Divide and Conquer

Lecture 2 a

Recurrence Relations

Many algorithms particularly divide and conquer algorithms have time complexities which are naturally modeled by recurrence relations

A recurrence relation is an equation which is defined in terms of itself-

Why are recurrences good things?

1. Many natural functions are easily expressed as recurrences

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n \quad (polynomial)$$

$$a_n = 2 * a_{n-1}, a_1 = 1, \longrightarrow a_n = 2^{n-1} \quad (exponential)$$

$$a_n = n * a_{n-1}, a_1 = 1, \longrightarrow a_n = n! \quad (weird\ function)$$

2. It is often easy to find a recurrence as the solution of a counting problem
Solving the recurrence can be done for many special cases as we will see although it is somewhat of an art

Recursion is Mathematical Induction

In both we have general and boundary conditions

With the general condition breaking the problem into smaller and smaller pieces

The initial or boundary condition terminate the recursion

Realize that linear, finite history, constant coefficient recurrences
always can be solved

Guess a solution and prove by induction

To guess the solution play around with small values for insight
Note that you can do inductive proofs with the big-Os notations just be sure you use it right

Example: Show that $T(n) \leq c \cdot n \lg n$ for large enough c and n.

Assume that it is true for $n/2$, then

$$\begin{aligned}
T(n) &\leq 2c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\
&\leq cn \lg(\lfloor n/2 \rfloor) + n \quad \text{dropping floors makes it bigger} \\
&= cn(\lg n - (\lg 2 = 1)) + n \quad \text{log of division} \\
&= cn \lg n - cn + n \\
&\leq cn \lg n \quad \text{whenever } c > 1
\end{aligned}$$

Starting with basis cases $T(2) = 4$, $T(3) = 5$, lets us complete the proof for $c \geq 2$.

- Try backsubstituting until you know what is going on
- Also known as the iteration method Plug the recur rence back into itself until you see a pattern

Example: $T(n) = 3T(\lfloor n/4 \rfloor) + n$. Try backsubstituting:

$$\begin{aligned} T(n) &= n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3\lfloor n/4 \rfloor + 9(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor)) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor) \end{aligned}$$

The $(3/4)^n$ term should now be obvious.

- Although there are only $\log_4 n$ terms before we get toT(1),
- it doesn't hurt to sum them all since this is a fast growing geometric series

$$T(n) \leq n \sum_{i=0}^{\infty} \frac{3}{4}^{i} + \Theta(n^{\log_4 3} \times T(1))$$

$$T(n) = 4n + o(n) = O(n)$$

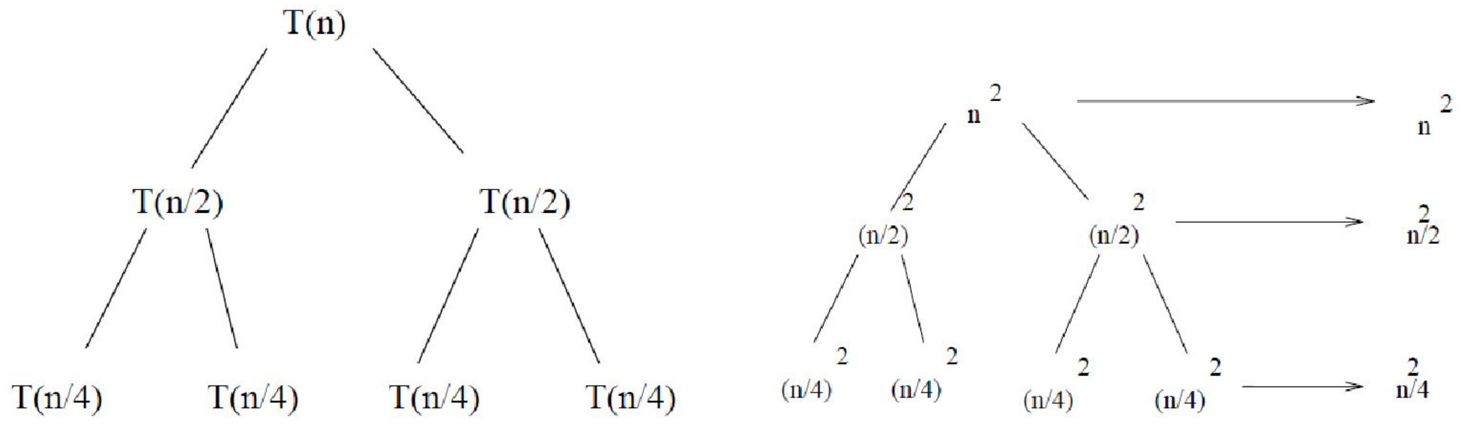
Recursion Trees

Drawing a picture of the backsubstitution process gives an idea of what is going on
We must keep track of two things

the size of the remaining argument to the recurrence and
the additive stuff to be accumulated during this call

Example: $T(n) = 2T(n/2) + n^2$

The remaining arguments are on the left
the additive terms on the right



Although this tree has height lg n the total sum at each level decreases geometrically so

$$T(n) = \sum_{i=0}^{\infty} n^2/2^i = n^2 \sum_{i=0}^{\infty} 1/2^i = \Theta(n^2)$$

