COMP 3711 Fall 2021
Design & Analysis of Algorithms

Written By: Ljm

Topic 2 **Divide & Conquer**

1 Intro: Binary Search

The main idea of Divide & Conquer is to solve a problem(such as of size n) by breaking it into one or more smaller(size less than n) problems. We use binary search example to illustrate that.

Problem: given an **sorted** array of length n, how to find the position of element x; if x does not exist in the array, output nil.

Since the array is already sorted, it has a good property that: **for each item a_i, those who are larger than a_i must be on its right side, while smaller than a_i must be on its left side.** Hence we come up with an idea that we check the middle item mid first, then we will be able to know which direction to go: left or right, depending on the comparison of mid and x(the item we're looking for). If we go left, then the right half will be directly abandoned. Then we continue this process, check middle item each time, and abandon half items each time.

```
Algorithm 1: BinarySearch($a[]$, $left$, $right$, $x$)
Data: $a[]$: the array given, $x$: the item to find
if $left = right$ then
    if $a[left] = x$ then
        return $left$
    else
        return nil
    end
else
    $mid = \lfloor (left + right)/2 \rfloor$
    if $x \leq a[mid]$ then
        BinarySearch($a[]$, $left$, $mid$, $x$)
    else
        BinarySearch($a[]$, $mid + 1$, $right$, $x$)
    end
end
```

First call: BinarySearch($a[]$, 1, n, x).

This algorithm is quite efficient, since each time we eliminate half of the array, with one additional comparison, until there is only one item left, when we will end the process.

Then let's analyse its time complexity. Let $T(n)$ be the number of comparisons needed for n elements, then we will have

$$T(n) = T(n/2) + 1,\ T(1) = 1$$

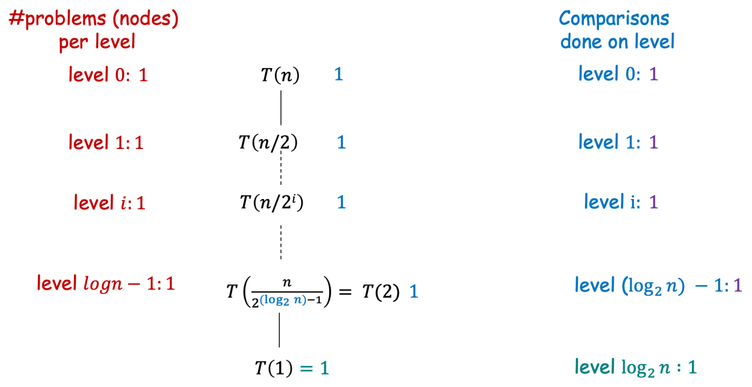
.

Solve this **recurrence**:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= [T(n/4) + 1] + 1 \\ &= T(n/4) + 2 \\ &= \cdots \\ &= T(n/2^i) + i \end{aligned}$$

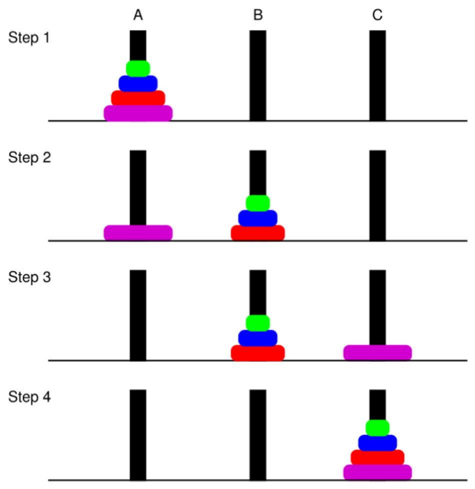
This process ends when reaching $T(1)$, i.e., $i = \log_2 n$, thus, $T(n) = T(1) + \log_2 n = \log_2 n + 1$.

We can also visualize this recurrence with recursion tree: (image from lecture note)



In each recursion step(level), we use 1 comparison(compare *mid* and x), then call recursion on a half of the original array. From the image above, we can easily notice there are total $1 + 1 + \cdots + 1 = 1 + \log_2 n$ comparisons.

2 Example: Towers of Hanoi



In this example, we want to design an algorithm to move all n discs from peg A(start) to peg C(end), with the constraints: (1) move one disc at a time, and (2) cannot put larger disc on a smaller one. We are given another peg B(helper) where we can temporary storage our discs.

