Problem Solving Strategies and Computational Approaches SCS1304

Handout 4 : Brute Force & Divide-and-Conquer

Prof Prasad Wimalaratne PhD(Salford),SMIEEE

Overview

- Brute Force
  - Origin and Use of the Term "Brute Force“
  - Brute Force in Computing
  - Brute Force Algorithms
  - Time Complexities of Brute Force Algorithms
  - Advantages and Disadvantages
  - Optimizing Brute Force
  - Alternatives to Brute Force
  - Brute Force Applications
- Divide and Conquer
  - Merge Sort
  - Quick Sort
  - Strassen’s Matrix Multiplication

Origin and Use of the Term "Brute Force"

- Understanding where the term 'Brute Force' originates from can be helpful in grasping its connotations.
- In historical military jargon, the phrase referred to direct, unconcealed, and overwhelming attacks.
- The use in computer science applies the same concept—overwhelming a problem with sheer effort rather than clever stratagem.
- Imagine a castle with a locked gate, and you have lost the key.
    i. A strategic plan may involve climbing the walls, finding a secret entrance, or picking the lock.
    ii. Brute Force, however, would mean you attack the gate with enough force until it breaks down—no subtlety, no strategy, just sheer power.

https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/

Brute Force in Computing

- Brute force in computer science refers to a straightforward approach to problem-solving, directly addressing the problem's possible solutions without applying any strategic logic or established algorithms.
- This method may involve guessing all possible combinations until the correct one is found or systematically going through each option one by one.
- In programming, a Brute Force algorithm solves a problem by generating all possible solutions and testing each one until it finds an answer that works.
- It is not sophisticated or efficient, but it is guaranteed to deliver an answer if one exists.

https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/

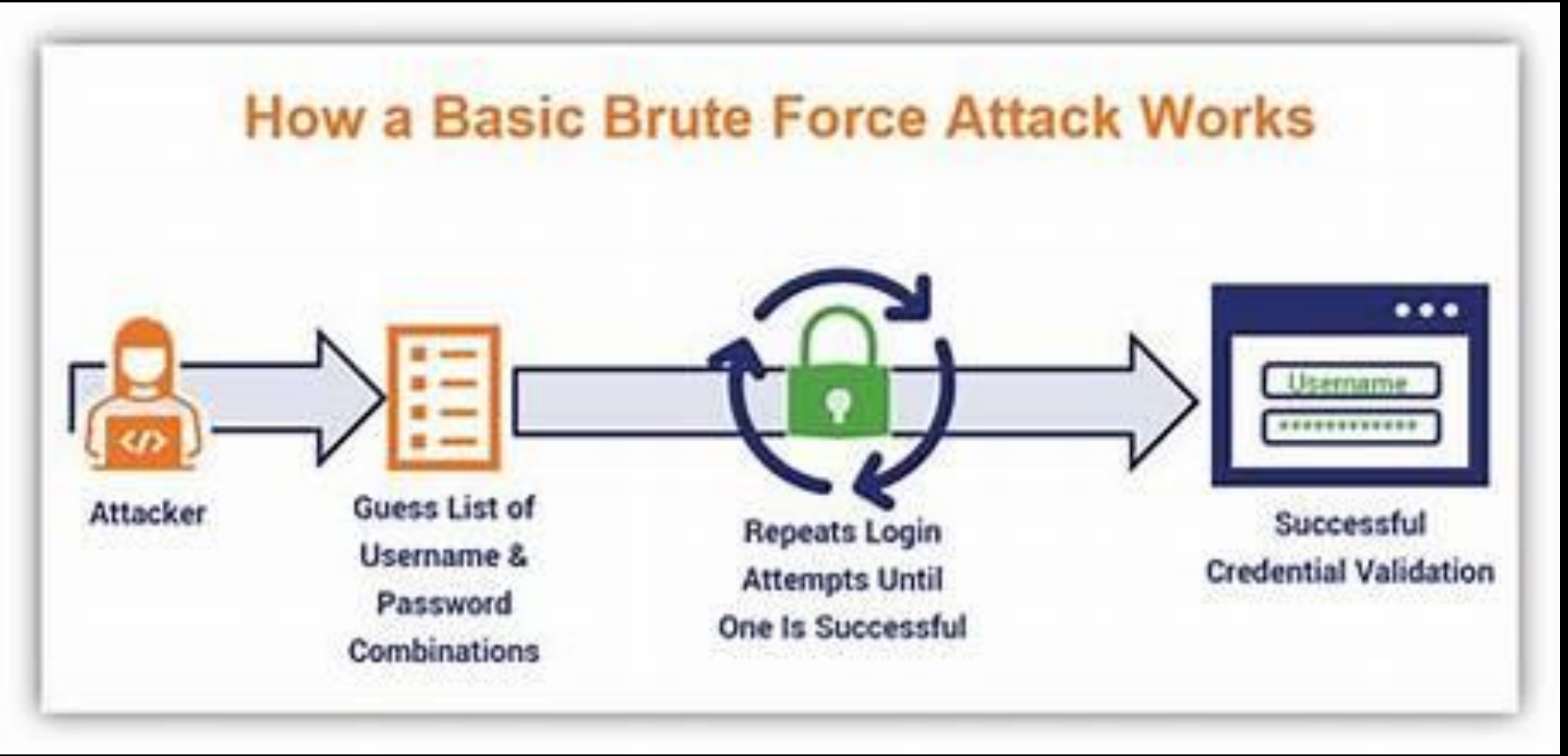
Brute Force Algorithms

- Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.
  - For example, imagine you have a small padlock with 4 digits, each from 0-9. You forgot your combination, but you don't want to buy another padlock. Since you can't remember any of the digits, you have to use a brute force method to open the lock.
  - Set all the numbers back to 0 and try them one by one: 0001, 0002, 0003, and so on until it opens. In the worst case scenario, it would take 10^4, or 10,000 attempts to find your combination.

https://www.freecodecamp.org/news/brute-force-algorithms-explained/

Brute force Attacks in Cyber Security?

example



https://tse2.mm.bing.net/th?id=OIP.eaYHm-c6tkeLapZPhvmrXQHaDk&pid=Api&P=0&h=220

TSP in Brute force approach

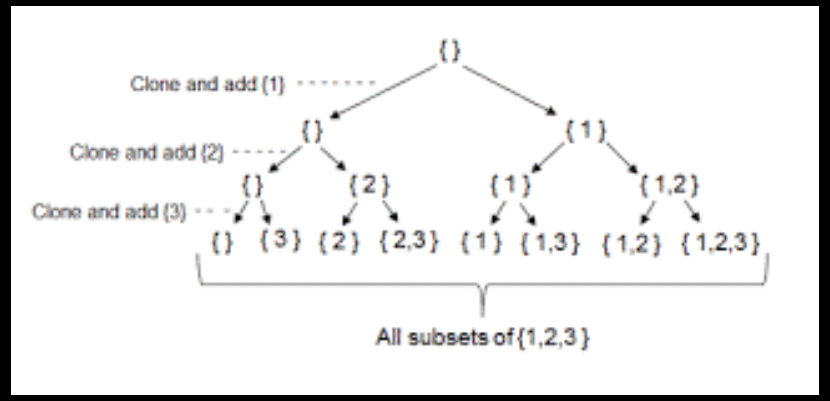
- To solve the Travelling Salesman Problem (TSP) using the brute-force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one
- This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.
    - The TSP is a classic algorithmic problem in the field of computer science and operations research, focusing on optimization. It seeks the shortest possible route that visits every point in a set of locations just once.
    - The TSP problem is highly applicable in the logistics sector, particularly in route planning and optimization for delivery services. TSP solving algorithms help to reduce travel costs and time.
    - Real-world applications often require adaptations because they involve additional constraints like time windows, vehicle capacity, and customer preferences.
        - Advances in technology and algorithms have led to more practical solutions for real-world routing problems. These include heuristic and metaheuristic approaches that provide good solutions quickly. (Heuristics are mental shortcuts that help people make quick decisions.)
        - Tools like Routific use sophisticated algorithms and artificial intelligence to solve TSP and other complex routing problems, transforming theoretical solutions into practical business applications.

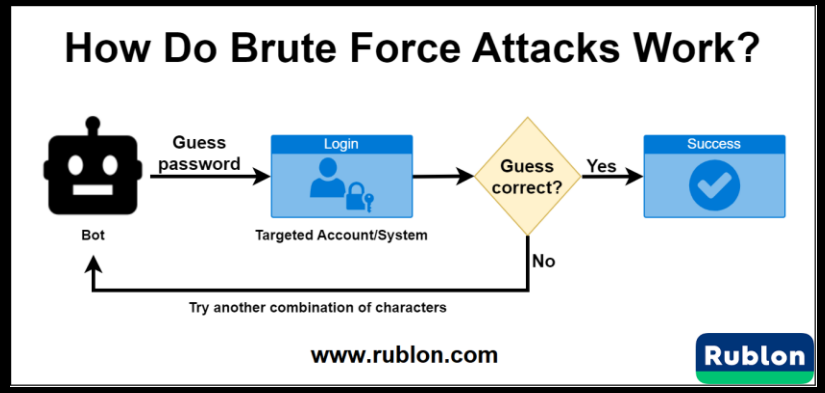
https://www.freecodecamp.org/news/brute-force-algorithms-explained/

https://www.routific.com/blog/travelling-salesman-problem

Time Complexities of Brute Force Algorithms

- Common time complexities for brute force algorithms include:
  - O(n!): Generating all permutations of n elements
  - O(2^n): Enumerating all subsets of an n-element set
  - O(n^m): Checking all possibilities for m variables, each taking n values
  - O(m^n): Trying all password combinations of length n with m possible characters





https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/

Brute Force for Traveling Salesman Problem?

- These exponential complexities make brute force impractical for large inputs.
  - For example, using brute force to solve the traveling salesman problem for n cities has a time complexity of O(n!). The following table shows how quickly this grows:

Number of Cities	Possible Routes	Approximate Time at 1 billion checks/sec
5	120	0.12 microseconds
10	3,628,800	3.63 seconds
15	1.3 × 10^12	36.5 years
20	2.4 × 10^18	77.1 billion years

https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/

Advantages and Disadvantages

- Brute force has several advantages:
  1. Simplicity and ease of implementation
  2. Guaranteed to find a solution if it exists
  3. Useful for solving small instances or one-off problems
  4. Provides a baseline for evaluating more optimized algorithms

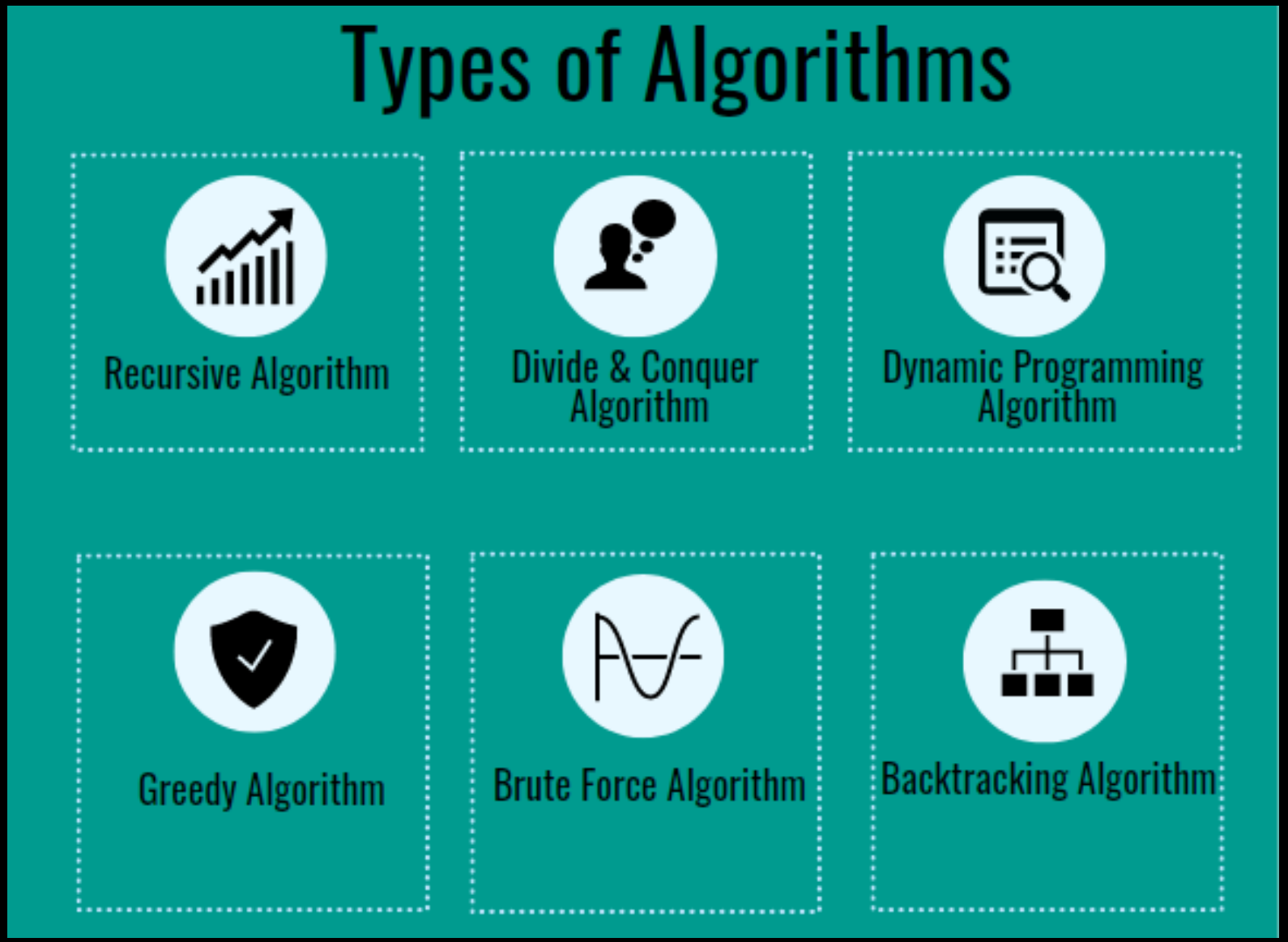
- However, it also has significant drawbacks:
  1. Poor scalability to large problem sizes
  2. Inefficient use of computational resources
  3. May be too slow for real-time or interactive applications
  4. Does not exploit problem-specific structure or heuristics

https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/

Optimizing Brute Force

- While brute force is inherently inefficient for large problems, there are still ways to optimize it when a complete search is necessary:
  1. Prune the search space by eliminating infeasible solutions early.
  2. Use heuristics to guide the search towards promising areas.
  3. Memonize or cache intermediate results to avoid redundant work.
  4. Parallelize independent subproblems to utilize multiple cores or machines.

https://www.bomberbot.com/algorithms/brute-force-algorithms-explained-a-comprehensive-guide/



- Algorithms can be categorized using different criteria. Above is an example

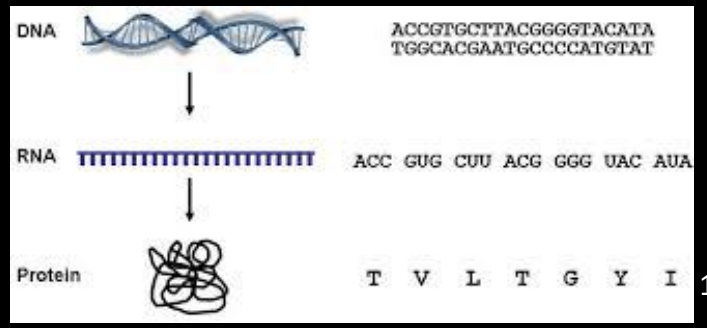
https://urwahmaqsood23.medium.com/algorithm-design-techniques-and-their-real-life-examples-bad97700e07c

Alternatives to Brute Force

- For most non-trivial problems, brute force is too slow. Alternative techniques that can provide better performance include:
  1. Divide and conquer: Recursively divide the problem into smaller subproblems, solve them independently, and combine the results. Merge sort and quick sort are classic examples. (covered later in this note)
  2. Greedy algorithms: Make the locally optimal choice at each stage, hoping to find a globally optimal solution. Dijkstra's shortest path algorithm and Huffman coding use greedy approaches.
  3. Dynamic programming: Break the problem into overlapping subproblems and store their solutions to avoid redundant work. This is effective for optimization problems like knapsack and longest common subsequence.
  4. Reduction: Transform the problem into an instance of another problem for which efficient algorithms are known. For example, the subset sum problem can be reduced to the knapsack problem.
  5. Approximation algorithms: Find a solution that is provably close to the optimum in polynomial time. This is useful for NP-hard problems like set cover and vertex cover.

