

# Part III

# Chapter 4- Divide and Conquer

# Divide and Conquer

Breaks down problems into smaller easier to solve subproblems, solves these subproblems and then combines the solutions of these subproblems to come up with a solution for the original problem.

# Divide and Conquer

Algorithms that take the divide and conquer approach are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.

# Paradigm

Given a problem of size  $n$ , divide the problem into “ $a$ ” subproblems of size  $n/b$ , where  $a \geq 1$  and  $b > 1$ ;

Solve each sub-problem recursively, the idea being that the problem becomes small enough to solve in a straightforward manner (**constant time**);

Then combine the solution to all of these subproblems into a solution for the original problem.

## 3 steps

**Divide:** the problem into a number of subproblems that are smaller instances of the same problem.

**Conquer:** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner. The recursion “bottoms out” when we reach the base case.

**Combine:** the solutions to the subproblems into the solution for the original problem.

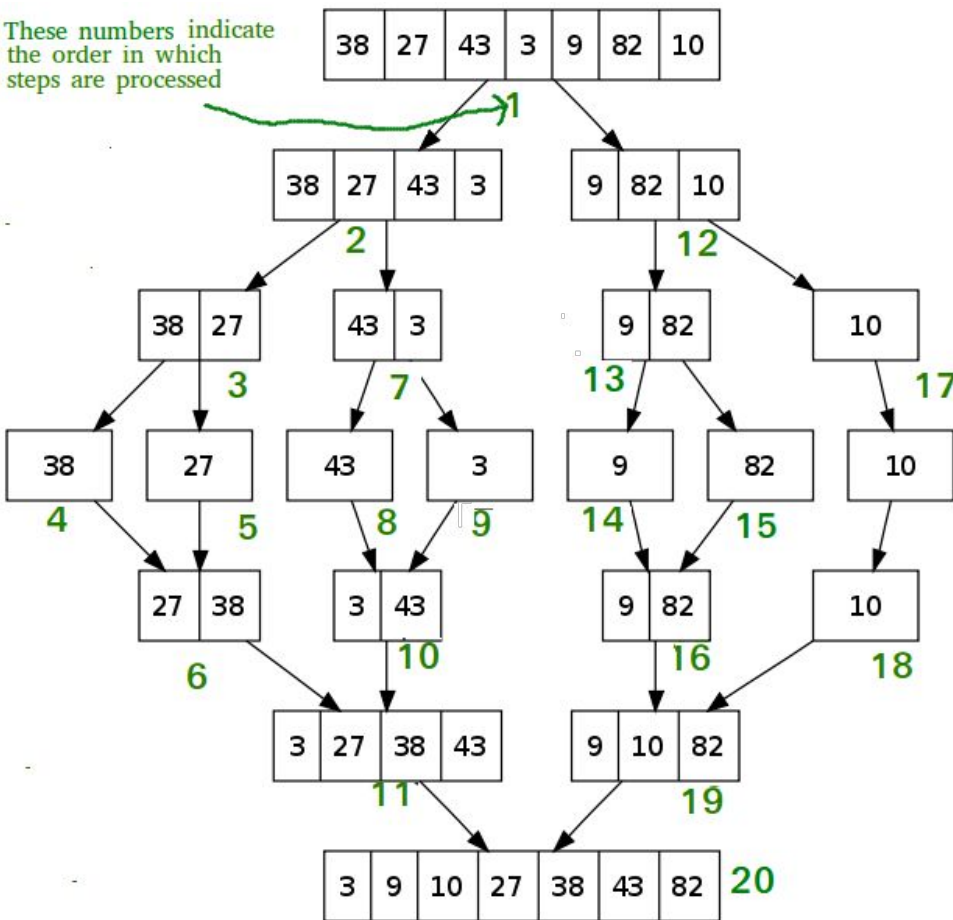
# Merge Sort

**Divide:** Divide the  $n$ -element sequence to be sorted into 2 subsequences of  $n/2$  elements each. ( $a$  and  $b$  are both 2)

