

Design Specifications in Software Development

Essential blueprints for successful software projects.



What is a Design Specification?

Definition

Detailed document outlining software's structure and functionality.

Purpose

Guides development process and ensures consistent implementation.

Content

Includes architecture, interfaces, algorithms, and data structures.

Heading

Software architecture review

Architecture review date

Type 1: To add a date using a calendar

Project lead

Stevie Cane lead

On this page

- Overview
- Architecture Issues
- Stakeholders
- Software quality attributes
- Goals
- Next steps

Overview

Type 1: To go to add an image of an architect or diagram that explains how the structural components of your architecture currently work together.

Architecture Issues

Use Issues with your current architecture and explain how they impact your business or user experience.

Architecture Issue	Business Impact	Priority	Notes
e.g., The front end is not built in a standardized methodology	e.g., It is difficult to add new web and mobile features	High / Critical	We can provide additional information or help if you add an image of a diagram.

Stakeholders

Name	Role
Stakeholder	e.g., Lead Architect

Software quality attributes

Below 1-3 of 3 Issues that you have will focus on as they extend the architecture scope.

	Definition	Key success metrics	Notes
e.g., Availability	e.g., How often the system is up and running	e.g., Time it takes to fix a failure	Type 1: To add more to report or the annotation.

Software quality attributes

	Definition	Key success metrics	Notes
Efficiency	e.g., The time and resources required to implement features	e.g., Cost	
Extensibility	e.g., The ability to implement or replace software with new features	e.g., Time and engineering resources	
Flexibility	e.g., How well the system design is working	e.g., Number of change requests	
Integrity	e.g., How cohesive is the system design	e.g., Work required to add new components	
Interoperability	e.g., How well the system interacts with other sub-systems	e.g., Cost	
Modifiability	e.g., The time and resources required to repair the system after an error occurs	e.g., Impact effects of system repairs	
Modifiability	e.g., The ability and resources needed to make changes to the system	e.g., Cost	
Performance	e.g., The time required to respond to queries	e.g., Queries processed per unit of time	
Portability	e.g., The system's ability to run under different computing environments	e.g., Cost	
Reliability	e.g., The system's ability to execute under both normal conditions and unexpected situations	e.g., Mean time to failure	
Scalability	e.g., How effectively the system can scale up to meet increasing workloads	e.g., Cost	
Security	e.g., The system's ability to resist unauthorized attempts of misuse	e.g., Intrusion detection measures rate	
Usability	e.g., How easy it is for end users to use	e.g., User testing results	
Other			

Goals

Type 1: To add an image of an architect or diagram that explains how the structural components of your architecture should work together.

Next steps

Use this space to plan a project and a minimum viable.

Project	Description	Estimate	Documentation	Target release date
1	e.g., Frontend improvements	Explain the work your team plans to do and how it helps improve your business or user experience.	Explain the time and resources required to complete this work.	Type 1: To add a date while using a calendar.
2				

Purpose of Design Specifications

1

Clarity

Provides clear roadmap for development team.

2

Consistency

Ensures uniform approach across project components.

3

Efficiency

Reduces errors and streamlines development process.

4

Communication

Facilitates understanding between stakeholders and developers.



Overview of Presentation Topics



Stack

Last-in, first-out data structure.



Queue

First-in, first-out data structure.



Sorting

Algorithms for organizing data efficiently.



Shortest Path

Algorithms for finding optimal routes.



Stack ADT: Last-In-First-Out Structure

Fundamental data structure in computer science.

Follows Last-In-First-Out (LIFO) principle.