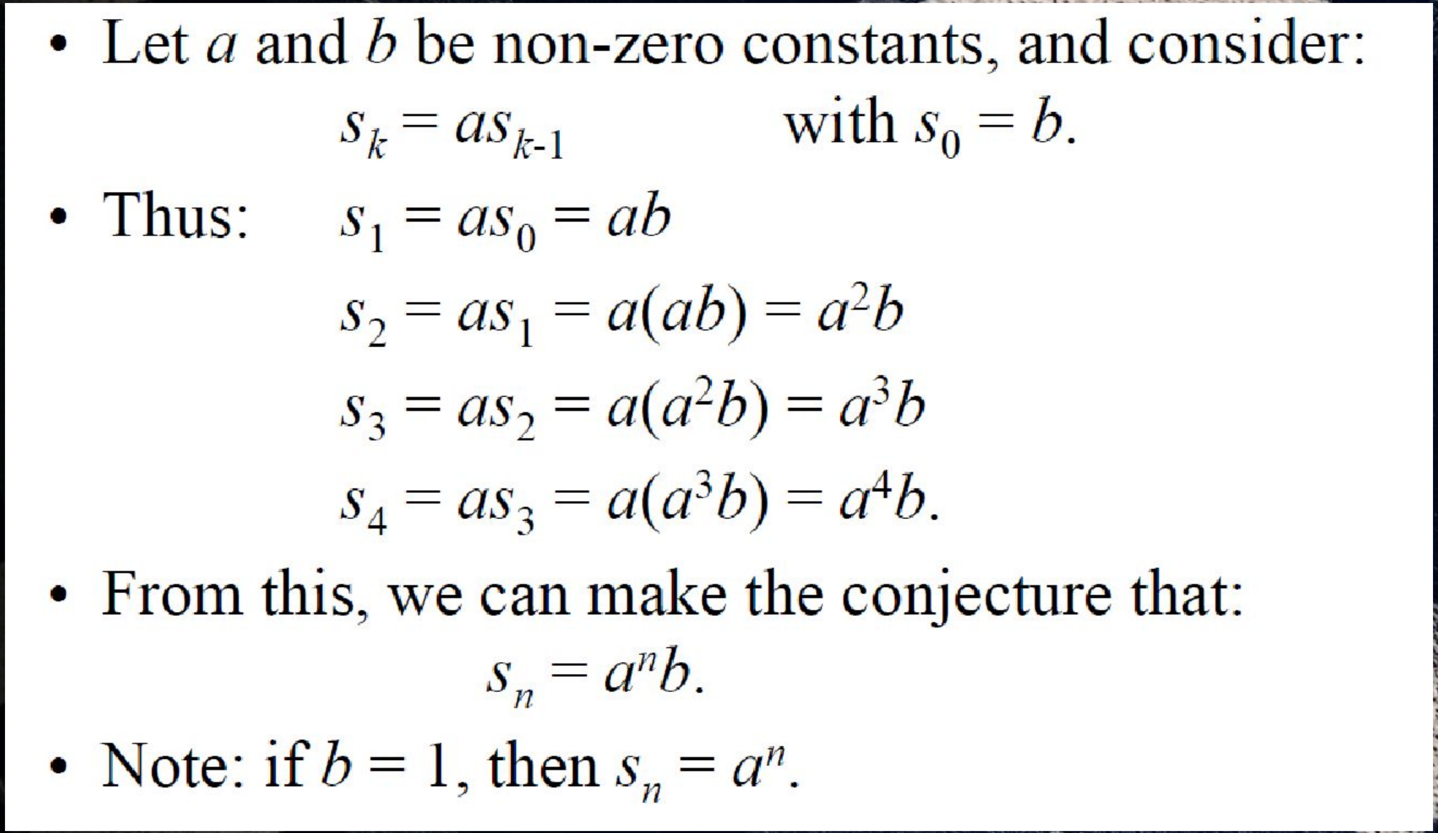
The recursion tree framework made this much easier to see than with algebraic backsubstitution

The Method of Iteration

- Let $\{a_i\}$ be the sequence defined by:
$$a_k = a_{k-1} + 2 \qquad \text{with } a_0 = 1.$$
- Plugging values of k into the relation, we get:
$$a_1 = a_0 + 2 = 1 + 2$$
$$a_2 = a_1 + 2 = 1 + 2 + 2 = 1 + 2(2)$$
$$a_3 = a_2 + 2 = 1 + 2 + 2 + 2 = 1 + 3(2)$$
$$a_4 = a_3 + 2 = 1 + 2 + 2 + 2 + 2 = 1 + 4(2)$$
- Continuing in this fashion reinforces the apparent pattern that $a_n = 1 + n(2) = 1 + 2n$.

This *brute force* technique is the *Method of Iteration*.

A Geometric Sequence



- Let a and b be non-zero constants, and consider:

$$s_k = as_{k-1} \qquad \text{with } s_0 = b.$$

- Thus:

$$\begin{aligned} s_1 &= as_0 = ab \\ s_2 &= as_1 = a(ab) = a^2b \\ s_3 &= as_2 = a(a^2b) = a^3b \\ s_4 &= as_3 = a(a^3b) = a^4b. \end{aligned}$$

- From this, we can make the conjecture that:

$$s_n = a^n b.$$

- Note: if $b = 1$, then $s_n = a^n$.