We still use the idea of **Divide & Conquer**, consider how we can turn a problem of n discs into a problem of $n-1$? One possible solution is that, we can call recursion on upper $n-1$ discs, i.e., move upper $n-1$ discs to peg B(helper peg), then move the remaining (the biggest) disc to peg C(end peg), and finally move the $n-1$ discs from peg B(helper) to peg C(end). The following pseudocode shows this idea.

Algorithm 2: MoveTower(n, *start*, *helper*, *end*)

```
Input: $n$: num of discs
1 if $n = 1$ then
2     move the only disc from start peg to end peg
3     return
4 else
      // move first $n-1$ from start peg to helper peg
      // so this time ``helper'' peg will be the old end peg
5     MoveTower($n-1$, start, end, helper)
6     move the only disc from start peg to end peg
      // finally move first $n-1$ from helper peg to end peg
      // this time ``helper'' peg will be the old start peg
7     MoveTower($n-1$, helper, start, end)
8 end
```

Now we would like to analyze the time complexity of this algorithm, in other words, how many **steps** are needed. Let $T(n)$ be the num of steps for n discs, each time, we first move $n-1$ disks from *start* to *helper*, costs $T(n-1)$ steps; then we move the biggest disk to *end* peg, costs only 1 step; finally we move $n-1$ disks from *helper* to *end*, again costs $T(n-1)$ steps. To sum up:

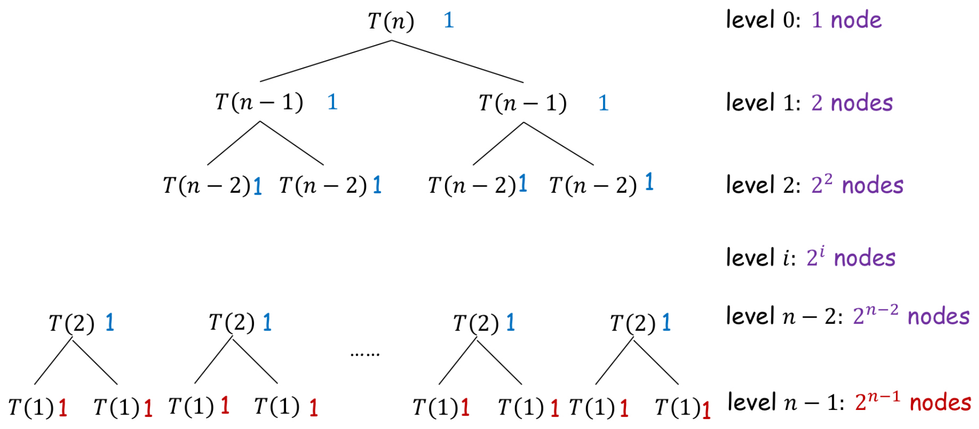
$$T(n) = 2T(n-1) + 1$$

when $n > 1$, and $T(1) = 1$.

Now we solve the recurrence by the **expansion method**:

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2[2T(n-2) + 1] + 1 \\
&= 4T(n-2) + 3 \\
&= 4[2T(n-3) + 1] + 3 \\
&= 8T(n-3) + 7 \\
&= \cdots \\
&= 2^i T(n-i) + (2^i - 1) \\
&= 2^{n-1} T(1) + (2^{n-1} - 1) \\
&= 2^n - 1
\end{aligned}$$

Or, with the recursion tree method:



There are, altogether, $1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-2} + 2^{n-1} = 2^n - 1$ nodes, and we are doing one work(step) each node, then the time complexity is again, $2^n - 1$.

3 Merge Sort

Now we again back to sorting, and we would like to introduce a new algorithm or sorting: Merge Sort. This is a typical example of divide & conquer, and its process is like: (1) we first divide array into two halves, (2) then we recursively sort each half, (which means we continuously divide it into halves, and then halves...) (3) finally **merge** two halves to get a whole.

The **merge** operation may confuse you most. Here it means combine two **sorted lists** into a whole sorted list. For example, given two sorted lists: $A = [2, 5, 7]$ and $B = [3, 4, 6, 10, 12]$, then after **merge** operation, we will get

$result = [2, 3, 4, 5, 6, 7, 10, 12]$. Since these two lists are sorted, we can do this process in $O(n)$ time, where n is the length of result list.(how many numbers in total) The basic idea is: we compare the first item of A and B, put the smaller one, say, $A[1]$, in the first position of result list, then we move on to the next item of A, but compare it still with the **first** item of B(since the first item of B has not yet inserted into result list), and again put the smaller one into result list, then continue move on. An example may help you understand the process:

(1) Compare first items: $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $2 < 3$, so $result = [2]$;

(2) then compare 2nd in A and 1st in B, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $3 < 5$, so $result = [2, 3]$;

(3) continue the process, similarly, $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $4 < 5$, so $result = [2, 3, 4]$;

(4) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $5 < 6$, so $result = [2, 3, 4, 5]$;

(5) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $6 < 7$, so $result = [2, 3, 4, 5, 6]$;

(6) $A = [2, 5, 7], B = [3, 4, 6, 10, 12]$, $7 < 10$, so $result = [2, 3, 4, 5, 6, 7]$;

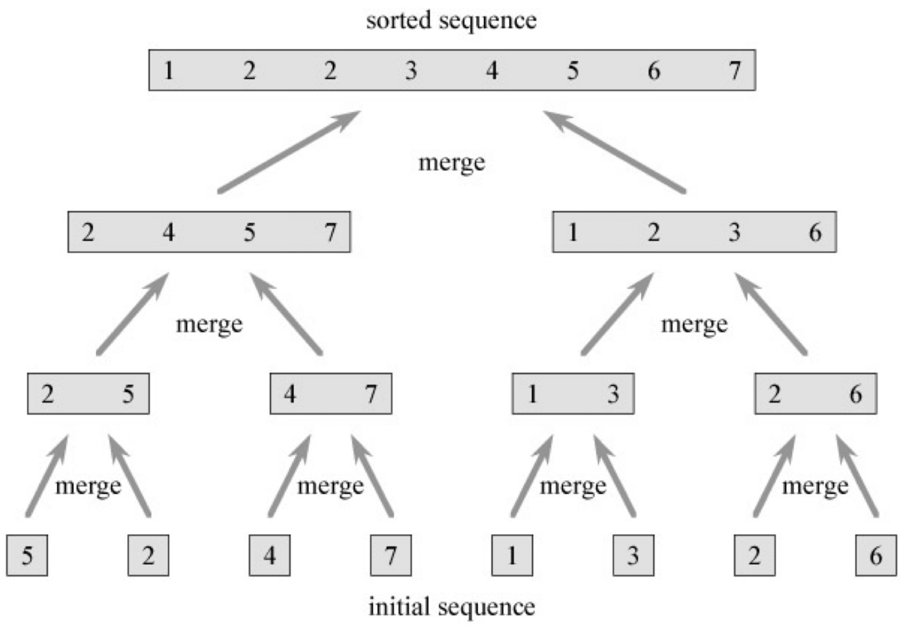
(7) Now, all items in A have already been inserted into result list so that no items can be compared with items in B. Then we simply add remaining items in B to result list, this will, obviously, ensure a sorted result list.(you may think of why) Hence $result = [2, 3, 4, 5, 6, 7, 10, 12]$

The pseudocode below shows the process: (below, append ∞ at the end of two lists can free us from considering the situation that one list is empty, like (7) above. Though different implementation, the idea is entirely the same)

Algorithm 3: Merge(A, $left$, mid, $right$)

```
// merge two sorted list: $A[left \cdots mid]$ and $A[mid+1 \cdots right]$
$L \leftarrow A[left \cdots mid]$, $R \leftarrow A[mid+1 \cdots right]$
append $\infty$ at the end of $L$ and $R$      // see explanation above
$i \leftarrow 1$, $j \leftarrow 1$     // two pointers point at items in $L$ and $R$
for $k \leftarrow left$ to $right$ do
    // always choose the smaller one to insert, and move on
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    end
end
```

After learning how **Merge** works, you now, hopefully, are able to understand how Merge Sort works, with the image below:



We break down array recursively, until one element left, and then merge from bottom to up. The complete pseudocode for Merge Sort is given below:

Algorithm 4: MergeSort(A, $left$, $right$)

```
1 if $left = right$ then
2     return
3 end
4 $mid \leftarrow \lfloor (left + right)/2 \rfloor$
  // recursively divide array into two halves
5 MergeSort($A$, $left$, $mid$)
6 MergeSort($A$, $mid + 1$, $right$)
  // then merge from bottom to up
7 Merge($A$, $left$, $mid$, $right$)
```

Firstly call **MergeSort(**A**, 1,** n**)** to sort array A.

As usual, we are interested in the running time of Merge Sort algorithm. Let $T(n)$ be the running time on an array of size n, it's not hard to find $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, when $n > 1$ and $T(1) = O(1)$.

Here we are actually able to simplify the equation. Firstly we can replace $\leq$ with $=$, since we are insterested in big-O upper bound of $T(n)$; and with the same reason, we can replace $O(n)$ with n, $O(1)$ with 1; finally, we can assume n is a power of 2 for the sake of simplicity but doesn't change the result at all, as $T(n) \leq T(n') \leq T(2n) = O(T(n))$ where n' is the smallest power of 2 such that $n' \geq n$.

Now we want to solve: $T(n) = 2T(n/2) + n$ for $n > 1$, and $T(1) = 1$.

$$\begin{aligned}
T(n) &= 2\left(\frac{n}{2}\right) + n \\
&= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2 \cdot \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 3n \\
&= \cdots \\
&= 2^k \cdot T\left(\frac{n}{2^k}\right) + kn
\end{aligned}$$

We know the process ends with $\frac{n}{2^k} = 1$ i.e. $k = \log_2 n$, thus

$$\begin{aligned}
T(n) &= 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n \cdot \log_2 n \\
&= n \log_2 n + n
\end{aligned}$$

In summary, merge sort runs in $O(n \log n)$ time. It is also worth pointing out that merge sort **always** runs in $O(n \log n)$ time, which means best case is the same as worst case, as you may think of it, the complexity of merge sort *does not depend on inputs*, it always break array down and then merge up.

4 Inversion Numbers

Given an array $A[1 \cdots n]$, we say two elements $A[i]$ and $A[j]$ are **inverted** if $i < j$ but $A[i] > A[j]$, i.e., $A[i]$ appears before $A[j]$ but is larger than $A[j]$. The number of inverted pairs is called the **inversion number** of array A. Actually this is a useful measure, it provides us with an intuitive idea about how "sorted" an array is, larger inversion number implies a more unsorted array.

What may surprise you is that inversion number has a close relation to insertion sort, and more concretely, **the number of swaps used by insertion sort is equals to inversion number.** We can prove it by induction:

Proof. Assuming the array has size n. Basic case $n = 2$ obviously holds.

Inductive step: assume correct for an array of size $n - 1$, i.e., the total number of swaps performed while insertion sorting $A[1 \cdots n-1]$ is equals to the inversion number of $A[1 \cdots n-1]$.

Let $x = A[n]$. Now, the remaining work by insertion sort is that we swap x with all items $A[j]$ such that $j < n$ and $A[j] > x$, notice that the number of those items is the same of inversions in which x **participates**. Therefore, adding these new inversions gives the full inversion number of $A[1 \cdots n]$. $\square$

Now we will consider how to compute the inversion number of a given array with size n. One possible method is we check all (i,j) pairs of given array, this requires $\binom{n}{2} = \Theta(n^2)$ running time. Another method uses the relation we proved above, running insertion sort and count the number of swaps we perform, but this also requires $\Theta(n^2)$ time since insertion sort requires $\Theta(n^2)$. How can we improve that? Come back to topic: divide and conquer!

Similar to previous problems, we divide array into two halves, and recursively count inversions in each half, but notice: we are missing something: we still need to count inversions where a_i and a_j are in different halves! We need to return the sum of those three quantities finally.

So the main problems is that, how we count the third quantity? Consider below situation, the two halves of array are: $[1,5,4,8,10,2]$ and $[6,9,12,11,3,7]$, how would you do that? You may count by hand, and knowing there are $5-3, 4-3, 8-6, \cdots$ and in total 9 inversions with one item in 1st array and another in 2nd. But, it's really time consuming and totally a mess! We have no efficient algorithm to do this but to count one by one.

Fortunately, things will become much better if those two arrays are *sorted*. For example, $A = [3,7,10,14,18,19]$ and $B = [2,11,16,17,23,25]$. How will we do then? We can scan progressively through both lists, and for each item in B, we only need to find the smallest A item larger than it. In the lists above, for example, $A[1] = 3$ is larger than $B[1] = 2$, so all items in A form an inversion pair with $B[1]$; then we move to $B[2] = 11$, we try to find the smallest item in A larger than 11, so we move the pointer in A, $A[2] = 7 < 11, A[3] = 10 < 11$, until $A[4] = 14 > 11$, so each item in $A[4] \cdots A[6]$ can form an inversion pair with $B[2]$. If we continue the process, we will finally get the inversion number formed between A and B, in $O(n)$ time. (Why is $O(n)$? Since we only iterate each item once during the whole process. You may find it quite similar to Merge operation in Merge Sort)