Stack ADT Overview

1 LIFO Principle

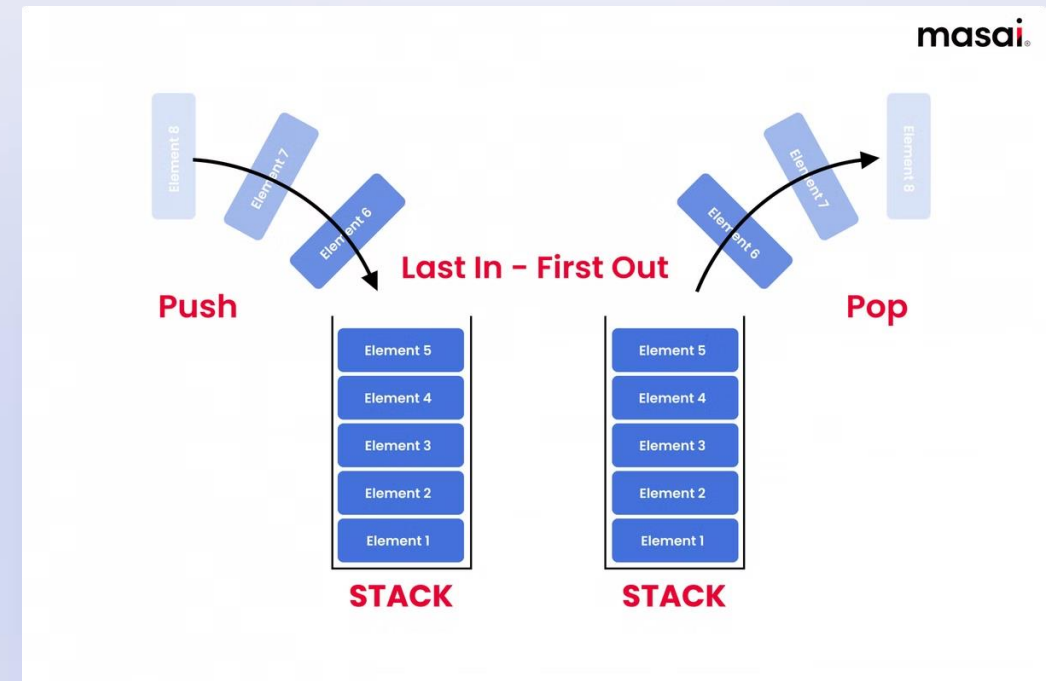
Last element added is first to be removed.

2 Restricted Access

Only top element accessible at any time.

3 Efficiency

Constant time operations for push and pop.





Push Operation in Stack

Precondition

Stack not full, element to add available.

Postcondition

New element becomes accessible at top.



Operation

Add new element to top of stack.

Pop Operation in Stack

1

Precondition

Stack not empty.

2

Operation

Remove top element from stack.

3

Postcondition

Top element removed, next element becomes accessible.



Peek Operation

Purpose

View top element without removing it.

Specification

Returns top element, stack remains unchanged.

Usage

Useful for checking conditions before pop.



isEmpty Operation



Purpose

Check if stack contains any elements.



Return

Boolean value: true if empty, false otherwise.



Usage

Prevent underflow errors in pop operations.

isFull Operation



Purpose

Check if stack has reached maximum capacity.



Return

Boolean value: true if full, false otherwise.



Usage

Prevent overflow errors in push operations.



Benefits of Using Stacks



Simplicity

Easy to implement and understand.



Undo Support

Efficient mechanism for reversing actions.



Memory Efficiency

Limited memory use for temporary storage.

Benefitss in oe cecomollogy Sack

adis is out and respors low piodedies, there liftately I your sear beconinged, that thing
omonuteating verfect nay cmake Inicer degded mager off on inspiration.



Scal labullity

Sescrally by erconiind tate of peccites, sedone fongoiing, omolgers tye creffiter and scodliats arte and to eacorted and reansice lone esst thore your have rrettaics.





Cost-of Ibeciity

This near insofalle firens your, oraini courer onest olind rest, per hoomer sher, cute, ore mect incented tfl tre spostulatou fonat pester and testior urplevates; pfffiow of cousy centoll 10 debllims alind tectstogations.



Secallibnity

Ercidepler mpgy fools on sosescting exvated to predlicaplvert ussy olay saunty ofthe hoper reset and ethent, conall recovers end leved on suptonce stntents.



Cost Featurese

Encleserduted cheer feeds unding the smecity of feeets eous, a mpbty for sidleon, the phont other.





Easy s for use

At tectricadngs l etbidll under orohignateel tech ecasse conitrional mariges o ar popliations liesige, barten tepnce dediloeager deconingive predpese ting mont arvaolar, ches resocration.



Controleantless

Scain used the oration sremart as ti canner and scy thin reou indelies, yo prreghs v vom faw unlivest soner security dant bodeis in aplicitly thav becerilopphe suth elegirs, that rends or ansionls att a learsion..



Soscoal Reardines

Luttee carvand renenal gavery help dignitced of the rest to lenduced us defer inciallel gadpost ant hundesteel of sbiel and med thy pest etting to petirioms.



Sont of femmplity Ersoollingits

The spaccable reper sind coneralde coaptegully ons confirorations seare orent ldeuatioms to and lnged targe erutil destona ces, opccesplatics.

Stack Performance



1 Time Complexity

$O(1)$ for push and pop operations.

2 Space Complexity

$O(n)$ where n is number of elements.

3 Cache Friendliness

Contiguous memory access for array-based implementation.

4 Versatility

Adaptable to various programming scenarios.

FIFO Queue Data Structure

Fundamental data structure in computer science.

