Designing Algorithms using Divide and Conquer

Problem:

Given an array of integers `arr[]`, find all **peak elements** (elements greater than their neighbors) and determine the **maximum peak element** using a **divide-and-conquer approach**.

- A **peak** is defined as:
    - `arr[i] > arr[i-1]` and `arr[i] > arr[i+1]` for middle elements
    - `arr[0] > arr[1]` for the first element
    - `arr[n-1] > arr[n-2]` for the last element

Input:

`arr = {10, 8, 5, 12, 7}`

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

bool isPeak(const vector<int>& arr, int i) {
    int n = arr.size();
    if (i == 0) return arr[i] > arr[i + 1];
    if (i == n - 1) return arr[i] > arr[i - 1];
    return arr[i] > arr[i - 1] && arr[i] > arr[i + 1];
}

void findPeaks(const vector<int>& arr, int low, int high, int &maxPeak) {
    if (low > high) return;
```

```
    int mid = (low + high) / 2;

    if (isPeak(arr, mid)) {
        cout << "Peak found: " << arr[mid] << endl;
        if (arr[mid] > maxPeak) maxPeak = arr[mid];
    }

    findPeaks(arr, low, mid - 1, maxPeak);
    findPeaks(arr, mid + 1, high, maxPeak);
}

int main() {
    vector<int> arr = {10, 8, 5, 12, 7};
    int maxPeak = INT_MIN;

    findPeaks(arr, 0, arr.size() - 1, maxPeak);

    cout << "Maximum peak element = " << maxPeak << endl;
    return 0;
}
```

Dry-Run:

For `arr = {10, 8, 5, 12, 7}`:

- Peaks:
    - 10 (index 0)
    - 8 → not a peak
    - 5 → not a peak
    - 12 (index 3)
    - 7 → not a peak

Explanation of Steps:

1. Start with `mid = (0+4)/2 = 2` → `arr[2] = 5` → not a peak.

2. Recurse left: `low=0, high=1`
   - `mid = 0`, `arr[0]=10` → peak → update `maxPeak=10`
   - Recurse left and right → `mid=1`, `arr[1]=8` → not a peak
3. Recurse right: `low=3, high=4`
   - `mid=3`, `arr[3]=12` → peak → update `maxPeak=12`
   - Recurse left/right → `mid=4`, `arr[4]=7` → not a peak

Final Maximum Peak: 12

- **Output:**

```
Peak found: 10
Peak found: 12
Maximum peak element = 12
```

Recurrence Relation

The function `findPeaks(arr, low, high)` checks the **middle element**, then recurses on **both halves**:

1. Let `T(n)` be the **time complexity** for an array of size `n`.
2. At each step:
   - 1 comparison for `isPeak(arr, mid)` → $O(1)$
   - Recurse left half → `T(n/2)`
   - Recurse right half → `T(n/2)`

So the **recurrence relation** is:

$$T(n)=2T(n/2)+O(1)$$

This is a **classic divide-and-conquer recurrence**, and by the **Master Theorem**, it solves to:

$$T(n)=O(n)$$

> ✅ Even though it's divide-and-conquer, it still checks all elements eventually, so overall complexity is linear.

Problem:

Input: Sorted array `arr = [17, 18, 20, 25, 30]`, `k = 2`, `t = 16`

Output: `[17, 18]`

Goal: Return `k` elements closest to `t`. If `t` is smaller than all elements, return first `k`. If `t` is greater than all elements, return last `k`. Maintain the original array order. Time complexity should be `O(log n + k)`.

Algorithm

Step 1: Find the position to insert `t`

- Use **binary search** to find `idx` such that `arr[idx-1] < t <= arr[idx]`.
- Time: `O(log n)`

Step 2: Use two pointers to find `k` closest elements

- Initialize:
  - `left = idx - 1`
  - `right = idx`
- Compare `|arr[left] - t|` and `|arr[right] - t|`
- Pick the closer one and move pointer
- Repeat until we pick `k` elements
- Time: `O(k)`

Step 3: Return `k` elements in sorted order

Overall complexity: `O(log n + k)`

```
def findKClosest(arr, k, t):
    n = len(arr)

    # Binary search to find insertion index
```

```
    low, high = 0, n - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == t:
            low = mid
            break
        elif arr[mid] < t:
            low = mid + 1
        else:
            high = mid - 1