A Perturbed Geometric Sequence

- Let a be non-zero constants, and consider:

$$s_k = as_{k\text{-}1} + 1 \qquad \text{with } s_0 = 1.$$

- Thus: $s_1 = as_0 + 1 = a + 1$

$$s_2 = as_1 + 1 = a(a+1) + 1 = a^2 + a + 1$$

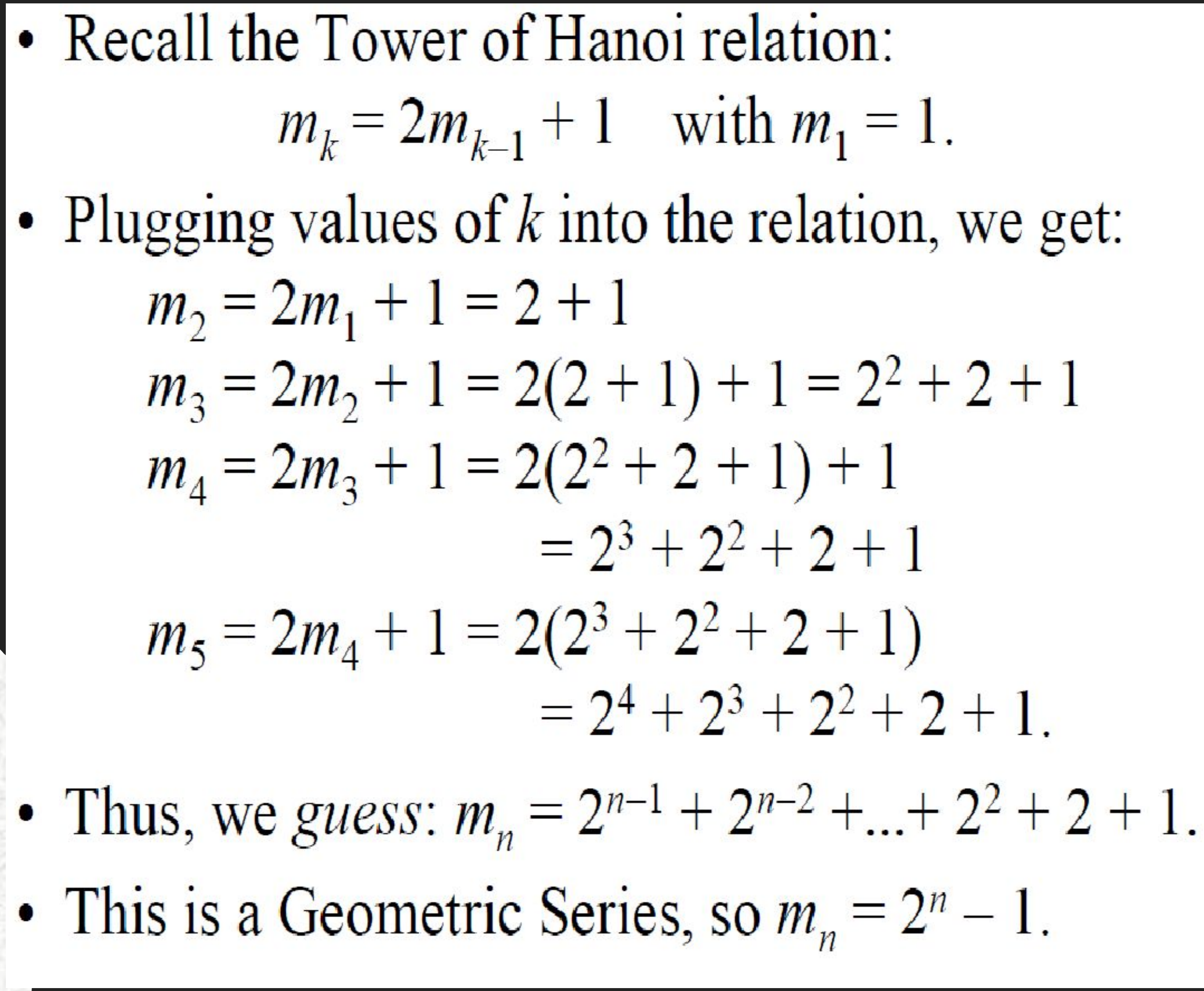
$$s_3 = as_2 + 1 = a(a^2 + a + 1) + 1$$
$$= a^3 + a^2 + a + 1$$

$$s_4 = as_3 + 1 = a(a^3 + a^2 + a + 1)$$
$$= a^4 + a^3 + a^2 + a + 1.$$

- It appears that $s_n = a^n + a^{n-1} + \ldots + a^2 + a + 1$, so

$$s_n = (a^{n+1} - 1) \,/\, (a - 1)\,.$$

The Tower of Hanoi



- Recall the Tower of Hanoi relation:

$$m_k = 2m_{k-1} + 1 \quad \text{with } m_1 = 1.$$

- Plugging values of k into the relation, we get:

$$\begin{aligned}
m_2 &= 2m_1 + 1 = 2 + 1 \\
m_3 &= 2m_2 + 1 = 2(2+1) + 1 = 2^2 + 2 + 1 \\
m_4 &= 2m_3 + 1 = 2(2^2 + 2 + 1) + 1 \\
&= 2^3 + 2^2 + 2 + 1 \\
m_5 &= 2m_4 + 1 = 2(2^3 + 2^2 + 2 + 1) \\
&= 2^4 + 2^3 + 2^2 + 2 + 1.
\end{aligned}$$

