we can just round up or down, depending on what the situation calls for.

The important thing to know about $\lg n$ is that it is a *very* slow-growing function, which is really good news for us! The following table shows how quickly the powers of 2 grow.

n	2^n
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	**1024**

At just $n = 10$, we have already surpassed 1000. A logarithm is really just, "What is the exponent needed to reach some number." It is the inverse of exponentiation. Powers grow big really fast! This means that logarithms grow *really* slowly.

n	$\log_2 n$
2	1
10	3.3219…
100	6.6439…
1000	**9.9658…**
10,000	13.2877…
100,000	16.6096…
10^6	19.9315…
10^9	29.8973…
10^{12}	39.8631…
10^{15}	49.8289…

See that $\lg 10^9 < 30$. That means at just $n = 30$, we can see that 2^{30} has already surpassed **one billion**! That means if we were to run our egg drop algorithm on a building with 10^9 floors, we would still *only need at most 30 tries to get the answer.* Logarithmic time is awesome!

This technique is called **binary search**, because we split the search space into two at each iteration.

Class Discussion. Would the algorithm still work if we rounded m up instead of down?

Answer. Note that we only get a non-integer m if the number of elements in the search space is odd. If we were searching through $n = 1000$ elements, then we can split our search space into sizes 500 and 500. If we were searching through $n = 1001$ elements, then the best we can do is split our search space into sizes 500 and 501. Changing whether we round up or round down doesn't matter too much, since all we're changing which half is the 500 and which half is the 501.

Class Discussion. If $\log_2 1000 = 9.96578\ldots$, then why do we round **up** to 10 iterations needed, and not **down** to 9?

Answer. Recall that we're trying to find the number of operations needed in the *worst* case. So, if the search space is partitioned into sizes 500 and 501, we have to be pessimistic and consider what happens if it goes into the bigger half—or if it *always* goes into the bigger half at each step. That's why we round up.

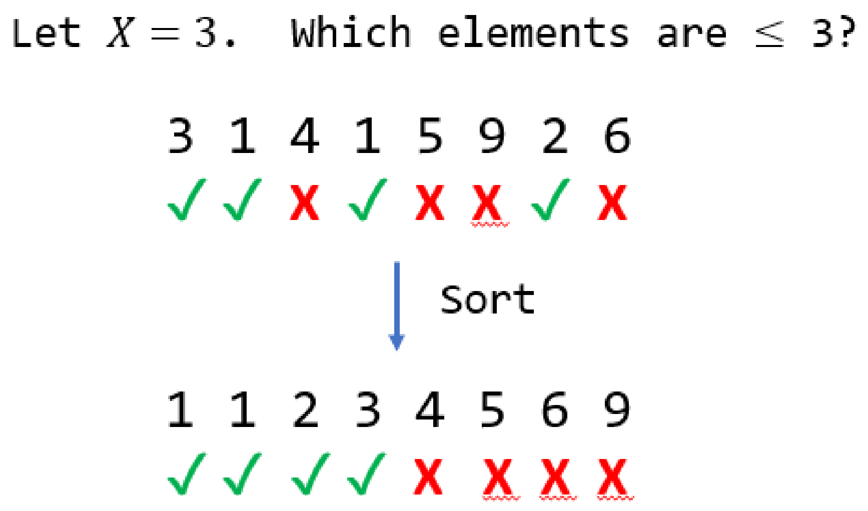
Try it on small cases to be convinced. If you need to search through 3 elements, do you need at most 1 or 2 checks?

Upper Bound and Lower Bound

Suppose you are given a list $A_0, A_1, A_2, \cdots, A_{n-1}$. We need to support several queries that ask us to find the largest element in A that is $\leq X$ (this X will be different from query to query, of course).

For example, suppose you are managing a restaurant and have a menu with items at various price points. Then, your customers each want to know what is the most expensive dish that they can afford (within their budget X).

A straightforward solution would be to inspect each of the elements in the list one-by-one, but that would give us a time of $\Theta(n)$ per query. Can we do better?



It turns out that yes, we can. The key is that we have to **sort** the list. If we do, observe what happens to the numbers that are less than or equal to X.
That looks just like our egg drop problem! If A is sorted, then all the numbers in A are $\leq X$ up until a certain point—then, from that point onwards, the numbers will always be $> X$. We just need to find this turning point.

If $X < A_m$ for some m, then that means $X < A_i$ for all $m < i$ as well, since the list is sorted, and all those numbers are even larger than A_m (which we learned is $> X$). So, the answer is strictly less than m.

If $A_m \leq X$ for some m, then we know that the index of the answer is at least m, since we want the largest valid element and are assured that A_m works.

We wish to find the greatest k such that $A_k \leq X$. We can apply the same principles from our egg drop solution. Let the initial search space be $[0, n)$

- What I know: I can find the answer in the range $[L, R)$
- If $L + 1 = R$ (so $[L, R)$ consists of just L), then we know that the answer is L.
- Otherwise, take $m = \frac{L+R}{2}$ (rounded down).
    - If $X < A_m$, then we know that k must be in the range $[L, m)$
    - If $A_m \leq X$, then we know k must be in the range $[m, R)$

Just like with egg drop, we are also roughly halving the size of our search space at every iteration, therefore we can answer each query in just $O(\log_2 n)$ time.

Exercise 1. Implement this solution in code

Hint: you just need to edit the egg drop algorithm at the appropriate places.

Exercise 2. Modify your code so that it instead finds the smallest element in the array that is greater than or equal to some X.

Sorting

Hold on, the last solution ran in $\mathcal{O}(\log_2 n)$ per query, but only if we sorted it. How efficiently can we sort elements anyway? That's a good question! Let's explore.

Given an array of *n* values $A_0, A_1, \dots, A_{N-1}$, rearrange the elements so that they are sorted in non-decreasing order (i.e. if $i < j$, then $a_i < a_j$). We will examine two different sorting algorithms and compare how their different approaches affect their running time.