Brute Force Technique Applications

- Despite its limitations, brute force has practical applications in various domains:
  i. Cryptography: Attacking weak encryption schemes by trying all possible keys.
  ii. Bioinformatics: Comparing DNA sequences to find mutations or aligning protein structures. https://en.wikipedia.org/wiki/Sequence_alignment
  iii. Puzzle solving: Exploring all possible moves in chess, sudoku, or Rubik‘s cube.
  iv. Generating test cases: Exhaustively testing all input combinations to find bugs in software.
  v. Coding competitions and job interviews: Brute force is often sufficient for small problem instances within the time limit.



https://www.strongdm.com/blog/brute-force-attack

2. Divide-and-Conquer
[ref - Chapter 2]

Divide and Conquer

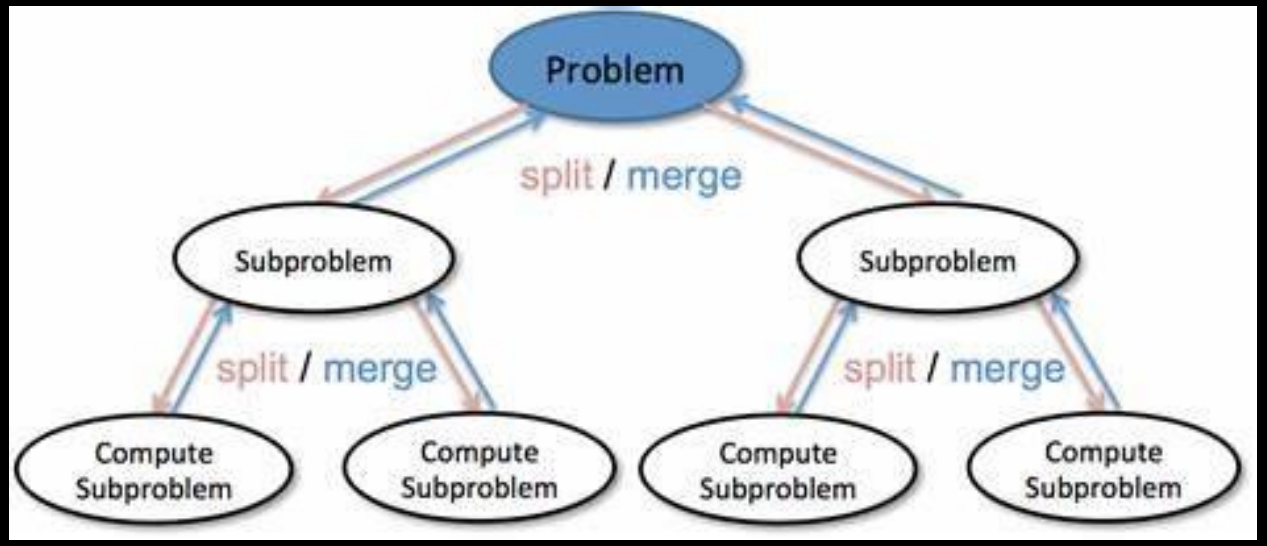
is a military tactic of bringing a large portion of one's own force to bear on small enemy units in sequence, rather than engaging the bulk of the enemy force all at once

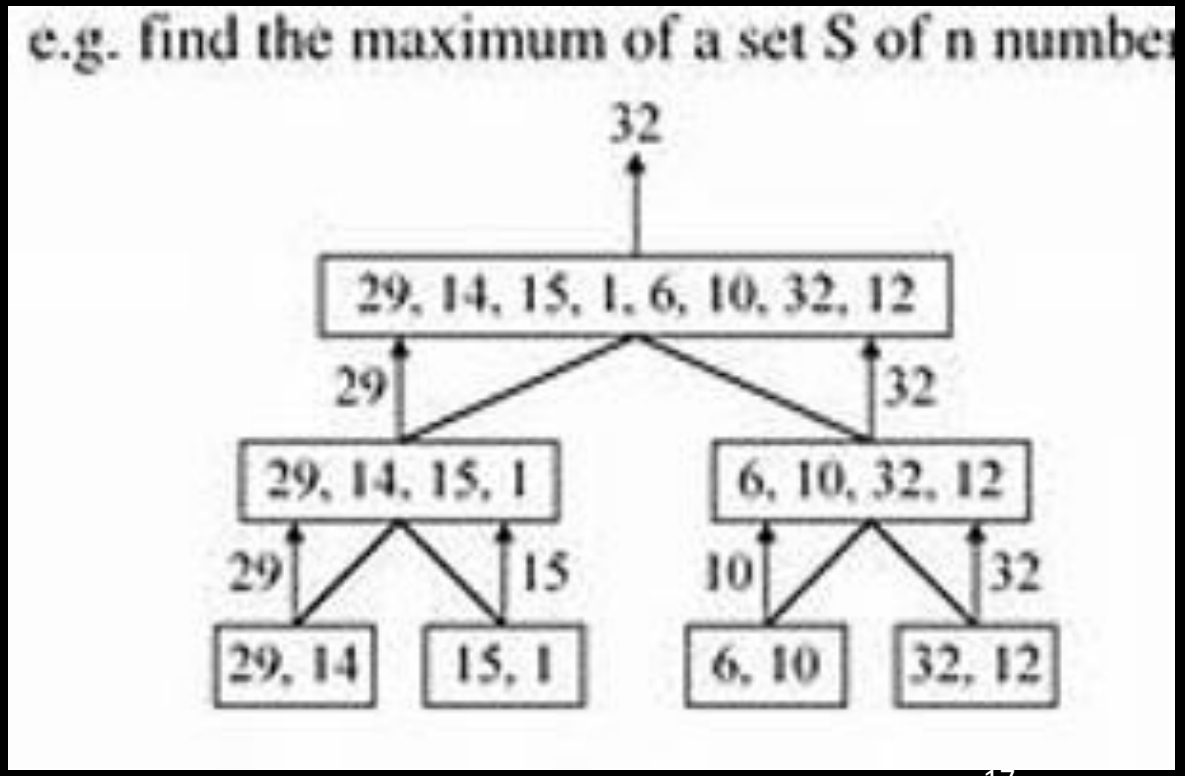
Napoleon knew this: One of his favorite tactics was to send the whole of his army against a fraction of the enemy army, and in this way, slowly wear down what was once a formidable force. Having divided the enemy into small, manageable pieces, Napoleon's conquest was assured

https://www.warhistoryonline.com/wp-content/uploads/sites/64/2015/12/Napoleon-Italy.jpg

Divide and Conquer Strategy

- When you divide the original problem into 2 smaller problem, solve them & combine the results, that's divide & conquer





Divide and Conquer Strategy

- The divide-and-conquer approach is a top-down approach.
- That is, the solution to a top-level instance of a problem is obtained by going down and obtaining solutions to smaller instances.

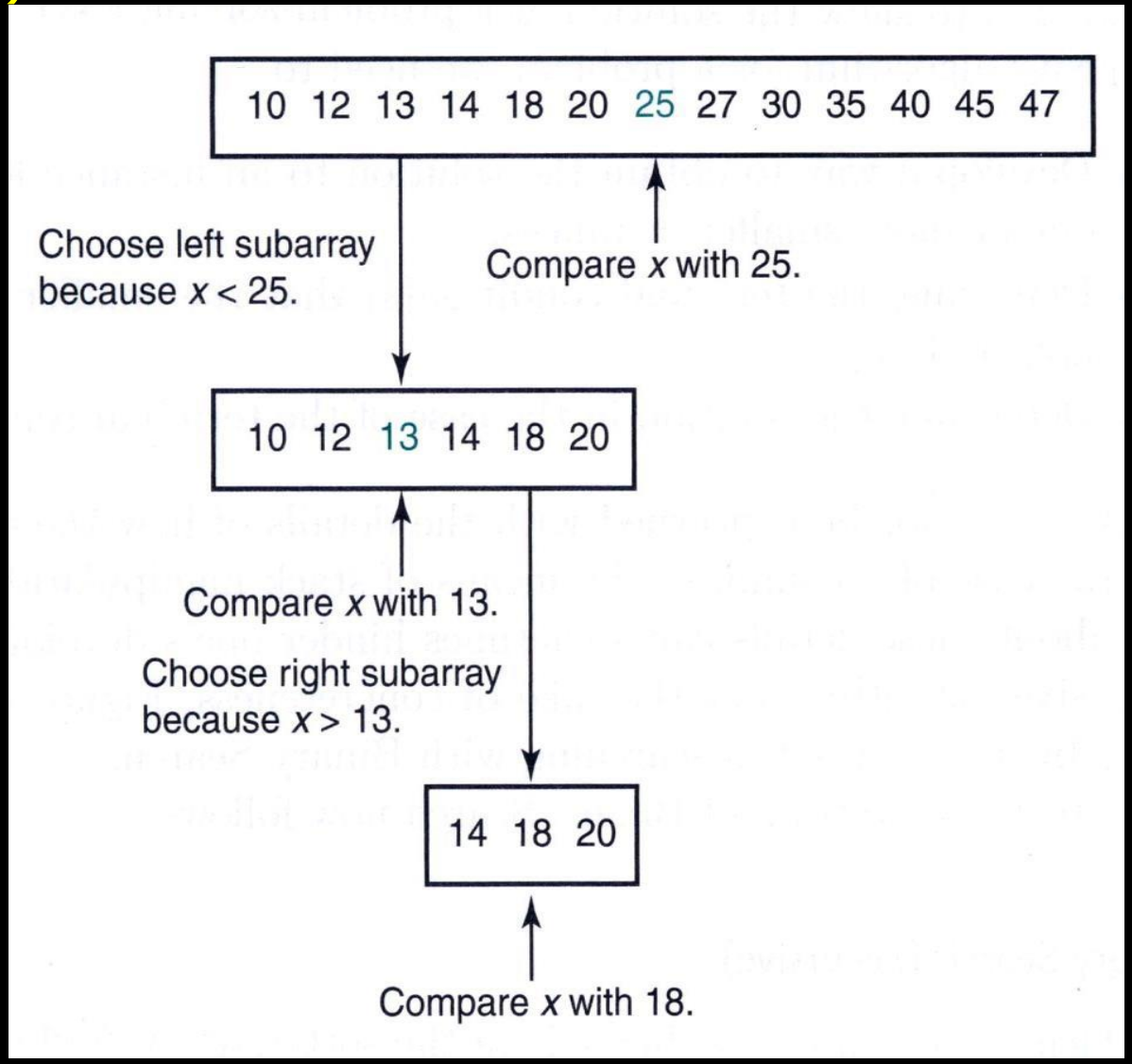
Divide-and-Conquer Approach

- Divide
  - It divides an instance of a problem into **two** or more smaller instances.
  - If the smaller instances are still too large to be solved readily, they can be divided into even smaller instances, until solutions are readily obtainable.
- Conquer
  - The smaller instances are usually instances of the original problem.
  - We may obtain solutions to the smaller instances readily.
- Combine
  - The process of dividing the instance can be obtained by combining these partial solutions.
- Top-down approach

Binary Search Algorithm Design

- Problem
  - Is the key x in the array S of n keys?
  - Determine whether x is in the sorted array S of n keys.
- Inputs (parameters)
  - Positive integer n, sorted (non-decreasing order) array of keys S indexed from 1 to n, a key x.
- Outputs
  - The location of x in S. (0 if x is not in S.)
- Design Strategy
  - *Divide* the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger, choose the right one.
  - *Conquer* (solve) the subarray by determining whether x is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.
  - *Obtain* the solution to the array from the solution to the subarray.

Figure 2.1 The steps done by a human when searching with Binary Search .(*note:* $x = 18$.)



Binary Search (Recursive algorithm)

```
index location (index low, index high) {
index mid;

    if (low > high)  // Not found.

        return 0;

    else {

        mid = (low + high) / 2      // Integer division.

     if (x == S[mid])      //  Found.
            return mid;
      else if (x < S[mid])
                return location(low, mid-1);      // Choose the left sub-array.
           else
                return location(mid+1, high);    // Choose the right sub-array.
    }
}
...
locationout = location(1, n);
...
```

Points of Observation

- Reason for using a local variable *locationout*
  - Input parameters, **n, S, x**, will not be changed during running the algorithm.
  - Dragging those unchanging variables for every recursive call would incur a source of unnecessary inefficiency.
- Other Points to Note
  - No operations are done after the recursive call.
  - It is straightforward to produce an iterative version. (see previous notes)
  - Recursion clearly illustrates the divide-and-conquer process.
  - However, running recursions is over-burdensome due to excessive uses of activation records.
  - A substantial amount of memory can be saved by eliminating the stack for activation records. (reason for preferring to iteration)
  - Iterative version is better only as constant factor. Order is same.

Time Complexity of Recursive Algorithms - Worst Case

- Formal Proof Methods (beyond basic counting):
- Induction:
  - For recursive algorithms, mathematical induction can be used to prove the time complexity. A base case is established, and then it is shown that if the complexity holds for a smaller input, it also holds for the current input.
- Recurrence Relations:
  - For recursive algorithms, a recurrence relation can be formulated that describes the time complexity in terms of smaller subproblems. Solving this recurrence relation (e.g., using the Master Theorem or substitution method) yields the Big O complexity.

Time Complexity of non-recursive algorithms

- Proving the worst-case time complexity of non-recursive algorithms primarily involves analyzing the number of basic operations performed in the execution path that maximizes resource consumption. This is typically achieved through the following methods:
- 1. Step-by-Step Analysis (Counting Operations):
  - Identify Basic Operations: Determine the fundamental operations that contribute most significantly to the algorithm's execution time (e.g., comparisons, assignments, arithmetic operations).
  - Determine Input Size Parameter (n): Define a parameter, n, that represents the size of the input and influences the number of operations.
  - Analyze Loop Iterations: For loops (e.g., for, while), determine the maximum number of times the loop body will execute in the worst-case scenario, often expressed in terms of n.
  - Sum Operations: Sum the number of basic operations performed by each part of the algorithm, considering the worst-case path.
  - Express as a Function of n: Represent the total number of operations as a function of n, for example, T(n)
  - Apply Asymptotic Notation: Use Big O notation to express the upper bound of T(n), discarding lower-order terms and constant factors.
- And.. Other methods exists

Worst-Case Time Complexity

As discussed in Section 1.2, one way the worst case can occur is when x is larger than all array items. If n is a power of 2 and x is larger than all the array items, each recursive call reduces the instance to one exactly half as big. For example, if $n = 16$, then $mid = \lfloor (1 + 16)/2 \rfloor = 8$. Because x is larger than all the array items, the top eight items are the input to the first recursive call. Similarly, the top four items are the input to the second recursive call, and so on. We have the following recurrence:

$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{Comparisons in recursive call}} + \underbrace{1}_{\text{Comparison at top level}}$$

Worst-Case Time Complexity ctd..

In an algorithm that searches an array, the most costly operation is usually the comparison of the search item with an array item. Hence

- Basic operation: the comparison of *x* with *S[mid]*
- Input size: *n*, the number of items in the array.
- Case 1: When n is a power of 2.

$$W(n) = W\left(\frac{n}{2}\right) + 1$$

$$W(1) = 1$$

$$W(1) = 1$$

$$W(2) = W(1) + 1 = 2$$

$$W(4) = W(2) + 1 = 3$$

$$W(8) = W(4) + 1 = 4$$

$$W(16) = W(8) + 1 = 5$$

$$W(2^k) = k + 1$$

$$W(n) = \lg n + 1$$

See page 713 *appendix for proof=>.

Note: There are two comparisons of x with S [mid] in any call to function location in which x does not equal S[mid].

Worst-Case Time Complexity ctd..

If $n = 1$ and x is larger than the single array item, there is a comparison of x with that item followed by a recursive call with *low* > *high*. At this point the terminal condition is true, which means that there are no more comparisons. Therefore, $W(1)$ is 1. We have established the recurrence

$$W(n) = W\left(\frac{n}{2}\right) + 1 \qquad \text{for } n > 1,\ n \text{ a power of } 2$$
$$W(1) = 1$$

This recurrence is solved in Example B.1 in Appendix B. The solution is

$$W(n) = \lg n + 1.$$

Mathematical Induction ctd..

- Some sums equal closed-form expressions. For example

$$1+2+\cdots+n=\frac{n(n+1)}{2}.$$

We can illustrate this equality by checking it for a few values of n, as follows:

$$1+2+3+4=10=\frac{4(4+1)}{2}$$

$$1+2+3+4+5=15=\frac{5(5+1)}{2}.$$

- However, because there are an infinite number of positive integers, we can never become certain that the equality holds for all positive integers n simply by checking individual cases.
- Checking individual cases can inform us only that the equality **appears** to be true.
- A powerful tool for obtaining a result for all positive integers n is mathematical induction.

Mathematical Induction ctd..

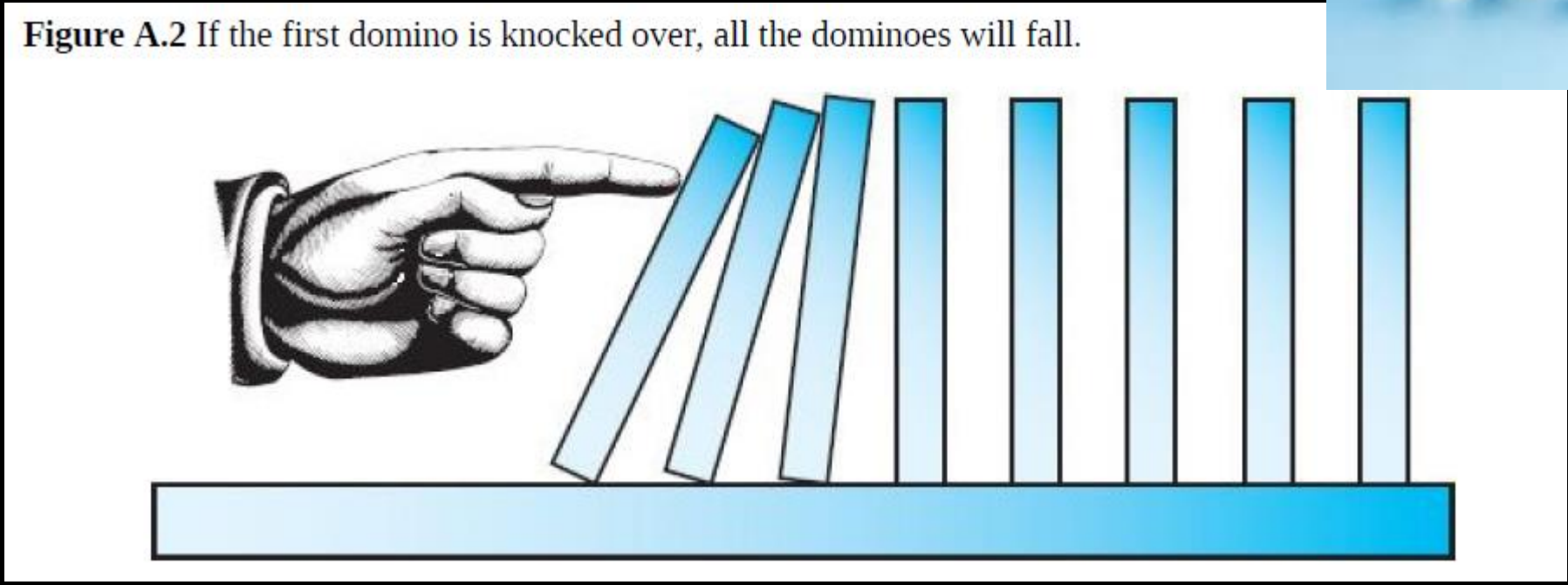


Figure A.2 If the first domino is knocked over, all the dominoes will fall.

- If we knock over the first domino, it will knock over the second, the second will knock over the third, and so on.
- In theory, we can have an arbitrarily large number of dominoes, and they all will fall.

Mathematical Induction ctd..

- Mathematical induction works in the same way as the domino principle.
- Figure A.2 illustrates that, if the distance between two dominoes is always less than the height of the dominoes, we can knock down all the dominoes merely by knocking over the first domino. We are able to do this because:
    1. We knock over the first domino.
    2. By spacing the dominoes so that the distance between any two of them is always less than their height, we guarantee that if the n^{th} domino falls, the $(n + 1)^{st}$ domino will fall.

Mathematical Induction ctd..

- An induction proof works in the same way. We first show that what we are trying to prove is true for n = 1.
- Next we show that if it is true for an arbitrary positive integer n, it must also be true for n + 1.
- Once we have shown this, we know that because it is true for n = 1, it must be true for n = 2; because it is true for n = 2, it must be true for n = 3; and so on, ad infinitum (to infinity).
- We can therefore conclude that it is true for all positive integers n. When using induction to prove that some statement concerning the positive integers is true, we use the following terminology:

Mathematical Induction ctd..

The *induction base* is the proof that the statement is true for $n = 1$ (or some other initial value).

The *induction hypothesis* is the assumption that the statement is true for an arbitrary $n \geq 1$ (or some other initial value).

The *induction step* is the proof that if the statement is true for n, it must also be true for $n + 1$.

- Analogy? The induction base amounts to knocking over the first domino, whereas the induction step shows that if the n^{th} domino falls, the $(n + 1)^{st}$ domino will fall

Mathematical Induction ctd..

Example A.1

We show, for all positive integers n, that

$$1 + 2 + \cdots + n = \frac{n\,(n+1)}{2}.$$

Induction base: For $n = 1$,

$$1 = \frac{1\,(1+1)}{2}.$$

Induction hypothesis: Assume, for an arbitrary positive integer n, that

$$1 + 2 + \cdots + n = \frac{n\,(n+1)}{2}.$$

Induction step: We need to show that

$$1 + 2 + \cdots + (n+1) = \frac{(n+1)\,[(n+1)+1]}{2}.$$

Mathematical Induction ctd..