- Thus, we *guess*: $m_n = 2^{n-1} + 2^{n-2} + \ldots + 2^2 + 2 + 1.$
- This is a Geometric Series, so $m_n = 2^n - 1.$

Observations

- In solving these recurrence relations, we point out the following observations:

1. Each recurrence relation looks only 1 step back; that is each relation has been of the form $s_n = \mathrm{F}(s_{n-1})$;

2. We have relied on *luck* to solve the relation, in that we have needed to observe a pattern of behavior and formulated the solution based on the pattern;

3. The initial condition has played a role in making this pattern evident;

4. Generating a formula from the generalization of the pattern looks back to our study of induction.

Verifying Solutions

- Once we "guess" the form of the solution for a recurrence relation, we need to verify it is, in fact,
  the solution.
- We use Mathematical Induction to do this.
- For example, in the Tower of Hanoi game, we conjecture that the solution is
  $m_n = 2^n - 1$.
- Basis Step: $m_1 = 1$ (by playing the game), and $2^1 - 1 = 2 - 1 = 1$,
  therefore $m_1 = 2^1 - 1$.
- Inductive Step: Recall the recurrence relation is
  $m_k = 2m_{k-1} + 1$. Assume $m_k = 2^k - 1$.

Verifying Solutions (*cont'd.*)

- Inductive Step: Show $m_{k+1} = 2^{k+1} - 1$.

$$\begin{aligned} \text{Now, } m_{k+1} &= 2m_k + 1 \\ &= 2(2^k - 1) + 1 \\ &= 2 \cdot 2^k - 2 + 1 \\ &= 2^{k+1} - 1. \end{aligned}$$

Therefore $m_k = 2^k - 1$ is the solution to $m_k = 2m_{k-1} + 1$, when $m_1 = 1$. QED

Iterative Vs Recursive Methods

The iteration method is a "brute force" method of solving a recurrence relation.

The general idea is to iteratively substitute the value of the recurrent part of the equation until a pattern (usually a summation) is noticed.
At a specific point the summation can be used to evaluate the recurrence.

Suppose we have the following recurrence relation:

$$T(1) = \Theta(1)$$
$$T(n) = c_1 + 2(T(n-1))$$

then:

$$T(n) = c_1 + 2(T(n-1))$$
$$T(n) = c_1 + 2(c_1 + 2(T(n-2)))$$
$$T(n) = c_1 + 2(c_1 + 2(c_1 + 2(T(n-3))))$$
$$T(n) = c_1 + 2(c_1 + 2(c_1 + 2(c_1 + 2(T(n-4)))))$$

Iterative Vs Recursive Methods

Each iteration, the recurrence is replaced with its value as established by the original recurrence relation. Now that we've done a few iterations, let's simplify and see if there is a recognizable pattern.

$$T(n) = c_1 + 2\left(c_1 + 2\left(c_1 + 2\left(c_1 + 2\left(T(n-4)\right)\right)\right)\right)$$

$$T(n) = c_1 + 2c_1 + 4\left(c_1 + 2\left(c_1 + 2\left(T(n-4)\right)\right)\right)$$

$$T(n) = c_1 + 2c_1 + 4c_1 + 8\left(c_1 + 2\left(T(n-4)\right)\right)$$

$$T(n) = c_1 + 2c_1 + 4c_1 + 8c_1 + 16\left(T(n-4)\right)$$

$$T(n) = 2^0c_1 + 2^1c_1 + 2^2c_1 + 2^3c_1 + 2^4\left(T(n-4)\right)$$

There definitely seems to be a pattern here. Each iteration we're adding a 2^i term; where *i* is the number of iterations that we have made. Now the question is: When is this going to stop?

From the original problem we know that:

. We can say that: $T(1) = \Theta(1)$

$$T(n-i) = 1 \text{ when } i = n-1$$

Iterative Vs Recursive Methods

Now we can write our simplified equation in terms of *i*

$$T(n) = 2^i T(n-i) + 2^{i-1}c_1 + 2^{i-2}c_1 + \ldots + 2^0 c_1$$

Now $n - i = n - (n-1) = 1$

and we know that $T(1) = \Theta(1)$

So, we can re-write our equation as:

$$T(n) = 2^i\Theta(1) + 2^{i-1}\Theta(1) + 2^{i-2}\Theta(1) + \ldots + 2^0\Theta(1)$$
$$T(n) = \Theta(1)\left(2^i + 2^{i-1} + 2^{i-2} + \ldots + 2^0\right)$$
$$T(n) = \Theta(1)\sum_{i=0}^{n-1} 2^i$$
$$T(n) = \Theta(1)\frac{2^{((n-1)+1)} - 1}{2-1}$$
$$T(n) = \Theta(1)\left(2^n - 1\right)$$
$$T(n) = \Theta\left(2^n\right)$$

So, the time complexity of this recurrence relations is

$$\Theta\left(2^n\right)$$

Divide and Conquer: Basics

Divide-and-conquer solves a large problem by recursively breaking it down into smaller subproblems until they can be solved directly.

Divide-and-conquer works in three steps:
divide,
conquer, and
combine.

It is more efficient than brute force approaches. It is prone to stack overflow error due to the use of recursion.