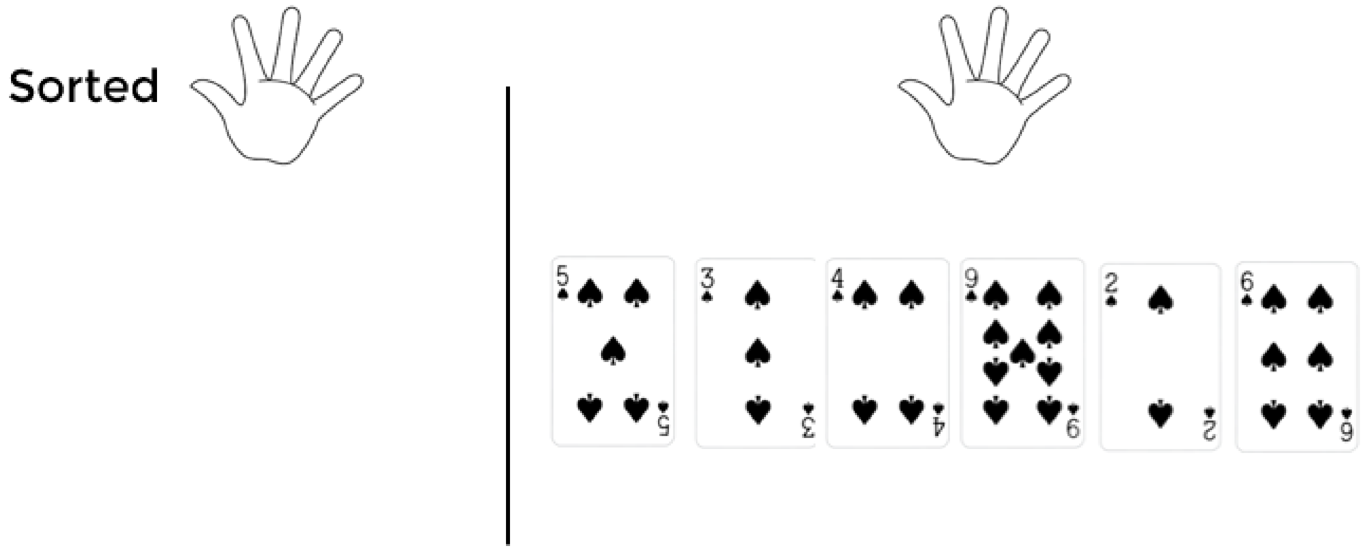
Also, we encourage you to take out some slips of paper that you can move around, and try executing these sorting algorithms by hand! It helps to get a concrete grasp of what the computer is doing.

Insertion Sort

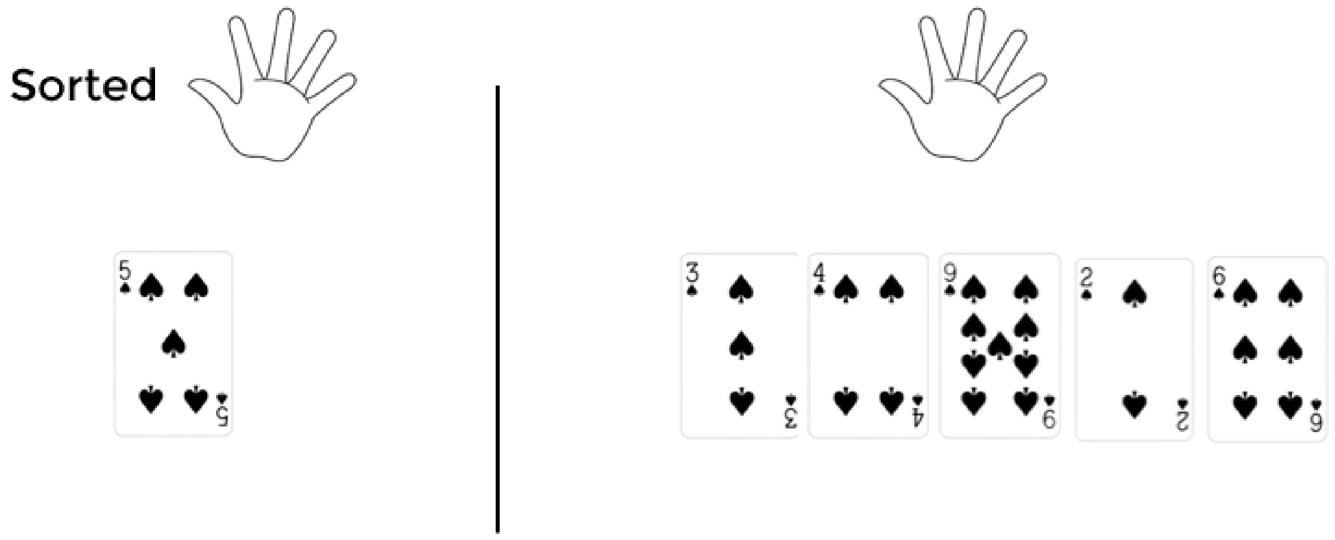
Have you ever played a card game, especially one like Pusoy Dos or Tong-Its, where you are dealt a "big" hand of 13+ cards? If so, you might have wanted to sort the cards in your hand in order to have it be better-organized. A natural algorithm when sorting a hand of cards might be as follows.

1. Create a "sorted" hand, which is initially empty.
2. One by one, choose a card from your hand and move it to the sorted hand. When you do, make sure to insert it in its "proper" position, so that the sorted hand always remains *sorted* at all times.
3. Repeat this until your original hand is empty, and all cards are in the sorted hand.

