Karlo Robert C. Wagan BSCS - 3 ITP 6 Assignment 3 Divide and Conquer

Divide-and-Conquer Algorithm for Finding the Largest Element

```
FIND-MAX-POSITION(A, low, high)
    if low == high
        return low

    mid = ⌊(low + high)/2⌋

    leftMaxPos = FIND-MAX-POSITION(A, low, mid)

    rightMaxPos = FIND-MAX-POSITION(A, mid + 1, high)

    if A[leftMaxPos] >= A[rightMaxPos]
        return leftMaxPos
    else
        return rightMaxPos

MAX-POSITION(A)
    return FIND-MAX-POSITION(A, 0, length(A) - 1)'
```

In [9]:

```python
#Example 1:
def find_max_position(arr, low, high):
    if low == high:
        return low

    mid = (low + high) // 2

    left_max_pos = find_max_position(arr, low, mid)

    right_max_pos = find_max_position(arr, mid + 1, high)

    if arr[left_max_pos] >= arr[right_max_pos]:
        return left_max_pos
    else:
        return right_max_pos

def max_position(arr):
    if not arr:
        return None
    return find_max_position(arr, 0, len(arr) - 1)

def main():
    arr = [3, 4, 5, 1, 2]
    print(max_position(arr))

    arr = [1, 2, 3, 4, 5]
    print(max_position(arr))
```

```python
    arr = [5, 4, 3, 2, 1]
    print(max_position(arr))

    arr = [1]
    print(max_position(arr))

    arr = []
    print(max_position(arr))

if __name__ == '__main__':
    main()
```

```
2
4
0
0
None
```

Divide-and-Conquer Algorithm for Finding Both Largest and Smallest Elements

```
FIND-MIN-MAX(A, low, high)
    if low == high
        return (A[low], A[low])

    if high == low + 1
        if A[low] < A[high]
            return (A[low], A[high])
        else
            return (A[high], A[low])

    mid = ⌊(low + high)/2⌋

    (leftMin, leftMax) = FIND-MIN-MAX(A, low, mid)

    (rightMin, rightMax) = FIND-MIN-MAX(A, mid + 1, high)

    if leftMin < rightMin
        min = leftMin
    else
        min = rightMin

    if leftMax > rightMax
        max = leftMax
    else
        max = rightMax

    return (min, max)

MIN-MAX(A)
    return FIND-MIN-MAX(A, 0, length(A) - 1)
```

In [10]:
```python
#Example 2:
def find_min_max(arr, low, high):
    if low == high:
        return (arr[low], arr[low])

    if high == low + 1:
        if arr[low] < arr[high]:
            return (arr[low], arr[high])
        else:
            return (arr[high], arr[low])

    mid = (low + high) // 2

    left_min, left_max = find_min_max(arr, low, mid)

    right_min, right_max = find_min_max(arr, mid + 1, high)

    min_val = left_min if left_min < right_min else right_min
    max_val = left_max if left_max > right_max else right_max

    return (min_val, max_val)

def min_max(arr):
    if not arr:
        return (None, None)
    return find_min_max(arr, 0, len(arr) - 1)

def main():
    arr = [3, 4, 5, 1, 2]
    print(min_max(arr))

    arr = [1, 2, 3, 4, 5]
    print(min_max(arr))

    arr = [5, 4, 3, 2, 1]
    print(min_max(arr))

    arr = [1]
    print(min_max(arr))

    arr = []
    print(min_max(arr))

if __name__ == '__main__':
    main()
```

```
(1, 5)
(1, 5)
(1, 5)
(1, 1)
(None, None)
```

Mergesort Application on E, X, A, M, P, L, E to sort the list in alphabetical order

```
Step 1: Divide the array until subarrays of size 1

[E, X, A, M, P, L, E]
/               \
[E, X, A, M]    [P, L, E]
/        \      /     \
[E, X]   [A, M] [P]   [L, E]
/  \     /  \         /  \
[E] [X] [A] [M]      [L] [E]

Step 2: Merge the subarrays in sorted order

Merge [E] and [X]:
    - Compare E and X: E < X
    - Result: [E, X]

Merge [A] and [M]:
    - Compare A and M: A < M
    - Result: [A, M]

Merge [L] and [E]:
    - Compare L and E: E < L
    - Result: [E, L]

Merge [E, X] and [A, M]:
    - Compare E and A: A < E
    - Compare E and M: E < M
    - Compare X and M: M < X
    - Result: [A, E, M, X]

Merge [P] and [E, L]:
    - Compare P and E: E < P
    - Compare P and L: L < P
    - Result: [E, L, P]

Merge [A, E, M, X] and [E, L, P]:
    - Compare A and E: A < E
    - Compare E and E: Equal, take both: [A, E, E]
    - Compare M and L: L < M
    - Compare M and P: M < P
    - Compare X and P: P < X
    - Result: [A, E, E, L, M, P, X]

Final sorted array: A, E, E, L, M, P, X
```

