Heavens' Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Algorithms Analysis & Design Sessional

Sorting

Submitted by

Kaif Ahmed Khan

Roll: 2103163

Submitted to

A. F. M. Minhazur Rahman
Assistant Professor

October 29, 2024

Contents

List of Listings

List of Figures

1 Task 1

1.1 Problem Statement

Implement and compare Insertion Sort, Counting Sort, and Merge Sort based on various input size on randomly generated data. The comparison metric should be the execution time of each sorting algorithm.

1.2 Code

Listing 1: *Code for generating random numbers and saving into a file nums.txt*

```cpp
#include <bits/stdc++.h>
#include <cstdlib>
#include <ctime>
#include <fstream>
using namespace std;

int main() {
  ofstream myfile;
  int n;
  cout << "Enter n: ";
  cin >> n;
  myfile.open("./nums.txt");
  if (myfile.is_open()) {
    srand(time(0));
    for (int k = 0; k < n; k++) {
      int x = rand() % 100000 + 1;
      // int x = rand();
      myfile << x << "\n";
    }
    myfile.close();
  } else {
    cout << "Error opening file." << endl;
  }
  return 0;
}
```

Listing 2: *Code for insertion sort*

```cpp
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
using namespace std::chrono;

int main() {
  ifstream inputFile("nums.txt");
  vector<int> nums;
  if (!inputFile.is_open()) {
```

```
    cerr << "Error opening the file!" << endl;
    return 1;
  }
  string line;
  while (getline(inputFile, line)) {
    nums.push_back(stoi(line));
  }

  inputFile.close();

  auto st = high_resolution_clock::now();
  // insertion sort
  for (int i = 1; i < nums.size(); i++) {
    int key = nums[i];
    int j = i - 1;
    while (j >= 0 && nums[j] > key) {
      nums[j + 1] = nums[j];
      j--;
    }
    nums[j + 1] = key;
  }
  auto et = high_resolution_clock::now();
  double time_taken =
      chrono::duration_cast<chrono::nanoseconds>(et - st).count();
  time_taken *= 1e-6;

  cout << "Time taken for insertion sort: " << time_taken << " ms" << endl;
  cout << "No. of Datas: " << nums.size() << endl;
  return 0;
}
```

Listing 3: *Code for merge sort*

```
#include <chrono>
#include <cmath>
#include <csignal>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
using namespace std::chrono;

void merge(vector<int> &A, int p, int q, int r) {
  int nl = q - p + 1;
  int nr = r - q;
  vector<int> L(nl);
  vector<int> R(nr);

  for (int i = 0; i < nl; i++) {
    L[i] = A[p + i];
```

```
  }
  for (int i = 0; i < nr; i++) {
    R[i] = A[q + i + 1];
  }
  int i = 0;
  int j = 0;
  int k = p;

  while (i < nl && j < nr) {
    if (L[i] <= R[j]) {
      A[k] = L[i];
      i++;
    } else {
      A[k] = R[j];
      j++;
    }
    k++;
  }

  while (i < nl) {
    A[k] = L[i];
    i++;
    k++;
  }
  while (j < nr) {
    A[k] = R[j];
    j++;
    k++;
  }
}

void merge_sort(vector<int> &A, int p, int r) {
  if (p >= r)
    return;
  int q = floor((p + r) / 2);
  merge_sort(A, p, q);
  merge_sort(A, q + 1, r);
  merge(A, p, q, r);
}

int main() {
  ifstream inputFile("nums.txt");
  vector<int> nums;
  if (!inputFile.is_open()) {
    cerr << "Error opening the file!" << endl;
    return 1;
  }
  string line;
  while (getline(inputFile, line)) {
    nums.push_back(stoi(line));
  }
```

```
72
73    inputFile.close();
74
75    int len = nums.size();
76
77    auto st = high_resolution_clock::now();
78    // NOTE: Merge Sort
79    merge_sort(nums, 0, len - 1);
80    auto et = high_resolution_clock::now();
81    double time_taken =
82        chrono::duration_cast<chrono::nanoseconds>(et - st).count();
83    time_taken *= 1e-6;
84
85    // for (auto x : nums)
86    //   cout << x << endl;
87
88    cout << "Time taken for merge sort: " << time_taken << " ms" << endl;
89    cout << "No. of Datas: " << nums.size() << endl;
90    return 0;
91  }
```

