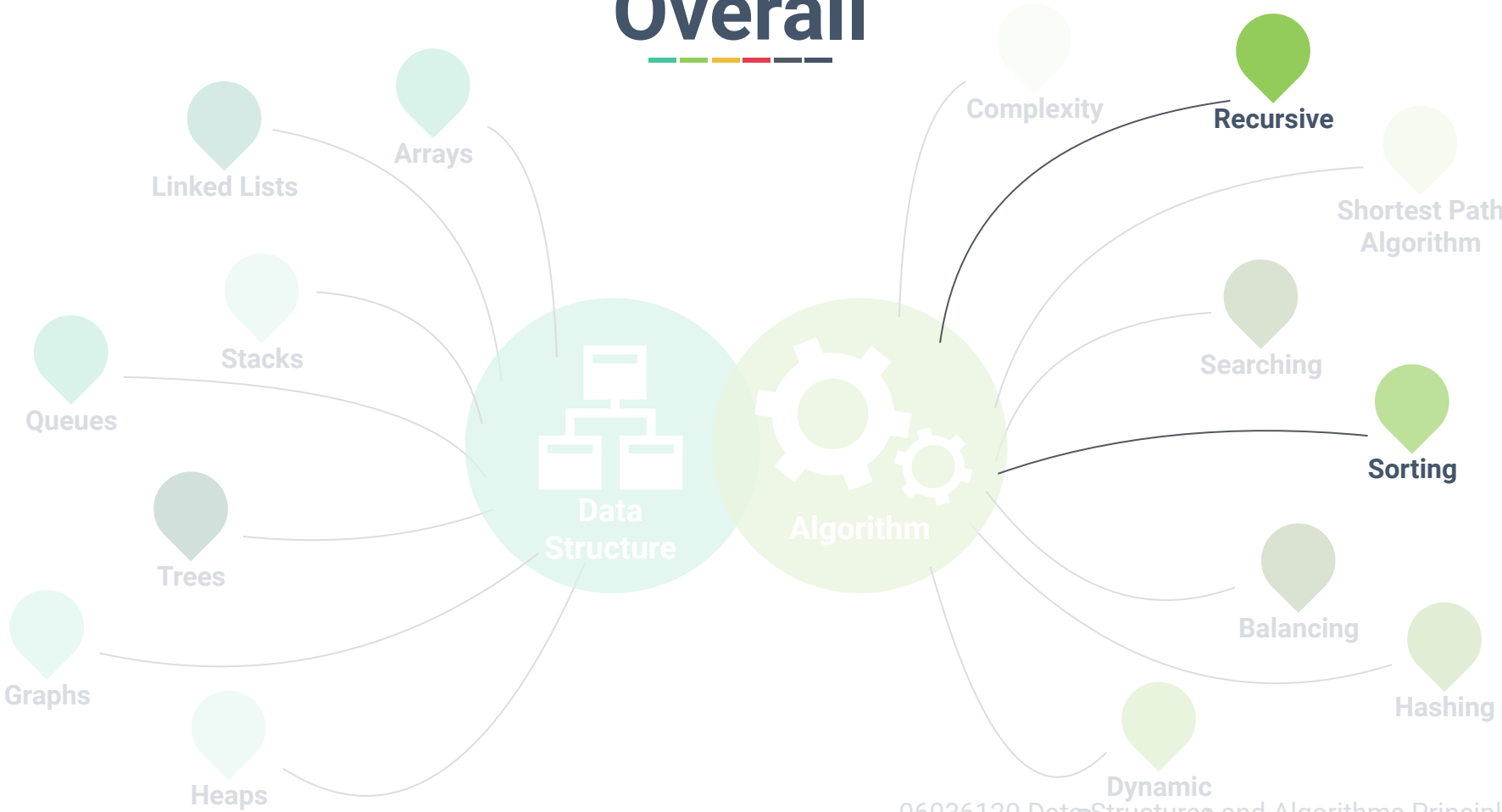
Chapter 10: Recursion & Sorting

Dr. Sirasit Lochanachit

Overall


Linked Lists
Arrays
Stacks
Queues
Trees
Graphs
Heaps
Data Structure
Algorithm
Complexity
Recursive
Shortest Path Algorithm
Searching
Sorting
Balancing
Hashing
Dynamic Programming

Outline

Recursion

- Components of Recursion
- Summation
- Exponentiation
- Factorial

Sorting (Cont.)