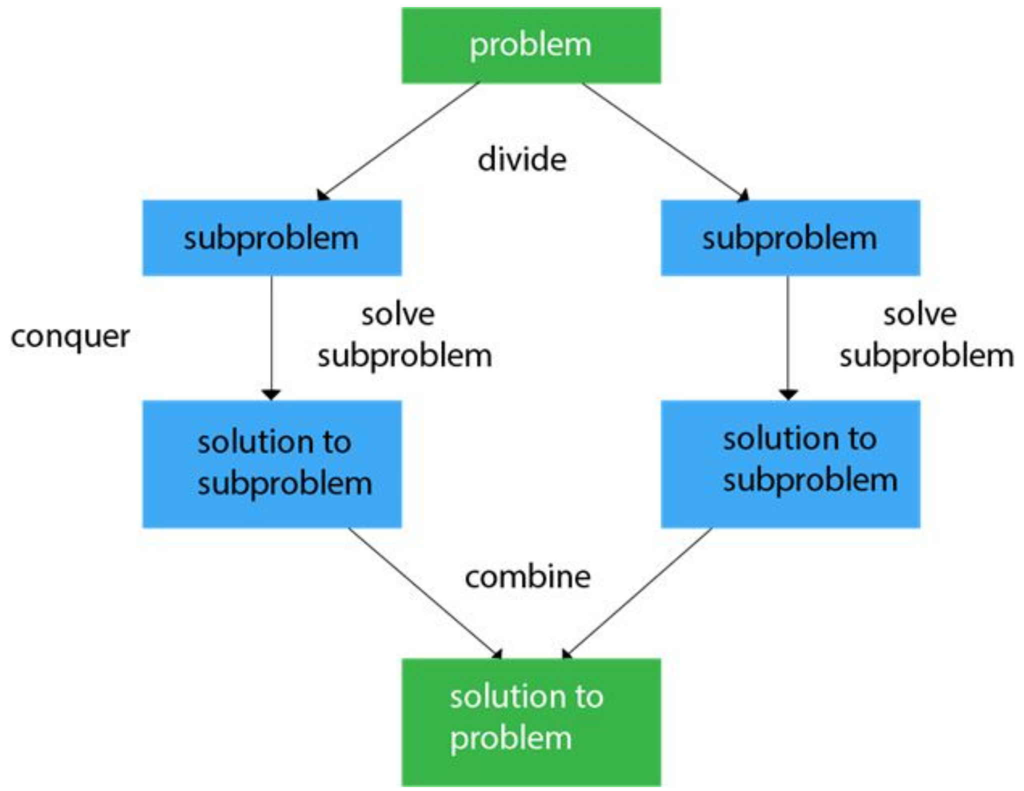
In algorithmic methods, the design is to take a dispute on a huge input,
break the input into minor pieces,
decide the problem on each of the small pieces, and
then merge the piecewise solutions into a global solution.

This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer: Basics

Divide and Conquer algorithm consists of a dispute using the following three steps.

- **Divide** the original problem into a set of subproblems.
- **Conquer:** Solve every subproblem individually, recursively.
- **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

3. **Relational Formula:** It is the formula that we generate from the given technique.
   After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. **Stopping Condition:** When we break the problem using Divide & Conquer Strategy,
   then we need to know that for how much time, we need to apply divide & Conquer.
   So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

Binary Search:

The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search.

It works by comparing the target value with the middle element existing in a sorted array.

After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half.

We will again consider the middle element and compare it with the target value.
The process keeps on repeating until the target value is met.

If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.

Applications of Divide and Conquer Approach:

Quicksort:

It is the most efficient sorting algorithm, which is also known as partition-exchange sort.

It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays.

The partition is made by comparing each of the elements with the pivot value.

It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

Merge Sort:

It is a sorting algorithm that sorts an array by making comparisons.

It starts by dividing an array into sub-array and then recursively sorts each of them.

After the sorting is done, it merges them back.

Quicksort

QuickSort *is a sorting algorithm based on the* [*Divide and Conquer algorithm*] *that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.*

The key process in ***quickSort*** *is a* ***partition()****.*

The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array

and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Quicksort

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

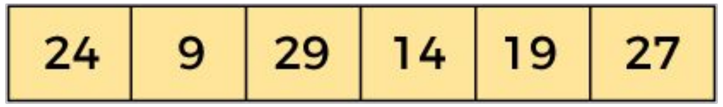
Combine: Combine the already sorted array.

Working of Quick Sort Algorithm

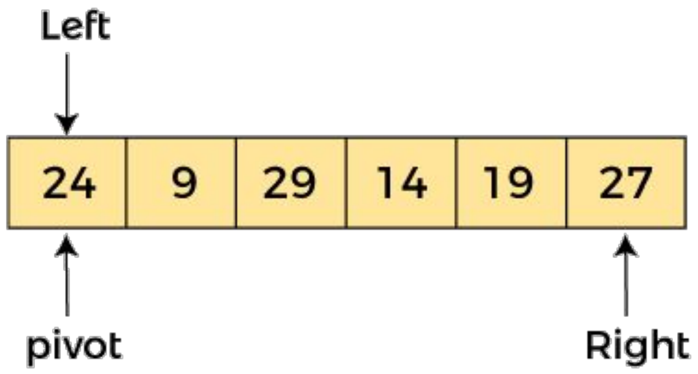
Now, examine the operation of the Quicksort Algorithm.

To understand the working of quick sort, let's take an unsorted array.

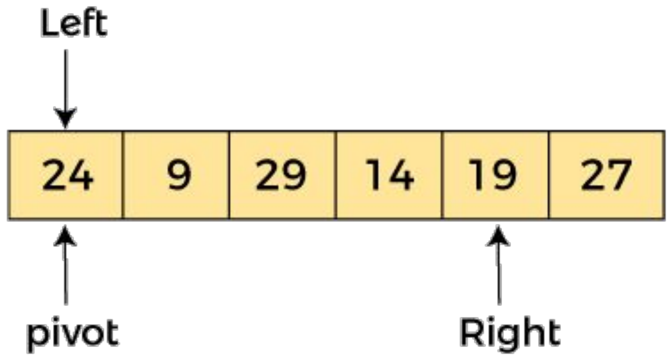
Let the elements of array are -



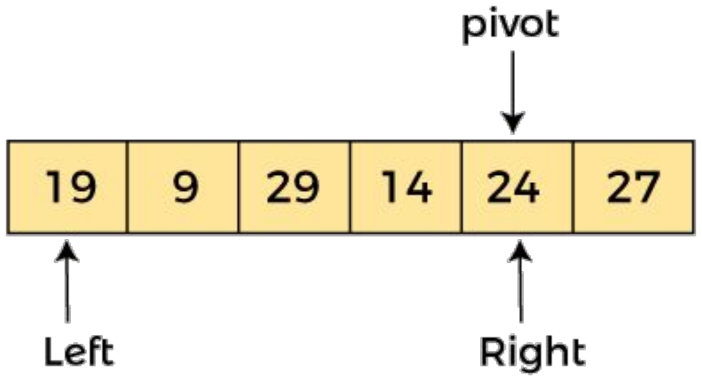
In the given array, we consider the leftmost element as pivot.
So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.
Since, pivot is at left, so algorithm starts from right and move towards left.



Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



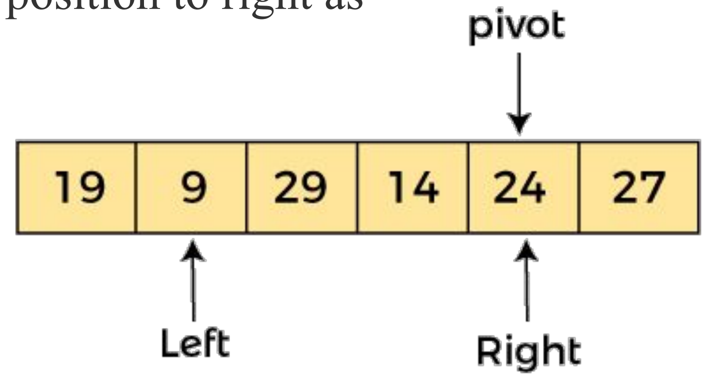
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
Because, a[pivot] > a[right],
so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



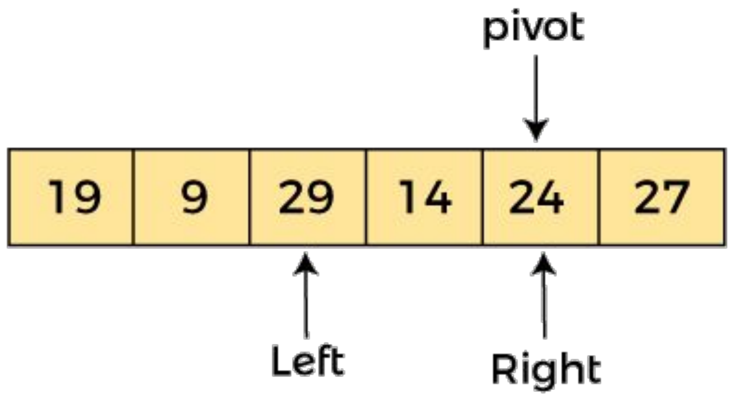
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24.
Since, pivot is at right, so algorithm starts from left and moves to right.

Working of Quick Sort Algorithm

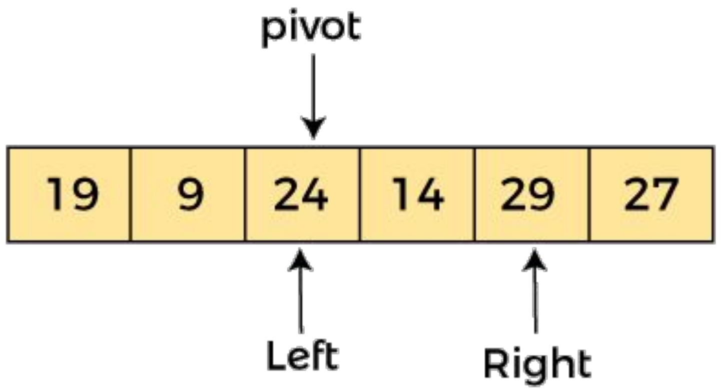
As a[pivot] > a[left], so algorithm moves one position to right as -



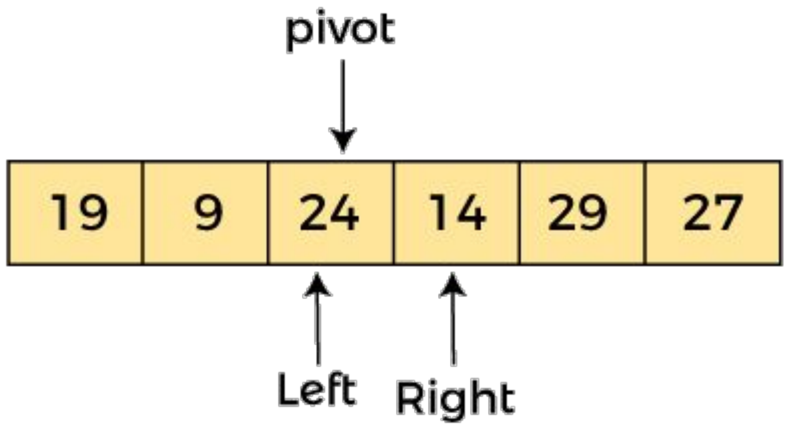
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24.
As a[pivot] > a[left], so algorithm moves one position to right as -



Now, a[left] = 29, a[right] = 24, and a[pivot] = 24.
As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -

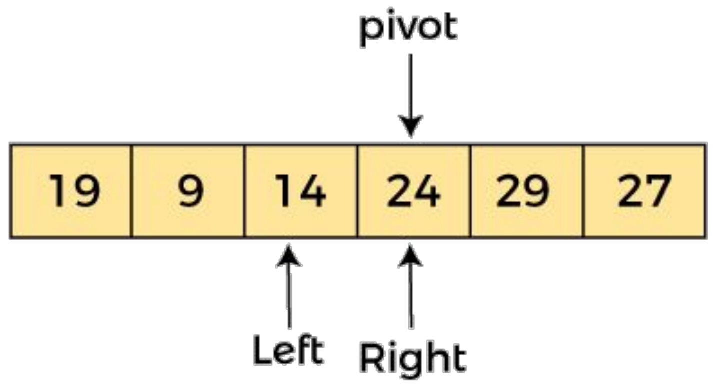


Since, pivot is at left, so algorithm starts from right, and move to left.
Now, a[left] = 24, a[right] = 29, and a[pivot] = 24.
As a[pivot] < a[right], so algorithm moves one position to left, as -

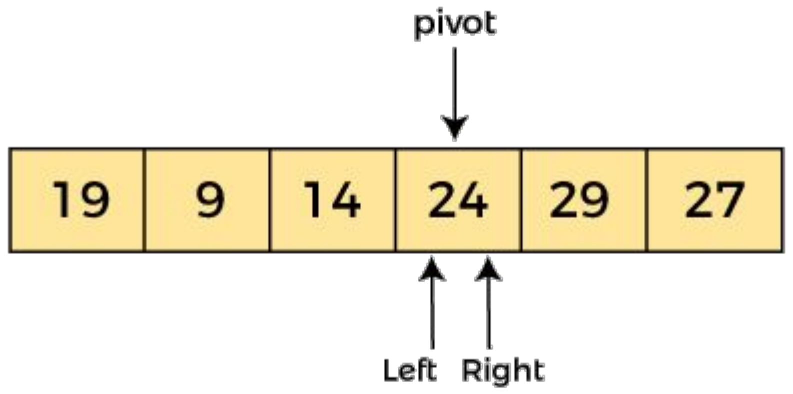


Working of Quick Sort Algorithm

Now, a[pivot] = 24, a[left] = 24, and a[right] = 14.
As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



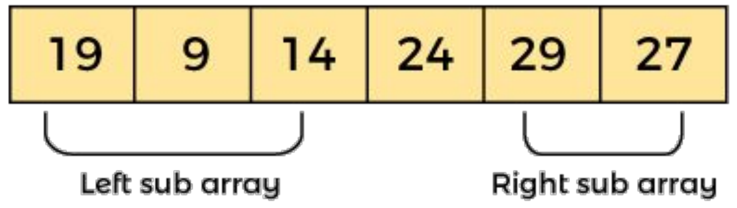
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



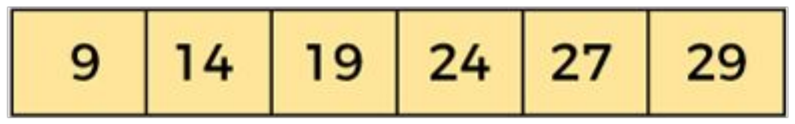
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24.
So, pivot, left and right are pointing the same element.
It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Quicksort Complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case.

Time Complexity

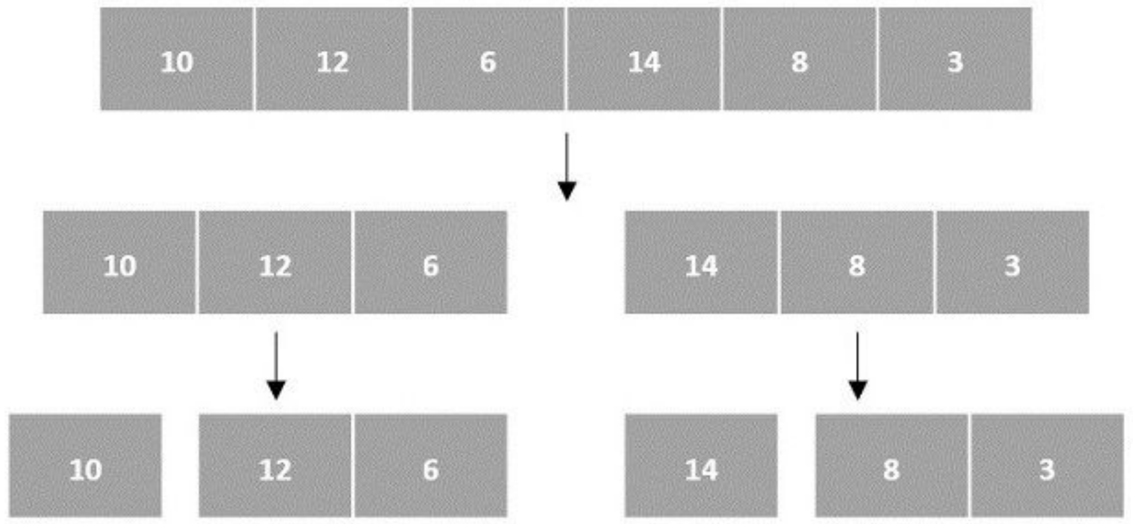
- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.