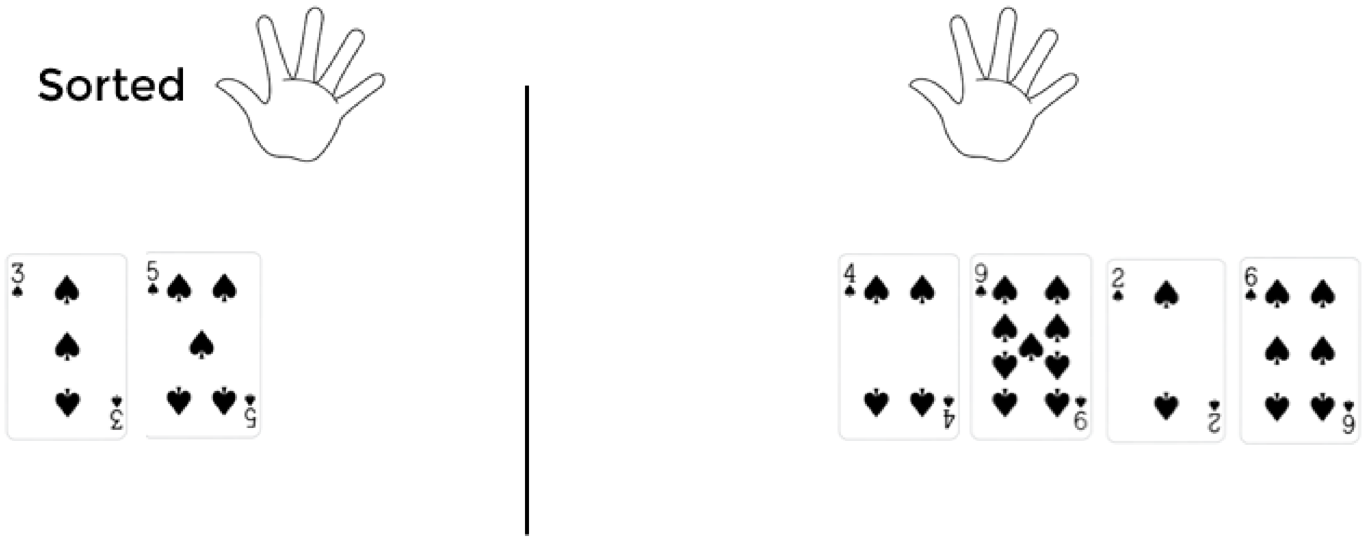
Let's look at this algorithm in practice. Suppose you were dealt the following hand.



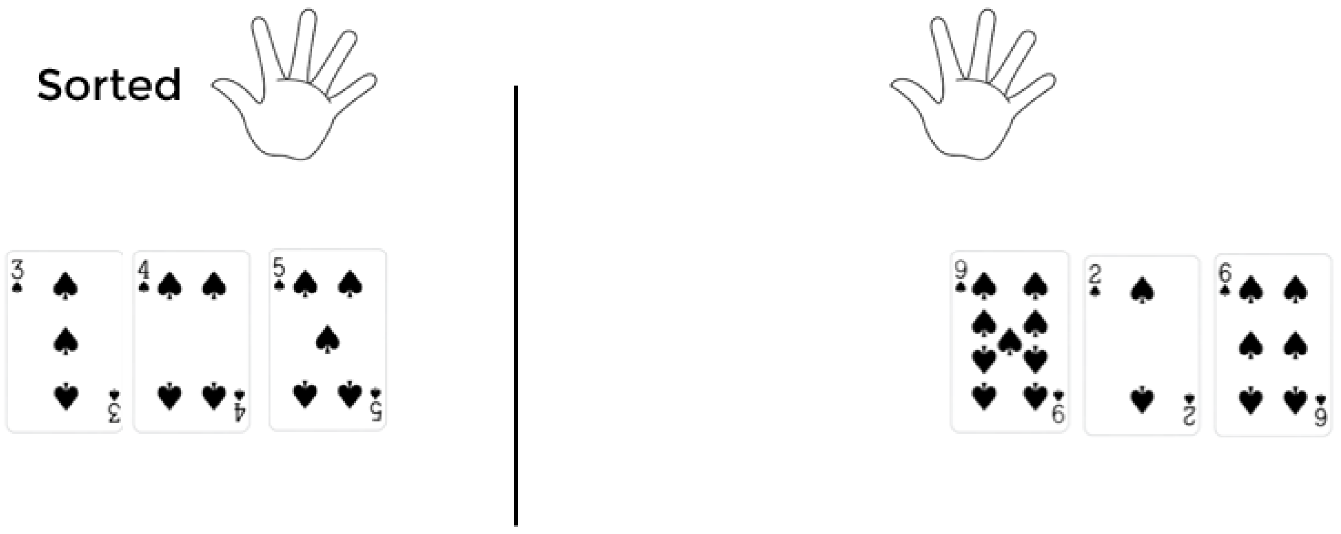
Let's take the first card in our hand, the 5, and move it to the sorted hand.



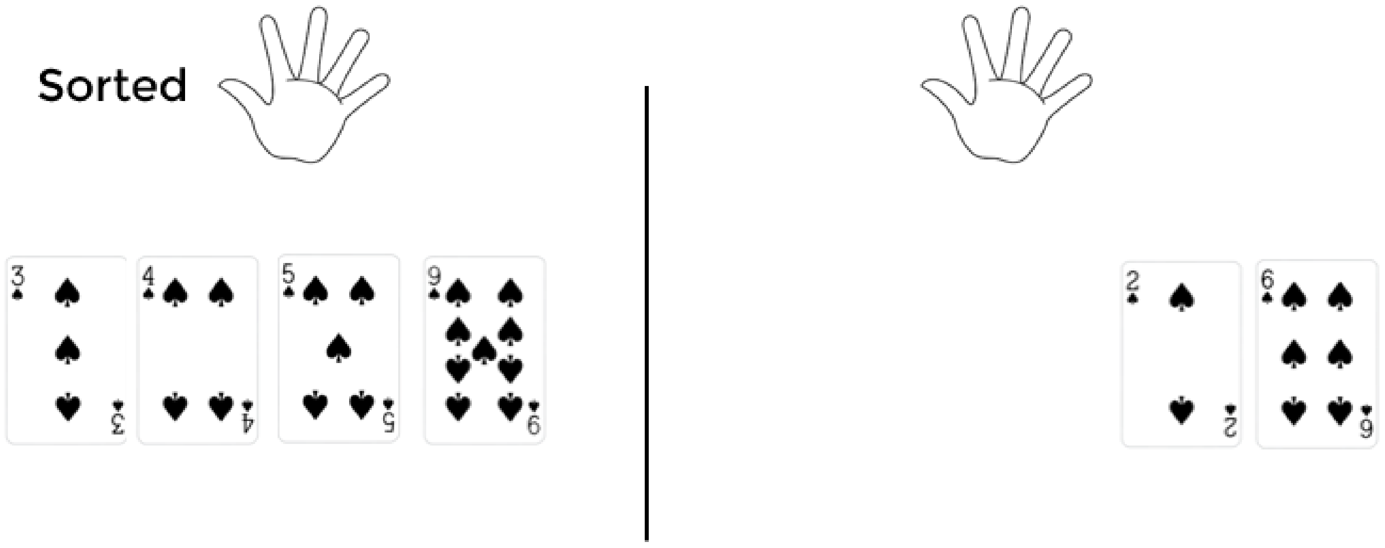
Then, we take next card, the 3, and move it to the sorted hand. When we do this, we must take care to insert it to the *left* of 5, so that the sorted hand remains sorted.