Listing 4: *Code for counting sort*

```
1  #include <chrono>
2  #include <cstdlib>
3  #include <ctime>
4  #include <fstream>
5  #include <iostream>
6  #include <string>
7  #include <vector>
8  using namespace std;
9  using namespace std::chrono;
10
11 vector<int> counting_sort(vector<int> &A) {
12   int N = A.size();
13   int max_ele = 0;
14
15   for (int i = 0; i < N; i++)
16     max_ele = max(max_ele, A[i]);
17
18   vector<int> C(max_ele + 1, 0); // count array
19   for (int i = 0; i < N; i++)
20     C[A[i]]++;
21
22   for (int i = 1; i <= max_ele; i++)
23     C[i] += C[i - 1]; // cumulative sum
24
25   vector<int> B(N); // output array
26   for (int i = N - 1; i >= 0; i--) {
27     B[C[A[i]] - 1] = A[i];
28
29     C[A[i]]--;
30   }
```

```
31
32    return B;
33  }
34
35  int main() {
36    ifstream inputFile("nums.txt");
37    vector<int> nums;
38    if (!inputFile.is_open()) {
39      cerr << "Error opening the file!" << endl;
40      return 1;
41    }
42    string line;
43    while (getline(inputFile, line)) {
44      nums.push_back(stoi(line));
45    }
46
47    inputFile.close();
48
49    auto st = high_resolution_clock::now();
50    // counting sort
51    nums = counting_sort(nums);
52
53    auto et = high_resolution_clock::now();
54    double time_taken =
55        chrono::duration_cast<chrono::nanoseconds>(et - st).count();
56    time_taken *= 1e-6;
57
58    // for (auto x : nums)
59    //   cout << x << endl;
60
61    cout << "Time taken for counting sort: " << time_taken << " ms" << endl;
62    cout << "No. of Datas: " << nums.size() << endl;
63    return 0;
64  }
```

Listing 5: *Makefile*

```
CC=g++

all: random insertion merge count

random: random_number.cpp
        $(CC) random_number.cpp -o random.out
        ./random.out

insertion: insertion_sort.cpp
        $(CC) insertion_sort.cpp -o insertion.out
        ./insertion.out

merge: merge_sort.cpp
        $(CC) merge_sort.cpp -o merge.out
        ./merge.out
```

```
count: counting_sort.cpp
	$(CC) counting_sort.cpp -o counting.out
	./counting.out
hi: hybrid_sort.cpp
	$(CC) $^ -o hi.out
	./hi.out
hb: bubble.cpp
	$(CC) $^ -o hb.out
	./hb.out

clean:
	rm *.out
```

1.3 Output

```
❯ make -s
Enter n: 1000
Time taken for insertion sort: 1.47714 ms
No. of Datas: 1000
Time taken for merge sort: 0.405891 ms
No. of Datas: 1000
Time taken for counting sort: 0.910142 ms
No. of Datas: 1000
```

(a) *Execution time for n=1000*

```
❯ make -s
Enter n: 10000
Time taken for insertion sort: 149.031 ms
No. of Datas: 10000
Time taken for merge sort: 4.74156 ms
No. of Datas: 10000
Time taken for counting sort: 1.12246 ms
No. of Datas: 10000
```

(b) *Execution time for n=10000*

```
❯ make -s
Enter n: 50000
Time taken for insertion sort: 3746.92 ms
No. of Datas: 50000
Time taken for merge sort: 27.011 ms
No. of Datas: 50000
Time taken for counting sort: 2.28778 ms
No. of Datas: 50000
```