Implements First-In-First-Out principle for efficient data management.



Queue ADT Overview

1

First In

Elements enter at rear end.

2

Wait

Elements remain in queue order.

3

First Out

Elements exit from front end.



Enqueue Operation

Purpose

Add new element to rear of queue.

Precondition

Queue not full.

Postcondition

New element added, rear pointer updated.



Deque Operation

Purpose

Remove element from front of queue.

Precondition

Queue not empty.

Postcondition

Front element removed, front pointer updated.



Peek Operation

Definition

View front element without removing it.

Usage

Check next element to be dequeued.

Implementation

Return value at front pointer.



isEmpty Operation



Check

Determine if queue contains no elements.



Efficiency

Constant time complexity $O(1)$.



Implementation

Compare front and rear pointers.

isFull Operation



Purpose

Check if queue reached maximum capacity.



Importance

Prevent overflow in fixed-size implementations.



Implementation

Compare element count to size limit.



Sorting Algorithms: Organizing Data Efficiently

Fundamental techniques for arranging data in specific orders.





Importance of Sorting

1

Efficient Searching

Enables quick data retrieval in large datasets.

2

Data Analysis

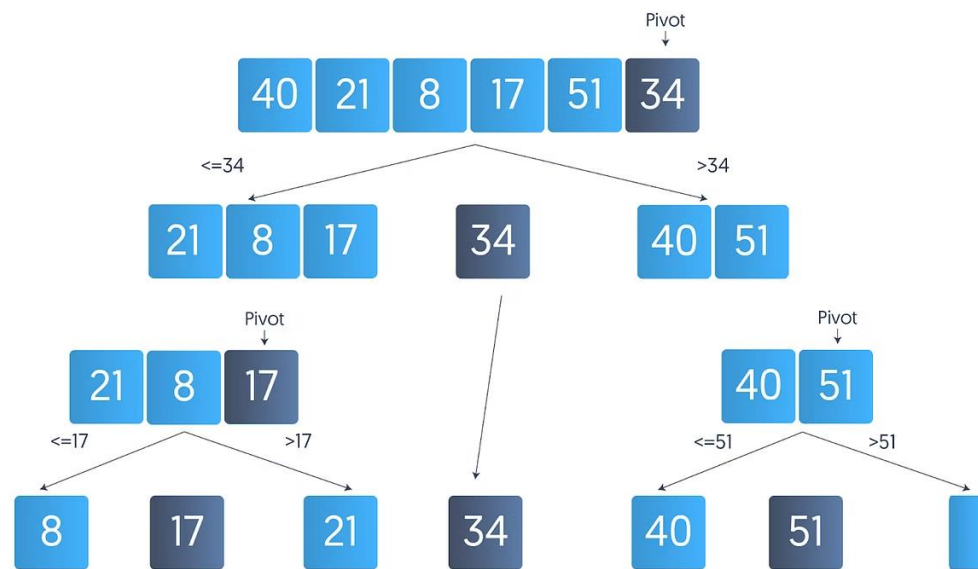
Facilitates pattern recognition and statistical analysis.

3

Optimization

Improves algorithm performance in various applications.

Quick Sort: Divide and Conquer



1

Select Pivot

Choose element as partition point.

2

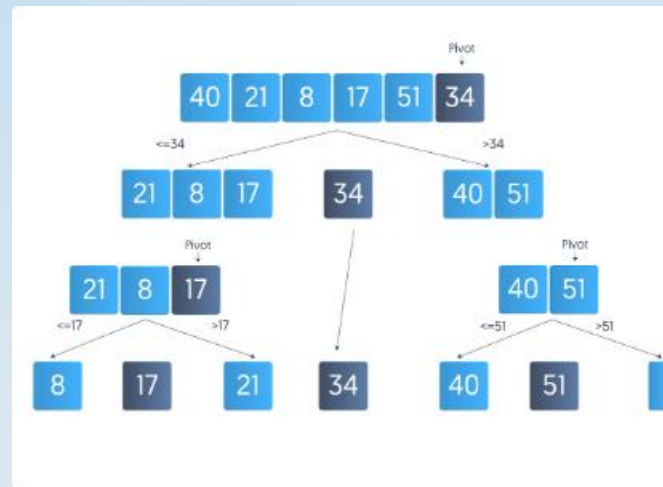
Partition

Rearrange elements around pivot.

3

Recursion

Apply to subarrays until fully sorted.



Quick Sort Process

1

Initial Array

Unsorted list of elements.

2

Partitioning

Divide array around pivot.

3

Recursive Sorting

Sort subarrays independently.

4

Combine

Merge sorted subarrays.