To that end,

$$
\begin{aligned}
1+2+\cdots+(n+1) &= \mathbf{1+2+\cdots+n}+n+1 \\
&= \frac{\boldsymbol{n\,(n+1)}}{\mathbf{2}}+n+1 \\
&= \frac{n\,(n+1)+2\,(n+1)}{2} \\
&= \frac{(n+1)\,(n+2)}{2} \\
&= \frac{(n+1)\,[(n+1)+1]}{2}.
\end{aligned}
$$

Mathematical Induction ctd..

In the induction step, we highlight the terms that are equal by the induction hypothesis. We often do this to show where the induction hypothesis is being applied. Notice what is accomplished in the induction step. By assuming the induction hypothesis that

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

and doing some algebraic manipulations, we arrive at the conclusion that

$$1 + 2 + \cdots + (n+1) = \frac{(n+1)[(n+1)+1]}{2}.$$

Therefore, if the hypothesis is true for n, it must be true for $n + 1$. Because in the induction base we show that it is true for $n = 1$, we can conclude, using the domino principle, that it is true for all positive integers n.

Mathematical Induction ctd..

- We can often derive a possibly true statement by investigating some cases and making an educated guess.
- This is how the equality in Example A.1 was originally conceived. Induction can then be used to verify that the statement is true. If it is not true, of course, the induction proof fails.
- It is important to realize that it is never possible to derive a true statement using induction.
- Induction comes into play only after we have already derived a possibly true statement.

Analysis of Algorithm 2.1

Worst-Case Time Complexity (Binary Search, Recursive)

In an algorithm that searches an array, the most costly operation is usually the comparison of the search item with an array item. Thus, we have the following:

Basic operation: the comparison of x with S [mid].

Input size: n, the number of items in the array.

We first analyze the case in which n is a power of 2. There are two comparisons of x with S [mid] in any call to function *location* in which x does not equal S [mid]. However, as discussed in our informal analysis of Binary Search in Section 1.2, we can assume that there is only one comparison, because this

```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid - 1);
    else
      return location(mid + 1, high);
  }
}
```

would be the case in an efficient assembler language implementation. Recall from Section 1.3 that we ordinarily assume that the basic operation is implemented as efficiently as possible.

As discussed in Section 1.2, one way the worst case can occur is when x is larger than all array items. If n is a power of 2 and x is larger than all the array items, each recursive call reduces the instance to one exactly half as big. For example, if $n = 16$, then $mid = \lfloor (1 + 16)/2 \rfloor = 8$. Because x is larger than all the array items, the top eight items are the input to the first recursive call. Similarly, the top four items are the input to the second recursive call, and so on. We have the following recurrence:

$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\substack{\text{Comparisons in} \\ \text{recursive call}}} + \underbrace{1}_{\substack{\text{Comparison at} \\ \text{top level}}}$$

If $n = 1$ and x is larger than the single array item, there is a comparison of x with that item followed by a recursive call with *low* > *high*. At this point the terminal condition is true, which means that there are no more comparisons. Therefore, $W(1)$ is 1. We have established the recurrence

$$W(n) = W\left(\frac{n}{2}\right) + 1 \qquad \text{for } n > 1,\ n \text{ a power of } 2$$
$$W(1) = 1$$

This recurrence is solved in Example B.1 in Appendix B. The solution is

$$W(n) = \lg n + 1.$$

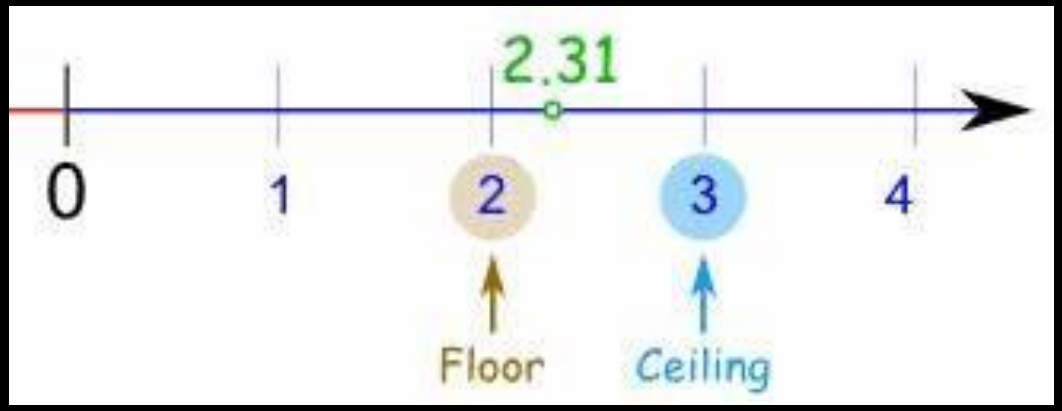
If n is not restricted to being a power of 2, then

$$W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n),$$

Floor

where $\lfloor y \rfloor$ means the greatest integer less than or equal to y. We show how to establish this result in the exercises.

Floor and Ceiling of numbers



$$\lfloor 3.3 \rfloor = 3 \qquad \left\lfloor \frac{9}{2} \right\rfloor = 4 \qquad \lfloor 6 \rfloor = 6$$

$$\lfloor -3.3 \rfloor = -4 \qquad \lfloor -3.7 \rfloor = -4 \qquad \lfloor -6 \rfloor = -6.$$

$$\lceil 3.3 \rceil = 4 \qquad \left\lceil \frac{9}{2} \right\rceil = 5 \qquad \lceil 6 \rceil = 6$$

$$\lceil -3.3 \rceil = -3 \qquad \lceil -3.7 \rceil = -3 \qquad \lceil -6 \rceil = -6.$$

Merge Sort

- A process related to sorting is merging. Uses divide and conquer and usually uses recursion.
- By two-way merging we mean combining two sorted arrays into one sorted array.
- By repeatedly applying the merging procedure, we can sort an array.
  i. For example, to sort an array of 16 items, we can divide it into two subarrays, each of size 8, sort the two subarrays, and then merge them to produce the sorted array.
  ii. In the same way, each subarray of size 8 can be divided into two subarrays of size 4, and these subarrays can be sorted and merged.
  iii. Eventually, the size of the subarrays will become 1, and an array of size 1 is trivially sorted.
- This procedure is called "Mergesort."

Refer: **Merge sort in 3 minutes**
https://www.youtube.com/watch?v=4VqmGXwpLqc

How does Merge Sort work?

- Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach to sort a given array of elements.
- Here is a step-by-step explanation of how merge sort works:
  1. Divide: Divide the list or array recursively into two halves until it can no more be divided.
  2. Conquer: Each subarray is sorted individually using the merge sort algorithm.
  3. Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

https://www.geeksforgeeks.org/merge-sort/

- Given an array with n items (for simplicity, let n be a power of 2), Mergesort involves the following steps:

1. *Divide* the array into two subarrays each with $n/2$ items.
2. *Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
3. *Combine* the solutions to the subarrays by merging them into a single sorted array.

Suppose the array contains these numbers in sequence:

27 10 12 20 25 13 15 22.

1. Divide the array:

27 10 12 20 and 25 13 15 22.

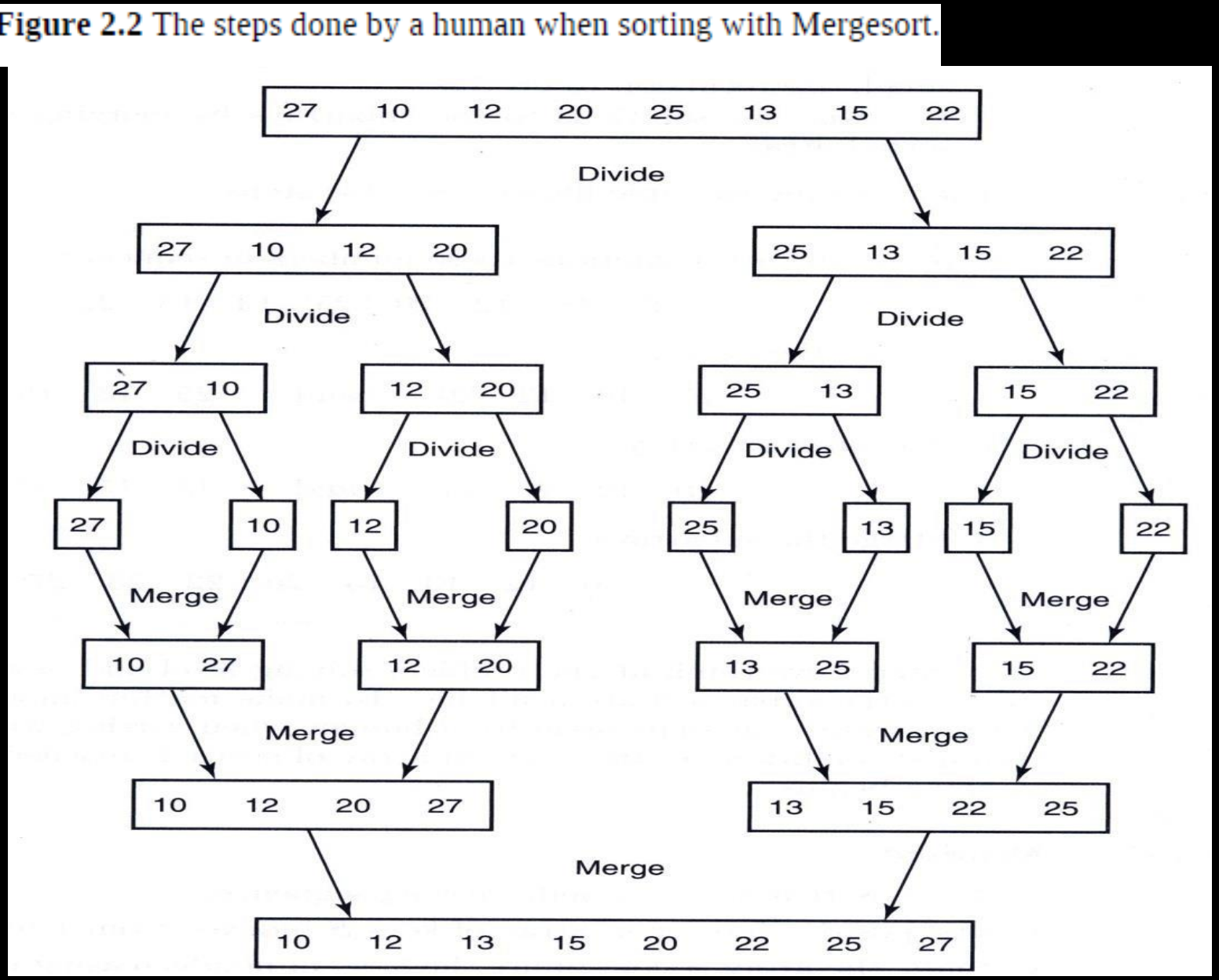
2. Sort each subarray:

10 12 20 27 and 13 15 22 25.

3. Merge the subarrays:

10 12 13 15 20 22 25 27.

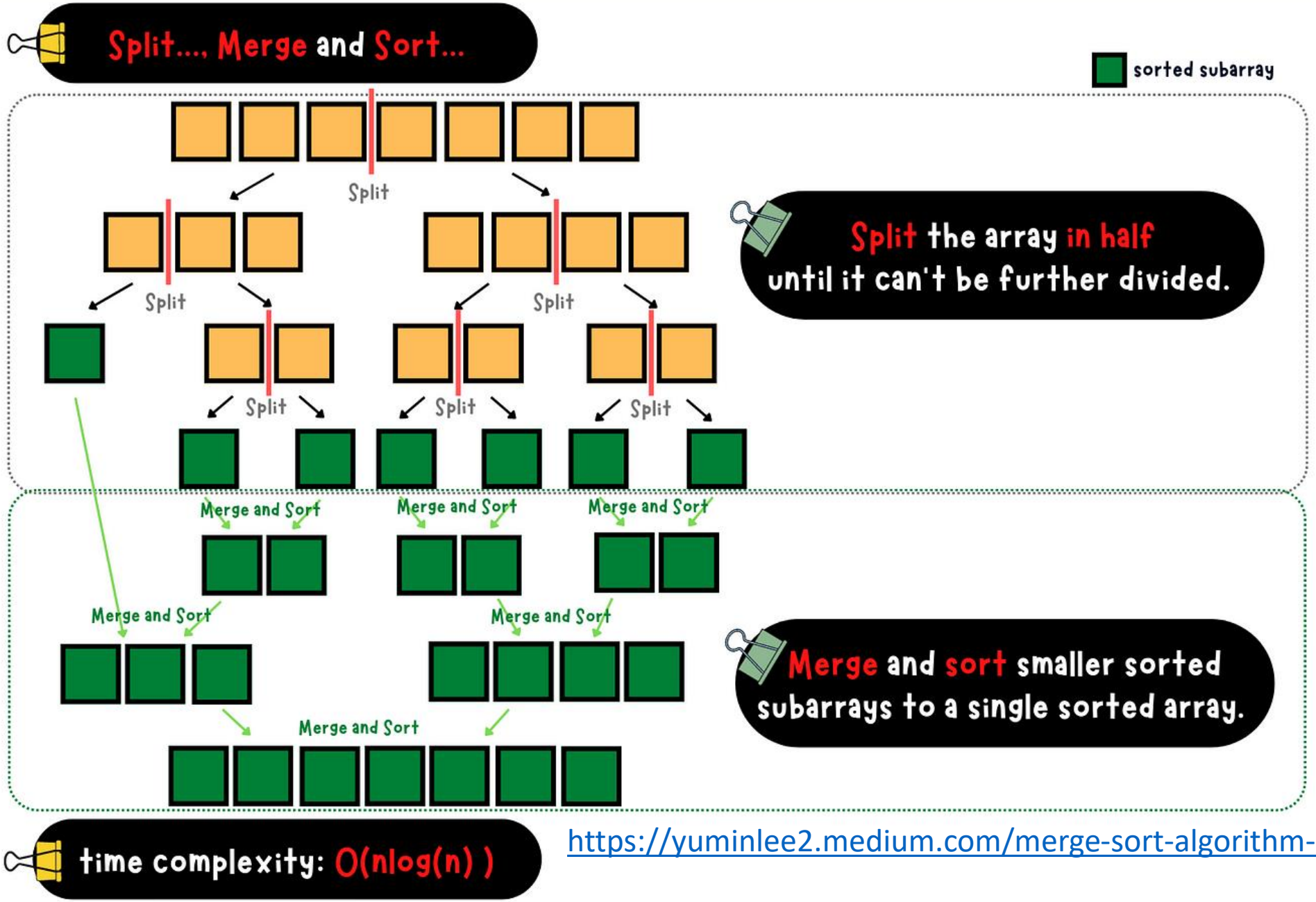
Figure 2.2 The steps done by a human when sorting with Mergesort.



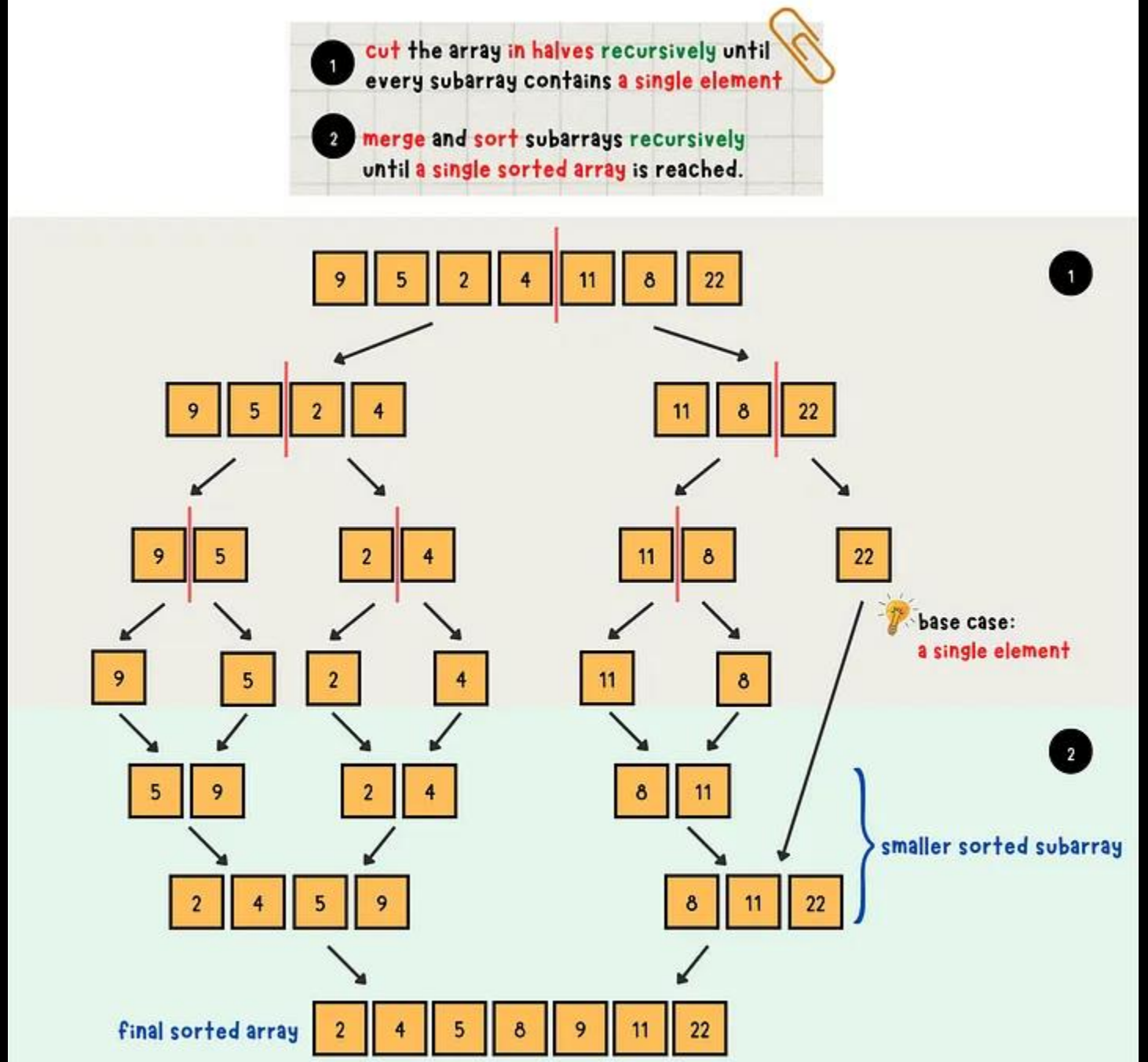
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

<= terminal condition occurs when an array of 1 item occurs; at this point merging starts