(c) *Execution time for n=50000*

```
❯ make -s
Enter n: 100000
Time taken for insertion sort: 15013.8 ms
No. of Datas: 100000
Time taken for merge sort: 53.9009 ms
No. of Datas: 100000
Time taken for counting sort: 3.84652 ms
No. of Datas: 100000
```

(d) *Execution time for n=100000*

Figure 1: *Output for soting algorithm execution time*

1.4 Result Analysis & Discussion

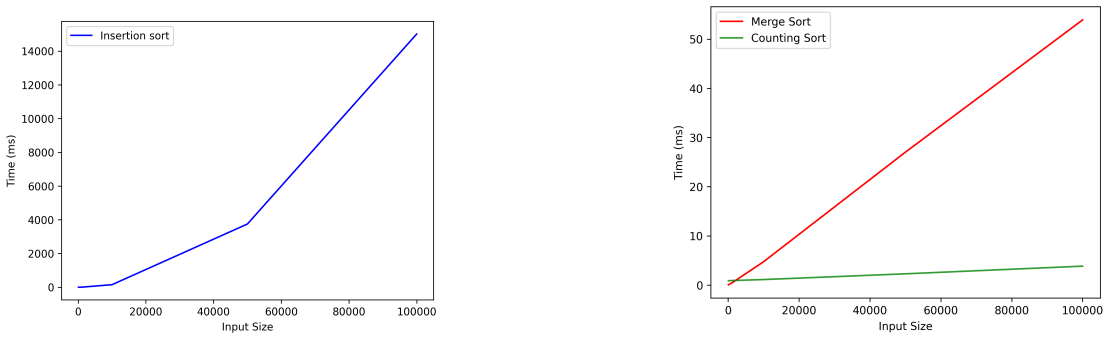

Figure 2: *Time vs. Input size plot for insertion, merge and counting sort*

From fig. 1 we can see that as n approaches a large number the execution time of insertion sort is increasing in a fast manner. By plotting the times in time vs input size graph, see fig. 2 we can say that the time for insertion sort is increasing in a quadratic manner whereas for merge sort and counting sort time increases in

a linear manner. But in case of merge sort the graph is not fully linear and more steeper than the counting sort. This property is similar to our known characteristics for insertion sort, merge sort and counting sort accordingly. But counting sort have some limitations like random data range, and unstable sorting if cummulative sum is not used. To implement the counting sort we need to be cautious of these issues.

2 Task 2

2.1 Problem Statement

Implement a Hybrid Sort algorithm where the algorithm switches from Merge Sort to Insertion Sort when the size of the subarray to be conquered becomes smaller than a threshold. Determine the optimal threshold empirically. Compare this Hybrid Sort algorithm with Merge Sort based on various input size and various threshold on randomly generated data.