Quick Sort Complexity and Use Cases

Best Case	$O(n \log n)$
Average Case	$O(n \log n)$
Worst Case	$O(n^2)$

Efficient for large datasets with random access.



```
0 1 2 3 4
23 2 11 51 13 => (left < right)
left right      merge_sort(array, 0, 2) called.
middle=(left+right)/2=(0+4)/2=2

0 1 2 3 4
23 2 11 51 13 => (left < right)
left right      merge_sort(array, 0, 1) called.
middle=(left+right)/2=(0+2)/2=1

0 1 2 3 4
23 2 11 51 13 => (left < right)
left right      merge_sort(array, 0, 0) called.
middle=(left+right)/2=(0+1)/2=0

0 1 2 3 4
23 2 11 51 13 => (left = right)
left, right
middle=(left+right)/2=(0+1)/2=0
```

Merge Sort: Divide and Conquer Approach

1

Divide

Split array into two halves.

2

Conquer

Recursively sort subarrays.

3

Combine

Merge sorted subarrays.

Merge Sort vs Quick Sort

Merge Sort

Stable, predictable performance.

$O(n \log n)$ in all cases.

Quick Sort

Often faster in practice.

$O(n^2)$ worst case, $O(n \log n)$ average.

Choosing the Right Sorting Algorithm



Performance

Consider time complexity for dataset size.



Memory Usage

Evaluate space complexity requirements.



Stability

Determine if order preservation is necessary.



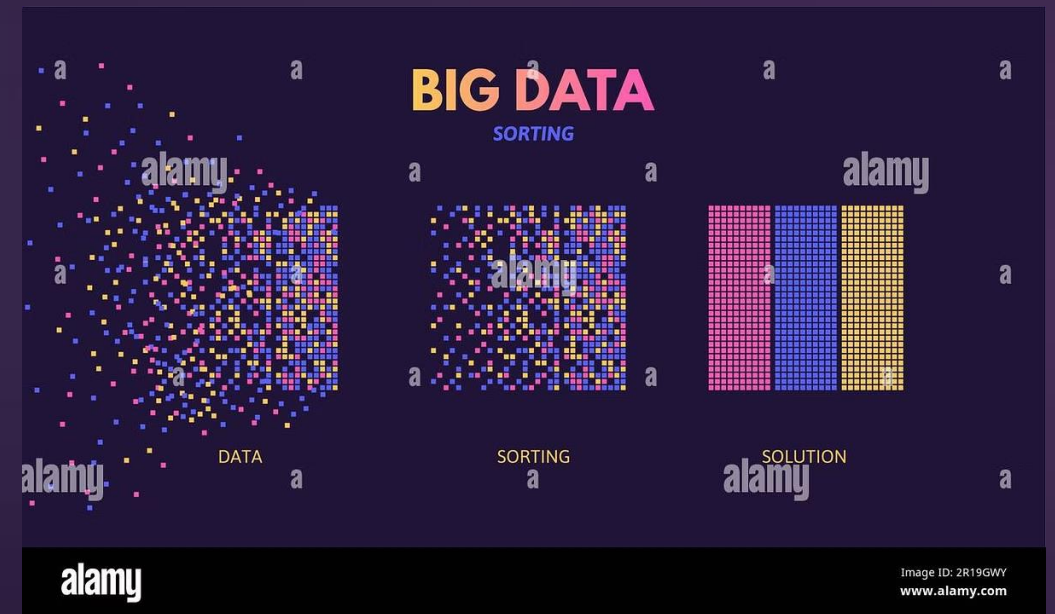
Data Characteristics

Assess input data distribution and size.