In [11]:
```python
#Example 3:
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

example_list = ['E', 'X', 'A', 'M', 'P', 'L', 'E']
sorted_list = merge_sort(example_list)
print(f"Sorted list: {sorted_list}")
```

```
Sorted list: ['A', 'E', 'E', 'L', 'M', 'P', 'X']
```

Quicksort Application on E, X, A, M, P, L, E

```
Initial array: E, X, A, M, P, L, E

Simulate quicksort using the first element as the pivot:

First recursive call:
Pivot = E
    - Elements less than E: A, E
    - Elements greater than E: X, M, P, L
    - After partition: [A, E, E, X, M, P, L]

Recursively sort [A, E] and [X, M, P, L]:

Left subarray [A, E]:
Pivot = A
    - Elements less than A: none
    - Elements greater than A: E
    - After partition: [A, E]
```

```
Right subarray [X, M, P, L]:
Pivot = X
    - Elements less than X: M, P, L
    - Elements greater than X: none
    - After partition: [M, P, L, X]

Recursively sort [M, P, L]:

Subarray [M, P, L]:
Pivot = M
    - Elements less than M: L
    - Elements greater than M: P
    - After partition: [L, M, P]

Recursively sort [L] and [P]:
Both are single elements, already sorted.

The tree of recursive calls:
        [E,X,A,M,P,L,E]
         /          \
    [A,E,E]         [X,M,P,L]
     /   \            /    \
  [A]   [E]     [M,P,L]    [X]
                /    \
              [L]    [M,P]
                     /   \
                   [M]   [P]

Final sorted array: A, E, E, L, M, P, X
```

In [12]:

```python
#Example 4:
def quick_sort(arr, low=0, high=None, depth=0, call_tree=None):
    if high is None:
        high = len(arr) - 1
        call_tree = []

    if low < high:
        call_tree.append((depth, arr[low:high+1]))

        pivot_idx = partition(arr, low, high)

        quick_sort(arr, low, pivot_idx - 1, depth + 1, call_tree)
        quick_sort(arr, pivot_idx + 1, high, depth + 1, call_tree)

    return arr, call_tree

def partition(arr, low, high):
    pivot = arr[low]
    i = low + 1

    for j in range(low + 1, high + 1):
        if arr[j] < pivot:
```

```
            arr[i], arr[j] = arr[j], arr[i]
            i += 1

    arr[low], arr[i-1] = arr[i-1], arr[low]
    return i - 1

example_list = ['E', 'X', 'A', 'M', 'P', 'L', 'E']
sorted_list, call_tree = quick_sort(example_list.copy())

print(f"Sorted list: {sorted_list}")
print("Quick Sort Recursive Call Tree:")
for depth, subarray in call_tree:
    print(f"{'  ' * depth}Call: {subarray}")
```

```
Sorted list: ['A', 'E', 'E', 'L', 'M', 'P', 'X']
Quick Sort Recursive Call Tree:
Call: ['E', 'X', 'A', 'M', 'P', 'L', 'E']
  Call: ['X', 'M', 'P', 'L', 'E']
    Call: ['E', 'M', 'P', 'L']
      Call: ['M', 'P', 'L']
```

Algorithm to Rearrange Negative Elements Before Positive

```
REARRANGE-NEGATIVE-POSITIVE(A)
    left = 0
    right = length(A) - 1

    while left < right
        while left < right and A[left] < 0
            left = left + 1

        while left < right and A[right] >= 0
            right = right - 1

        if left < right
            swap A[left] with A[right]
            left = left + 1
            right = right - 1

    return A
```

This algorithm for rearranging negative elements before positive ones uses a two-pointer approach:

1. Initialize two pointers, one at the beginning (left) and one at the end (right) of the array.
2. Move the left pointer until it finds a positive element.
3. Move the right pointer until it finds a negative element.
4. Swap these elements and continue the process until the pointers meet.

Time complexity: O(n) - each element are examined at most twice Space complexity: O(1) - only use a constant amount of extra space for the pointers

This approach is optimal in both time and space efficiency as it achieves the task in a single pass through the array without using any extra storage beyond a few variables.

In [13]:
```
#Example 5:
def rearrange_negative_positive(arr):
    left = 0
    right = len(arr) - 1

    while left < right:
        while left < right and arr[left] < 0:
            left += 1

        while left < right and arr[right] >= 0:
            right -= 1

        if left < right:
            arr[left], arr[right] = arr[right], arr[left]
            left += 1
            right -= 1

    return arr

test_array = [9, -3, 5, -2, -8, 6, 1, 3, -4, 7]
result = rearrange_negative_positive(test_array)
print(f"Rearranged array: {result}")
```

```
Rearranged array: [-4, -3, -8, -2, 5, 6, 1, 3, 9, 7]
```