**Conquer:** Sort the two subsequences recursively using merge sort. The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

**Combine:** **Merge** the two sorted subsequences to produce the sorted answer.

These numbers indicate  
the order in which  
steps are processed



# Divide and Conquer

## Merge Sort Demo



# Merge Sort

Sort the following array:

[17, 9, 2, -11, 7, 90, 40, -5]

- Merge sort sorts or recurses depth firstly.
- Once you have 2 sorted lists, you call the merge subroutine on the 2 lists.
- How many splits happen? How many times do we cut?
- How many base cases are there?

# Merge Sort Algorithm

```
Merge-Sort (A, p, r){  
    if p>=r    //A.length is at most 1  
        return A[p]  
    q=(p+r)/2  
    L=Merge-Sort(A,p,q)  
    R=Merge-Sort(A,q+1,r)  
    return merge(L,R)  
}
```

p- starting index

r- ending index

q- mid point

**What happens during the merge(L,R) procedure?**

- Merges both lists and produce a sorted list.

**What is the worst time running time for this merge procedure?**

# Merge Sort Algorithm

The merging: merges 2 sorted lists into a bigger sorted list.

**The algorithm:**

Merge( $L_1, L_2$ )

$n = L_1.length + L_2.length$

Total Comparisons:  $n-1$

$T(n) = \Theta(n)$

# Divide and Conquer: Merge Sort Algorithm

MERGE-SORT( $A, p, r$ )

```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

MERGE( $A, p, q, r$ )

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

# Divide and Conquer: Analysis

The time taken to run a divide and conquer algorithms comes from the 3 parts: the time taken to divide into subproblems, the time taken to conquer (solve) and the time taken to combine.

$T(n)$  can be expressed as a recurrence equation.

# Divide and Conquer: Analysis

## Recap

Given a problem of size  $n$ , divide the problem into “ $a$ ” subproblems of size  $n/b$ , where  $a \geq 1$  and  $b > 1$ ;

Solve each sub-problem recursively, the idea being that the problem becomes small enough to solve in a straightforward manner (constant time);

Then combine the solution to all of these subproblems into a solution for the original problem.

# Divide and Conquer: Analysis

- If it takes  $T(n)$  to solve a problem of size  $n$ , then it takes  $T(n/b)$  to solve each subproblem which is  $1/b$  size of  $n$ .
- If we have " $a$ " subproblems, then it takes a total of  $aT(n/b)$  to solve all the subproblems.
- It takes  $D(n)$  time to divide the problem into the subproblems.
- And it takes  $C(n)$  time to combine the solution to all of the subproblems.

# Divide and Conquer: Analysis

In total

$$T(n) = \begin{cases} aT(n/b) + D(n) + C(n) \\ \Theta(1) \end{cases}, \text{ if } n \text{ is small enough}$$



# Divide and Conquer: Merge Sort Analysis

$$T(n) = n \log(n) + (n)$$

$$T(n) = \Theta(n \log(n))$$

# Additional Notes on Divide and Conquer

Hybrid Algorithms- allow for larger base cases, reduces the number of bases cases.

Optimal for parallel processing.

**These algorithms could be inefficient if you keep recalculating solutions to already solved subproblems. \***

Question: Is binary search a divide and conquer algorithm?

# Divide and Conquer: Maximum SubArray Problem

Given an array of positive and negative numbers, find a contiguous subarray with the maximum sum.

[ -1, 3, 4, 0, -5, 9, 1, -2]

What is the maximum sum of consecutive values?

What is the brute-force approach?

# Divide and Conquer: Maximum SubArray Problem

Divide and Conquer Approach:

[ -1, 3, 4, 0, -5, 9, 1, -2]

Start by dividing into 2 subarrays. Will finding the maximum subarray in the 2 subarrays suffice to answer the original problem?

# Divide and Conquer: Maximum SubArray Problem

Therefore, after splitting the array into 2 subarrays, the maximum subarray could lie:

- entirely in the left subarray      //find the max subarray for left subarray
- entirely in the right subarray      //find the max subarray for right subarray
- across the middle boundary      //find the max subarray for cross subarray

# Divide and Conquer: Maximum SubArray Problem

How can we efficiently find the maximum subarray that crosses the mid boundary?

The crossing subarray solves a major problem, it gives us a definitive starting point to search for the max subarray.

Starting from the mid point, spread out left and right comparing sums.

# Divide and Conquer: Maximum SubArray Problem

[ -1, 3, 4, 0 | -5, 9, 1, -2 ]  
Left max sum      Right max sum

Cross subarray max sum = left max sum + right max sum

However, this is not the definitive answer, we need to compare the cross subarray max sum with the left subarray max and the right subarray max and return the highest of these values.

# Divide and Conquer: Merge Sort Analysis

For merge sort,  $a=2$  and  $b=2$ ,  $D(n)=\Theta(1)$ ,  $C(n)=n-1$  or  $\Theta(n)$  :-

$$T(n) = 2(T(n/2)) + n \quad // D(n) \text{ not taken into account because it's a constant}$$

$$T(n/2) = ?$$

...

$$T(1) = c \quad // \text{there are no splits, no comparisons } (1-1==0)$$



# Divide and Conquer: Maximum SubArray Problem