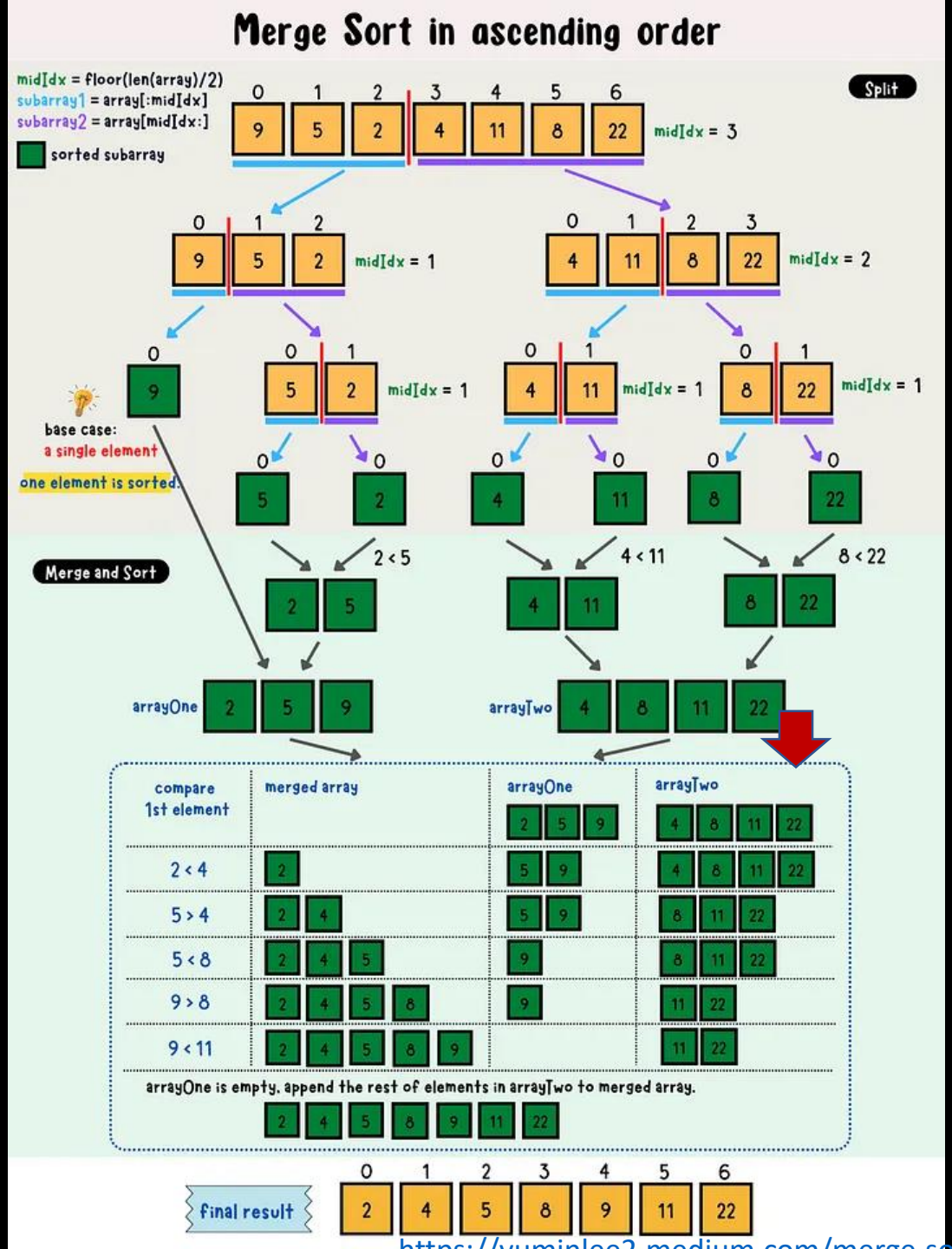
Merge Sort Algorithm



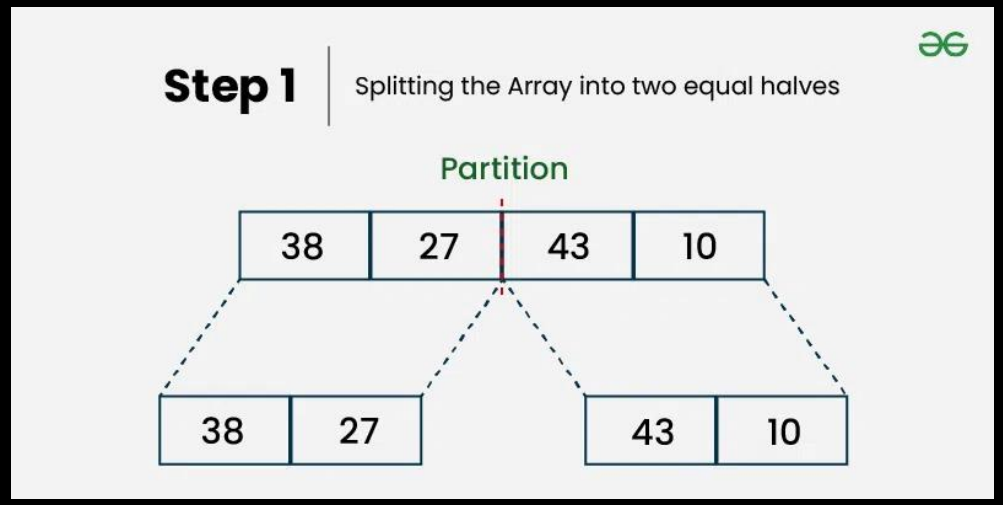
https://yuminlee2.medium.com/merge-sort-algorithm-d4a3ad52a900

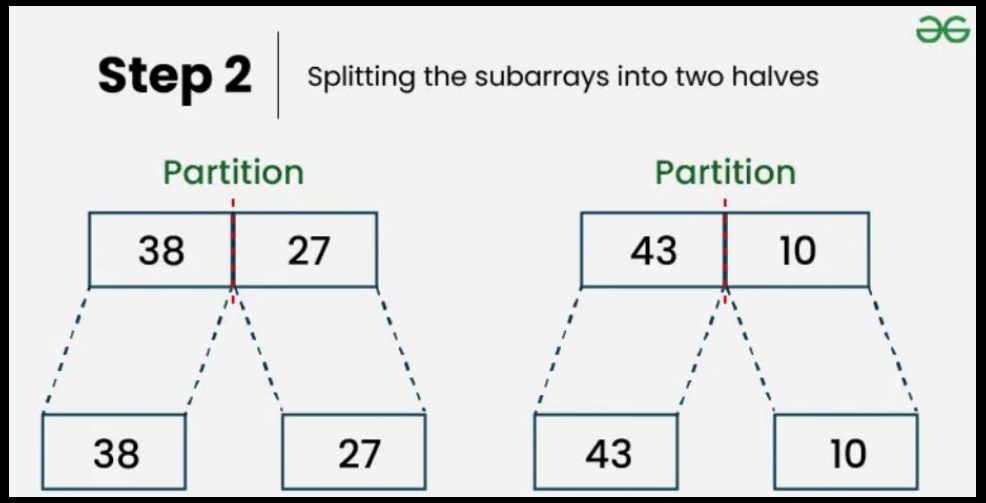


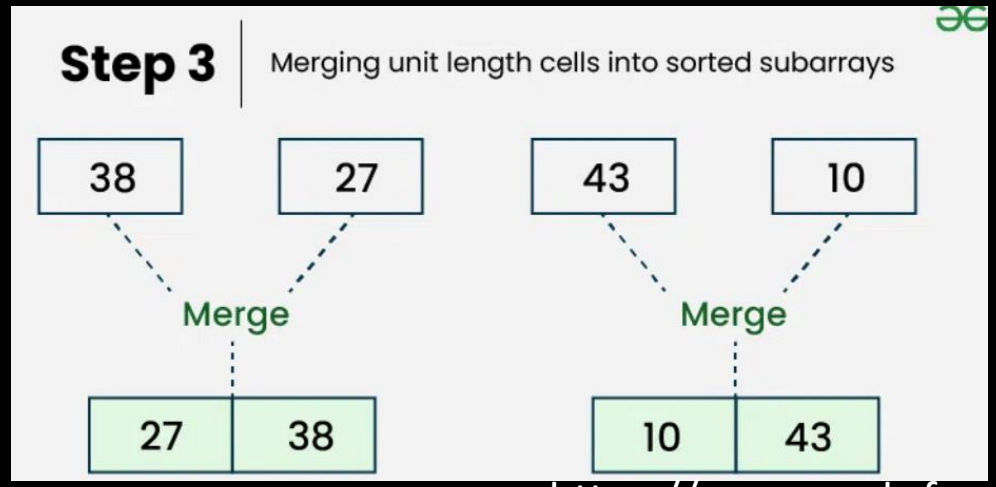
https://yuminlee2.medium.com/merge-sort-algorithm-d4a3ad52a900

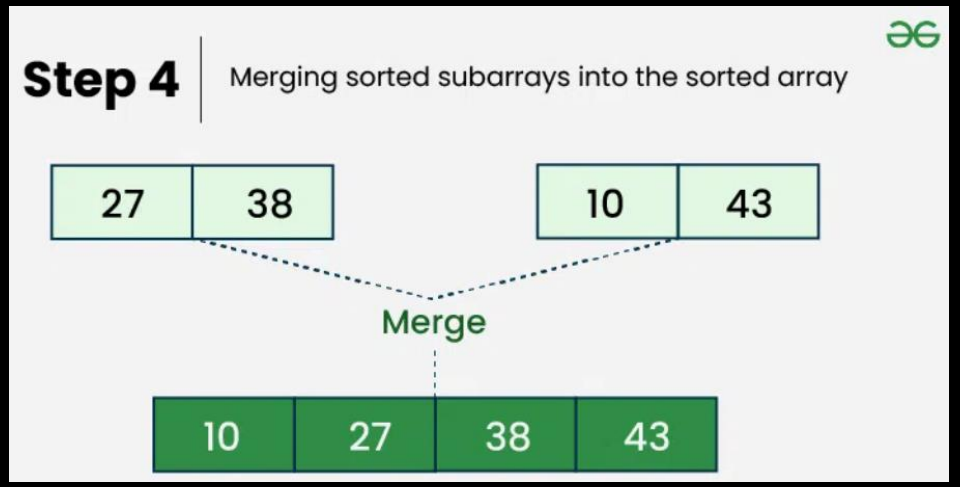

Merge Sort in ascending order
midIdx = floor(len(array)/2)
subarray1 = array[:midIdx]
subarray2 = array[midIdx:]
sorted subarray
Split
midIdx = 3
midIdx = 1
midIdx = 2
base case:
a single element
one element is sorted.
Merge and Sort
2 < 5
4 < 11
8 < 22
arrayOne
arrayTwo
compare 1st element
merged array
2 < 4
5 > 4
5 < 8
9 > 8
9 < 11
arrayOne is empty. append the rest of elements in arrayTwo to merged array.
final result

Merge Sort





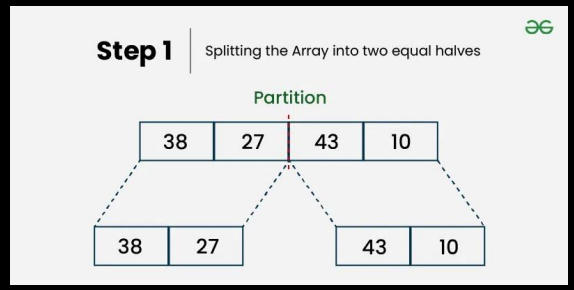


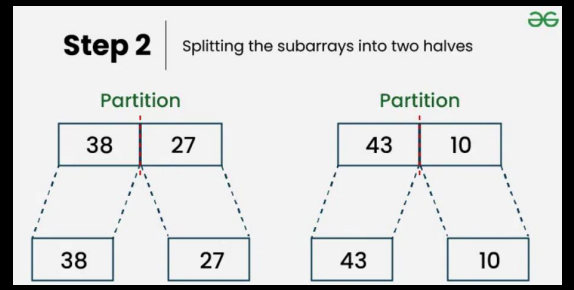


https://www.geeksforgeeks.org/merge-sort/

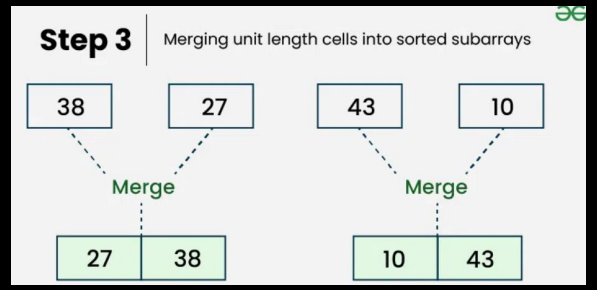
Divide and Conquer – Merge Sort

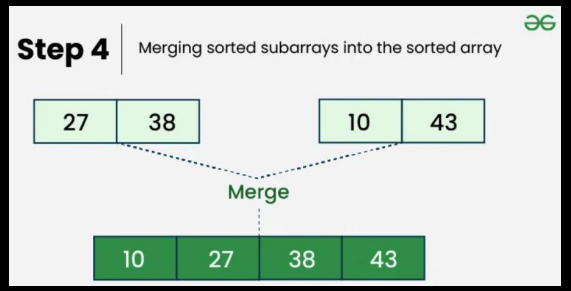
- **Divide**
  - [38, 27, 43, 10] is divided into [38, 27] and [43, 10] .
  - [38, 27] is divided into [38] and [27] .
  - [43, 10] is divided into [43] and [10] .





- **Conquer:**
  - [38] is already sorted.
  - [27] is already sorted.
  - [43] is already sorted.
  - [10] is already sorted.





- **Merge:**
  - Merge [38] and [27] to get [27, 38] .
  - Merge [43] and [10] to get [10,43] .
  - Merge [27, 38] and [10,43] to get the final sorted list [10, 27, 38, 43]
  - Therefore, the sorted list is [10, 27, 38, 43] .

1. *Divide* the array into two subarrays each with $n/2$ items.
2. *Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
3. *Combine* the solutions to the subarrays by merging them into a single sorted array.

https://www.geeksforgeeks.org/merge-sort/

Algorithm 2.2

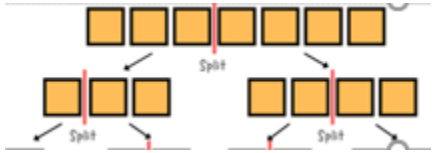
Mergesort

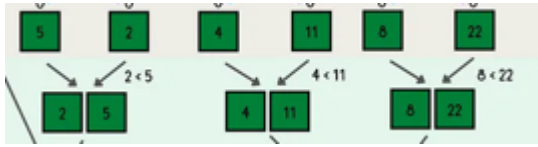
Problem: Sort n keys in nondecreasing sequence.

Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

```
void mergesort (int n, keytype S[])
{
  if (n>1) {
    const int h = ⌊n/2⌋, m = n − h;
    keytype U[1..h], V[1..m];
    copy S[1] through S[h] to U[1] through U[h];
    copy S[h+1] through S[n] to V[1] through V[m];
    mergesort(h, U);
    mergesort(m, V);
    merge(h, m, U, V, S);
  }
}
```





Before we can analyze Mergesort, we must write and analyze an algorithm that merges two sorted arrays.

Algorithm 2.3

Merge

Problem: Merge two sorted arrays into one sorted array.

Inputs: positive integers h and m, array of sorted keys U indexed from 1 to h, array of sorted keys V indexed from 1 to m.

Outputs: an array S indexed from 1 to $h + m$ containing the keys in U and V in a single sorted array.

h and m are size of sub arrays U and V resulting size of S is h +m

```
void merge (int h, int m, const keytype U[],
                          const keytype V[],
                                keytype S[])
{
  index i, j, k;

  i = 1; j = 1;  k = 1;
  while (i <= h && j <= m){
     if (U[i] < V[j]) {
        S[k] = U[i];
        i++;
     }
     else {
        S[k] = V[j];
        j++;
     }
     k++;
  }
  if (i>h)
     copy V[j] through V[m] to S[k] through S[h+m];
  else
     copy U[i] through U[h] to S[k] through S[h+m];
}
```

Table 2.1 An example of merging two arrays U and V into one array S*

k	i U	j V	S (Result)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← Final values

Table 2.1 illustrates how procedure *merge* works when merging two arrays of size 4.

Analysis of Algorithm 2.3

Worse-Case Time Complexity (Merge)

As mentioned in Section 1.3, in the case of algorithms that sort by comparing keys, the comparison instruction and the assignment instruction can each be considered the basic operation. Here we will consider the comparison instruction. When we discuss Mergesort further in Chapter 7, we will consider the number of assignments. In this algorithm, the number of comparisons depends on both h and m. We therefore have the following:

Basic operation: the comparison of $U[i]$ with $V[j]$.

Input size: h and m, the number of items in each of the two input arrays.

h and m are size of sub arrays U and V resulting size of S is h +m

The worst case occurs when the loop is exited, because one of the indices— say, i—has reached its exit point $h + 1$ whereas the other index j has reached m, 1 less than its exit point. For example, this can occur when the first $m - 1$ items in V are placed first in S, followed by all h items in U, at which time the loop is exited because i equals $h + 1$. Therefore,

$$W(h, m) = h + m - 1.$$

Analysis of Algorithm 2.2

Worst-Case Time Complexity (Mergesort)

The basic operation is the comparison that takes place in *merge*. Because the number of comparisons increases with h and m, and h and m increase with n, we have the following:

Basic operation: the comparison that takes place in *merge*.

Input size: n, the number of items in the array S.

The total number of comparisons is the sum of the number of comparisons in the recursive call to *mergesort* with U as the input, the number of comparisons in the recursive call to *mergesort* with V as the input, and the number of comparisons in the top-level call to *merge*. Therefore,

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h+m-1}_{\text{Time to merge}}$$

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h+m-1}_{\text{Time to merge}}$$

We first analyze the case where n is a power of 2. In this case,

$$h = \lfloor n/2 \rfloor = \frac{n}{2}$$

Floor of n/2

$$m = n - h = n - \frac{n}{2} = \frac{n}{2}$$

Substitute h from above in m

$$h + m = \frac{n}{2} + \frac{n}{2} = n.$$

Our expression for $W(n)$ becomes

$$W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1$$

Substitute h & m

$$= 2W\left(\frac{n}{2}\right) + n - 1.$$

When the input size is 1, the terminal condition is met and no merging is done. Therefore, $W(1)$ is 0. We have established the recurrence

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1 \qquad \text{for } n > 1,\ n \text{ a power of } 2$$
$$W(1) = 0$$

This recurrence is solved in Example B.19 in Appendix B. The solution is

$$W(n) = n \lg n - (n-1) \in \Theta(n \lg n).$$

For n not a power of 2, we will establish in the exercises that

floor ceiling

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1,$$

where $\lceil y \rceil$ and $\lfloor y \rfloor$ are the smallest integer $\geq y$ and the largest integer $\leq y$, respectively. It is hard to analyze this case exactly because of the floors ($\lfloor\ \rfloor$) and ceilings ($\lceil\ \rceil$). However, using an induction argument like the one in Example B.25 in Appendix B, it can be shown that $W(n)$ is nondecreasing. Therefore, Theorem B.4 in that appendix implies that

$$W(n) \in \Theta(n \lg n).$$

In-place sorting

- In-place sorting refers to a category of sorting algorithms that rearrange the elements of an array or list directly within the original memory location, without requiring a significant amount of additional memory for temporary storage.
- in-place algorithm is an algorithm which transforms input using a data structure with a small, constant amount of extra storage space.
- In-place sorting means sorting without any extra space requirement.
- This means that the algorithm does not need to create a new array to store the sorted elements.
- Quicksort is one example of In-Place Sorting.

Complexity Analysis of Merge Sort

- Best Case: Ω(n log n), When the array is already sorted or nearly sorted.
- Average Case: θ(n log n), When the array is randomly ordered.
- Worst Case: O(n log n), When the array is sorted in reverse order.
- Auxiliary Space: O(n), Additional space is required for the temporary array used during merging.

```
keytype U[1..h], V[1..m];
copy S[1] through S[h] to U[1] through U[h];
copy S[h+1] through S[n] to V[1] through V[m];
mergesort(h, U);
mergesort(m, V);
merge(h, m, U, V, S);
```

https://www.geeksforgeeks.org/merge-sort/

Space Complexity Analysis

- In-place sort ?
  - A sorting algorithm that does not use any extra space beyond that needed to store the input.
  - Mergesort() is not an in-place sorting algorithm.
  - New arrays U and V will be created when *mergesort* is called.
  - The total number of extra array items created is $n+\frac{n}{2}+\frac{n}{4}+\cdots=2n$
  - In other words, the space complexity is

    $2n \in O(n)$

  - We may reduce the extra space to n.
  - But it is not possible to make mergesort algorithm to be an in-place sort.

Advantages of Merge Sort

i. Stability : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
   e.g 1,3,6,7,7,8,6,8,8,9 – sort students according to height where two students are same height
ii. Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N log N) , which means it performs well even on large datasets.
iii. Simple to implement: The divide-and-conquer approach is straightforward.
iv. Naturally Parallel : We independently merge subarrays that makes it suitable for parallel processing.

https://www.geeksforgeeks.org/merge-sort/

Disadvantages of Merge Sort

i. Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.

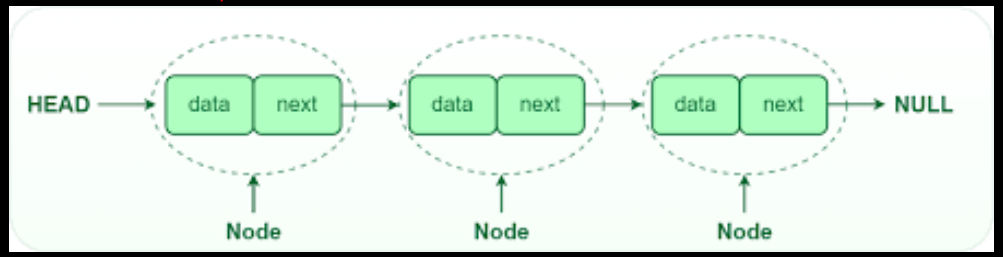
ii. Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.

iii. Slower than QuickSort in general. QuickSort is more cache friendly because it works in-place.

https://www.geeksforgeeks.org/merge-sort/

Applications of Merge Sort

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.



https://www.geeksforgeeks.org/merge-sort/

2.3 The Divide-and-Conquer Approach

Having studied two divide-and-conquer algorithms in detail, you should now better understand the following general description of this approach.

The ***divide-and-conquer*** design strategy involves the following steps:

1. *Divide* an instance of a problem into one or more smaller instances.
2. *Conquer* (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, *combine* the solutions to the smaller instances to obtain the solution to the original instance.

The reason we say "if necessary" in Step 3 is that in algorithms such as Binary Search Recursive (Algorithm 2.1) the instance is reduced to just one smaller instance, so there is no need to combine solutions.

2.4 Quicksort (Partition Exchange Sort)

Next we look at a sorting algorithm, called "Quicksort," that was developed by Hoare (1962). Quicksort is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively. In Quicksort, however, the array is partitioned by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it. The pivot item can be any item, and for convenience we will simply make it the first one. The following example illustrates how Quicksort works.

- It is still a commonly used algorithm for sorting. Overall, it is slightly faster than merge sort and heapsort for randomized data, particularly on larger distributions. Quick sort can be used to sort ascending or descending order

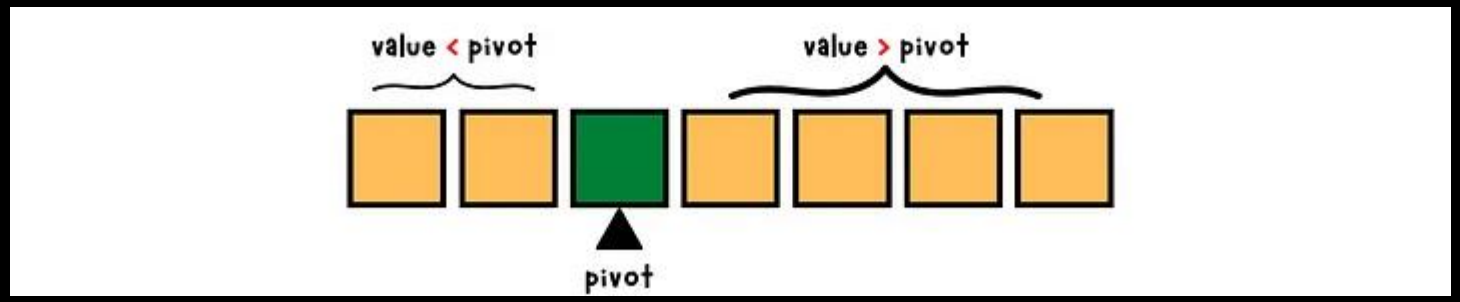
Quick Sort

- The quick sort uses divide and conquer to gain the same advantages as the merge sort, while NOT using additional storage.
- As a limitation, however, it is possible that the list may not be divided in half. When this happens, the performance diminishes(i.e.reduces).
- A quick sort first selects a value, which is called the pivot value. (can be middle, first, last etc)
- The role of the pivot value is to assist with splitting the list.
- The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.

How does Quick Sort work?

- The key process in Quick Sort 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.



https://www.geeksforgeeks.org/quick-sort-algorithm/

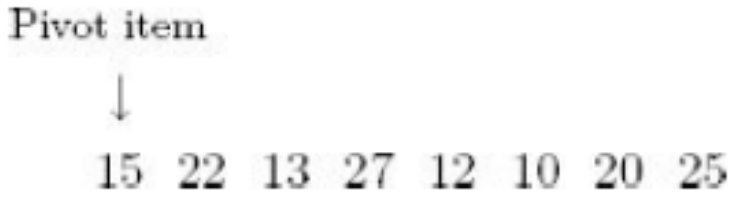
Pivot Selection

- There are many different choices for picking pivots
- Examples
  i. Always pick the first element as a pivot
  ii. Always pick the last element as a pivot
  iii. Pick a random element as a pivot
  iv. Pick the middle as the pivot

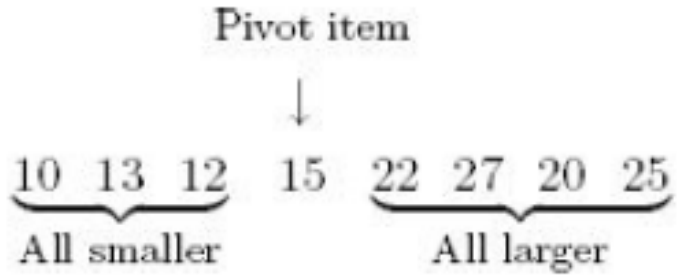
https://www.geeksforgeeks.org/quick-sort-algorithm/

Example 2.3

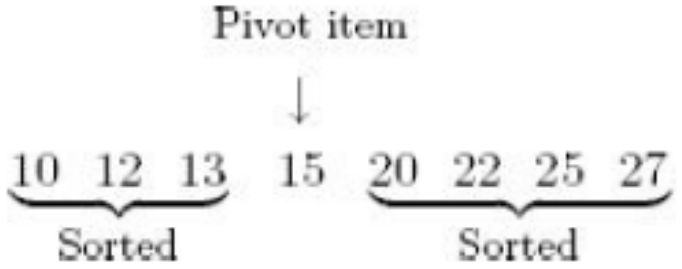
Suppose the array contains these numbers in sequence:



1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:



2. Sort the subarrays:



After the partitioning, the order of the items in the subarrays is unspecified and is a result of how the partitioning is implemented. We have ordered them according to how the partitioning routine, which will be presented shortly, would place them. The important thing is that all items smaller than the pivot item are to the left of it, and all items larger are to the right of it. Quicksort is then called recursively to sort each of the two subarrays. They are partitioned, and this procedure is continued until an array with one item is reached. Such an array is trivially sorted. Example 2.3 shows the solution at the problem-solving level. Figure 2.3 illustrates the steps done by a human when sorting with Quicksort. The algorithm follows.

Algorithm 2.6

Quicksort

Problem: Sort n keys in nondecreasing order.

Inputs: positive integer n, array of keys S indexed from 1 to n.

Outputs: the array S containing the keys in nondecreasing order.

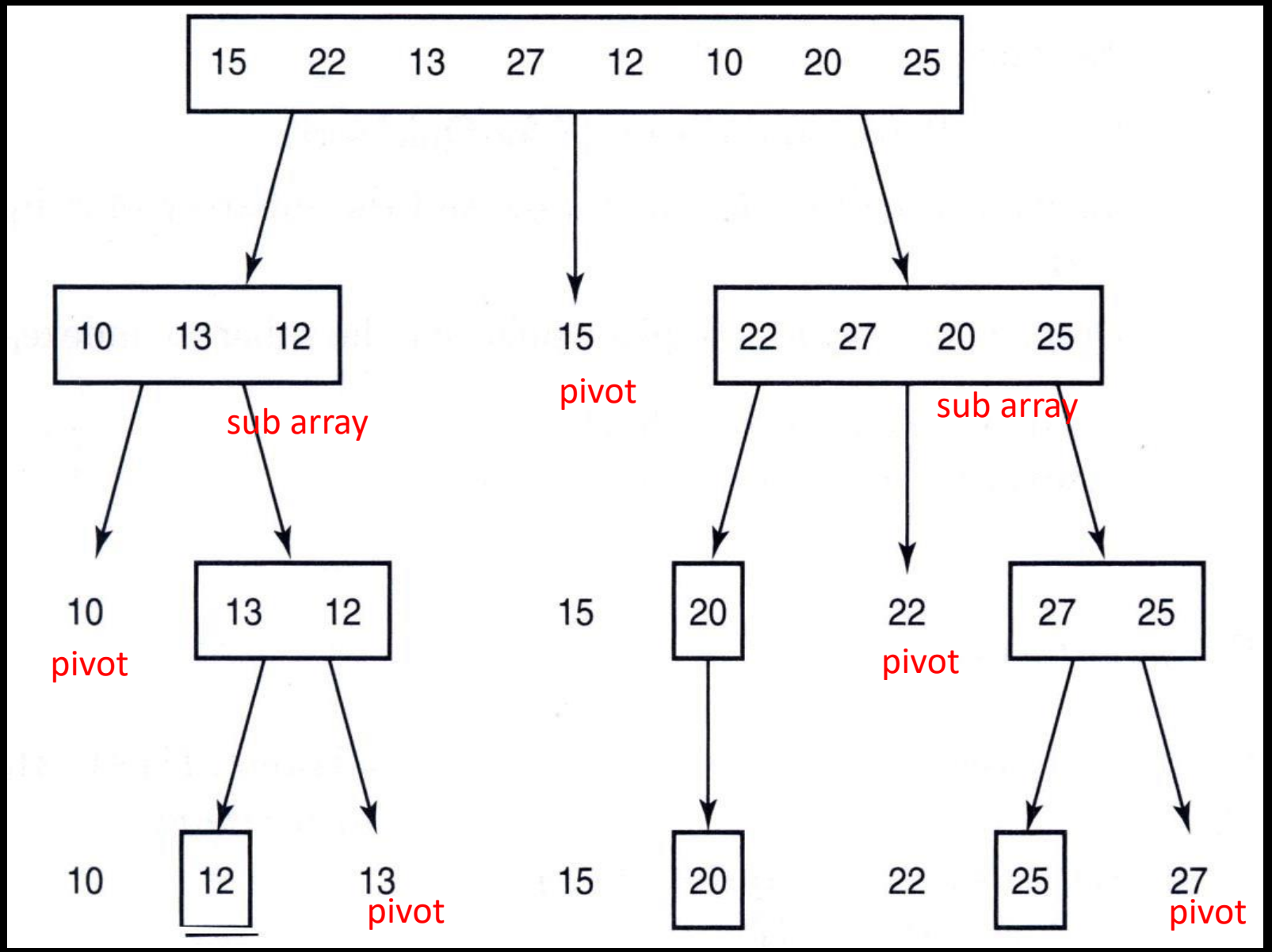
```
void quicksort (index low, index high)
{
  index pivotpoint;

  if (high > low){
     partition(low, high, pivotpoint);
     quicksort(low, pivotpoint - 1);
     quicksort(pivotpoint + 1, high);
  }
}
```

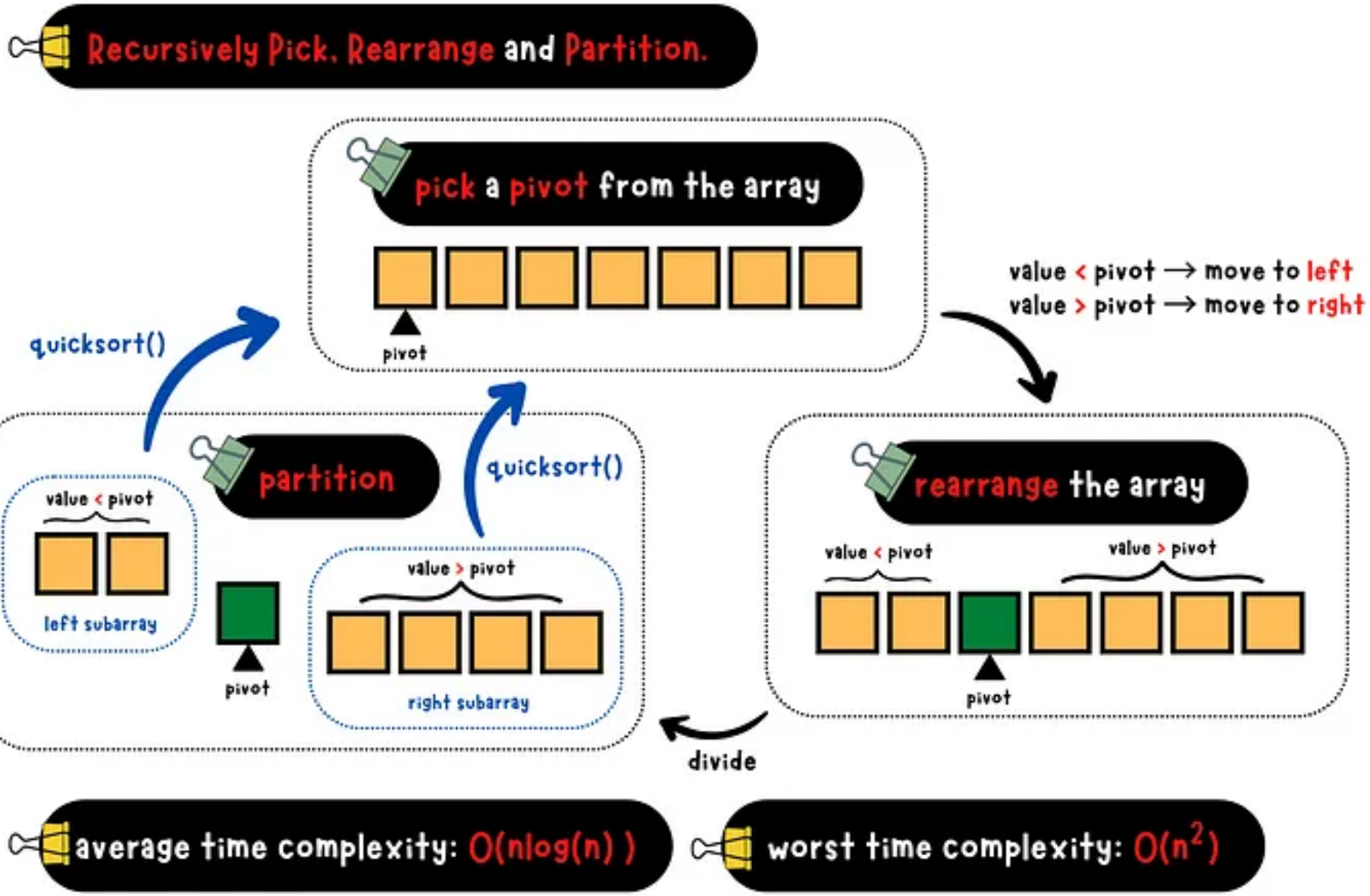
Following our usual convention, n and S are not parameters to procedure *quicksort*. If the algorithm were implemented by defining S globally and n was the number of items in S, the top-level call to *quicksort* would be as follows:

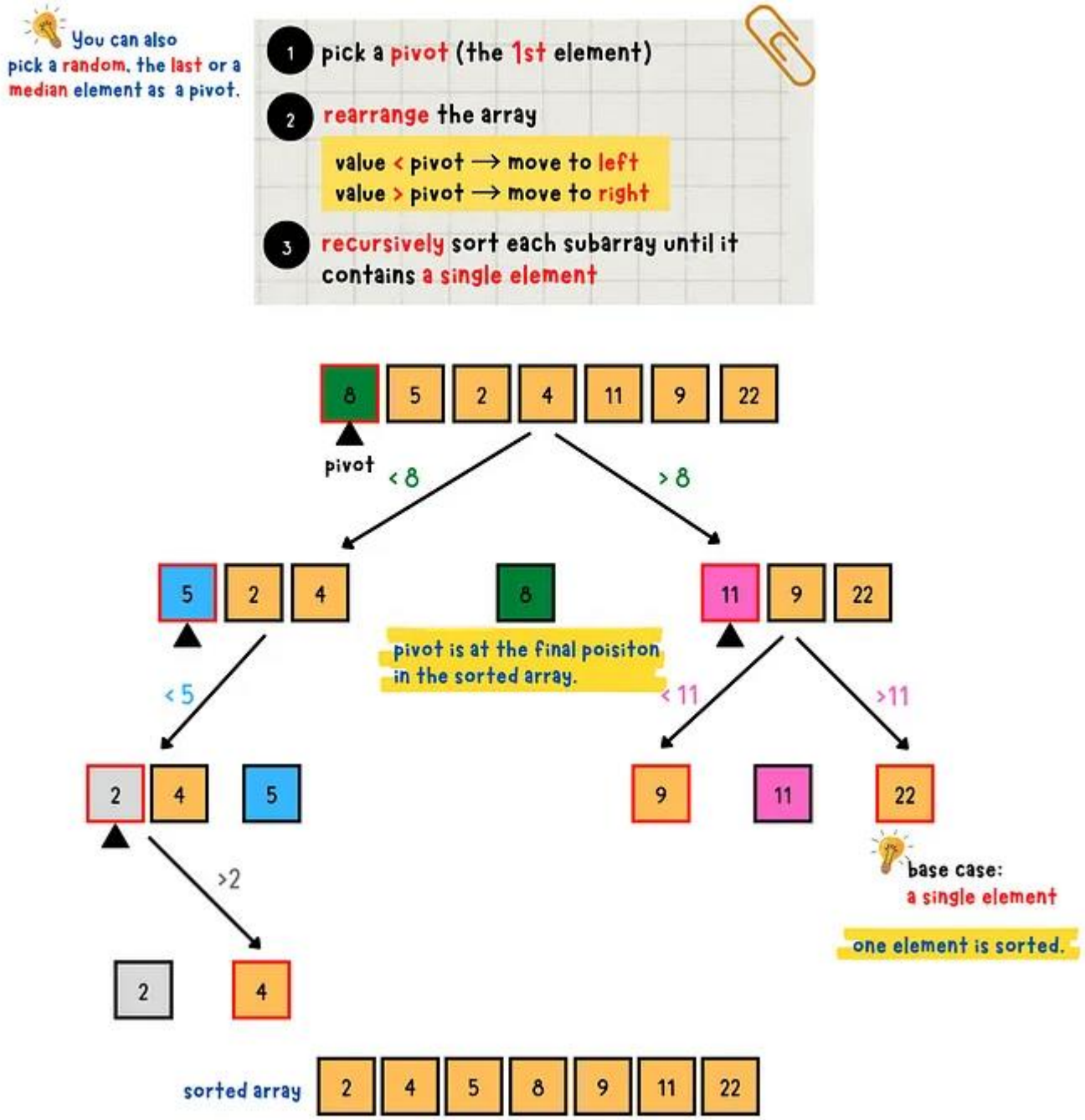
quicksort (1 , n) ;

Figure 2.3 The steps done by a human when sorting with Quicksort.
The subarrays are enclosed in rectangles whereas the pivot points are free.