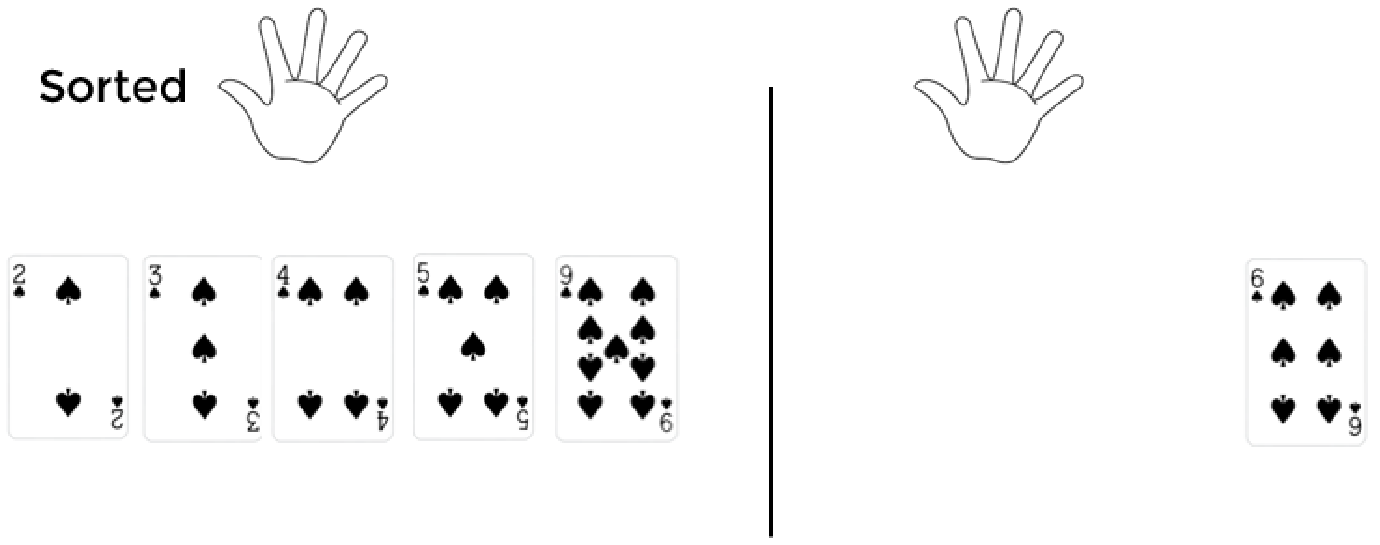


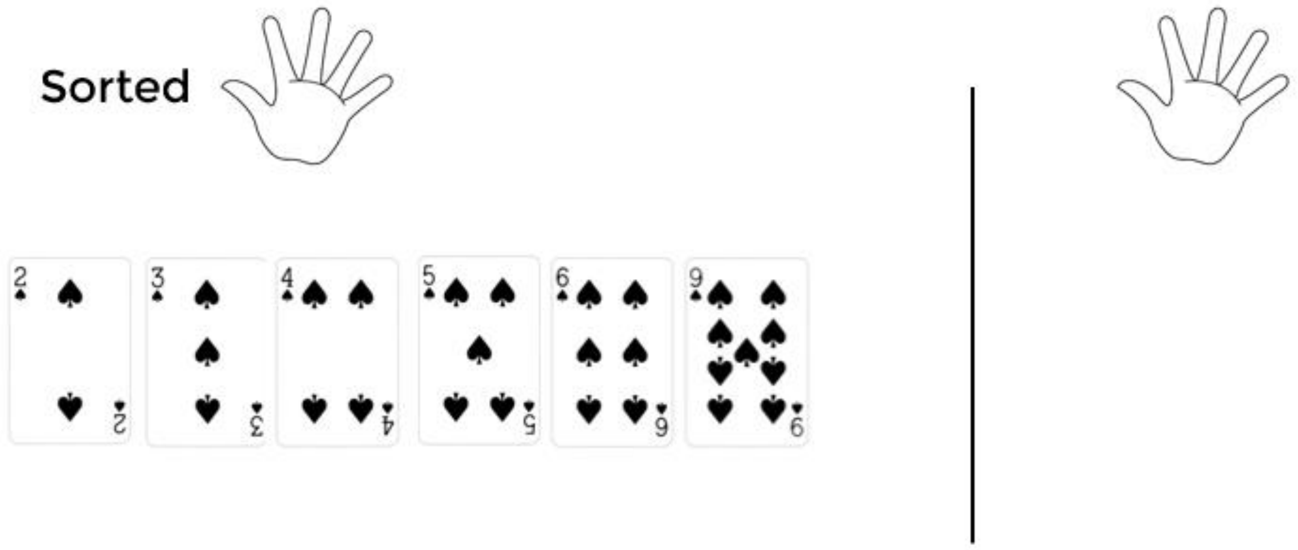
Next, move the 4. Make you sure you insert it between 3 and 5.



We finish the algorithm by doing the same for the 9, the 2, and the 6, each time inserting it into its proper place in the sorted list.

