The Algorithm chosen: Quick Sort algorithm.

Initial Quick Sort Algorithm

The standard Quick Sort algorithm involves:

1. **Partitioning**: Choosing a pivot and partitioning the array into elements less than the pivot and elements greater than the pivot.
2. **Recursively Sorting**: Recursively applying Quick Sort to the sub-arrays formed by partitioning.

The basic implementation of Quick Sort:

```
void quickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        cmp++;
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
            basic_operations++;
        }
    }
    swap(arr[i + 1], arr[high]);
    basic_operations++;
    return i + 1;
}
```

Optimization 1: Choosing a Better Pivot

Initial Version: The basic version selects the last element as the pivot, which can lead to poor performance if the array is already sorted or nearly sorted.

Refinement: Using a better pivot selection strategy, such as the **Median-of-Three** method, can improve performance. This strategy selects the pivot as the median of the first, middle, and last elements of the array. This approach reduces the likelihood of worst-case performance.

An implementation of Median of three:

```
int medianOfThree(vector<int> &arr, int low, int high) {
    int mid = low + (high - low) / 2;
    int a = arr[low], b = arr[mid], c = arr[high];
    if (a < b) {
        if (b < c) return b;
        return (a < c) ? c : a;
    } else {
        if (a < c) return a;
        return (b < c) ? c : b;
    }
}
```

Optimization 2: Insertion Sort for Small Arrays

Initial Version: The basic Quick Sort algorithm recursively sorts small arrays, which might be inefficient due to the overhead of recursion.

Refinement: For small sub-arrays (e.g., size less than 10), using **Insertion Sort** instead of Quick Sort can be more efficient because Insertion Sort has less overhead for small datasets.

```cpp
void quickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        if (high - low + 1 <= 10) {
            insertionSort(arr, low, high);
            return;
        }
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Comparison:

```
(base) lokeshbudda@Lokeshs-MacBook-Air Q2 % ./optimizedQuickSort
Choose input method:
1. Generate Random Numbers (Average case)
2. Read from Input File (worst_case.txt)
2
Enter the input file name: worst_case.txt

Normal QuickSort Results:
Comparisons: 499500
Swaps: 500499
Basic operations: 500499
Execution time: 9 ms
Memory usage: 4000 bytes

Optimized QuickSort Results:
Comparisons: 6753
Swaps: 3598
Basic operations: 4216
Execution time: 0 ms
Memory usage: 4000 bytes
```

Note: The code this comparison is there in the folder.

By applying these refinements, Quick Sort can be significantly improved in both theoretical performance (fewer comparisons and swaps) and empirical execution times, demonstrating the practical benefits of algorithmic optimizations.