- Merge Sort
- Quick Sort

What is Recursion?

- It breaks down a large problem into smaller subproblems that are simpler to solve[1].
- A function that is calling itself one or more times[2].

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011
[2] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

What is Recursion?

Ref: https://www.smbc-comics.com/comic/recursion

Recursion

Real-life example:

- Fractal patterns in art and nature
- **Matryoshka** doll
  - Also known as Russian doll or nested doll
  - Set of wooden dolls where smaller dolls are placed inside another.
  - A symbol of motherhood and fertility.

Image Source: https://images.theconversation.com/files/163337/original/image-20170330-4592-1n4ji0f.jpg, https://cdn.pixabay.com/photo/2017/06/13/13/45/matryoshka-2398918_960_720.jpg, https://tf-cmsv2-smithsonianmag-media.s3.amazonaws.com/filer/1d/ff/1dffc703-583f-43d9-a645-96870f5a1d7b/ice-queen.jpg

When to use recursion?

- For problems that contain smaller instances of the same problem.
- To repeat a computer program, **while-loop** and **for-loop** can be used.
- Alternatively, recursion repeats by calling a function itself one or more times.

[2] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Image Source: https://cdn.pixabay.com/photo/2017/06/13/13/45/matryoshka-2398918_960_720.jpg

Components of Recursion

- Base Case: The smallest subproblem that can be solved easily.

- Recursive Case: A subproblem that reduces the size of the input and move toward the base case

Summation

- Summing list elements recursively.

Summation

- Summing list elements recursively.

Power

Power function takes 2 numbers as input and return the value of *base* to the power of *exp*.

$$\text{power}(base, exp) = \begin{cases} 1 & \text{if } exp = 0 \\ base * \text{power}(base, exp-1) & \text{otherwise} \end{cases}$$

```
def getPower(base,exp):
  """ Return the multiplciation of  base with exp times. """
  if exp==0:
    return 1
  else:
    return base*getPower(base, exp-1)
```

Factorial

Factorial function takes a number as an input and returns the factorial of that number

- n! = Product of positive integers from 1 to n
- If n = 0, then n! = 1 by convention

factorial(1)

factorial(2)

factorial(3)

factorial(4)

factorial(5)

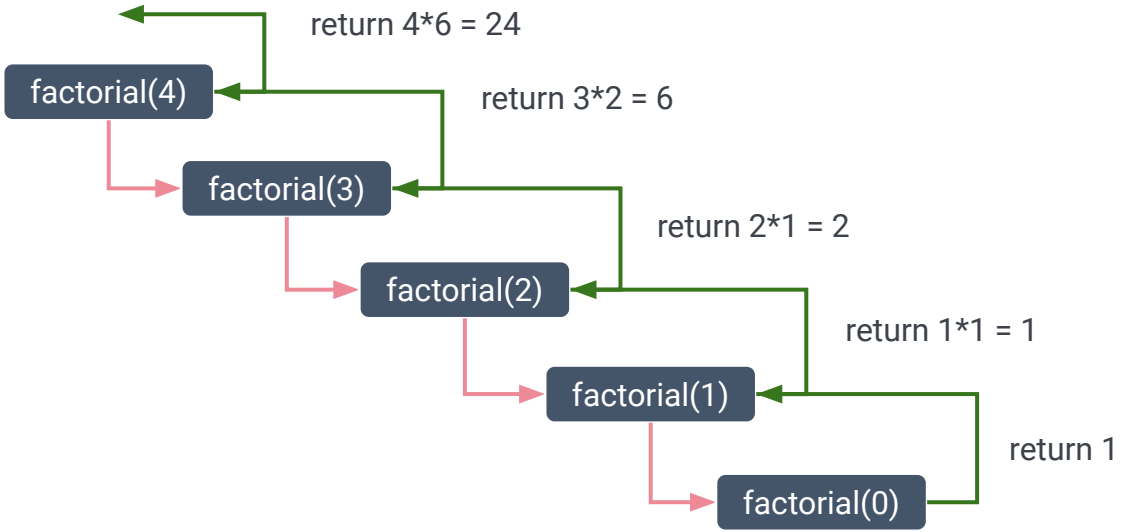
Factorial

Factorial function takes a number as an input and returns the factorial of that number

- n! = Product of positive integers from 1 to n
- If n = 0, then n! = 1 by convention

Recursion

- For recursive, repetition is conducted by repeatedly invoke the function call.