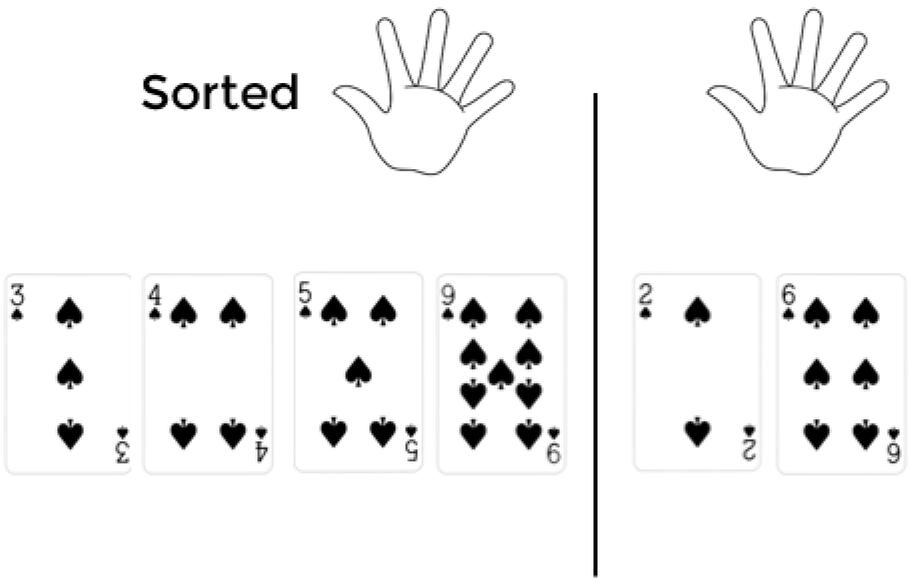






And we are done!

First, a note on implementation. Rather than actually moving the sorted cards to a separate hand, most players will instead just keep it a single hand, but *move sorted cards to the front of their hand*. They will just remember that after i insertions, the first i cards in their hand are already sorted. In the illustrations above, notice that the six cards can actually still be arranged in a single row, with the vertical line separating the sorted cards from the unsorted cards.



Many implementations of insertion sort operate on a similar principle, when sorting a list. There is no need to create another separate list. After i insertions have been made, the *first i elements* of the list should be sorted.

If we wanted to write this algorithm recursively, it would make sense for "one step" to be one insertion. So, we could phrase it recursively as follows. The following algorithm sorts a list of n integers.

1. Sort the first $n-1$ elements.
2. Figure out where to insert the nth element to make the whole list sorted.

Exercise 1. Recursive algorithms need a base case. What could be a base case for sorting?

Solution. If $n = 1$, then the list (with just one element) is already sorted, so we don't have to do anything.

What is the running time of this algorithm?

To insert the kth element, we would have to perform $O(k)$ steps to scan the sorted part, find the proper position of the kth element, and "shift" the greater elements forward so that we can insert the new value. But, we have do these $O(k)$ operations for *all* k from 1 to n. So, the total amount of operations is $0 + 1 + 2 + \cdots + (n-1)$, which you might recognize from before as being equal to $\frac{n(n-1)}{2}$. So, this entire algorithm is $O(n^2)$.

Can we do better? Let's try to explore.

The main reason for the running time devolving into $O(n^2)$ is that there are a lot of "levels" of recursive calls. Because each step only handles one element, there are $O(n)$ levels. How can we have fewer levels?

Merge Sort

We have a lot of "levels" because we're only making *one* insertion at each recursive call. So, we are basically sorting the list one element at a time. We can do better than that. What if, instead of one at a time, we find a way to do a lot of operations *together* to save time?

Consider the following outline of an algorithm.

1. Partition the array into two roughly equally-sized smaller arrays—its left half, and its right half.

2. Recursively sort each half.

3. **Merge** the two sorted halves together somehow.

This algorithm is known as **merge sort**. Here's how that might look like in Python.

```
def sort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a) // 2
        return merge(sort(a[:mid]), sort(a[mid:]))
```

Notice here that compared to insertion sort, we are partitioning the work much more evenly. Instead of having "pieces" of size 1 and $n-1$, we instead have two pieces that are both around $n/2$ in size. This intuitively makes merge sort a lot more *balanced* than insertion sort.

The only missing detail is the **merge** step—given two sorted lists, how do we quickly merge them together into one large, sorted list? For insertion sort, this was easy—we just had a single element that we had to insert somewhere in the bigger list. But this isn't trivial in merge sort, since the two lists that we need to merge are roughly the same size. We need to *weave* these two lists together into a large, sorted list.

Let's give a concrete analogy that we can imagine.

Suppose we are trying to line up the students in a class in increasing order of height. The boys are already lined up by height, and the girls are also already lined up by height. Now, let's build the merged line as follows.

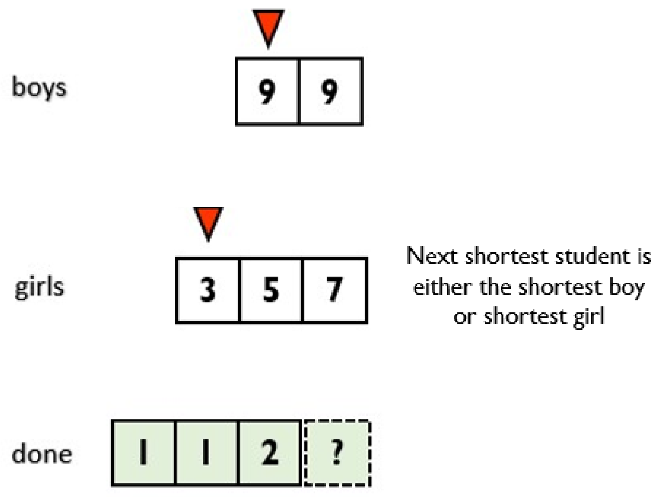
1. Find the shortest student who hasn't been sorted yet.

2. Place that student in the "next" spot in the merged line.

3. Repeat while there are still students that haven't been merged.

However, we already know where the next shortest student in class is—the shortest student must be either the shortest boy or the shortest girl! So, in the step, "*Find the shortest student who hasn't been sorted yet,*" rather than having to check *all* remaining students at each step, we only have *at most two* candidates that we need to check.

So, let's complete our implementation of merge sort by filling in the details of the merge function. Place the shortest boy/girl in front, then repeat this until we are done.


boys
9 9
girls
3 5 7
Next shortest student is either the shortest boy or shortest girl
done
1 1 2 ?

```
def merge(L, R): # L and R are sorted
    if L[0] < R[0]:
        return [L[0]] + merge(L[1:], R)
    else:
        return [R[0]] + merge(L, R[1:])

def sort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a) // 2
        return merge(sort(a[:mid]), sort(a[mid:]))
```

Question. What about the cases where `L[0] == R[0]`? Are they handled correctly?

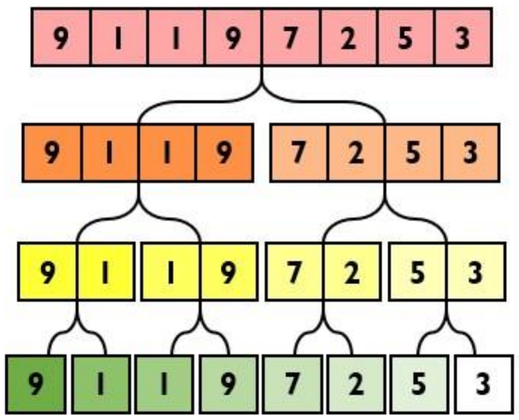
Answer. Yes, because then it doesn't matter who you put in front anyway.

Class Discussion. Try implementing this and running it. Do you get an error? What's wrong?

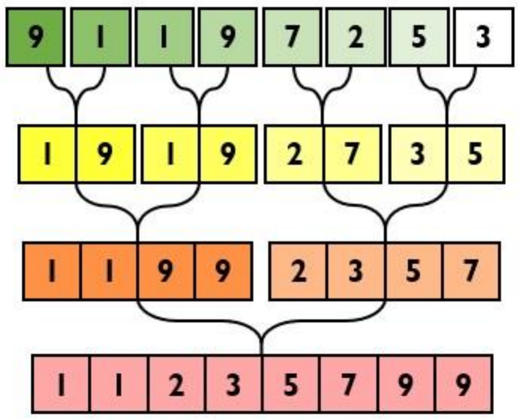
Answer. We aren't properly handling the cases when `L` or `R` are empty! Add some if statements to fix that.

To summarize, at each call of `sort`, we recursively sort the left half and the right half, and then use `merge` to merge the sorted lists together. The following diagram shows how the algorithm would look, sorting $n = 8$ elements.

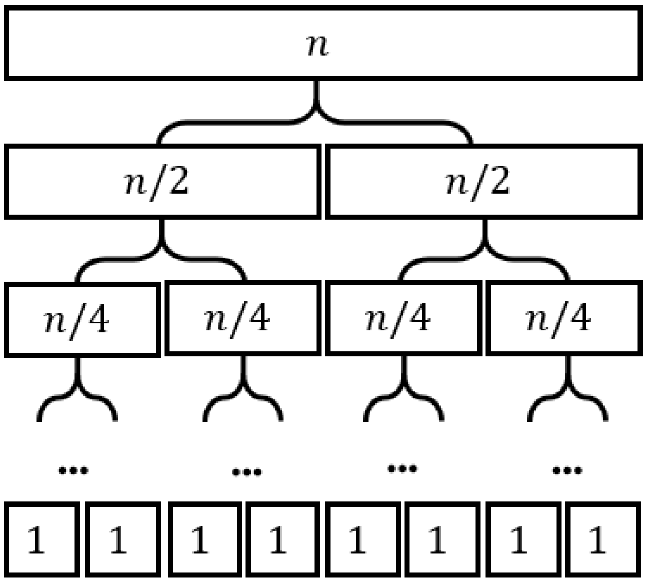
Dividing the list into halves



Sorting and merging the lists



Now for the big question – what is the running time of this algorithm? To understand this, let’s try drawing the tree in the previous diagram, but for some more general n.



Notice that we can group together the recursive calls at the same "level" – for example, at the level two steps down from the whole n, there are four recursive calls, and each of those calls handles a list that is $n/4$ in size. This holds true in general – the total length of all the lists at a certain level should sum up to approximately n.

Thus, the running time of merge sort is equal to the number of levels, multiplied by the amount of time needed for each level (which has n elements).

First question—how many levels are there? There are as many levels as there are times that we can keep dividing n by 2 until we hit 1—and you'll remember from earlier that this is $\mathcal{O}(\log_2 n)$

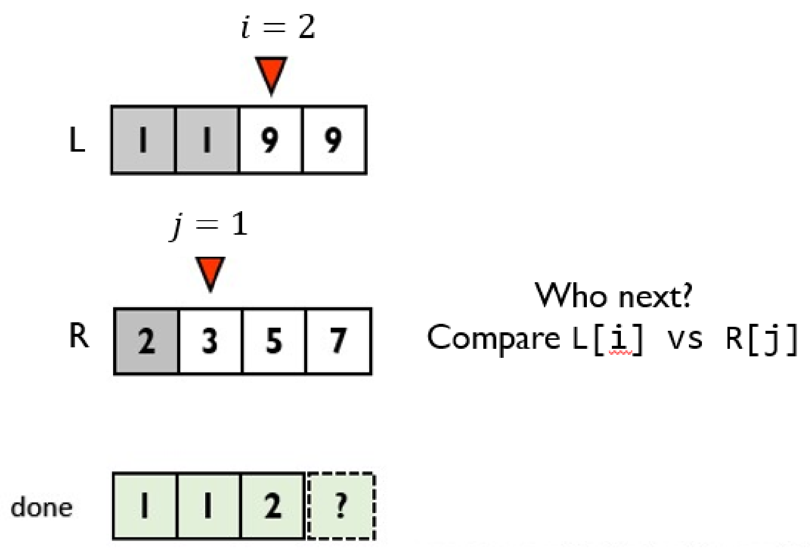
How much work is needed at each level? Notice that the main thing that is done at each level is the `merge` step. So, let's ask the question—what is the running time for applying `merge` to n total elements?