Algorithm 5: Count(A, l, mid, r)

```
// $l$ means left, while $r$ means right.
// notice here, $L[1 \cdots (mid-l+1)]$ is corresponding to $A[l \cdots mid]$, the subscript changes, try
   not be confused later.  $R$ also changes.
1  $L \leftarrow A[l \cdots mid]$, $R \leftarrow A[mid+1 \cdots r]$
2  (here assume) $L, R$ already sorted
3  $i \leftarrow 1$, $j \leftarrow 1$    // two pointers for $L$ and $R$
4  $ans \leftarrow 0$    // total inversion number
   // let $i, j$ iterator over two arrays
5  while $i \le mid-l+1$ and $j \le r-mid$ do
       // looking for smallest $L$ item larger than $R$
6      if $L[i] \le R[j]$ then
7          $i \leftarrow i+1$
8      else
           // Found $L[i] > R[j]$!
           // then $L[i] \cdots L[mid-l+1]$ each can form an inversion pair with $R[j]$, remind here $L$
              subscript is diff from $A$, as stated above
           // so inversion num for $R[j]$ is $mid-l+1-i+1 = mid-l-i+2$
9          $inv \leftarrow (mid-l-i+2)$
10         $ans \leftarrow ans + inv$
11         $j \leftarrow j+1$
12     end
13 end
14 return $ans$
```

And, the whole algorithm for counting the inversion number will be:

Algorithm 6: Count-Inversion(A, l, r)

```
if $l = r$ then
    return 0
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$c_1 \leftarrow$ Count-Inversion($A$, $l$, $mid$)
$c_2 \leftarrow$ Count-Inversion($A$, $mid+1$, $r$)
MergeSort($A$, $l$, $mid$)
MergeSort($A$, $mid+1$, $r$)
$c_3 \leftarrow$ Count($A$, $l$, $mid$, $r$)
return $c_1 + c_2 + c_3$
```

First call: Count-Inversion(A, 1, n).

So far, you may think this is an excellent algorithm since we only use $O(n)$ in each recursion step. However, it isn't! Remember, we have assumed each half is already sorted, but in fact they are random. If we firstly run some sort algorithm, say, Merge Sort, and then do the counting above, the whole running time will be:

$$T(n) = 2T(n/2) + \Theta(n \log n + n) = 2T(n/2) + \Theta(n \log n)$$

One can show $T(n) = \Theta(n \log^2 n)$.

This is, to a certain degree, acceptable, compared to previous $\Theta(n^2)$, but we still want to improve that. We can easily notice the main problem lies in sorting, which uses $\Theta(n \log n)$ in each resursion step. How can we reduce, or even avoid this process?

This is indeed hard to think about, but we can combine the sorting process (more concretely, Merge sort) with the process which we count inversion pairs that form between the two halves. In other words, previously we only do counting between two halves, now we also perform Merge at the same time. What will this lead to? Consider from recursion bottom(1 item), to top, each time we Merge the two halves, as what we did in Merge Sort, and at the same time, count inversion pairs that cross the two halves. And since we Merge from bottom to top, the two halves will always be sorted.(this is exactly the same Merge in Merge Sort)

The paragraph above is still so abstract, at least for myself, perhaps it's better to look at how the algorithm implements.

Algorithm 7: Merge-and-Count(A, l, mid, r)

```
// same as previous algorithm, subscripts for $L, R$ and $A$ are different, remember this
$L \leftarrow A[l \cdots mid],\ R \leftarrow A[mid+1 \cdots r]$
append $\infty$ at the end of $L$ and $R$
$i \leftarrow 1,\ j \leftarrow 1$    // two iteration pointers for $L$ and $R$
$count \leftarrow 0$    // counter for inversion number
for $k \leftarrow l$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i+1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j+1$
        $count \leftarrow count + (mid - l - i + 2)$
    end
end
return $count$
```

As you can find out above, apart from count inversion pairs between L and R, we merge them into a new array A, this is exactly what we did in merge sort, which maintain the "sorted" invariant. With the function above, the complete algorithm for finding inversion number for an array is displayed below.

