Bubble Sort

Bubble sort is a simple sorting algorithm. It repeatedly compares adjacent elements and swaps them if they are in the wrong order. This algorithm is easy to implement but inefficient for large lists.

Pseudocode:

BUBBLE-SORT(A)

for i from 0 to n-1 for j from 0 to n-2 if A[j] > A[j+1] swap A[j] and A[j+1] return A

In [1]:
```
def bubble_sort(a):
    n=len(a)
    for i in range(n):
        for j in range(n-1-i):
            if a[j]>a[j+1]:
                temp=a[j];a[j]=a[j+1];a[j+1]=temp
    return a
```

In [2]:
```
arr = [5,3,1,4,2]

result = bubble_sort(arr)

print(result)
```

```
[1, 2, 3, 4, 5]
```

In [3]:
```
arr = [9,7,5,3,1]

print(bubble_sort(arr))
```

```
[1, 3, 5, 7, 9]
```

Complexity Analysis

Time Complexity: Worst case: O(n^2) Best case: O(n^2) Space Complexity: O(1)

Quick Sort

Quick sort is a divide and conquer sorting algorithm. It selects a pivot element and partitions the array into two parts: elements smaller than the pivot and elements greater than the pivot. Then it recursively sorts the subarrays.

Pseudocode:

QUICK-SORT(A)

if length(A) <= 1 return A

pivot = A[0] create empty list LEFT create empty list RIGHT

for each element x in A[1:] if x <= pivot add x to LEFT else add x to RIGHT

return QUICK-SORT(LEFT) + pivot + QUICK-SORT(RIGHT)

In [4]:
```python
import random

def quick_sort_random(a):
    if len(a)<=1:
        return a
    pivot=random.choice(a)
    left=[]
    equal=[]
    right=[]
    for x in a:
        if x<pivot:
            left.append(x)
        elif x>pivot:
            right.append(x)
        else:
            equal.append(x)
    return quick_sort_random(left)+equal+quick_sort_random(right)


def quick_sort_middle(a):
    if len(a)<=1:
        return a
    first=a[0]
    middle=a[len(a)//2]
    last=a[-1]
    pivot=sorted([first,middle,last])[1]
    left=[]
    equal=[]
    right=[]
    for x in a:
        if x<pivot:
            left.append(x)
        elif x>pivot:
            right.append(x)
        else:
            equal.append(x)
    return quick_sort_middle(left)+equal+quick_sort_middle(right)
```

In [5]:
```python
arr = [8,3,1,7,0,10,2]

print(quick_sort_random(arr))
print(quick_sort_middle(arr))
```

```
[0, 1, 2, 3, 7, 8, 10]
[0, 1, 2, 3, 7, 8, 10]
```

In [6]:
```python
arr = [5,4,3,2,1]
```

```
print(quick_sort_random(arr))
print(quick_sort_middle(arr))
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

Complexity Analysis

Time Complexity: Best case: O(n log n) Average case: O(n log n) Worst case: O(n^2) Space Complexity: O(n)

Merge Sort

Merge sort is a divide and conquer sorting algorithm. It splits the list into two halves, sorts each half, and merges them.

Pseudocode:

MERGE-SORT(A)

if length(A) <= 1 return A

split A into LEFT and RIGHT LEFT = MERGE-SORT(LEFT) RIGHT = MERGE-SORT(RIGHT)

return MERGE(LEFT, RIGHT)

In [7]:
```
def merge_sort(a):
    if len(a)<=1:return a
    mid=len(a)//2
    left=merge_sort(a[:mid])
    right=merge_sort(a[mid:])
    result=[]
    while left and right:
        if left[0]<=right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result+=left
    result+=right
    return result
```

In [8]:
```
print(merge_sort([5,3,1,4,2]))
print(merge_sort([9,7,5,3,1]))
```

```
[1, 2, 3, 4, 5]
[1, 3, 5, 7, 9]
```

Complexity Analysis

Time Complexity: T(n) = 2T(n/2) + Θ(n) By Master Theorem (Case 2): T(n) = Θ(n log n) Space Complexity: O(n)

Heap Sort

Heap sort builds a max-heap and repeatedly swaps the largest element with the last element. After each swap, it restores the heap property.

Pseudocode:

HEAP-SORT(A) BUILD-MAX-HEAP(A) for end from n-1 down to 1 swap A[0] and A[end] SIFT-DOWN(A,0,end-1) BUILD-MAX-HEAP(A)

for i from parent of last node down to 0 SIFT-DOWN(A,i,n-1) SIFT-DOWN(A,start,end) root=start

while root has left child within range child=left child if right child exists and is larger child=right child if A[child]>A[root] swap A[root] and A[child] root=child else break

In [11]:
```
def heap_sort(a):
    n=len(a)
    def sift_down(start,end):
        root=start
        while True:
            child=2*root+1
            if child>end:
                break
            if child+1<=end and a[child]<a[child+1]:
                child=child+1
            if a[root]<a[child]:
                temp=a[root]
                a[root]=a[child]
                a[child]=temp
                root=child
            else:
                break
    start=(n-2)//2
    while start>=0:
        sift_down(start,n-1)
        start-=1
    end=n-1
    while end>0:
        temp=a[0]
        a[0]=a[end]
        a[end]=temp
        sift_down(0,end-1)
        end-=1
    return a
```

In [12]:
```
print(heap_sort([5,3,1,4,2]))
print(heap_sort([9,7,5,3,1]))
```

```
[1, 2, 3, 4, 5]
[1, 3, 5, 7, 9]
```

Complexity Analysis

Time Complexity: O(n log n) Space Complexity: O(1)

Sorting Algorithms Summary

Bubble Sort Time: O(n^2) Space: O(1)

Quick Sort (Random / Median-of-three) Best: O(n log n) Average: O(n log n) Worst: O(n^2) Space: O(n)

Merge Sort Time: O(n log n) Space: O(n)

Heap Sort Time: O(n log n) Space: O(1)

In [13]:

```
import random
def test_sort(fn,name):
    tests=[[5,3,1,4,2],[9,7,5,3,1],[3,1,2,1,3,0]]
    for n in [10,50]:
        random.seed(n)
        tests.append([random.randint(-100,100) for _ in range(n)])
    for a in tests:
        if fn(a[:])!=sorted(a):
            print(name,"FAIL")
            return
    print(name,"OK")
```

In [14]:

```
test_sort(bubble_sort,"Bubble")
test_sort(quick_sort_random,"Quick random")
test_sort(quick_sort_middle,"Quick middle")
test_sort(merge_sort,"Merge")
test_sort(heap_sort,"Heap")
```

```
Bubble OK
Quick random OK
Quick middle OK
Merge OK
Heap OK
```

In [ ]: