

Divide and Conquer

General Method:

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from $O(n^2)$ to $O(n \log n)$ to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

- Divide: Divide the problem into a number of sub problems. The sub problems are solved recursively.
- Conquer: The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A **control abstraction** is a procedure in which the flow of control is clearly defined, but the primary operations are specified by other procedures whose exact details are not defined.

In the Divide and Conquer technique, the control abstraction is represented as:

DANDC(P)

Where:

- P is the problem to be solved

```
DANDC(P)
{
    if SMALL(P) then
        return S(P);
    else
    {
        divide P into smaller instances P1, P2, ..., Pk (k ≥ 1);
        apply DANDC to each of these subproblems;
        return COMBINE(DANDC(P1), DANDC(P2), ...,
            DANDC(Pk));
    }
}
```

Explanation of Components

1. SMALL(P)

- A Boolean function.
- Determines whether the problem size is small enough to be solved directly.
- If true, the problem is solved without further division.

2. S(P)

- The direct solution method used when the problem is small.
- Represents the base case of recursion.

3. Divide Step

- The original problem P is divided into smaller subproblems:

$$P_1, P_2, \dots, P_k$$

- Each subproblem is a smaller instance of the same problem.

4. Conquer Step

- Each subproblem is solved recursively using DANDC.

5. Combine Step

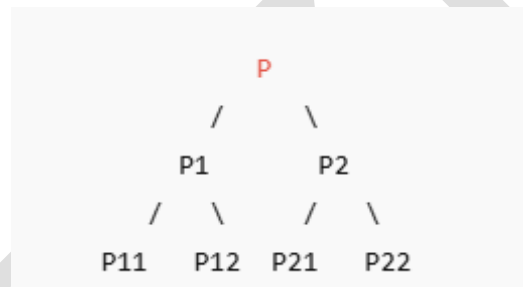
- The solutions of the subproblems are combined to obtain the final solution.

If the sizes of the two subproblems are approximately equal, then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n), & n \text{ small} \\ 2T(n/2) + f(n), & \text{otherwise} \end{cases}$$

Where:

- $T(n)$ is the time for DANDC on n inputs.
- $g(n)$ is the time to complete the answer directly for small inputs.
- $f(n)$ is the time for Divide and Combine.

Recursive Tree Representation:

- Each level represents further division.
- Leaves represent base cases.
- Results are combined upward.

Key Characteristics

- Recursive in nature.
- Requires a base condition.
- Efficient for large problems.
- Often leads to recurrence relations of the form:

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

Examples of Divide and Conquer Algorithms

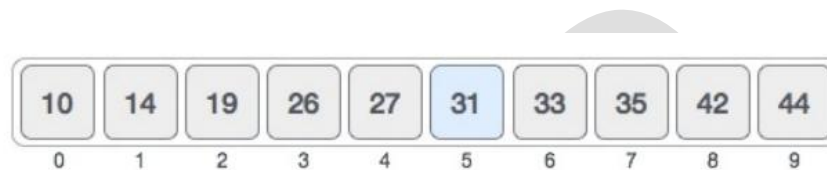
- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication

Binary Search:

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

How Binary Search Works?

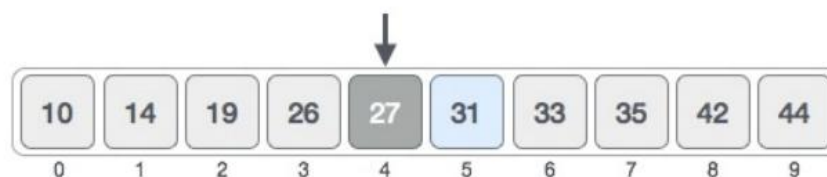
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula –

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Here it is, $0 + (9 - 0) / 2 = 4$ (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

$$\begin{aligned} \text{low} &= \text{mid} + 1 \\ \text{mid} &= \text{low} + (\text{high} - \text{low}) / 2 \end{aligned}$$

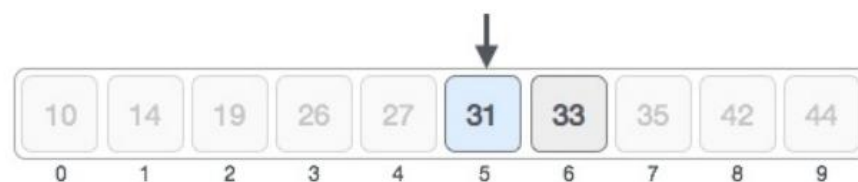
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5. Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Binary Search – Pseudo Code (Iterative)

Algorithm BinarySearch(A, n, key)

Input:

A → Sorted array of n elements

n → Number of elements

key → Element to be searched

Output:

Position of key (if found) or -1

BEGIN

low ← 0

high ← n - 1

WHILE low ≤ high DO

mid ← (low + high) DIV 2

IF A[mid] = key THEN

RETURN mid

ELSE IF A[mid] < key THEN

low ← mid + 1

ELSE

high ← mid - 1

END IF

END WHILE

RETURN -1

END

Binary Search – Pseudo Code (Recursive)

```

Algorithm RecursiveBinarySearch(A, low, high, key)
Input:
  A → Sorted array
  low → Starting index
  high → Ending index
  key → Element to search

BEGIN
  IF low > high THEN
    RETURN -1
  END IF

  mid ← (low + high) DIV 2

  IF A[mid] = key THEN
    RETURN mid
  ELSE IF key < A[mid] THEN
    RETURN RecursiveBinarySearch(A, low, mid - 1, key)
  ELSE
    RETURN RecursiveBinarySearch(A, mid + 1, high, key)
  END IF
END

```

Binary Search – Complexity Analysis**Time Complexity**

Binary Search reduces the search space by half in each step.

Recurrence relation:

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

Solving this gives:

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

Time Complexity:

Case	Time Complexity	Explanation
Best Case	O(1)	Key found at middle in first comparison
Average Case	O(log n)	Search space repeatedly divided by 2
Worst Case	O(log n)	Element found at last level or not found

Space Complexity:

Version	Space Complexity	Explanation
Iterative Binary Search	O(1)	Uses only a few variables (low, high, mid)
Recursive Binary Search	O(log n)	Due to recursion stack (log n recursive calls)

- Time Complexity (Overall): **O(log n)**
- Space Complexity (Iterative): **O(1)**
- Space Complexity (Recursive): **O(log n)**

Applications of Binary Search

Binary Search is used in various practical and algorithmic problems:

- **Searching in Sorted Arrays**
 - Finding an element in a sorted list.
 - Used in databases and large datasets.
- **Finding First and Last Occurrence**
 - Used when duplicates exist.
 - Example: Count number of occurrences of an element.
- **Finding Square Root**
 - Used to compute square roots using approximation methods.
- **Finding Peak Element**
 - Used in unimodal or mountain arrays.
- **Searching in Rotated Sorted Arrays**
 - Used in competitive programming and system optimization problems.
- **Lower Bound and Upper Bound Problems**
 - Used in STL (C++), searching range queries.
- **Optimization Problems**
 - Used in “Binary Search on Answer” problems.
 - Example: Minimum capacity, maximum possible value under constraints.
- **Finding Position to Insert Element**
 - Used in maintaining sorted order dynamically.

Advantages of Binary Search Algorithm

- **Efficiency in Time Complexity:** Binary search has a time complexity of $O(\log n)$, where n is the number of elements in the array. This makes it significantly faster than linear search ($O(n)$), especially for large datasets. By halving the search space in each iteration, binary search quickly narrows down the location of the target element.
- **Optimal for Sorted Data:** Binary search is specifically designed for sorted data. If the data is already sorted, binary search is one of the most efficient algorithms for finding an element.
- **Low Space Complexity:** Binary search has a space complexity of $O(1)$ when implemented iteratively. It does not require additional memory proportional to the input size, making it memory efficient.
- **Scalability:** Due to its logarithmic time complexity, binary search scales well with large datasets. The performance remains consistent even as the size of the dataset grows.
- **Deterministic Nature:** Binary search always follows a predictable pattern of dividing the search space, ensuring consistent performance across different runs.

Disadvantages of Binary Search Algorithm

- **Requires Sorted Data:** Binary search only works on sorted arrays or lists. If the data is unsorted, it must first be sorted, which adds an additional $O(n \log n)$ time complexity.
- **Inefficient for Small Datasets:** For small datasets, binary search may not be significantly faster than a linear search. The overhead of repeatedly dividing the search space can outweigh the benefits.
- **Not Suitable for Linked Lists:** Binary search relies on random access to elements, which is not efficient in linked lists. Linked lists require sequential access, making binary search impractical.
- **Complex Implementation:** While the concept of binary search is simple, implementing it correctly can be tricky. Issues like off-by-one errors or incorrect midpoint calculations can lead to bugs.
- **Static Data Requirement:** Binary search is not well-suited for dynamic datasets where elements are frequently inserted or deleted or modified. Maintaining the sorted order of the data can be costly.

Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made, and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n \log n)$ and $O(n^2)$, respectively.

Pseudo Code:

Algorithm PARTITION(A, low, high)

```
BEGIN
  pivot ← A[low]
  i ← low + 1
  j ← high

  WHILE TRUE DO

    WHILE i ≤ high AND A[i] ≤ pivot DO
      i ← i + 1
    END WHILE

    WHILE A[j] > pivot DO
      j ← j - 1
    END WHILE

    IF i < j THEN
      SWAP A[i] and A[j]
    ELSE
      BREAK
    END IF

  END WHILE

  SWAP A[low] and A[j]

  RETURN j
END
```

Main Algorithm

Algorithm QUICK_SORT(A, low, high)

```
BEGIN
  IF low < high THEN
    pivotIndex ← PARTITION(A, low, high)
    QUICK_SORT(A, low, pivotIndex - 1)
    QUICK_SORT(A, pivotIndex + 1, high)
  END IF
END
```

Working Logic

- Pivot is chosen as first element.
- Pointer i moves from left to right.
- Pointer j moves from right to left.
- If $A[i] > \text{pivot}$ and $A[j] < \text{pivot}$, swap them.
- When $i \geq j$, swap pivot with $A[j]$.
- Pivot reaches its correct position.

Quick Sort – Complexity Analysis

Quick Sort uses Divide and Conquer:

- Partition takes $O(n)$ time.
- Recursively sorts subarrays.

Time Complexity**Recurrence Relation**

- **Best / Average Case:**

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

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

- **Worst Case:**

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

$$T(n) = O(n^2)$$

Time Complexity:

Case	Time Complexity	Explanation
Best Case	$O(n \log n)$	Pivot divides array into two equal halves
Average Case	$O(n \log n)$	Partitions are reasonably balanced
Worst Case	$O(n^2)$	Pivot always smallest/largest (highly unbalanced)

Space Complexity

Quick Sort is an **in-place algorithm** (no extra array used).

Case	Space Complexity	Explanation
Best Case	$O(\log n)$	Balanced recursion depth
Average Case	$O(\log n)$	Stack depth $\approx \log n$
Worst Case	$O(n)$	Recursion depth becomes linear

Applications

- **General Sorting**
Used for sorting large datasets efficiently.
- **Database Systems**
Sorting records based on fields like ID, salary, or marks.
- **Competitive Programming**
Preferred due to average-case $O(n \log n)$ performance.
- **E-commerce Platforms**
Sorting products by price, rating, or popularity.
- **Selection Problems**
Used in Quick Select algorithm to find kth smallest/largest element.
- **Operating Systems & System Software**
Used in scheduling and internal data organization.

Advantages of Quick Sort

- **High Efficiency:** With an average time complexity of $O(n \log n)$, it is one of the fastest sorting algorithms available in practice.
- **In-Place Sorting:** It sorts within the original array, requiring minimal additional memory (typically $O(\log n)$ for recursion stack space), which makes it memory-efficient.
- **Cache Friendly:** Due to its localized and sequential memory access patterns, it utilizes CPU cache effectively, improving practical performance.
- **Versatility:** Works well for large datasets and is often used as the default sorting algorithm in many standard libraries (with optimizations).
- **Adaptable:** Using techniques like random pivot selection or median-of-three, the risk of worst-case behavior can be minimized.

Disadvantages of Quick Sort

- **Worst-Case Performance:** In rare scenarios (e.g., already sorted array with poor pivot selection), performance can degrade to $O(n^2)$.
- **Not Stable:** It does not preserve the relative order of equal elements.
- **Pivot Dependency:** Efficiency is highly dependent on the choice of the pivot element.
- **Recursion Depth:** Deep recursion in poorly partitioned arrays can increase stack usage and may lead to stack overflow for very large datasets.