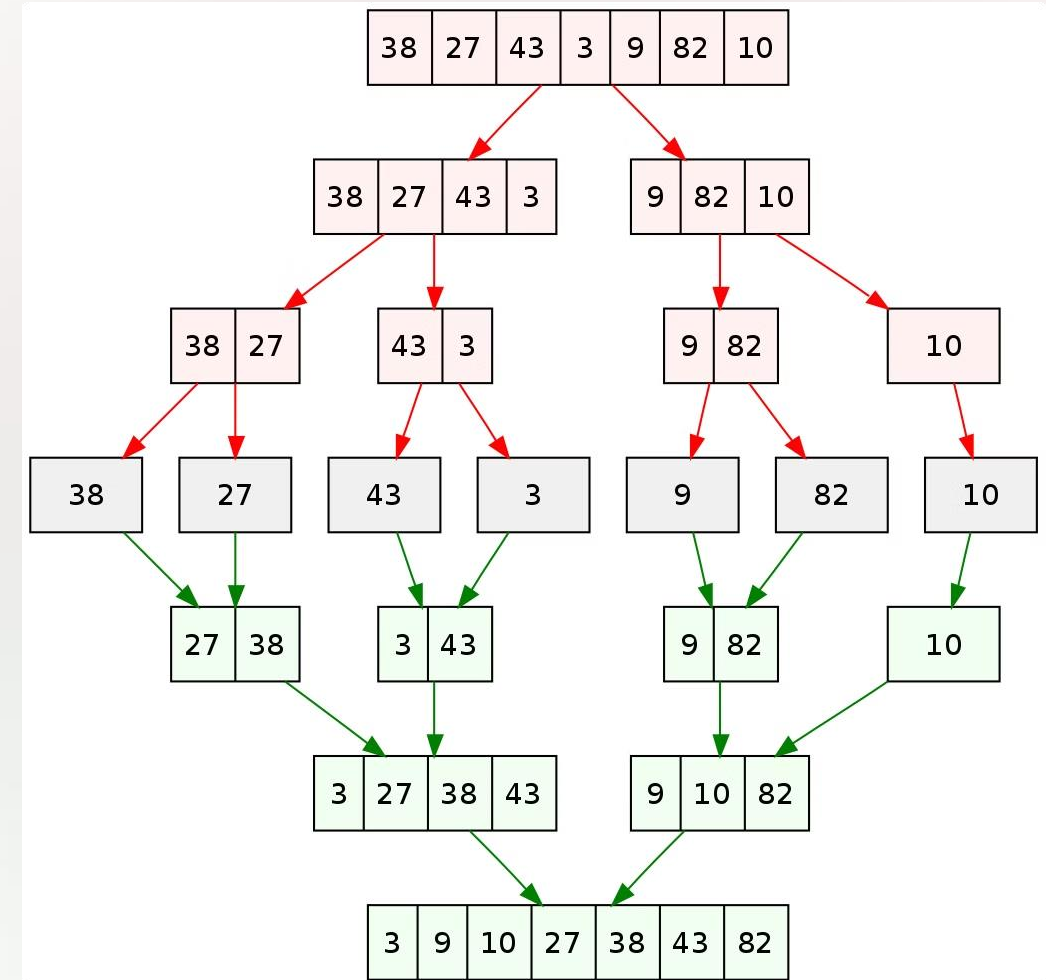