At each step, we remove the smallest element from each list, but the way that we do that is by list slicing. Remember that list slicing creates a *copy* of all the indicated elements. So, if `L` has n elements, then taking the slice `L[1:]` requires us to *make a brand new list* with the last $n-1$ elements. Eventually we need to merge all the elements from `L` (and of `R`) so we end up performing $(n-1)+(n-2)+\cdots+2+1+0 \in \mathcal{O}(n^2)$ operations. So, this seems even worse than insertion sort!

We can fix this. Note that the only reason why the merging devolved into $\mathcal{O}(n^2)$ was because each "remove the smallest element" operation took $\mathcal{O}(n)$. To speed this up, we need to "delete" each smallest element in $\mathcal{O}(1)$.

Rather than literally removing the smallest element of each list, we can just keep track of two values—how many elements from `L` have been merged already, and how many elements from `R` have been merged already. If we keep track of these two values i and j, then we will be able to recover who is at the front of each queue—they're `L[i]` and `R[j]`. Now to "delete" elements, we just increment these counters, without having to actually delete elements in `L` and `R`!

Since all we're doing is incrementing these two counters, the "remove the smallest element" step only takes $\mathcal{O}(1)$ each. So, the merge can be done in $\mathcal{O}(\text{size of the array})$.



So, here is our final version of merge sort.

```python
def merge(L, R): # L and R are sorted
    ans = []
    i = 0
    j = 0
    while i < len(L) or j < len(R):
        if L[i] < R[j]:
            ans.append(L[i])
            i += 1
        else:
            ans.append(R[j])
            j += 1
    return ans

def sort(a):
    if len(a) == 1:
        return a
    else:
        mid = len(a) // 2
        return merge(sort(a[:mid]), sort(a[mid:]))
```

There are $O(\lg n)$ levels, and each level is doing $O(n)$ work in total, so the overall running time of merge sort is $O(n \lg n)$.

Exercise. If you try implementing the above code, you'll *still* get an error. The `merge` function *again* is not properly handling the case when one of the lists becomes empty (or here, when `i == len(L)` or `j == len(R)`). Add some if statements to fix that so that this function is finally complete.

Note. You will almost never have to implement your own sort function instead of using `sort()` or `sorted()`, but knowing how it works is still important! These built-in functions run in $\mathcal{O}(n \lg n)$ as well, using similar ideas.

Now, notice that insertion sort had the following strategy.

1. Split the list into pieces of size $n-1$ and 1.
2. Sort the $n-1$ piece.
3. Insert the 1 element into the sorted $n-1$ piece.

Merge sort actually follows a similar strategy, except with more balanced pieces.

1. Split the list into pieces of size $n/2$ and $n/2$.
2. Sort each of the smaller pieces.
3. Merge the two pieces together.

Even binary search from egg drop had a similar pattern for finding the optimal k.

1. Split the search space into two equally-sized partitions.
2. Narrow down which partition k is in and continue the search there.

This reveals a more general strategy for attacking problems, making an algorithm faster than brute force.

1. **Divide:** Split the problem into parts.
2. **Conquer:** (Recursively) Solve each part independently.
3. **Combine:** (Optionally) Combine the answers from each part.

This approach is called **divide and conquer** (and sometimes, combine).

Maximum Subarray Sum

Let's tackle one last problem, applying divide and conquer so that we can solve it efficiently.

Consider a list of integers $A_0, A_1, \dots, A_{n-1}$. The **subarray** from l to r (where $0 \le l \le r < n$) is the *contiguous* sequence of elements $A_l, A_{l+1}, \dots, A_r$ contained within the original list.

Among all subarrays of A, find the one such that the sum of its elements is **maximum** possible. Note that we only consider subarrays with at least one element, so we can't pick an empty subarray.

Note that the integers in A can be negative (if they were all positive, it wouldn't be much of a problem!) For example, consider the list $[3, 4, -5, 4, -1]$. Then, the maximum subarray sum is 7, by taking the first two elements 3 and 4. We'd have liked to be able to take the other 4 as well, but it's separated from the other two elements by a -5, and our subarray has to be contiguous.

There exists an $\mathcal{O}(n^3)$ brute force by complete search, and if you think about it a bit, you might realize how to bring that down to an $\mathcal{O}(n^2)$ solution—but still by complete search.

Exercise. Show that there are $\mathcal{O}(n^2)$ subarrays of a list.

Answer. There are n subarrays that start at index 0, then $n-1$ subarrays that start at index 1, then $n-2$ subarrays that start at index 2, and so on. So, in total, we have $n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$ subarrays of the list.

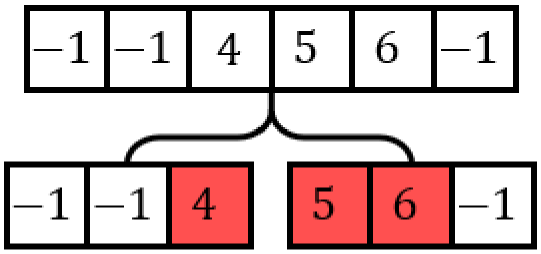
A complete search solution is at the very least going to have to check all those subarrays, so that approach is stuck at $\mathcal{O}(n^2)$ at least. Can we do better?

Let's apply divide and conquer!

1. **Divide:** Split the list into pieces of size $n/2$ and $n/2$, the left and right half.
2. **Conquer:** Find the maximum subarray sum in each half.
3. **Combine:** ???

Question. True or False: The maximum subarray sum of A is the greater of the maximum subarray of its left half, and the maximum subarray sum of its right half.