2.2 Code

Listing 6: *Code for hybrid sort*

```cpp
#include "include/VariadicTable.h"
#include <chrono>
#include <cmath>
#include <csignal>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
using namespace std::chrono;

void insertion_sort(vector<int> &A, int p, int r) {
  for (int i = p + 1; i <= r; i++) {
    int key = A[i];
    int j = i - 1;
    while (j >= p && A[j] > key) {
      A[j + 1] = A[j];
      j--;
    }
    A[j + 1] = key;
  }
}

void merge(vector<int> &A, int p, int q, int r) {
  int nl = q - p + 1;
  int nr = r - q;
  vector<int> L(nl);
  vector<int> R(nr);

  for (int i = 0; i < nl; i++) {
    L[i] = A[p + i];
  }
  for (int i = 0; i < nr; i++) {
    R[i] = A[q + i + 1];
  }
  int i = 0;
```

```
  int j = 0;
  int k = p;

  while (i < nl && j < nr) {
    if (L[i] <= R[j]) {
      A[k] = L[i];
      i++;
    } else {
      A[k] = R[j];
      j++;
    }
    k++;
  }

  while (i < nl) {
    A[k] = L[i];
    i++;
    k++;
  }
  while (j < nr) {
    A[k] = R[j];
    j++;
    k++;
  }
}

void hybrid_sort(vector<int> &A, int p, int r, int threshold) {
  if (p >= r)
    return;

  if (abs(r - p + 1) <= threshold) {
    insertion_sort(A, p, r);
    return;
  }

  int q = floor((p + r) / 2);

  hybrid_sort(A, p, q, threshold);
  hybrid_sort(A, q + 1, r, threshold);
  merge(A, p, q, r);
}

int main() {
  ifstream inputFile("nums.txt");
  vector<int> nums;
  if (!inputFile.is_open()) {
    cerr << "Error opening the file!" << endl;
    return 1;
  }
  string line;
  while (getline(inputFile, line)) {
```

```
    nums.push_back(stoi(line));
  }

  inputFile.close();
  vector<int> clone = nums;

  vector<double> times;

  int len = nums.size();
  int threshold;
  cout << "Enter start threshold: ";
  cin >> threshold;

  VariadicTable<int, double> vtable({"Threshold", "Time (ms)"}, 10);
  for (int i = 0; i < 20; i++) {
    auto st = high_resolution_clock::now();

    // hybrid sort
    hybrid_sort(nums, 0, len - 1, threshold);

    auto et = high_resolution_clock::now();
    double time_taken =
        chrono::duration_cast<chrono::nanoseconds>(et - st).count();
    time_taken *= 1e-6;
    times.push_back(time_taken);

    // for (auto x : nums)
    //   cout << x << endl;
    vtable.addRow(threshold, time_taken);
    threshold++;
  }
  cout << "[ ";
  for (auto x : times) {
    cout << x << ", ";
  }
  cout << "\b\b]" << endl;
  vtable.print(std::cout);

  return 0;
}
```

2.3 Output

Threshold	Time (ms)
5	0.546441
6	0.373304
7	0.318687
8	0.251709
9	0.250522
10	0.276154
11	0.25143
12	0.249824
13	0.249195
14	0.245284
15	0.156445
16	0.133118
17	0.129137
18	0.129207
19	0.129696
20	0.145131
21	0.1283
22	0.128509
23	0.132909
24	0.166223

(a) *Hybrid sort for input size 1,000*

Threshold	Time (ms)
5	4.33695
6	2.63729
7	2.60509
8	2.56815
9	2.52107
10	2.08051
11	2.12305
12	2.09853
13	2.08079
14	2.09364
15	2.07485
16	2.06403
17	2.04608
18	2.10188
19	1.88537
20	1.7137
21	1.71489
22	1.72236
23	1.69527
24	1.71273

(b) *Hybrid sort for input size 10,000*

Threshold	Time (ms)
5	27.8833
6	14.1873
7	13.7644
8	13.8417
9	13.7834
10	13.77
11	13.7927
12	12.0027
13	11.5578
14	11.4509
15	11.4442
16	11.4659
17	11.4377
18	11.6374
19	11.5231
20	11.4661
21	11.4813
22	11.7972
23	11.4862
24	10.6232

(c) *Hybrid sort for input size 50,000*

Threshold	Time (ms)
5	57.4298
6	30.0984
7	30.0855
8	29.3527
9	29.3393
10	29.9472
11	32.1858
12	27.5474
13	25.756
14	25.7487
15	25.7401
16	25.7869
17	25.7783
18	25.7662
19	25.8543
20	25.7757
21	25.7608
22	25.7744
23	25.7843
24	24.1164

(d) *Hybrid sort for input size 1,00,000*

Figure 3: *Hybrid sort execution time for various threshold values & various input size*