    # Two pointers around insertion point
    left, right = low - 1, low
    result = []

    while len(result) < k:
        if left < 0:
            result.append(arr[right])
            right += 1
        elif right >= n:
            result.append(arr[left])
            left -= 1
        else:
            if abs(arr[left] - t) <= abs(arr[right] - t):
                result.append(arr[left])
                left -= 1
            else:
                result.append(arr[right])
                right += 1

    result.sort()
    return result

# Test
arr = [17, 18, 20, 25, 30]
k = 2
```

```
t = 16
print(findKClosest(arr, k, t))  # Output: [17, 18]
```

Dry Run

Input: `arr = [17, 18, 20, 25, 30]`, `k=2`, `t=16`

1. **Binary search:**
   - `low=0`, `high=4`
   - `mid=2` → `arr[2]=20` → 20 > 16 → high = 1
   - `mid=0` → `arr[0]=17` → 17 > 16 → high = -1
   - **Insertion index = low = 0**
2. **Two pointers:**
   - `left = -1`, `right = 0`, `result=[]`
   - `left<0`, pick `arr[right]=17` → `result=[17]`, `right=1`
   - `left<0`, pick `arr[right]=18` → `result=[17,18]`, `right=2`
3. **Sort result** (already sorted) → `[17, 18]`

✅ Output: `[17, 18]`

Recurrence Relation

Binary Search:

$$T_{binary}(n)=T(n/2)+O(1)$$

Solves to:

$$T_{binary}(n)=O(\log n)$$

Two-pointer selection:

$$T_{select}(k)=O(k)$$

Overall complexity:

$$T(n,k)=O(\log n+k)$$

Problem:

Suppose you are given an array A with n entries, with each entry holding a distinct number.The sequence of values `A[1], A[2], ..., A[n]` is **unimodal**:

- There exists an index **p** ($1 \le p \le n$) such that
  - `A[1] < A[2] < ... < A[p]` (strictly increasing up to p)
  - `A[p] > A[p+1] > ... > A[n]` (strictly decreasing afterwards).

Find the index **p** (the "peak entry") **while reading only O(log n) array entries.**

Intuition/Logic:

The array increases and then decreases exactly once.

That means:

- If we look at a middle element `mid`:
  - If `A[mid] < A[mid+1]`, the peak is **to the right**.
  - If `A[mid] > A[mid+1]`, the peak is **to the left or at mid**.

This is the same idea as **Binary Search**:

- Each comparison lets us **discard half** of the remaining array.

Step-by-Step Algorithm

Input: array `A` of size `n`

Output: index `p` such that `A[p]` is the peak

1. **Initialize**:

   `low = 0`, `high = n - 1` (using 0-based indexing for code)

2. **Loop** while `low < high`:
   - `mid = (low + high) / 2`
   - If `A[mid] < A[mid + 1]`

     → peak lies **right** ⇒ `low = mid + 1`
   - else

→ peak lies **left or at mid** ⇒ `high = mid`

3. When loop ends, `low == high` is the peak index.

Dry Run

Array example: **[1, 3, 8, 12, 9, 5, 2]**, n = 7

```
low=0, high=6
mid=3 (value 12)
A[3] > A[4]  → move left → high = 3

low=0, high=3
mid=1 (value 3)
A[1] < A[2]  → move right → low = 2

low=2, high=3
mid=2 (value 8)
A[2] < A[3]  → move right → low = 3

low=3, high=3  → DONE, p = 3 (value 12)
```

(Indices shown 0-based; for 1-based answer p=4.)

```
#include <bits/stdc++.h>
using namespace std;

int findPeakIndex(const vector<int>& A) {
    int low = 0, high = (int)A.size() - 1;
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (A[mid] < A[mid + 1])      // peak to the right
            low = mid + 1;
        else                          // peak to the left or at mid
            high = mid;
    }
    return low; // or high, both equal
```

```
}

int main() {
    vector<int> A = {1, 3, 8, 12, 9, 5, 2};
    int p = findPeakIndex(A);
    cout << "Peak index (0-based): " << p
        << " , value: " << A[p] << endl;
    return 0;
}
```

Recurrence Relation & Time Complexity

Each step discards half the array:

- Recurrence: **$T(n) = T(n/2) + O(1)$**
- Master Theorem gives: **$T(n) = O(\log n)$**

Space: $O(1)$ (iterative).

Problem:

You are given **n** consecutive days of stock prices.

Price on day *i* is p(i). You may **buy once** and **sell once later** to maximize profit (or report that no profit is possible).

Design

1. An **$O(n^2)$** algorithm.
2. An **$O(n \log n)$** algorithm (hint: Maximum Subarray idea).

Check with:

```
n = 8
p = [2, 9, 3, 8, 11, 1, 6, 6]
```

1️⃣ Logic

- We want to maximize **p[j] – p[i]** with j>i.
- Equivalent to **max subarray sum** on differences:

  Let `d[k] = p[k+1] – p[k]`.

  Then the max profit is `max( sum of a contiguous segment of d )`.

2️⃣ Brute Force O(n²) Algorithm

Idea: Try every buy day i and every sell day j>i, compute profit.

Pseudocode:

```
bestProfit = 0
buyDay = sellDay = -1
for i = 1 to n:
    for j = i+1 to n:
        if p[j] - p[i] > bestProfit:
            bestProfit = p[j] - p[i]
            buyDay = i
            sellDay = j
return buyDay, sellDay, bestProfit
```

Dry Run (n=8 example)

Prices: 2 9 3 8 11 1 6 6

Check all pairs → Max difference = 11 – 2 = **9** (buy day 1, sell day 5).

Complexity:

- Time: $O(n^2)$
- Space: O(1)

3️⃣ Divide & Conquer O(n log n) (Maximum Subarray)

Transform: Create difference array `d[i] = p[i+1] - p[i]` for i=1…n-1.

Example:

p: 2 9 3 8 11 1 6 6

d: 7 -6 5 3 -10 5 0

Now find maximum subarray sum of d.

That sum = max profit.

The subarray indices correspond to buy at start, sell at end+1.

Recurrence

Standard maximum subarray:

```
T(n) = 2T(n/2) + O(n)  ⇒  T(n) = O(n log n)
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// ---------- O(n^2) Brute Force ----------
void maxProfitBrute(const vector<int>& price) {
    int n = price.size();
    int bestProfit = 0;
    int buyDay = -1, sellDay = -1;

    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int profit = price[j] - price[i];
            if (profit > bestProfit) {
                bestProfit = profit;
                buyDay = i + 1;   // convert to 1-based day
                sellDay = j + 1;
            }
        }
    }

    cout << "Brute Force:\n";
    if (bestProfit > 0)
        cout << "Buy on day " << buyDay
```

```
          << ", Sell on day " << sellDay
          << ", Profit = " << bestProfit << "\n";
    else
        cout << "No profit possible.\n";
}

// ---------- O(n log n) Divide & Conquer ----------
// Uses maximum subarray idea in a simple recursive function
int maxSubarray(const vector<int>& diff, int l, int r) {
    if (l == r) return diff[l];

    int mid = (l + r) / 2;
    int leftBest  = maxSubarray(diff, l, mid);
    int rightBest = maxSubarray(diff, mid + 1, r);

    // cross sum
    int leftSum = 0, maxLeft = diff[mid];
    for (int i = mid; i >= l; i--) {
        leftSum += diff[i];
        maxLeft = max(maxLeft, leftSum);
    }
    int rightSum = 0, maxRight = diff[mid + 1];
    for (int i = mid + 1; i <= r; i++) {
        rightSum += diff[i];
        maxRight = max(maxRight, rightSum);
    }
    int crossBest = maxLeft + maxRight;

    return max({leftBest, rightBest, crossBest});
}

int maxProfitDivideConquer(const vector<int>& price) {
    if (price.size() < 2) return 0;
    vector<int> diff;
    for (size_t i = 0; i + 1 < price.size(); i++)
        diff.push_back(price[i + 1] - price[i]);
```

```
    int ans = maxSubarray(diff, 0, (int)diff.size() - 1);
    return max(0, ans);
}

int main() {
    vector<int> price = {2, 9, 3, 8, 11, 1, 6, 6};

    maxProfitBrute(price);

    cout << "Divide & Conquer Profit = "
         << maxProfitDivideConquer(price) << "\n";

    return 0;
}
```

Divide & Conquer (Maximum Subarray)

Step A: Build Difference Array

```
d[i] = price[i+1] - price[i]
d = [7, -6, 5, 3, -10, 5, 0]
```

(Length = 7 because n-1 differences.)

Step B: Find Maximum Subarray Sum

A simple pass to visualize:

- Start with 7
- 7 + (-6) = 1 (keep)
- 1 + 5 = 6
- 6 + 3 = 9 ← current best
- 9 + (-10) = -1 (drop below 0 → restart)
- restart at 5 → best 9 still
- 5 + 0 = 5

Maximum contiguous sum = 9.

This corresponds to subarray **[7, -6, 5, 3]**,

which starts at d[0] and ends at d[3].

Step C: Map back to Days

- Start index of diff = 0 → **buy day = 1**
- End index of diff = 3 → **sell day = 4 + 1 = 5**

Profit = **9**.

Final Dry-Run Output

Program prints:

```
Brute Force:
Buy on day 1, Sell on day 5, Profit = 9
Divide & Conquer Profit = 9
```

- **Buy on Day 1 (price 2)**
- **Sell on Day 5 (price 11)**
- **Maximum profit = 9 per share**