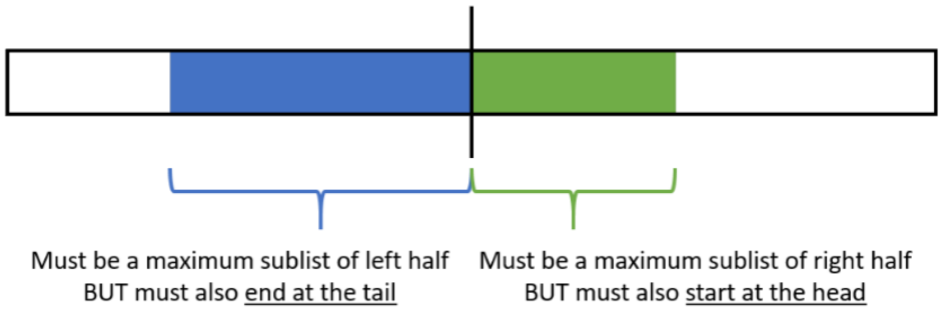
Answer. Not necessarily. Those are certainly possibilities, but it could also be the case that the optimal answer *cuts through the midpoint* and has a little bit in each half. See the following diagram for an example.



So, the maximum for the entire list is the maximum of the following.

- The maximum of the left half.
- The maximum of the right half.
- The maximum which *crosses* the midpoint. Call these *boundary-crossing subarrays*.

What does the maximum boundary-crossing subarray look like?



Must be a maximum sublist of left half BUT must also end at the tail
Must be a maximum sublist of right half BUT must also start at the head

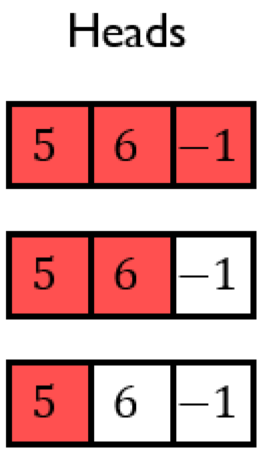
The portion which rests in the left half must *end at the tail* (include its last element), while the portion which rests in the right half must *start at the head* (include its first element). So, the maximum boundary-crossing subarray takes the maximum subarray of the left half *that ends at its tail* and the maximum subarray of the right half *that starts at the head*.

So, the maximum subarray sum of the entire list is one of the following.

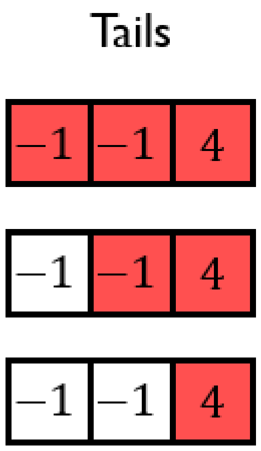
- The maximum of the left half.
- The maximum of the right half.
- The maximum *tail* of the left half + the maximum *head* of the right half.

But… now we have a new problem! We need to be able to compute the maximum tail of the left half, and the maximum head of the right half. What now?

Well… can we find those by brute force? There may be $\mathcal{O}(n^2)$ subarrays in a list, but what if *we fix* one of the endpoints of the subarray? If we force ourselves to include the first element (so this is a head), then there are only $\mathcal{O}(n)$ choices for the right endpoint of the subarray; thus there are only $\mathcal{O}(n)$ different heads to check.



Similarly, there are only $\mathcal{O}(n)$ different tails to check.



So, let's just check all of them! Merge Sort showed that we can afford to do $\mathcal{O}(n)$ work at each level and still end up with a "good" running time.

I want to emphasize here—we do **not** check all head-tail pairs. *First* we get the maximum tail of the left. *Second,* we get the maximum head of the right. *Then*, we add them together to get the maximum boundary-crossing subarray. The maximum checks are independently done on the heads, and on the tails.

Putting that all together, we get the following algorithm.

```python
def best(a):
    if len(a) == 1:
        return a[0]
    else:
        m = len(a) // 2
        L = a[:m]
        R = a[m:]

        max_left = best(L)
        max_right = best(R)

        max_left_tail = max([sum(L[i:]) for i in range(len(L))])
        max_right_head = max([sum(R[:i]) for i in range(len(R))])

        return max(max_left, max_right, max_left_tail + max_right_head)
```

Class Discussion. Note that performing `sum(a[:i]) for i in range(n)` actually computes all the heads (and tails) in $\mathcal{O}(n^2)$ time, due to the $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$ reasoning from earlier. We need to change this. How can we compute all the heads in $\mathcal{O}(n)$?

Answer. A lot of those computations are redundant because of *overlaps* between the heads. After all, note that `sum(a[:i]) == sum(a[:i-1]) + a[i]`. In other words, the sum of the first i elements is equal to the sum of the first $i - 1$ elements, plus the ith element. We can have a loop which "adds the next element" to a running total, keeping track of all values that the running total has seen so far.

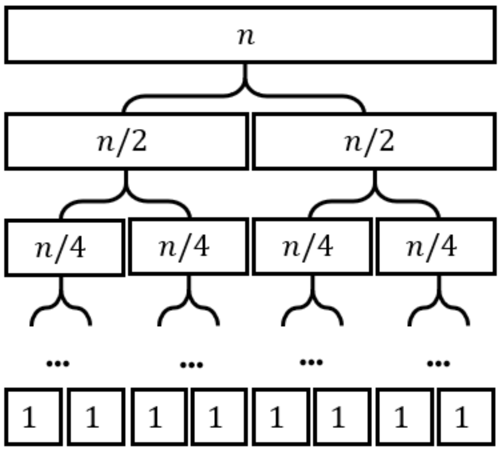
Use something like the following to get all the `heads` of the right half.

```python
heads = []
total = 0
for i in range(len(R)):
    total += R[i]
    heads.append(total)
```

You can do something similar to find all the `tails` of the left half.

Then, put all together, the answer would be `max(max_left, max_right, max(tails) + max(heads))`.

Now, what is the overall running time of this algorithm? Since we're doing divide and conquer, let's again draw that tree diagram which shows the recursive calls of the function.



Just like with merge sort, we divide each list into two equal halves at each step. This gives us $\mathcal{O}(\lg n)$ total levels in the tree, and the total length of all lists at the same level is $\mathcal{O}(n)$. Since the amount of work done at each level is also $\mathcal{O}(n)$ (to find the maximum head/tails), that means the total amount of work done overall is $\mathcal{O}(n \lg n)$.

Much better than $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$!