```
MaxSubarray(A,p,r){  
    if p==r  
        return A[p]  
    q=(p+r)/2  
    L=MaxSubarray(A,p,q)  
    R=MaxSubarray(A,q+1,r)  
    C=MaxCrossingSubArray(A,p,q,r)  
    return max(L,R,C)  
}
```

Remember that you have to return the index when the maximum subarray begins and ends as well, not just the maximum sum.

# Divide and Conquer: Maximum SubArray Algorithm

FIND-MAX-CROSSING-SUBARRAY(*A*, *low*, *mid*, *high*)

```
1  left-sum =  $-\infty$ 
2  sum = 0
3  for i = mid downto low
4      sum = sum + A[i]
5      if sum > left-sum
6          left-sum = sum
7          max-left = i
8  right-sum =  $-\infty$ 
9  sum = 0
10 for j = mid + 1 to high
11     sum = sum + A[j]
12     if sum > right-sum
13         right-sum = sum
14         max-right = j
15 return (max-left, max-right, left-sum + right-sum)
```

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
1  if high == low
2      return (low, high, A[low])          // base case
3  else mid =  $\lfloor (\textit{low} + \textit{high}) / 2 \rfloor$ 
4      (left-low, left-high, left-sum) =
          FIND-MAXIMUM-SUBARRAY(A, low, mid)
5      (right-low, right-high, right-sum) =
          FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
6      (cross-low, cross-high, cross-sum) =
          FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
8          return (left-low, left-high, left-sum)
9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10         return (right-low, right-high, right-sum)
11     else return (cross-low, cross-high, cross-sum)
```

# Divide and Conquer: Maximum SubArray Analysis

```
MaxSubarray(A,p,r){
```

```
    if p==r
```

```
        return A[p]
```


```
    q=(p+r)/2  Divide
```

```
    L=MaxSubarray(A,p,q)
```

**Subproblems**

```
    R=MaxSubarray(A,q+1,r)
```

**Work done at this level**

```
    C=MaxCrossingSubArray(A,p,q,r) 
```

```
    return max(L,R,C)  Combine
```

```
}
```

$T(n)$

Base case  $T(1)=0$

$D(n)=\Theta(1)$  -constant

$T(n/2)$

$T(n/2)$

$n$  or  $\Theta(n)$

$C(n)=\Theta(1)$

# Divide and Conquer: Maximum SubArray Analysis

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

Same as merge sort

Linear work gets done at each level.

# Solving Recurrences

# Divide and Conquer: Analysis

In total

$$T(n) = \begin{cases} aT(n/b) + D(n) + C(n) \\ \Theta(1) \end{cases}, \text{ if } n \text{ is small enough}$$

# Solving Recurrences

3 ways:

- Substitution Method
- Recursion Tree
- Master Theorem

# Substitution Method

2 steps:

1. Guess the solution (the running time)
2. Use mathematical induction to find the constants and show that the solution works

Try to proof for  $n$  by proofing for some  $m$ , where  $m < n$ .

This method is powerful, but we must be able to guess the form of the answer in order to apply it.



# Substitution Method: Examples

Merge Sort:  $T(n) = 2 T(n/2) + n$

# Substitution Method: Examples

Merge Sort:  $T(n) = 2 T(n/2) + n$

Step: Guess an answer ( a possible bounding function, say big oh), assume it's true for all  $m < n$  and prove that the function bounds the running time equation.

# Substitution Method: Examples

Merge Sort:  $T(n) = 2 T(n/2) + n$

Guess:  $T(n) = O(n \lg n)$  which means  $T(n) \leq c n \lg n$  ←prove this

By assuming  $T(m) \leq c m \lg m$  is true for all  $m < n$

# Substitution Method: Concerns

1. No general way to guess the correct solutions to recurrences.
2. Sometimes you might correctly guess an asymptotic bound on the solution of a recurrence, but somehow the math fails to work out in the induction. The problem frequently turns out to be that the inductive assumption is not strong enough to prove the detailed bound.
3. Prone to errors: we need to prove the exact form of the inductive hypothesis

# Substitution Method: Examples

Solve

1.  $T(n) \leq T(n-1) + 1$
2.  $T(n) \leq 2 T(n/2) + 1$
3.  $T(n) = 2 T(\sqrt{n}) + \lg n$
4.  $T(n) \leq 3 T(n/7) + 2 T(n/4) + n$ , guess  $T(n) = O(n)$

# Recursion Tree

Best used to generate good guesses to prove using the substitution method.

You often can afford to be sloppy with your assumptions since you will be verifying your guess later on.

Example:  $T(n) = 3T(n/4) + \Theta(n^2)$

$$T(n) = T(n/3) + T(2n/3) + \Theta(n)$$

# Master Theorem

The master method provides a “cookbook” method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where  $a \geq 1$ ,  $b > 1$  are constants and  $f(n)$  is an asymptotically positive function. The function  $f(n)$  encompasses the cost of dividing the problem and combining the results of the subproblems.

To use the master method, you will need to memorize three cases.

# Master Theorem

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the nonnegative integers by the recurrence:

$$T(n) = aT(n/b) + f(n)$$

where we interpret  $n/b$  to mean either  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ . Then  $T(n)$  has the following asymptotic bounds:



# Master Theorem

1. If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$
2. If  $f(n) = \Theta(n^{\log_b a})$ , then  $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$  and if  $af(n/b) \leq c f(n)$  for some constant  $c < 1$  and  $n > n_0$ , then  $T(n) = \Theta(f(n))$

# Master Theorem: Examples

1.  $T(n) = 2T(n/2) + n$

2.  $T(n) = 9T(n/3) + n$

3.  $T(n) = T(2n/3) + 1$

4.  $T(n) = 3T(n/4) + n \lg n$

5.  $T(n) = 2T(n/2) + n/\log n$

a	b	$\log_a b$	$f(n)$	Case	$\Theta(T(n))$
---	---	------------	--------	------	----------------

# Master Theorem: Concerns

Needs a single  $b$  ( subarrays of same sizes) to work.

The three cases do not cover all the possibilities for  $f(n)$ , cases may fall between case 1 & 2 or case 2 & 3.

$a$  (the number of subarrays) needs to be a constant.

$T(n)$  needs to grow in relation to  $f(n)$