Algorithm 8: Sort-and-Count(A, l, r)

```
if $l = r$ then
    return 0
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$c_1 \leftarrow$ Sort-and-Count($A$, $l$, $mid$)
$c_2 \leftarrow$ Sort-and-Count($A$, $mid+1$, $r$)
$c_3 \leftarrow$ Merge-and-Count($A$, $l$, $mid$, $r$)
return $c_1 + c_2 + c_3$
```

First call: Sort-and-Count(A, 1, n)

5 The Maximum Subarray Problem

Problem: Given an array of size n, the task is to find the largest possible sum of a contiguous subarray. For example, given $[3, 2, 1, -7, 5, 2, -1, 3, -1]$, subarray $[5, 2, -1, 3]$ has the largest sum among all subarrays, we need to output $5 + 2 + (-1) + 3 = 9$.

We will provide a lot of algorithms to solve this problem.

5.1 brute force algorithm

The simplest idea is, for each pair (i, j), we calculate $A[i] + A[i+1] + \cdots + A[j]$, and record maximum value we have seen along the process.

Algorithm 9: Max-Subarray-Brute-Force(A)

```
maxSum ← A[1]      // can also use -inf to initialize
for i ← 1 to n do
    for j ← i to n do
        // calculate A[i] + ··· + A[j]
        sum ← 0
        for k ← i to j do
            sum ← sum + A[k]
        end
        // if current sum is larger, update maxSum
        if sum > maxSum then
            maxSum ← sum
        end
    end
end
return maxSum
```

This is a very simple algorithm, but requires $\Theta(n^3)$ running time.

5.2 prefix sum

In brute force algorithm, we notice that each time when we calculate $A[i] + A[i+1] + \cdots + A[j]$, we need to iterate through these items, and add them together, which requires a lot of redundant work. The **prefix sum**, say $S[i]$, is defined as $\sum_{j=1}^{i} A[i]$, i.e., the sum of all items before(and include) $A[i]$. If we have a table of all $S[i]$ values, we can now rewrite $\sum_{k=i}^{j} A[k] = S[j] - S[i-1]$. See? That's a $\Theta(1)$ job!

Algorithm 10: Get-Prefix-Sum(A)

```
// S[] records the prefix sum of array A
S[0] = 0
for i = 1 to n do
    S[i] ← S[i − 1] + A[i]
end
return S
```

Algorithm 11: Max-Subarray-Prefix-Sum(A)

```
$maxSum \leftarrow A[1]$
$S \leftarrow$ Get-Prefix-Sum($A$)      // get prefix sum
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        $sum \leftarrow S[j] - S[i-1]$      // calculate $A[i] + \cdots + A[j]$
        if $sum > maxSum$ then
            $maxSum \leftarrow sum$
        end
    end
end
return $maxSum$
```

This reduces the running time of our algorithm to $\Theta(n^2)$. (Calculating prefix sum only requires $\Theta(n)$, so overall $\Theta(n^2 + n) = \Theta(n^2)$)

5.3 divide and conquer

Again, we return to our topic, and again, we try to cut the array into two halves. Similar to **Inversion Number** example, we classified all subarrays into three cases:

1. entirely in the first half
2. entirely in the second half
3. crosses the cut

I think it will not surprise you that the third situation is the most difficult one, while the first two cases, can still be found recursively.

Algorithm 12: Max-Subarray-Divide-Conquer(A, l, r)

```
if $l = r$ then
    return $A[l]$
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$max_1 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $l$, $mid$)
$max_2 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $mid + 1$, $r$)
$max_3 \leftarrow$ Max Subarray that crosses the cut
return $\max\{max_1, max_2, max_3\}$
```

So how can we efficiently calculate max_3? Firstly, consider what do we mean by "crosses the cut"? That should be, the subarray will *at least include both* $A[mid]$ *and* $A[mid+1]$ in order to "cross". Hence these kind of subarray can always be divided into two parts: $A[i \cdots mid]$ and $A[mid+1 \cdots j]$ for some i and j. So in order to find max $A[i] + \cdots + A[j]$, we just find max $A[i] + \cdots + A[mid]$, and $A[mid+1] + \cdots + A[j]$, and finally add them together, this will definitely give us the max value.

Alright, so how can we find i?(and can use exactly the same method to find j) It should be the index that maximize $A[i] + \cdots + A[mid]$. This is much easier since one end, say, mid, is fixed. We initialize $maxSum$ to $-\infty$, then scan from mid towards left, each step we add an item to temporary sum, and update $maxSum$ if sum is larger. When we reach l(left end), we will have already iterated all possible indices i and stored the max sum in $maxSum$.