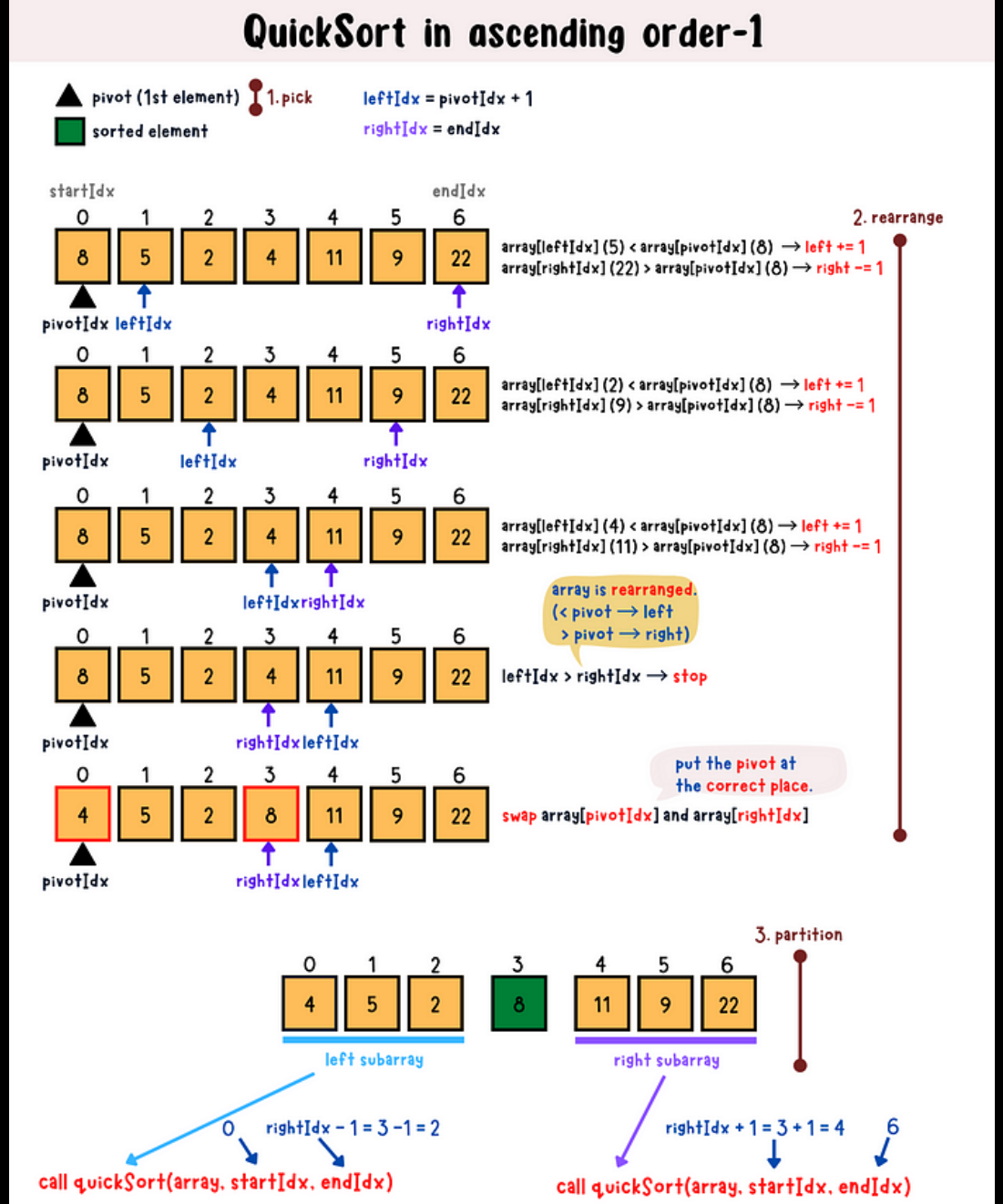


QuickSort Algorithm

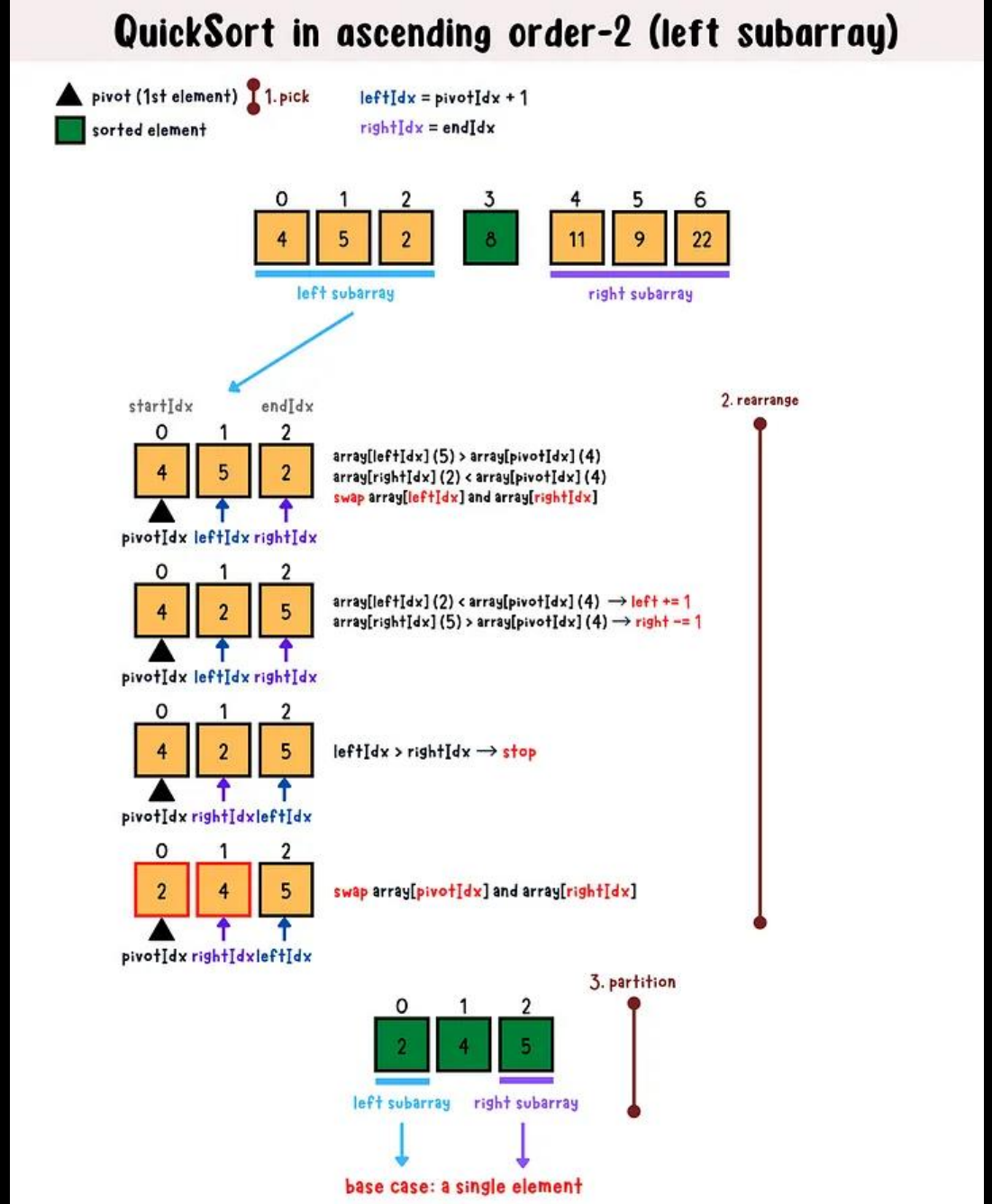




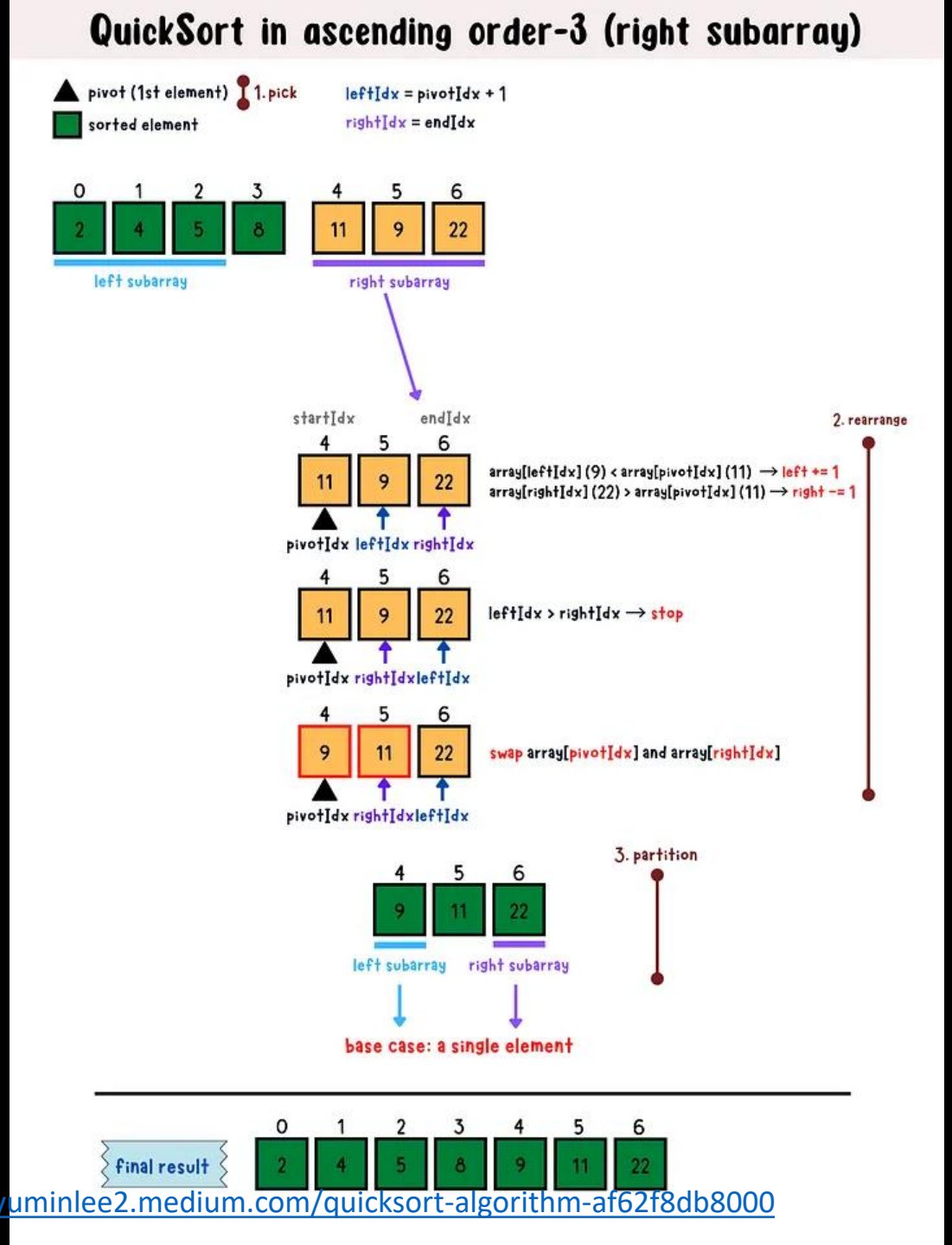
https://yuminlee2.medium.com/quicksort-algorithm-af62f8db8000



https://yuminlee2.medium.com/quicksort-algorithm-af62f8db8000



https://yuminlee2.medium.com/quicksort-algorithm-af62f8db8000

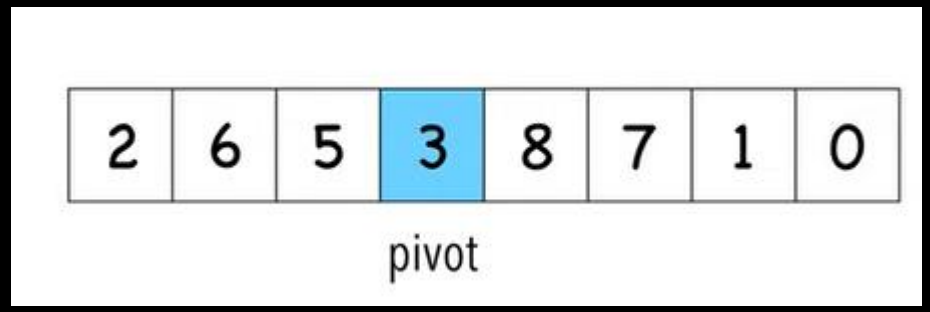


https://yuminlee2.medium.com/quicksort-algorithm-af62f8db8000

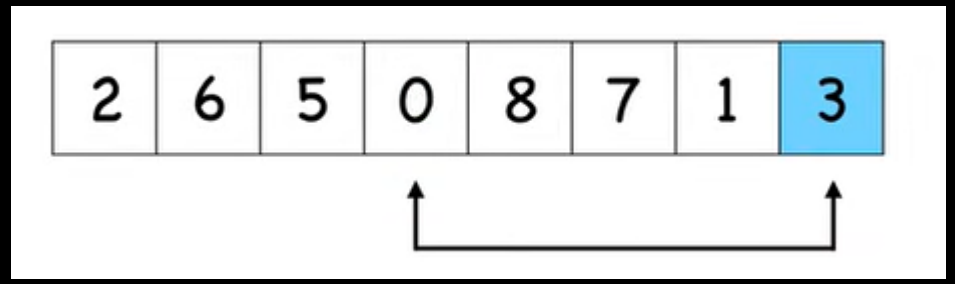
Quick Sort Ascending Further Example

Pivot?

1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger

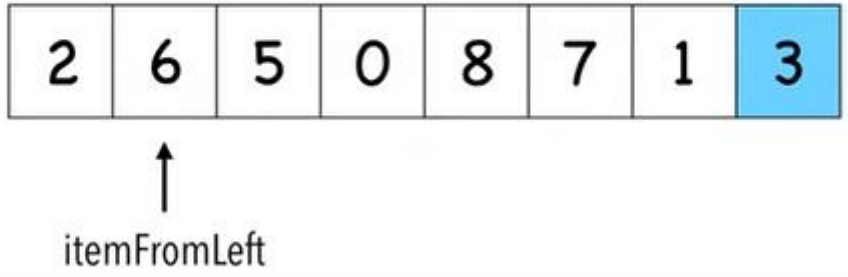


(1)



(2)

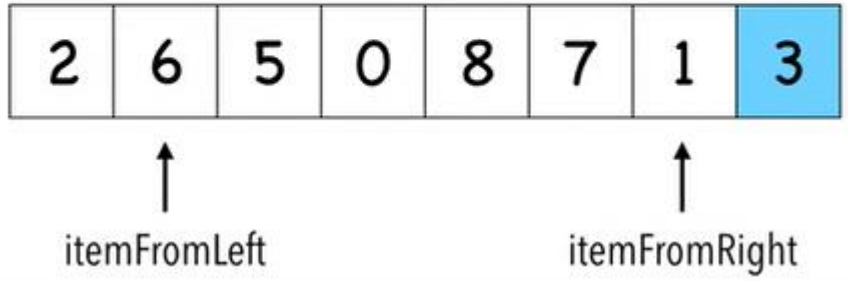
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



(3)

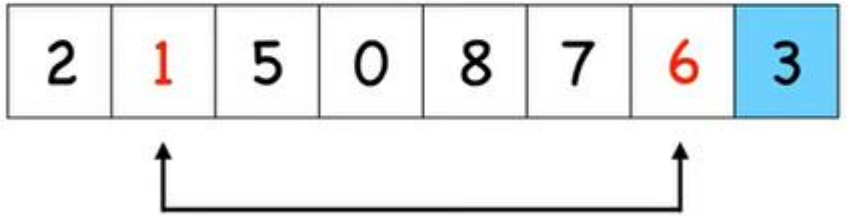
Refer: **Quick sort in 4 minutes**
https://www.youtube.com/watch?v=Hoixgm4-P4M&t=130s

1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

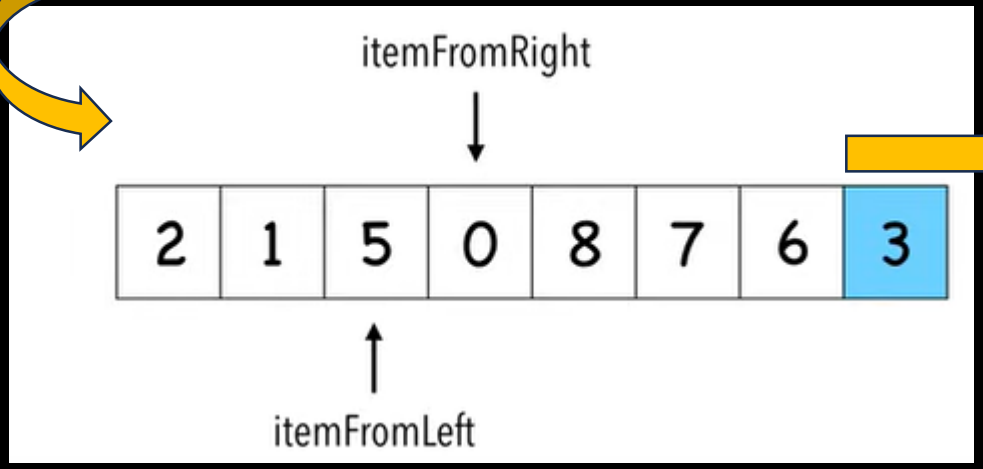


(4)

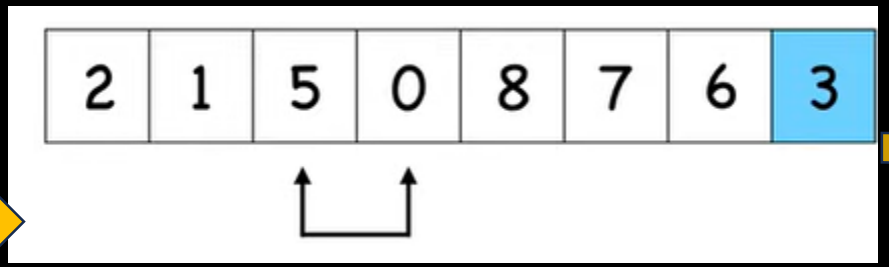
1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot



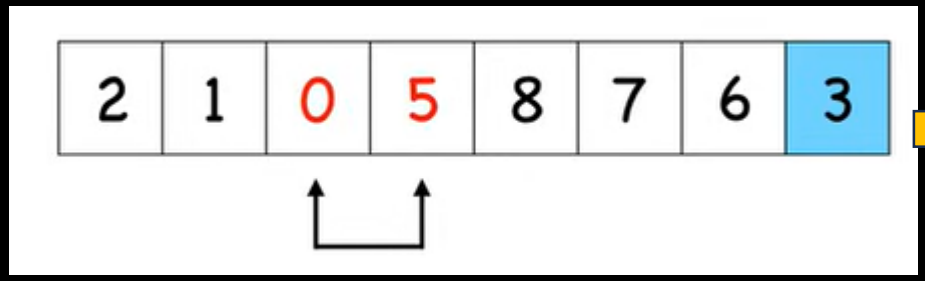
(5)



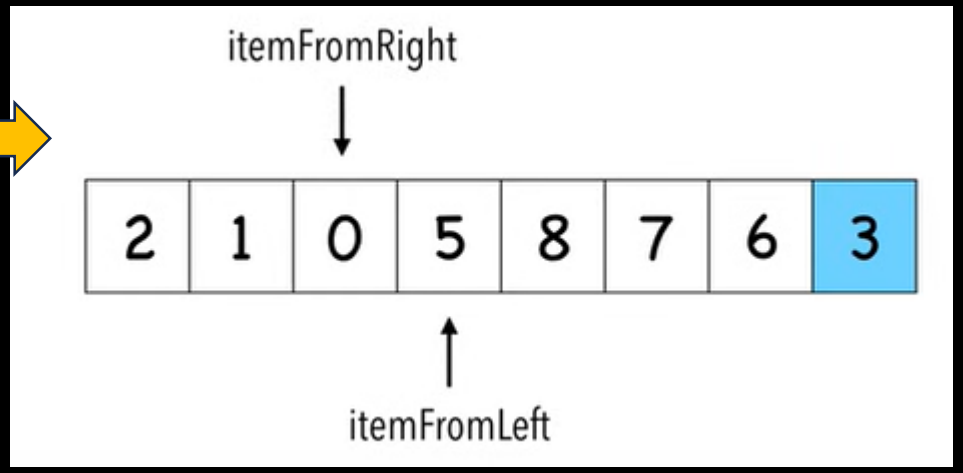
(6)



(7)

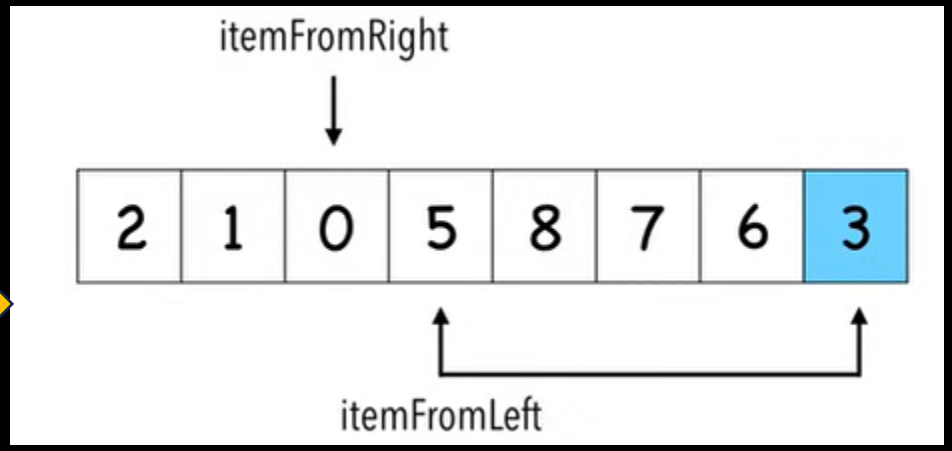


(8)



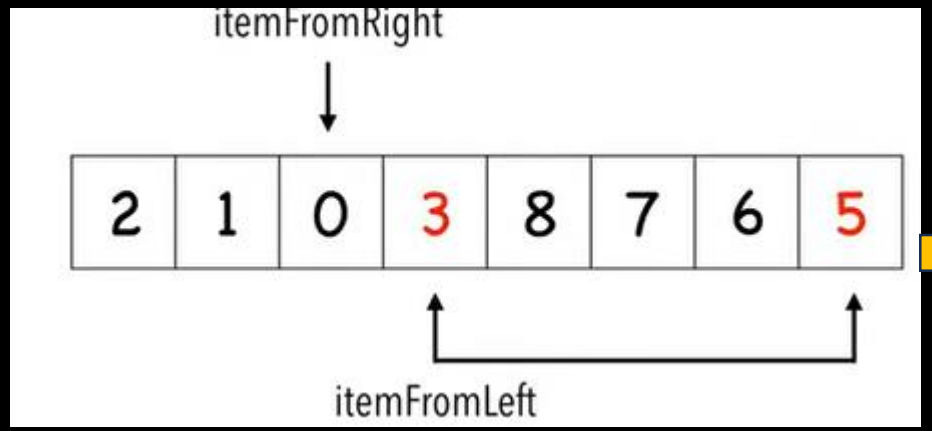
Stop when item from left > Item from right=> Done!

(9)



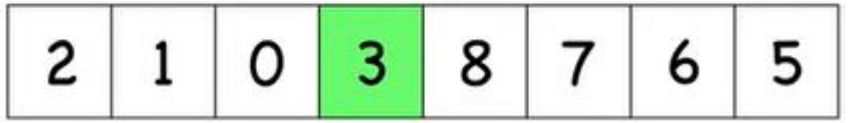
(10)

Swap item from Left with Pivot

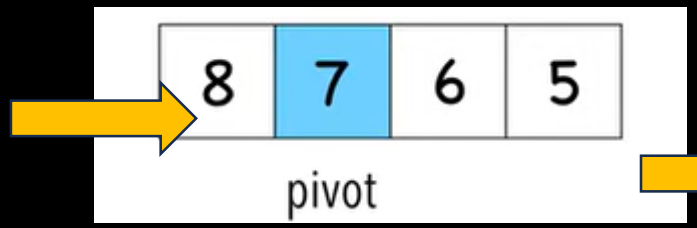


(11)

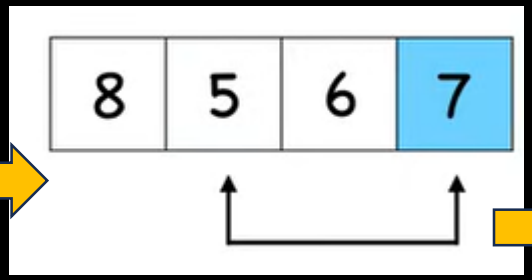
1. Correct position in final, sorted array
2. Items to the left are smaller
3. Items to the right are larger



(12)



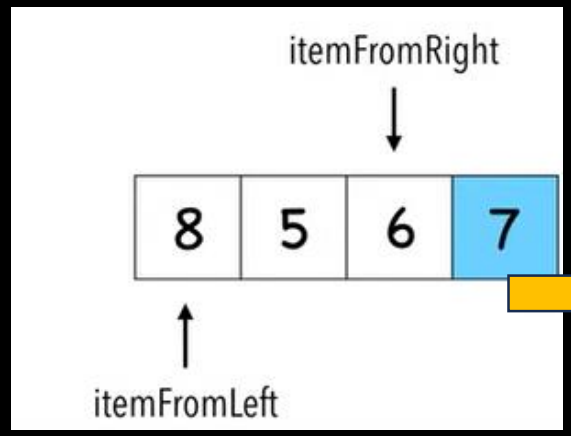
(13)



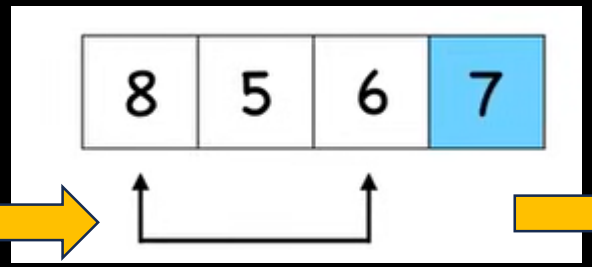
(14)

1. itemFromLeft that is larger than pivot
2. itemFromRight that is smaller than pivot

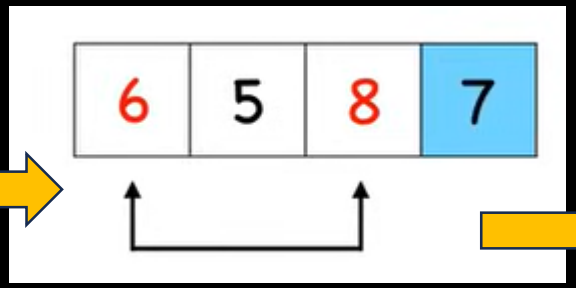
(15)



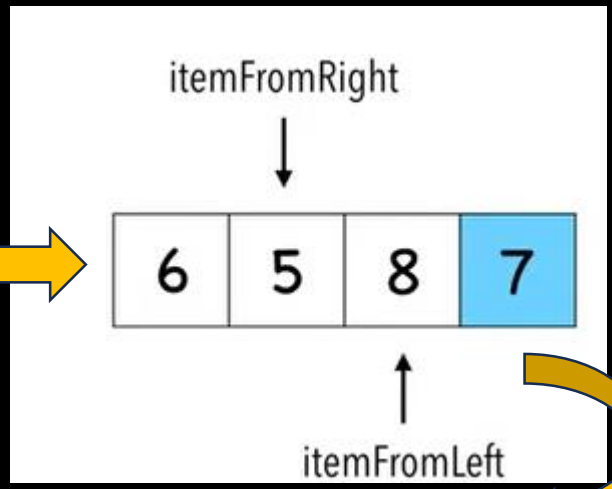
(16)



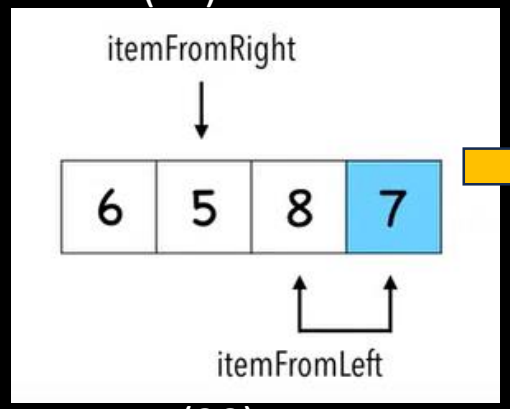
(17)



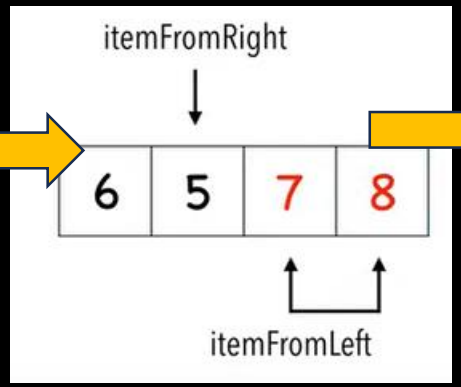
(18)



(19)



(20)



(21)

(22)

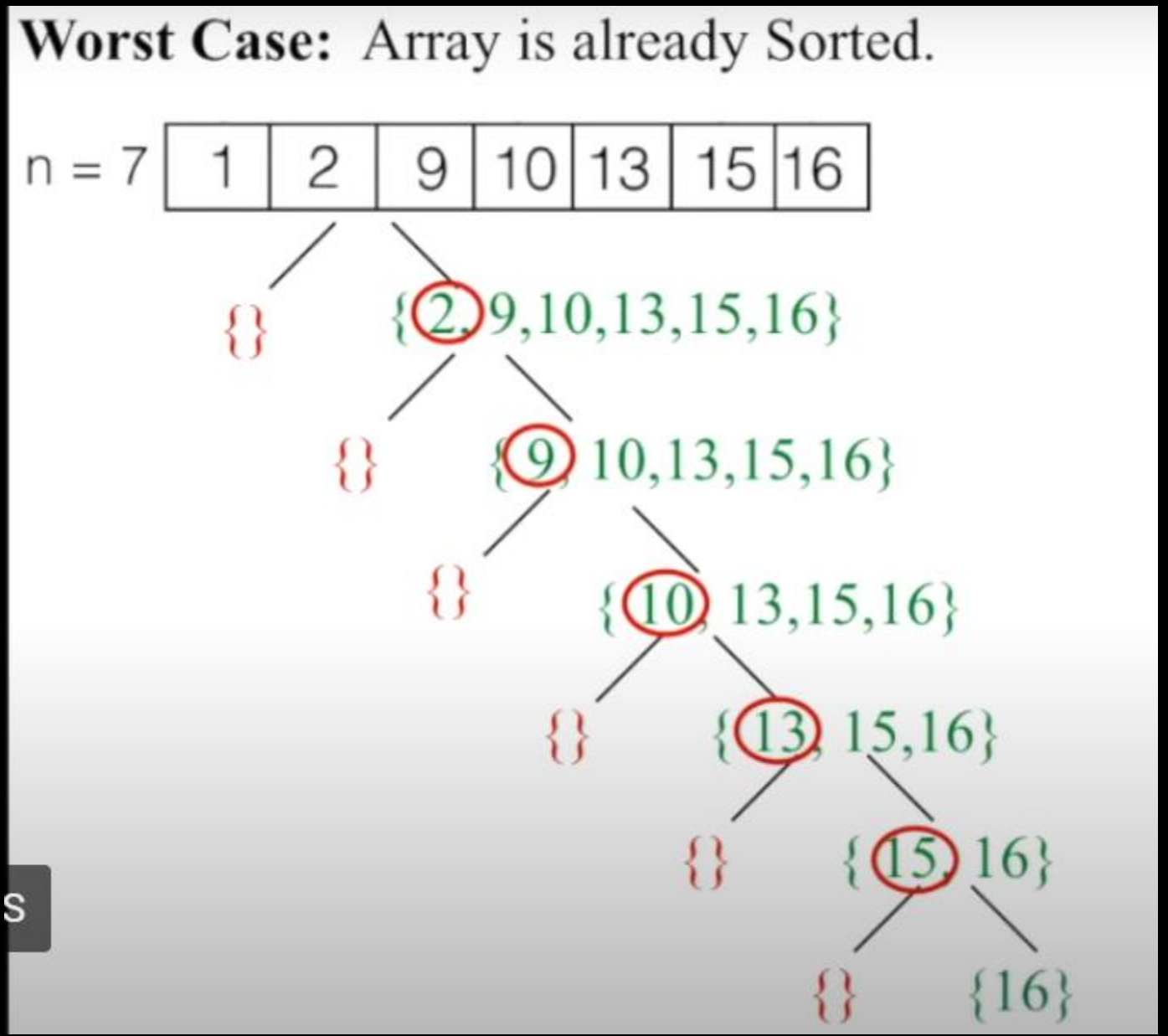
(23)

Quick Sort Time Complexity

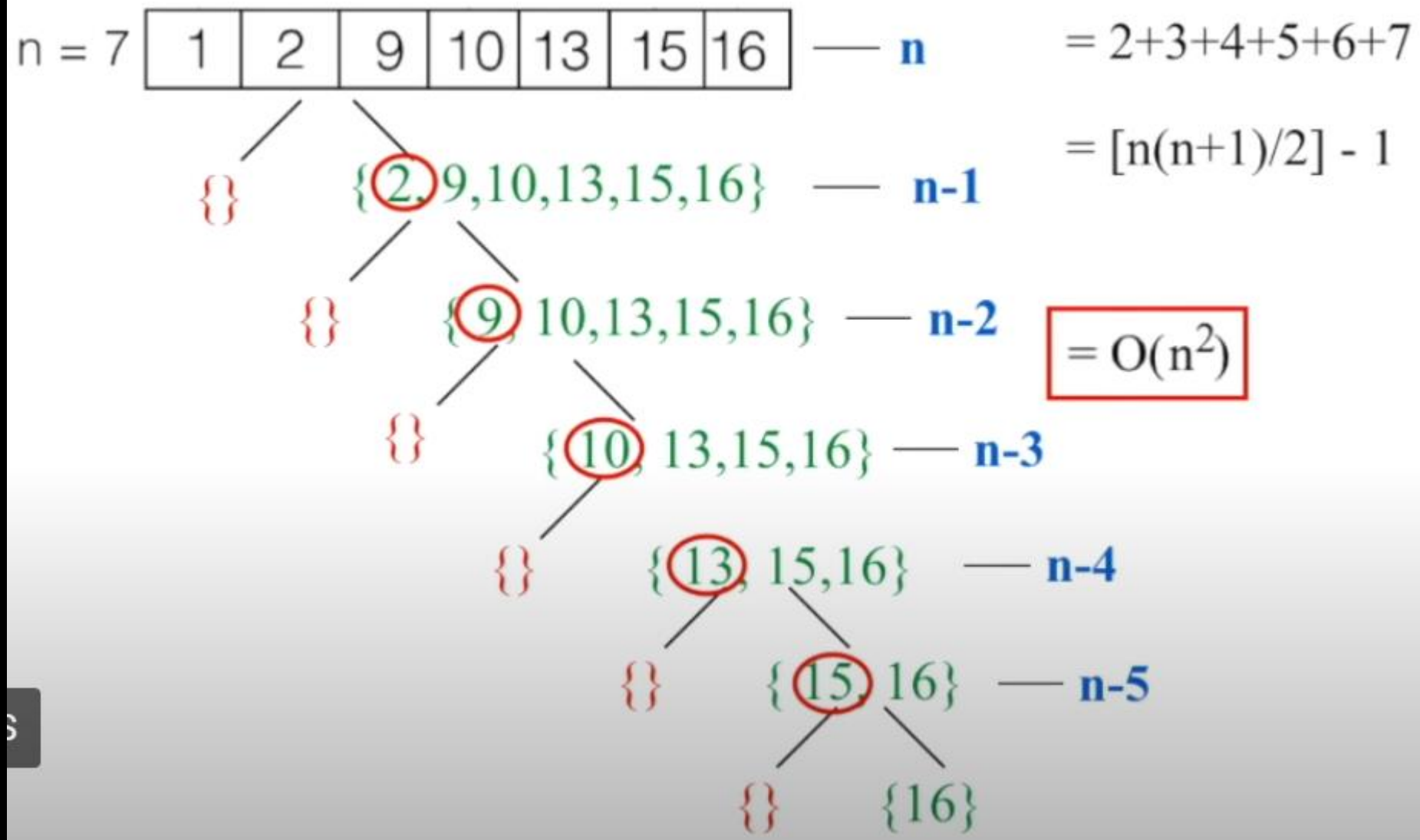
- Best Case : Ω (N log (N))
  - The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
  - In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- Average Case: θ (N log (N))
  - Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- Worst Case: O(N ^ 2) – see next slide
  - The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.
- Auxiliary Space: O(1)
  - If we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O (N).

https://www.geeksforgeeks.org/quick-sort-algorithm/

Worst Case: Array is already Sorted.



Worst Case: Array is already Sorted.



$= 2+3+4+5+6+7$

$= [n(n+1)/2] - 1$

$= O(n^2)$

The partitioning of the array is done by procedure *partition*. Next we show an algorithm for this procedure.

Algorithm 2.7

Partition

Problem: Partition the array *S* for Quicksort.

Inputs: two indices, *low* and *high*, and the subarray of *S* indexed from *low* to *high*.

Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*.

```
void partition (index low, index high,
                index& pivotpoint)
{
   index i, j;
   keytype pivotitem;

   pivotitem = S[low];                    // Choose first item for
   j = low;                               // pivotitem.
   for (i = low + 1; i <= high; i++)
      if (S[i] < pivotitem){
         j++;
         exchange S[i] and S[j];
      }
   pivotpoint = j;
   exchange S[low] and S[pivotpoint]; // Put pivotitem at pivotpoint.
}
```

If an element if smaller than pivot item the increment j and swap

• **Table 2.2** An example of procedure *partition*[*]

i	j	$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$	
—	—	15	22	13	27	12	10	20	25	← Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	[13]	[22]	**27**	12	10	20	25	
5	3	**15**	13	**22**	27	**12**	10	20	25	
6	4	**15**	13	[12]	27	[22]	**10**	20	25	
7	4	**15**	13	12	[10]	22	[27]	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	[10]	13	12	[15]	22	27	20	25	← Final values

[*] Items compared are in boldface. Items just exchanged appear in squares.

Procedure *partition* works by checking each item in the array in sequence. Whenever an item is found to be less than the pivot item, it is moved to the left side of the array. Table 2.2 shows how *partition* would proceed on the array in Example 2.3.

Next we analyze Partition and Quicksort.

Analysis of Algorithm 2.6

Worst-Case Time Complexity (Quicksort)

Basic operation: the comparison of $S[i]$ with *pivotitem* in *partition*.

Input size: n, the number of items in the array S.

Oddly enough, it turns out that the worst case occurs if the array is already sorted in nondecreasing order. The reason for this should become clear. If the array is already sorted in nondecreasing order, no items are less than the first item in the array, which is the pivot item. Therefore, when *partition* is called at the top level, no items are placed to the left of the pivot item, and the value of *pivotpoint* assigned by *partition* is 1. Similarly, in each recursive call, *pivotpoint* receives the value of *low*. Therefore, the array is repeatedly partitioned into an empty subarray on the left and a subarray with one less item on the right. For the class of instances that are already sorted in nondecreasing order, we have

$$T(n) = \underbrace{T(0)}_{\text{Time to sort left subarray}} + \underbrace{T(n-1)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

Worst Case: O(N ^ 2)

The worst case occurs when the array is already sorted because we always choose the first item for the pivot item. Therefore, if we have reason to believe that the array is close to being sorted, this is not a good choice for the pivot item. When we discuss Quicksort further in Chapter 7, we will investigate other methods for choosing the pivot item. If we use these methods, the worst case does not occur when the array is already sorted. But the worst-case time complexity is still $n\,(n-1)/2$.

In the worst case, Algorithm 2.6 is no faster than Exchange Sort (Algorithm 1.3). Why then is this sort called Quicksort? As we shall see, it is in its average-case behavior that Quicksort earns its name.

Analysis of Algorithm 2.6

Quicksort's average-case time complexity is of the same order as Mergesort's time complexity.

Average-Case Time Complexity (Quicksort)

Basic operation: the comparison of $S[i]$ with *pivotitem* in *partition*

Input size: n, the number of items in the array S.

We will assume that we have no reason to believe that the numbers in the array are in any particular order, and therefore that the value of *pivotpoint* returned by *partition* is equally likely to be any of the numbers from 1 through n. If there was reason to believe a different distribution, this analysis would not be applicable. The average obtained is, therefore, the average sorting time when every possible ordering is sorted the same number of times. In this case, the average-case time complexity is given by the following recurrence:

$$A(n) = \sum_{p=1}^{n} \underbrace{\frac{1}{n}}_{\substack{\text{Probability} \\ \textit{pivotpoint} \text{ is } p}} \underbrace{[A(p-1) + A(n-p)]}_{\substack{\text{Average time to} \\ \text{sort subarrays when} \\ \textit{pivotpoint} \text{ is } p}} + \underbrace{n-1}_{\substack{\text{Time to} \\ \text{partition}}} \qquad (2.1)$$

$$\approx 1.38\,(n+1)\lg n \in \Theta(n \lg n).$$

Advantages of Quick Sort

i. It is a divide-and-conquer algorithm that makes it easier to solve problems.
ii. It is efficient on large data sets.
iii. It has a low overhead, as it only requires a small amount of memory to function.
iv. It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
v. Fastest general purpose algorithm for large data when stability is not required.
vi. It is tail recursive and hence all the tail call optimization can be done. - Tail call optimization (TCO) is an optimization technique that eliminates the need for allocating a new stack frame for a function when the function call is the last operation performed in the calling function.

https://www.geeksforgeeks.org/quick-sort-algorithm/

Disadvantages of Quick Sort

i. It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.

ii. It is not a good choice for small data sets.

iii. It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

https://www.geeksforgeeks.org/quick-sort-algorithm/

2.5 Strassen's Matrix Multiplication Algorithm

Recall that Algorithm 1.4 (Matrix Multiplication) multiplied two matrices strictly according to the definition of matrix multiplication. We showed that the time complexity of its number of multiplications is given by $T(n) = n^3$, where n is the number of rows and columns in the matrices. We can also analyze the number of additions. As you will show in the exercises, after the algorithm is modified slightly, the time complexity of the number of additions is given by $T(n) = n^3 - n^2$. Because both of these time complexities are in $\Theta(n^3)$, the algorithm can become impractical fairly quickly. In 1969, Strassen published an algorithm whose time complexity is better than cubic in terms of both multiplications and additions/subtractions. The following example illustrates his method.

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is $\mathbf{O(n^3)}$, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from $\mathbf{O(n^3)}$ to $\mathbf{O(n^{\log 7})}$.

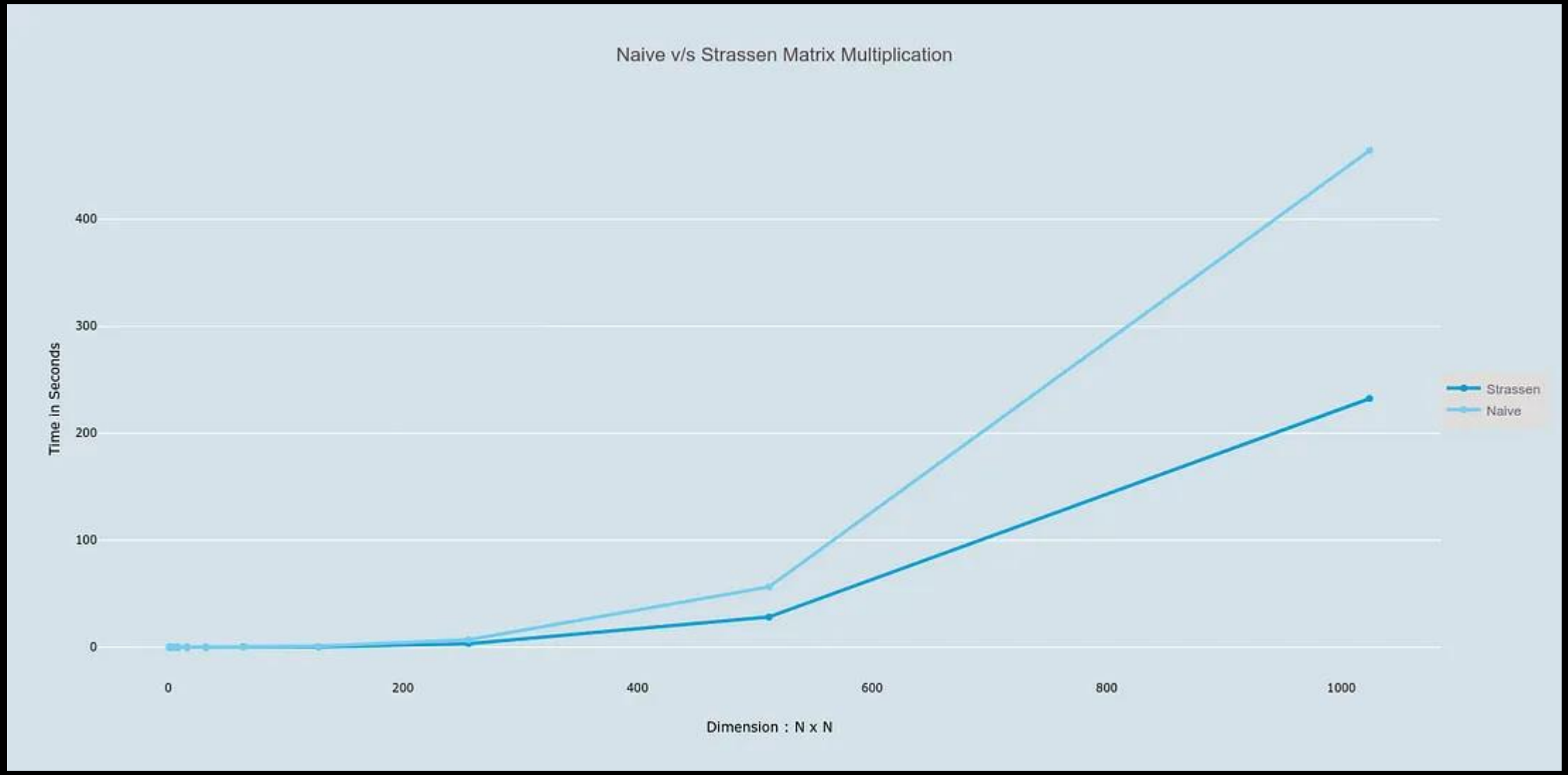
Naive Method

First, we will discuss Naive method and its complexity. Here, we are calculating Z=$\boldsymbol{X}$X × Y. Using Naive method, two matrices (**X** and **Y**) can be multiplied if the order of these matrices are **p** × **q** and **q** × **r** and the resultant matrix will be of order **p** × **r**. The following pseudocode describes the Naive multiplication –

```
Algorithm: Matrix-Multiplication (X, Y, Z)
for i = 1 to p do
   for j = 1 to r do
      Z[i,j] := 0
      for k = 1 to q do
         Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

Here, we assume that integer operations take $\mathbf{O(1)}$ time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes $\mathbf{O(n^3)}$ time to execute.

https://www.tutorialspoint.com/data_structures_algorithms/strassens_matrix_multiplication_algorithm.htm



https://medium.com/swlh/strassens-matrix-multiplication-algorithm-936f42c2b344

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four **(n/2)×(n/2)** matrices as represented below –

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following −

$$M_1 := (A + C) \times (E + F)$$

$$M_2 := (B + D) \times (G + H)$$

$$M_3 := (A - D) \times (E + H)$$

$$M_4 := A \times (F - H)$$

$$M_5 := (C + D) \times (E)$$

$$M_6 := (A + B) \times (H)$$

$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$

$$J := M_4 + M_6$$

$$K := M_5 + M_7$$

$$L := M_1 - M_3 - M_4 - M_5$$

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and}$$

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix}$$

https://www.tutorialspoint.com/data_structures_algorithms/strassens_matrix_multiplication_algorithm.htm

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

https://www.geeksforgeeks.org/strassens-matrix-multiplication/

Example 2.4

Suppose we want the product C of two 2×2 matrices, A and B. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})\, b_{11} \\ m_3 &= a_{11}\,(b_{12} - b_{22}) \\ m_4 &= a_{22}\,(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})\, b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}), \end{aligned}$$

the product C is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

In the exercises, you will show that this is correct.

To multiply two 2×2 matrices, Strassen's method requires seven multiplications and 18 additions/subtractions, whereas the straightforward method requires eight multiplications and four additions/subtractions. We have saved ourselves one multiplication at the expense of doing 14 additional additions or subtractions. This is not very impressive, and indeed it is not in the case of 2×2 matrices that Strassen's method is of value. Because the commutativity of multiplications is not used in Strassen's formulas, those formulas pertain to larger matrices that are each divided into four submatrices. First we divide the matrices A and B, as illustrated in Figure 2.4. Assuming that

n is a power of 2, the matrix A_{11}, for example, is meant to represent the following submatrix of A:

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1,n/2} \\ a_{21} & a_{22} \cdots a_{2,n/2} \\ & \vdots \\ a_{n/2,1} & \cdots a_{n/2,n/2} \end{bmatrix}.$$

Using Strassen's method, first we compute

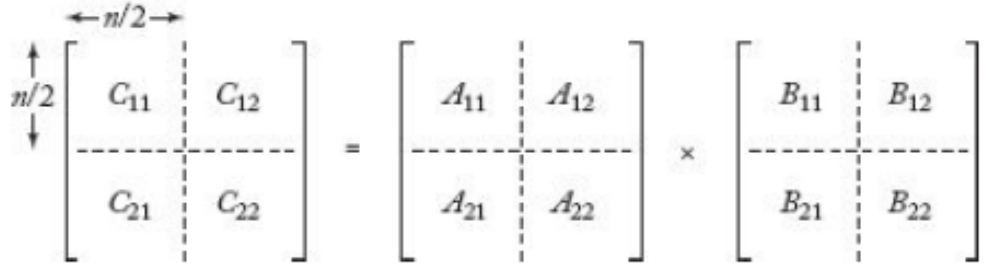
$$M_1 = (A_{11} + A_{22})\,(B_{11} + B_{22}),$$

where our operations are now matrix addition and multiplication. In the same way, we compute M_2 through M_7. Next we compute

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

and C_{12}, C_{21}, and C_{22}. Finally, the product C of A and B is obtained by combining the four submatrices C_{ij}. The following example illustrates these steps.

Figure 2.4 The partitioning into submatrices in Strassen's algorithm.



Example 2.5

Suppose that

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}.$$

Figure 2.5 illustrates the partitioning in Strassen's method. The computations proceed as follows:

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
&= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\
&= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}.
\end{aligned}
$$

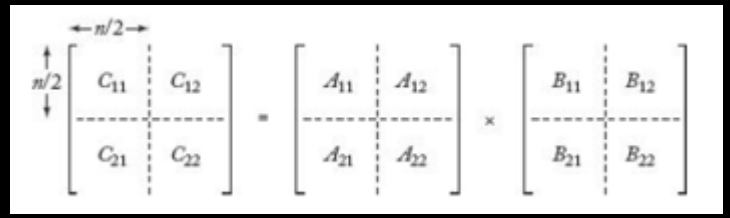
Figure 2.5 The partitioning in Strassen's algorithm with $n = 4$ and values given to the matrices.

$$
\begin{array}{c} \overset{\leftarrow 2 \rightarrow}{} \\ \updownarrow 2 \end{array}
\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right]
=
\left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right]
\times
\left[\begin{array}{cc|cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]
$$

When the matrices are sufficiently small, we multiply in the standard way. In this example, we do this when $n = 2$. Therefore,

$$
\begin{aligned}
M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\
&= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}.
\end{aligned}
$$

After this, M_2 through M_7 are computed in the same way, and then the values of C_{11}, C_{12}, C_{21}, and C_{22} are computed. They are combined to yield C.



Next we present an algorithm for Strassen's method when n is a power of 2.

Algorithm 2.8

Strassen

Problem: Determine the product of two $n \times n$ matrices where n is a power of 2.

Inputs: an integer n that is a power of 2, and two $n \times n$ matrices A and B.

Outputs: the product C of A and B.

```
void strassen (int n
                    n × n_matrix A,
                    n × n_matrix B,
                    n × n_matrix& C)
{
  if (n <= threshold)
     compute C = A × B using the standard algorithm;
  else{
     partition A into four submatrices A11, A12, A21, A22;
     partition B into four submatrices B11, B12, B21, B22;
     compute C = A × B using Strassen's method;
        // example recursive call:
        // strassen(n/2, A11 + A22, B11 + B22, M1);
  }
}
```

The value of *threshold* is the point at which we feel it is more efficient to use the standard algorithm than it would be to call procedure *strassen* recursively. In Section 2.7 we discuss a method for determining thresholds.

Analysis of Algorithm 2.8

Every-Case Time Complexity Analysis of Number of Additions/Subtractions (Strassen)

Basic operation: one elementary addition or subtraction.

Input size: n, the number of rows and columns in the matrices.

Again we assume that we keep dividing until we have two 1×1 matrices. When $n = 1$, no additions/subtractions are done. When we have two $n \times n$ matrices with $n > 1$, the algorithm is called exactly seven times with an $(n/2) \times (n/2)$ matrix passed in each time, and 18 matrix additions/subtractions are done on $(n/2) \times (n/2)$ matrices. When two $(n/2) \times (n/2)$ matrices are added or subtracted, $(n/2)^2$ additions or subtractions are done on the items in the matrices. We have established the recurrence

$$T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \qquad \text{for } n > 1,\ n \text{ a power of } 2$$
$$T(1) = 0.$$

This recurrence is solved in Example B.20 in Appendix B. The solution is

$$T(n) = 6n^{\lg 7} - 6n^2 \approx 6n^{2.81} - 6n^2 \in \Theta\left(n^{2.81}\right).$$

When n is not a power of 2, we must modify the previous algorithm. One simple modification is to add sufficient numbers of columns and rows of 0s to the original matrices to make the dimension a power of 2. Alternatively, in the recursive calls we could add just one extra row and one extra column of 0s whenever the number of rows and columns is odd. Strassen (1969) suggested the following, more complex modification. We embed the matrices in larger ones with $2^k m$ rows and columns, where $k = \lfloor \lg n - 4 \rfloor$ and $m = \lfloor n/2^k \rfloor + 1$. We use Strassen's method up to a ***threshold*** value of m and use the standard algorithm after reaching the threshold. It can be shown that the total number of arithmetic

operations (multiplications, additions, and subtractions) is less than $4.7n^{2.81}$.

Table 2.3 compares the time complexities of the standard algorithm and Strassen's algorithm for n a power of 2. If we ignore for the moment the overhead involved in the recursive calls, Strassen's algorithm is always more efficient in terms of multiplications, and for large values of n, Strassen's algorithm is more efficient in terms of additions/subtractions. In Section 2.7 we will discuss an analysis technique that accounts for the time taken by the recursive calls.

• **Table 2.3** A comparison of two algorithms that multiply $n \times n$ matrices

	Standard Algorithm	Strassen's Algorithm
Multiplications	n^3	$n^{2.81}$
Additions/Subtractions	$n^3 - n^2$	$6n^{2.81} - 6n^2$

Shmuel Winograd developed a variant of Strassen's algorithm that requires only 15 additions/subtractions. It appears in Brassard and Bratley (1988). For this algorithm, the time complexity of the additions/subtractions is given by

$$T(n) \approx 5n^{2.81} - 5n^2.$$

Coppersmith and Winograd (1987) developed a matrix multiplication algorithm whose time complexity for the number of multiplications is in $O(n^{2.38})$. However, the constant is so large that Strassen's algorithm is usually more efficient.

It is possible to prove that matrix multiplication requires an algorithm whose time complexity is at least quadratic. Whether matrix multiplications can be done in quadratic time remains an open question; no one has ever created a quadratic-time algorithm for matrix multiplication, and no one has proven that it is not possible to create such an algorithm.

One last point is that other matrix operations such as inverting a matrix and finding the determinant of a matrix are directly related to matrix multiplication. Therefore, we can readily create algorithms for these operations that are as efficient as Strassen's algorithm for matrix multiplication.

Reference

- **Strassens Matrix Multiplication**

https://www.youtube.com/watch?app=desktop&v=0oJyNmEbS4w

- Strassen's Matrix Multiplication Algorithm

https://www.knowprogram.com/algorithm/strassens-matrix-multiplication/

- **Strassen's Matrix Multiplication**

https://www.tutorialspoint.com/data_structures_algorithms/strassens_matrix_multiplication_algorithm.htm

Questions?