Recap

- Components:
  - Base case and Recursive case
- Examples
  - Summation
  - Power
  - Factorial
  - Binary search
- One rule for recursion: there must be a stop condition.

Image Source: https://cdn.pixabay.com/photo/2017/06/13/13/45/matryoshka-2398918_960_720.jpg

Divide and Conquer Strategy

3 Steps[2]:

1. **Divide**: Divide the input data into two or more disjoint subsets.
2. **Recur**: Recursively solve the subproblems associated with the subsets.
3. **Conquer**: Take the solutions to the subproblems and “merge” them into a solution to the original problem.

[2] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Merge Sort

Merge sort is a recursive sorting technique based on divide and conquer technique by continually splits a list into equal halves[1].

Base case: If the list is empty or has only one item, it is sorted.

Recursive case: If the list has > 1 item, split the list and recursively merge sort on both halves.

After the two halves are sorted, then a merge is invoked, combining them together into a single sorted new list.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

Merge Sort Real-life Demo

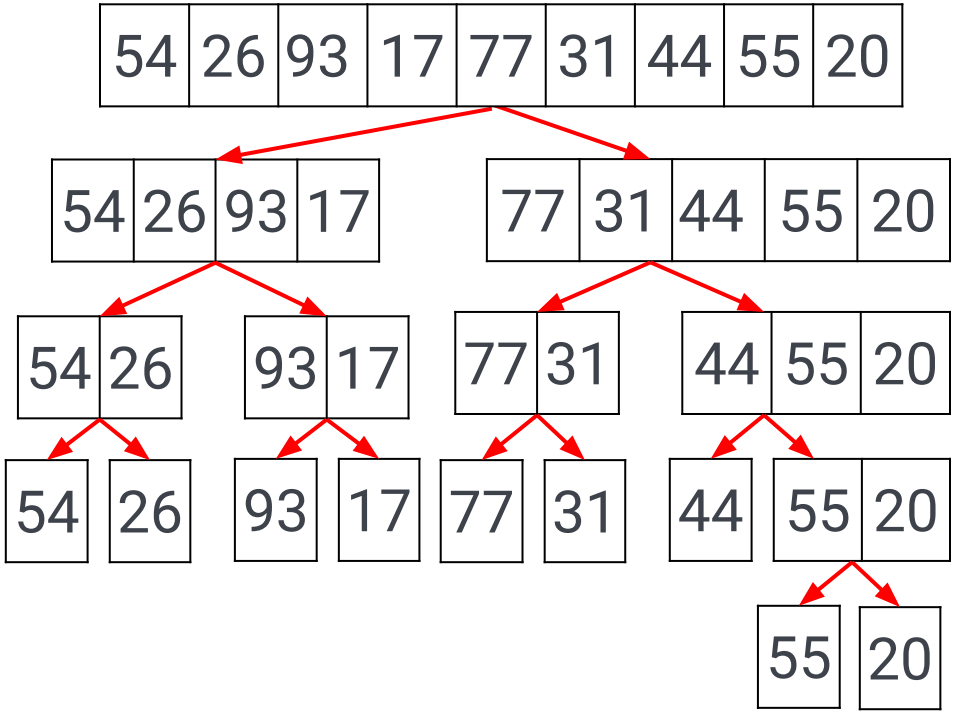
Merge-sort with Transylvanian-saxon (German) folk dance

Merge Sort Example

Merge Sort Example

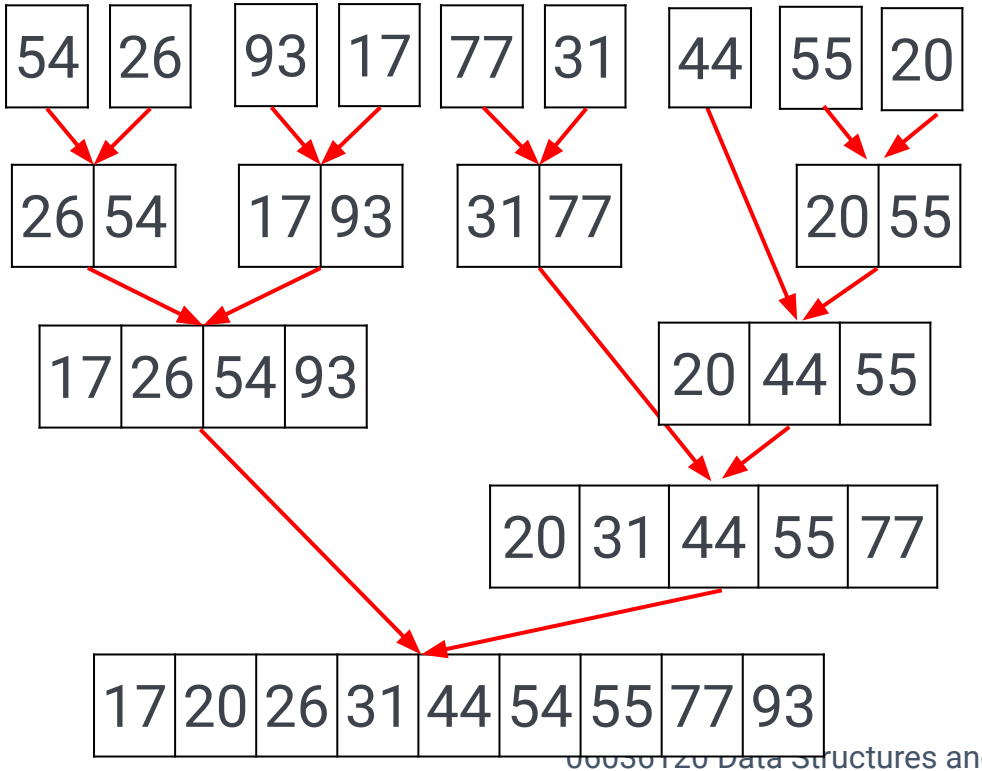
Merge Sort Example

Even elements



Merge Sort Example

Even elements



Merge Sort Performance

2 Processes[1]:

1. Splitting the list into halves
2. Merge operation and sorting

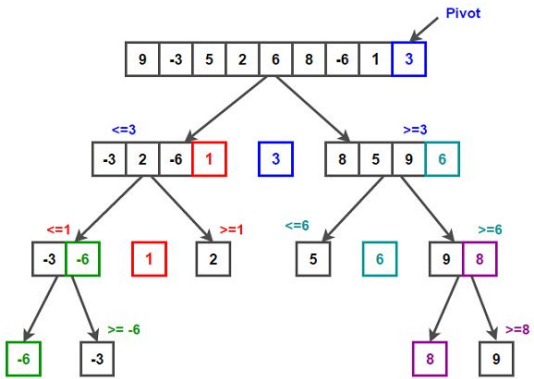
Merge sort requires extra memory space to hold two halves as they are continually splitting.

- This can be a critical factor if the list is large.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

```
Algorithm mergeSort(list):
    if length(list) ≤ 1:
        return list
    else:
        middle = length(list) / 2
        divide list into two unsorted sublists, left and right, using middle
        left = mergeSort(left)
        right = mergeSort(right)
        result = merge(left, right)
        return result
    end if
End mergeSort
```

Quick Sort



Quick Sort is one of the most efficient sorting algorithms and is based on the splitting of a list into smaller ones without using additional storage[1].

However, it is possible that the list may not be divided in half.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011
Ref: https://www.interviewbit.com/tutorial/quicksort-algorithm/

Quick Sort

Example: Sort the papers containing the names of the students, by name from A-Z.

1. Select any splitting character such as ‘L’.
    - The splitting value is also known as **pivot value**.
2. Divide the stack of papers into two. A-L and M-Z.
    - It is not necessary that the piles should be equal.
3. Repeat the above two steps with the A-L pile, splitting it into its significant two halves. And M-Z pile, split into its halves. The process is repeated until the piles are small enough to be sorted easily.
4. Ultimately, the smaller piles can be placed one on top of the other to produce a fully sorted and ordered set of papers.

Quick Sort

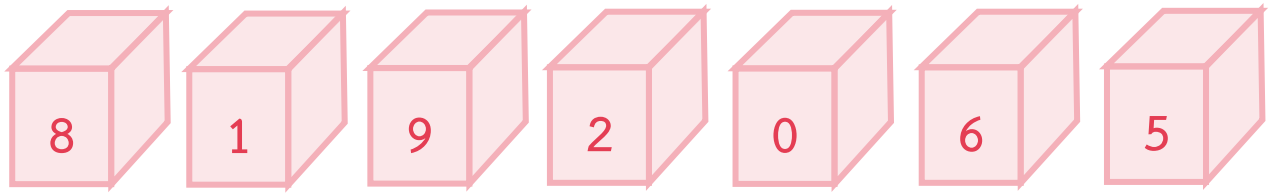
Technically, quick sort follows the below steps:

1. Make any element as **pivot**.
2. Partition the list on the basis of pivot.
3. Apply quick sort on left partition recursively.
4. Apply quick sort on right partition recursively.

Choosing a Pivot

To determine the pivot element, there are 3 general approaches[3]:

1) Choosing the first or last element of the unsorted list.
2) Random method.
3) Median-of-three method. It considers three elements including the first, last, and middle element in the list.



[3] Robert Sedgewick, Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1–4, 1998

Quick Sort Real-life Demo

Quick-sort with Hungarian (Kükullőmenti legényes) folk dance

Quick Sort Example

Example: Sort this list with divide and conquer strategy without using extra space.

50	23	9	18	61	32

Step 1: Make any element as **pivot**.

- For convenience, the rightmost index is selected as pivot.
- 'Low' and 'High' pointers corresponds to the first index and last index respectively.
- In this example, the low is at index 0 and high is at index 5.
- The value at pivot is 32.

Quick Sort Example

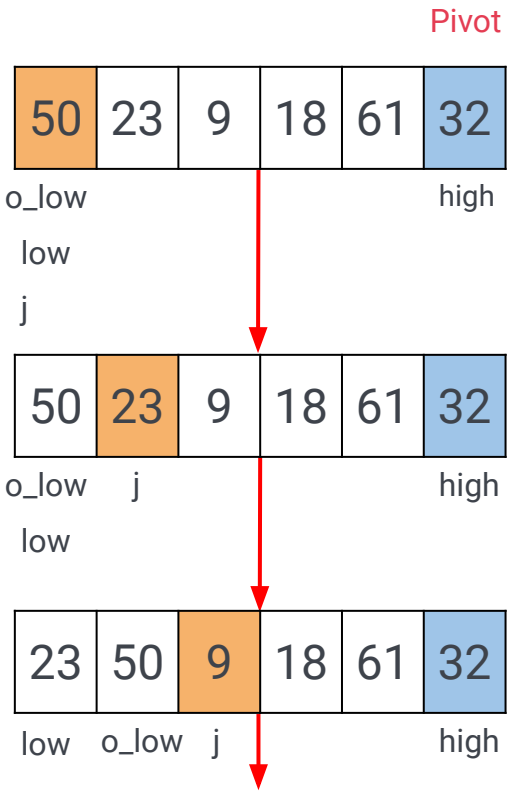
Example: Sort this list with divide and conquer strategy without using extra space.

50	23	9	18	61	32

Step 2: Partition the list on the basis of pivot.

- Rearranges the list in such a way that pivot(32) is at its proper position.
  - To the left of the pivot, the value of all elements is less than or equal to it.
  - To the right of the pivot, the value of all elements is higher than it.

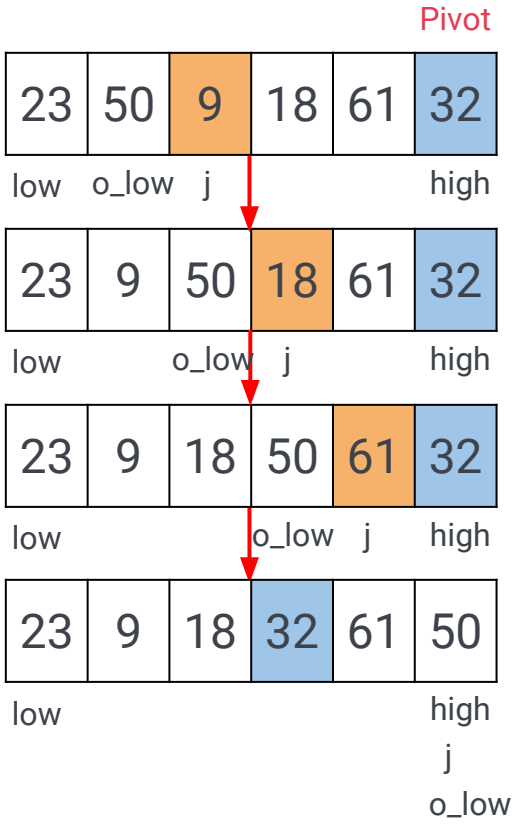
Quick Sort Example




```
original_low = low
pivot = alist[high]

for j in range(low, high):
        if alist[j] <= pivot:
                swap(alist[original_low], alist[j])
                original_low += 1
        else:
                continue
```