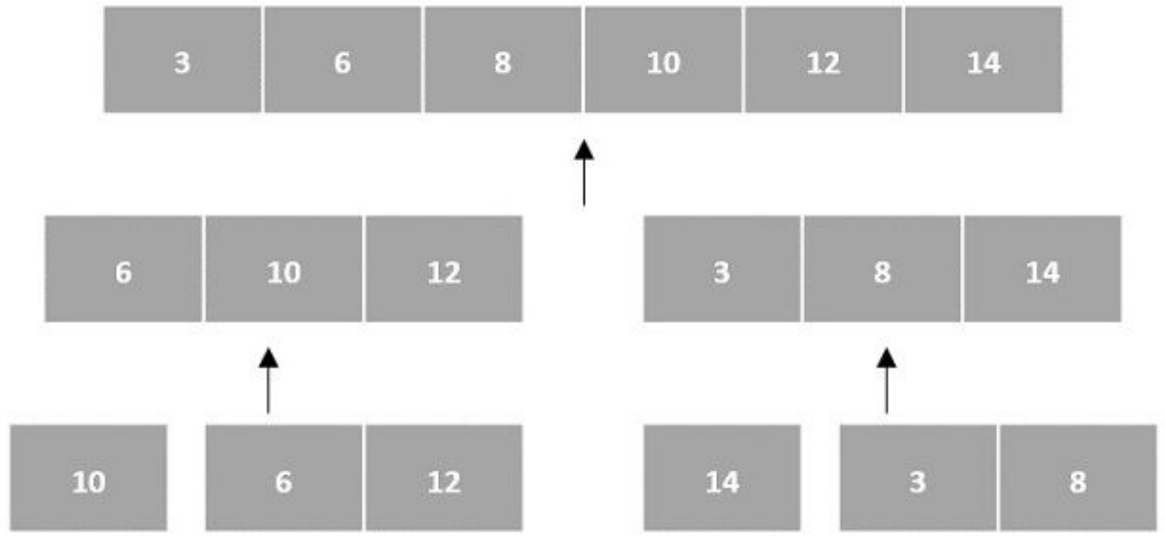
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element.
  Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice.
Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways.
Worst case in quicksort can be avoided by choosing the right pivot element.

In the input for a sorting algorithm below, the array input is divided into subproblems until they cannot be divided further



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).



Merge sort :Pseudocode

Step 1: If it is only one element in the list, consider it already sorted, so return.

Step 2: Divide the list recursively into two halves until it can no more be divided.

Step 3: Merge the smaller lists into new list in sorted order.

```
var l1 as array = a[0] ... a[n/2]
    var l2 as array = a[n/2+1] ... a[n]
    l1 = mergesort( l1 )
    l2 = mergesort( l2 )
    return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while
  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while
  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c
end procedure
```

Time Complexity Analysis:
Consider the following terminologies:

$T(k)$ = time taken to sort k elements
$M(k)$ = time taken to merge k elements

So, it can be written

$T(N) = 2 * T(N/2) + M(N)$
$= 2 * T(N/2) + \text{constant} * N$

These N/2 elements are further divided into two halves. So,

$T(N) = 2 * [2 * T(N/4) + \text{constant} * N/2] + \text{constant} * N$
$= 4 * T(N/4) + 2 * N * \text{constant}$
. . .
$= 2^k * T(N/2^k) + k * N * \text{constant}$

It can be divided maximum until there is one element left.
So, then $N/2^k = 1$. $k = \log_2 N$

$T(N) = N * T(1) + N * \log_2 N * \text{constant}$
$= N + N * \log_2 N$

Therefore, the time complexity is $O(N * \log_2 N)$.

Applications of Divide and Conquer Approach:

Closest Pair of Points:

It is a problem of computational geometry.

This algorithm emphasizes finding out the closest pair of points in a metric space,

given n points, such that the distance between the pair of points should be minimal.

Applications of Divide and Conquer Approach:

Strassen's Algorithm:

It is an algorithm for matrix multiplication, which is named after Volker Strassen.
It has proven to be much faster than the traditional algorithm when works on large matrices.

Karatsuba algorithm for fast multiplication:

It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962.

It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Divide and Conquer: Advantages

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle.
  It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively.
  This algorithm is much faster than other algorithms.

- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.

- It is more proficient than that of its counterpart Brute Force technique.

- Since these algorithms inhibit parallelism, it does not involve any modification and is handled effectively by systems incorporating parallel processing.

Divide and Conquer: Disadvantages

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Thank You