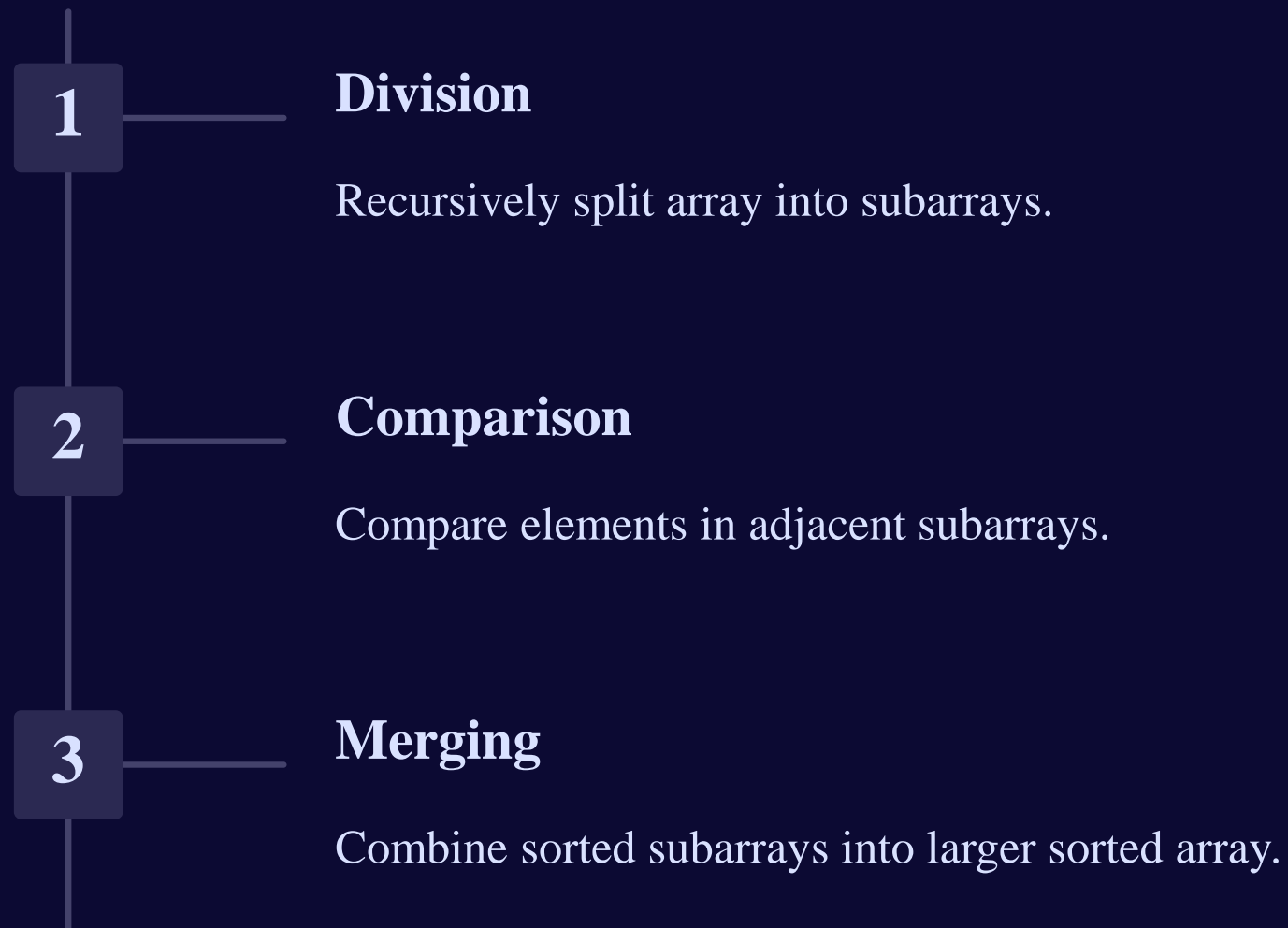