Algorithm 13: Max-Subarray-Divide-Conquer(A, l, r)

```
if $l = r$ then
    return $A[l]$
end
$mid \leftarrow \lfloor (l+r)/2 \rfloor$
$max_1 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $l$, $mid$)
$max_2 \leftarrow$ Max-Subarray-Divide-Conquer($A$, $mid+1$, $r$)
// now let's count $max_3$
$L_m \leftarrow -\infty,\ R_m \leftarrow -\infty$
$sum \leftarrow 0$
for $i = mid$ down to $l$ do
    $sum \leftarrow sum + A[i]$
    if $sum > L_m$ then
        $L_m \leftarrow sum$
    end
end
$sum \leftarrow 0$
for $i = mid + 1$ to $r$ do
    $sum \leftarrow sum + A[i]$
    if $sum > R_m$ then
        $R_m \leftarrow sum$
    end
end
return $\max\{max_1, max_2, L_m + R_m\}$
```

First call **Max-Subarray-Divide-Conquer(A, 1, n)**.

It's not difficult to find out the process of finding max_3 requires $O(n)$ time, since we just scan throught the array. If let $T(n)$ be the running time of whole algorithm, we will get:

$$T(n) = 2T(n/2) + O(n)$$

This gives $T(n) = O(n \log n)$.

5.4 linear time?

Review the idea of calculating max_3 above, we said that finding max $A[i] + \cdots + A[mid]$ is much easier since mid is a fixed ending point. This gives us an inspiration: for a *fixed* j, finding largest $A[i] + \cdots + A[j] = S[j] - S[i-1]$, is the same as finding the smallest $S[i-1]$.(Recall that $S[]$ is the prefix sum) If we can find the smallest $S[i-1]$ for each j, we will able to find the max subarray.(here $i-1$ must be strictly smaller than j, otherwise the subarray is null)

The process of finding smallest $S[i-1]$ can be easily done during the iteration through array. More concretely, we only need to update $minS = \min\{minS, A[i]\}$ at each step. Below shows the entire algorithm.

Algorithm 14: Max-Subarray-Linear(A)

```
// Here we initialize $minS$ to 0 because at least we can do $S[j] - S[0]$ to ensure the sum is at
   least not smaller than $S[j]$
$maxSum \leftarrow -\infty,\ minS \leftarrow 0$
// for each $j$, find $minS$, and then find $S[j] - minS$
for $j \leftarrow 1$ to $n$ do
    // update overall answer
    if $S[j] - minS > maxSum$ then
        $maxSum \leftarrow S[j] - minS$
    end
    // update $minS$ so far
    if $S[j] < minS$ then
        $minS \leftarrow S[j]$
    end
end
return $maxSum$
```

This algorithm can also be written without calculating prefix sum before, since each time we only use $S[j]$, we only need one variable to record $S[j]$ and accumulate it each time.

Algorithm 15: Max-Subarray-Linear2(A)

```
// $S$ will be all prefix sum so far, i.e., $A[1] + \cdots + A[j]$
$maxSum \leftarrow -\infty,\ minS \leftarrow 0,\ S \leftarrow 0$
for $j \leftarrow 1$ to $n$ do
    $S \leftarrow S + A[j]$      // calculate prefix sum so far
    if $S - minS > maxSum$ then
        $maxSum \leftarrow S - minS$
    end
    // update $minS$ so far
    if $S < minS$ then
        $minS \leftarrow S[j]$
    end
end
return $maxSum$
```

As you can see, this algorithm only requires linear $\Theta(n)$ time. It is indeed a difficult progress that we optimize the algorithm from $\Theta(n^3)$ down to $\Theta(n)$, with lots of new ideas come out. We say this is "More art than science".

5.5 dynamic programming

By using **dynamic programming** ideas, as we will formally introduce later, we can also design quite efficient algorithms, but efficient always requires more thinking. Here we just give you a first taste on DP.

We define $d[i]$ be the max sum of subarray that *ends with* $A[i]$. Here we must contain $A[i]$ in $d[i]$, otherwise, we cannot get $d[i+1]$ from $d[i]$, because it will break the "consecutive" subarray requirement. But if you ask me why we define in such a way, I cannot explain it, and that is the "art of dynamic programming" :).

