ans_02

Question: Rewrite the above algorithm to find 3rd-largest element.

```
#include <algorithm>
#include <iostream>
#include <vector>

typedef double DataType;
std::vector<DataType> given;

struct DataSet {
  size_t begin, count;
};

std::vector<DataType> solve(DataSet input) {
  if (input.count == 1) {
    return {given[input.begin]};
  }

  DataSet left, right;
  left.begin = input.begin;
  left.count = input.count / 2;
  right.begin = input.begin + left.count;
  right.count = input.count - left.count;

  auto leftMax = solve(left);
  auto rightMax = solve(right);

  std::vector<DataType> merged;
  merged.insert(merged.end(), leftMax.begin(), leftMax.end());
  merged.insert(merged.end(), rightMax.begin(), rightMax.end());

  std::sort(merged.begin(), merged.end(), std::greater<DataType>());
  if (merged.size() > 3) {
    merged.resize(3);
  }
  return merged;
```

```
}

int main() {
  given = {5, 33, 3, 45, 8, 9, 2, 1, 7, 0, 55};
  DataSet data = {0, given.size()};

  auto top3 = solve(data);

  std::cout << "Top 3 elements: ";
  for (auto num : top3) {
    std::cout << num << " ";
  }
  return 0;
}
```

Question: When satisfied otherwise, try to proof there are no $c > c_{crit}$ can let $f(n) = \Omega(n^c)$.

$$Let \quad f(n) = \Theta(n^{c_{crit}}(\log n)^k$$

$$if \quad f(n) = \Omega(n^c),$$

$$then \quad f(n) \geq C \cdot n^c, \quad C > 0$$

$$(\log n)^k \geq C \cdot n^{\,c - log_b a}$$

$$because \quad c > c_{crit},$$

polynomial increment is greater than logoritihm increment. so there are no $c > c_{crit}$ can let $f(n) = \Omega(n^c)$.

Question: Try to analyse the time complexity of merge sort with $k = 5$ by master theorem.

For Merge Sort dividing the problem into 5 parts (k=5):

$$T(n) = 5T(n/5) + f(n)$$

where: `a = 5` (number of subproblems) `b = 5` (division factor) `f(n)` = cost to merge 5 sorted subarrays

Merge Step Complexity Merging 5 sorted arrays using a min-heap: - Each heap operation takes O(log5) time (constant) - Total of n operations needed

$$f(n) = O(nlog5) = O(n)$$

Critical Exponent

$$c_{crit} = log_b a = log_5 5 = 1$$

Case Analysis Compare f(n) with $n^{c_{crit}}$:

$$f(n) = O(n) = \Theta(n^1) = \Theta(n^{c_{crit}})$$

This falls under **Case 2 (Otherwise)** of Master Theorem where:

$$f(n) = \Theta(n^{c_{crit}}(logn)^k) \quad with \quad k = 0$$

Final Complexity According to Master Theorem:

$$T(n) = \Theta(n^{c_{crit}}(logn)^{k+1}) = \Theta(nlogn)$$

Question: Suppose given elements are not unique, please modify the algorithm to merge same values and count the frequent.

```
#include <iostream>
#include <vector>

struct Element {
  int value;
  int count;
};

std::vector<Element> mergeSortWithCount(std::vector<int> &arr, int left,
                                        int right);
std::vector<Element> countFrequencies(std::vector<int> &arr);

int main() {
  std::vector<int> array{3, 2, 3, 5, 2, 3, 4, 5, 6, 22};
  std::vector<Element> Merged = mergeSortWithCount(array, 0, array.size() - 1);

  bool first = true;
  for (auto &M : Merged) {
    if (!first)
      std::cout << ", ";
    std::cout << "{" << M.value << ", " << M.count << "}";
    first = false;
  }
  return 0;
}

std::vector<Element> mergeSortWithCount(std::vector<int> &arr, int left,
                                        int right) {
```

```
  // Base case: single element
  if (left == right) {
    return {{arr[left], 1}};
  }

  // Divide into 2 parts
  int mid = left + (right - left) / 2;

  // Conquer: sort and count each half
  auto leftResult = mergeSortWithCount(arr, left, mid);
  auto rightResult = mergeSortWithCount(arr, mid + 1, right);

  // Merge the results
  std::vector<Element> merged;
  size_t i = 0, j = 0;

  while (i < leftResult.size() && j < rightResult.size()) {
    if (leftResult[i].value < rightResult[j].value) {
      merged.push_back(leftResult[i]);
      i++;
    } else if (leftResult[i].value > rightResult[j].value) {
      merged.push_back(rightResult[j]);
      j++;
    } else {
      // Same value - merge counts
      merged.push_back(
          {leftResult[i].value, leftResult[i].count + rightResult[j].count});
      i++;
      j++;
    }
  }

  // Add remaining elements
  while (i < leftResult.size()) {
    merged.push_back(leftResult[i]);
    i++;
  }

  while (j < rightResult.size()) {
    merged.push_back(rightResult[j]);
    j++;
  }

  return merged;
}
```

```
// Wrapper function
std::vector<Element> countFrequencies(std::vector<int> &arr) {
  if (arr.empty())
    return {};
  return mergeSortWithCount(arr, 0, arr.size() - 1);
}
```