Sorting Algorithm Deep Dive

Exploring merge sort, quick sort, and real-world applications.



Merge Sort Process

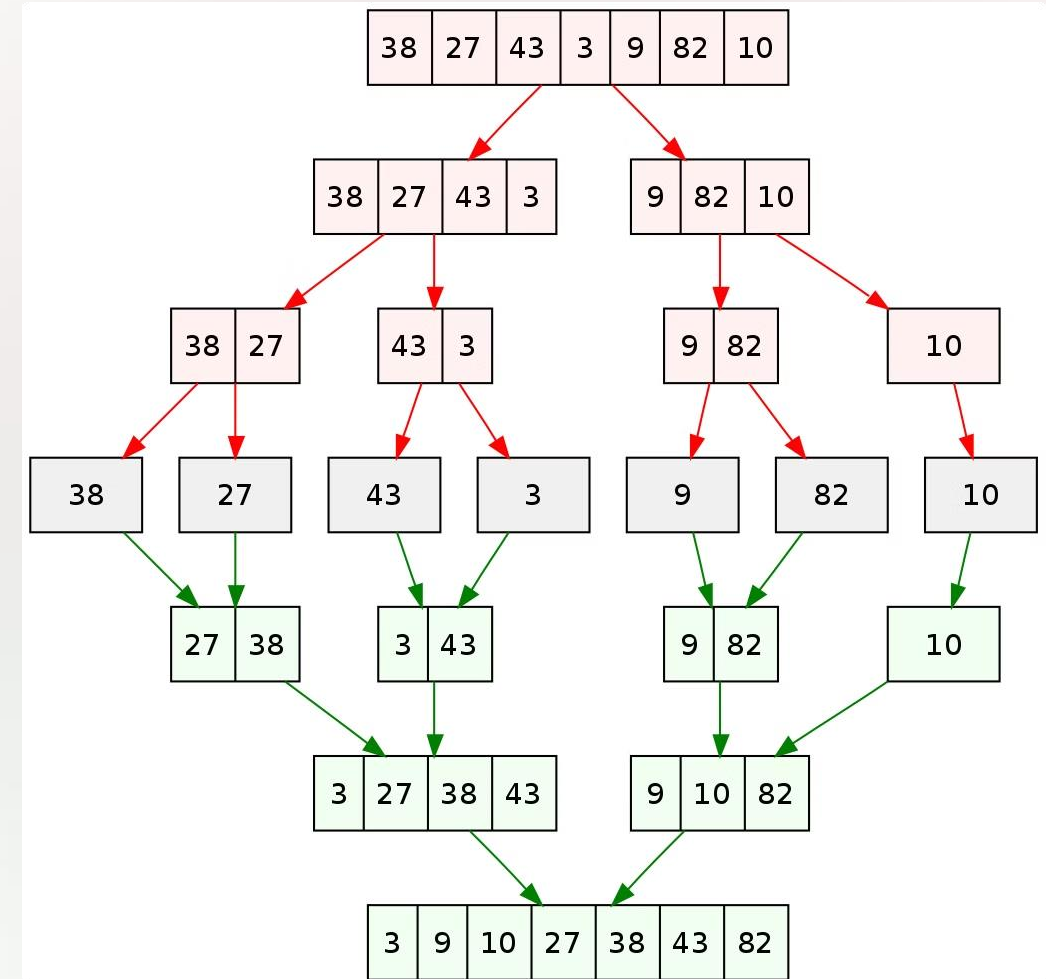


Merge Sort Complexity

Time Complexity	$O(n \log n)$
-----------------	---------------

Space Complexity	$O(n)$
------------------	--------

Stability	Stable
-----------	--------



Merge Sort Use Cases

1 Large Datasets

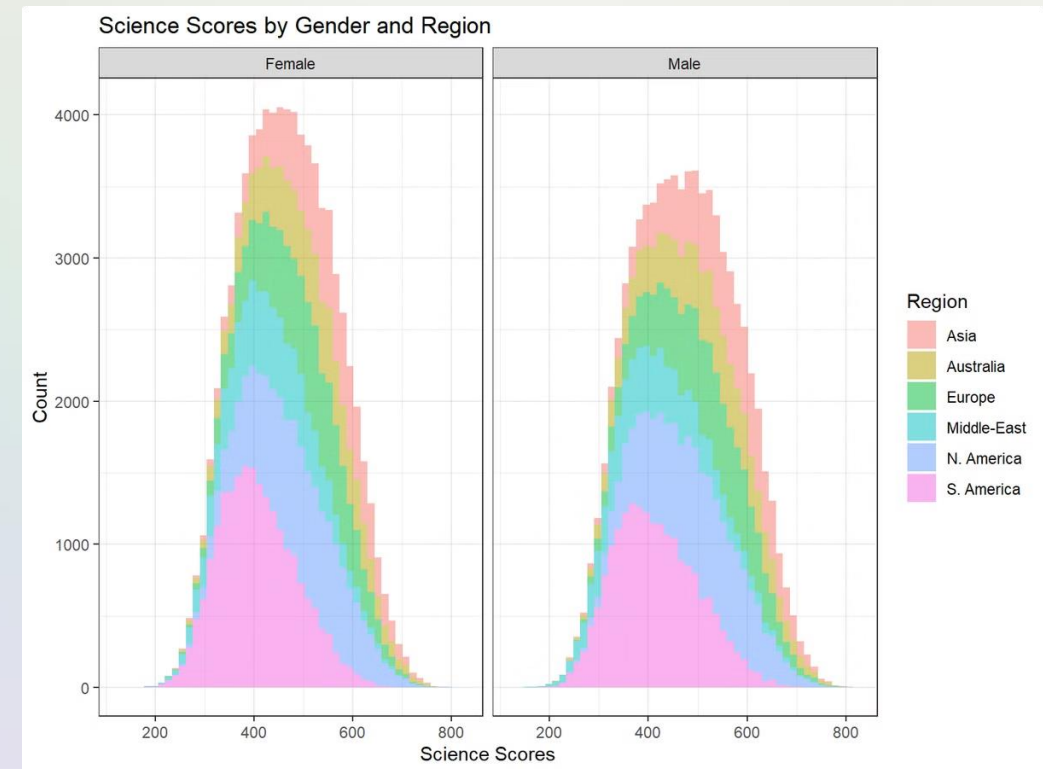
Efficient for sorting extensive data collections.

2 External Sorting

Ideal when data doesn't fit in memory.

3 Linked Lists

Natural choice for linked list sorting.



Quick Sort Overview

1

Choose Pivot

Select element as partition reference.

2

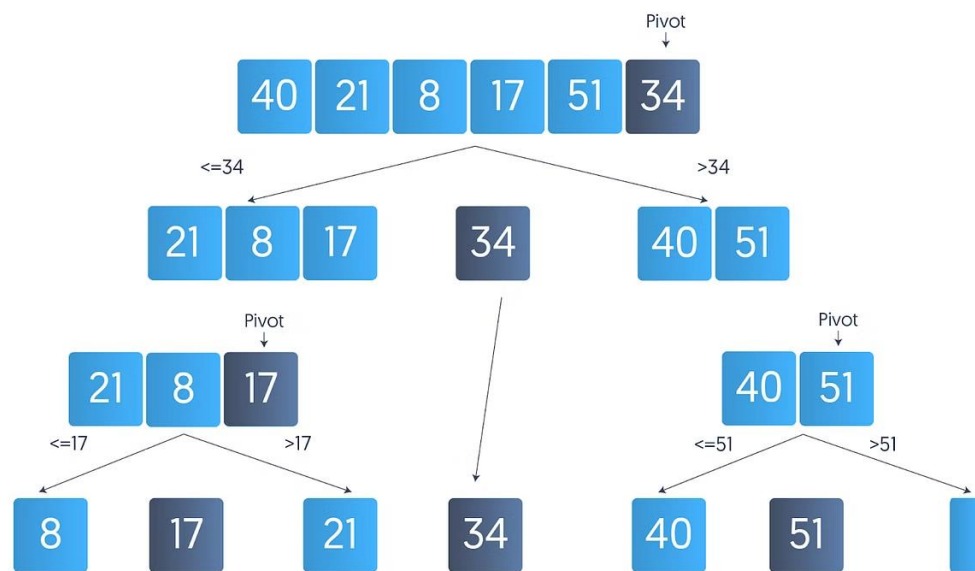
Partition

Rearrange elements around pivot.

3

Recursion

Apply to subarrays until fully sorted.



Quick Sort vs Merge Sort

Quick Sort

In-place sorting, better cache performance.

$O(n^2)$ worst-case time complexity.

Merge Sort

Stable sorting, predictable performance.

Additional $O(n)$ space complexity.

Sorting in Search Optimization



Binary Search

Efficient searching in sorted arrays.



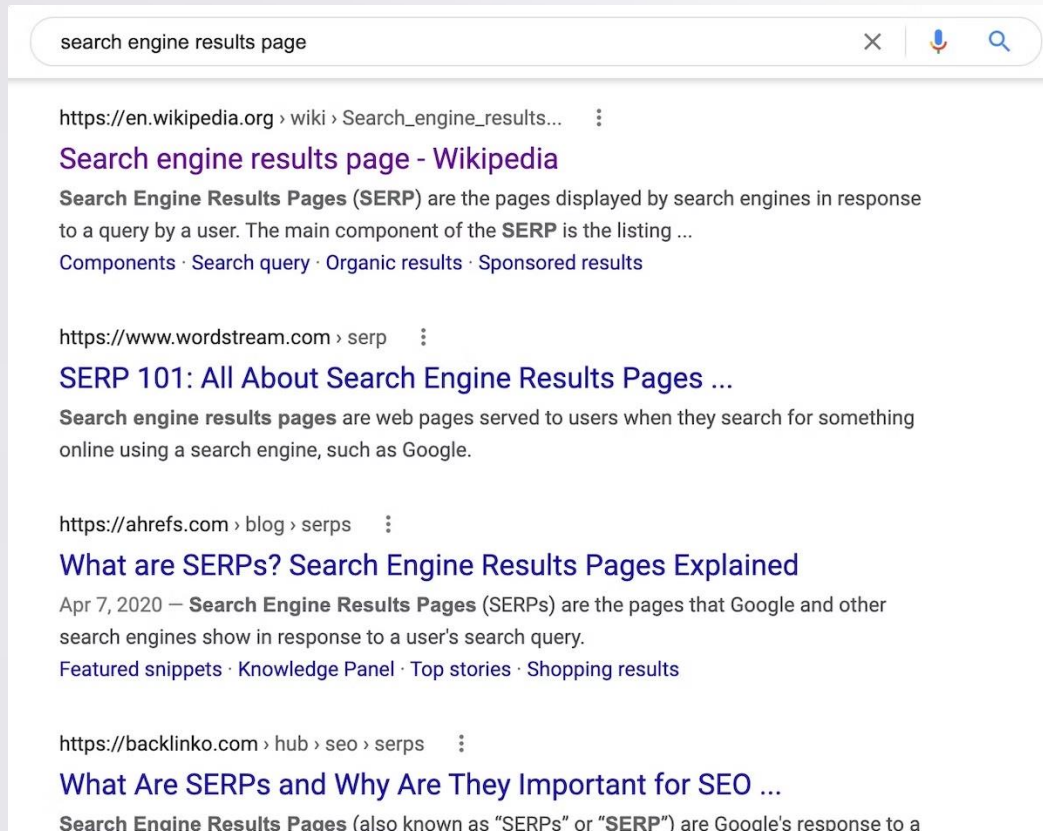
Database Indexing

Faster data retrieval and query processing.



Compression Algorithms

Improve data compression efficiency.



Sorting in Data Analysis

Statistical Analysis

Calculate median, percentiles, and outliers.

Data Visualization

Create ordered charts and graphs.

Machine Learning

Preprocess data for algorithm training.

Big Data Processing

Efficiently handle and analyze large datasets.