So when we calculating $d[i]$, we only need to check $d[i-1]$ and $A[i]$, this is the basic idea of dynamic programming, that is, get value from previous values. And if $d[i-1] \leq 0$, which means $d[i-1] + A[i]$ is not larger than $A[i]$ itself! So why should we include $d[i-1]$ then, we just let $d[i] = A[i]$, this will be the max sum with $A[i]$ included. On the contray, if $d[i-1] > 0$, then we should let $d[i] = d[i-1] + A[i]$, since include $d[i-1]$ makes the sum larger, and that is exactly what we want.

Algorithm 16: Max-Subarray-DP(A)

```
$d[0] \leftarrow 0$    // max sum of subarrays end with no item is 0
$maxSum \leftarrow A[1]$    // used to record max sum so far, notice here cannot initialize to 0
for $i = 1$ to $n$ do
    if $d[i-1] \leq 0$ then
        $d[i] \leftarrow A[i]$
    else
        $d[i] \leftarrow d[i-1] + A[i]$
    end
    if $d[i] > maxSum$ then
        $maxSum = d[i]$
    end
end
return $maxSum$
```

This is also a $\Theta(n)$ algorithm, but also, it is not easy to think. But, we can still reduce the space it takes, i.e., space complexity. Notice here we use an array $d[i]$ to record the max sum of subarrays end with $A[i]$, but each time, say, when we calculate $d[i]$, we only use the previous one, say, $d[i-1]$. So it is no need that we use an array to track this: we only need a variable to record the previous d value, that's enough! So we can slightly modify the algorithm as below:

Algorithm 17: Max-Subarray-DP(A)

```
$previousD \leftarrow 0$
$maxSum \leftarrow A[1]$
for $i = 1$ to $n$ do
    if $previousD \leq 0$ then
        $previousD \leftarrow A[i]$
    else
        $previousD \leftarrow previousD + A[i]$
    end
    if $previousD > maxSum$ then
        $maxSum = previousD$
    end
end
return $maxSum$
```

6 The Master Theorem

6.1 Theorem and its proof

The Master Theorem: Let $a \geq 1, b > 1, c \geq 0$ be constants, if $T(n) = aT(n/b) + n^d$, then:

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n), & \text{if } d = \log_b a \\ O(n^{\log_b a}), & \text{if } d < \log_b a \end{cases}$$

There is one kind of proof given in lecture slide, using expansion method. But personally, I like the method below.

Proof. Consider $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$, for the 0th layer of recursion(since we haven't begun recursion), running time is $c \cdot n^d$; for the 1st layer, there are a branches, and each branch has running time $c \cdot \left(\frac{n}{b}\right)^d$, in total $c \cdot \left(\frac{a}{b^d}\right) \cdot n^d$; for the 2nd layer, each branch in 1st layer has a branches, so there are a^2 branches now, with each requires $c \cdot \left(\frac{n}{b^2}\right)^d$, and $c \cdot \left(\frac{a}{b^d}\right)^2 \cdot n^d$ in total. We can easily find the pattern: the running time of k-th layer is $c \cdot \left(\frac{a}{b^d}\right)^k \cdot n^d$.

Add all of them together, we get

$$T(n) = c \cdot n^d \cdot \left[1 + \left(\frac{a}{b^d}\right) + \cdots + \left(\frac{a}{b^d}\right)^k\right]$$

Recall the sum of geometric sequence:

$$1 + p + p^2 + \cdots + p^k = \begin{cases} k+1, & \text{if } p = 1 \\ \frac{p^{k+1} - 1}{p - 1}, & \text{if } p \neq 1 \end{cases}$$

Condition 1: when $a = b^d$, ratio is 1, so $T(n) = O(n^d \log n)$.

Condition 2: when $a < b^d$, the sequence is decreasing, so the sum is determined by the first item(you can also infer from the equation of sum above), then $T(n) = O(n^d)$.

Condition 3: when $a > b^d$, the sequence is increasing, the sum is determined by the last item, this gives $T(n) = n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{\left(b^{\log_b n}\right)^d}\right) = a^{\log_b n} = \left(n^{\log_n a}\right)^{\log_b n} = n^{\left(\frac{\ln a}{\ln n} \cdot \frac{\ln n}{\ln b}\right)} = n^{\log_b a}$. □

6.2 equalities, inequalities and more

to be added.

7 Integer Multiplication

You may first think of using primary school method: i.e., “long multiplication”, and this requires $\Theta(n^2)$ time. We will show that we can do better than this, but the ideas are quite difficult to think of, and actually people used quite a long time to invent the algorithms.