Quick Sort Example




```
original_low = low
pivot = alist[high]


for j in range(low, high):
        if alist[j] <= pivot:
                swap(alist[original_low], alist[j])
                original_low += 1
        else:
                continue
swap(alist[original_low], alist[high])
```

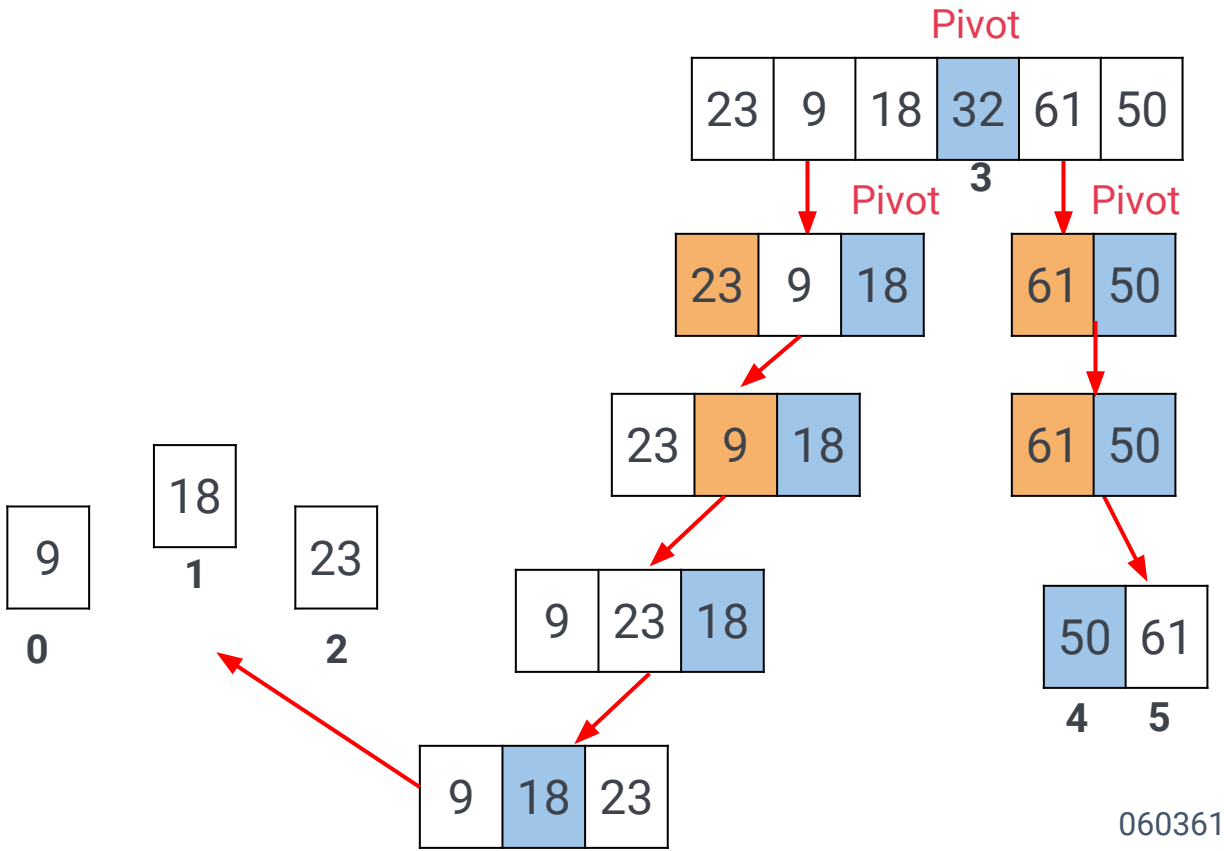
Quick Sort Example

50	23	9	18	61	32

Step 3: Divide the list into two parts - left and right sublists.

Step 4: Repeat the steps for the left and right sublists recursively.

Quick Sort Example



Quick Sort Performance

No need for additional memory as in the merge sort process.

If partition occurs in the middle of the list: log *n* times to split[1]

Each item checked against the pivot value: *n* times[1]

However, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division.

For example, sorting a list of *n* items divides into sorting a list of 0 items and a list of n−1 items.

[1] Bradley N. Miller, David L. Ranum, Problem Solving with Algorithms and Data Structures using Python, 2011

Comparing Sorting Algorithms

Sorting Algorithms Animations