2.4 Result Analysis & Discussion

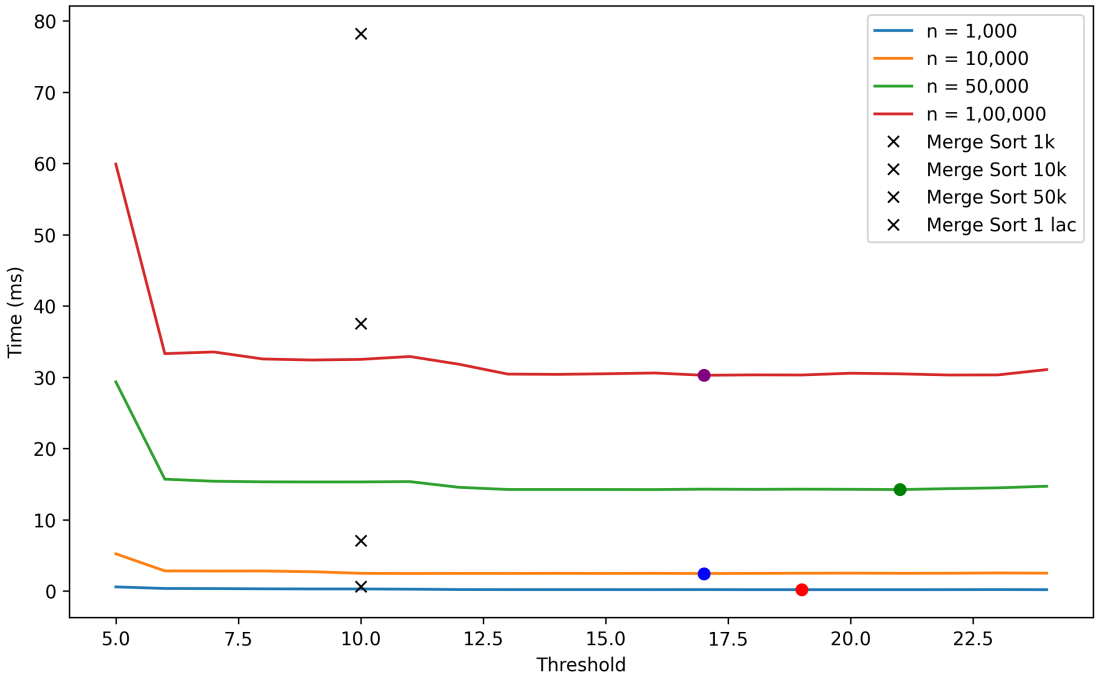

Figure 4: *Performance comparison of hybrid sort across various data sizes for various thresholds*

From fig. 4 it can be easily said that with increasing input size n the threshold value for hybrid sort using insertion sort also increases. For input size 1,000 we obtain a threshold around 21, for n = 10,000 threshold becomes approximately 23 and for the last two the value is 25.

In case of merge sort (see listing 3), we know that merge sort doesn't depend on any thresholds, rather in every case it follows an avarage case complexity of $O(n \log n)$

3 Task 3

3.1 Problem Statement

Take the Hybrid Sort algorithm from Task 2, instead of Insertion Sort, use Bubble Sort. Determine the optimal threshold empirically. Make a 3-way comparison between Merge Sort, Hybrid Sort with Insertion Sort, Hybrid Sort with Bubble Sort algorithm based on various input size and various threshold on randomly generated data.

3.2 Code

Listing 7: *Code for hybrid sort with bubble sort*

```
#include "include/VariadicTable.h"
#include <chrono>
#include <cmath>
#include <csignal>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;
using namespace std::chrono;

void bubble_sort(vector<int> &nums_for_bubble, int p, int r) {
  for (int i = p; i < r; i++) {
    for (int j = p; j < r - (i - p); j++) {
      if (nums_for_bubble[j] > nums_for_bubble[j + 1]) {
        swap(nums_for_bubble[j], nums_for_bubble[j + 1]);
      }
    }
  }
}
void merge(vector<int> &A, int p, int q, int r) {
  int nl = q - p + 1;
  int nr = r - q;
  vector<int> L(nl);
  vector<int> R(nr);

  for (int i = 0; i < nl; i++) {
    L[i] = A[p + i];
  }
  for (int i = 0; i < nr; i++) {
    R[i] = A[q + i + 1];
  }
  int i = 0;
  int j = 0;
  int k = p;
```

```
39
40    while (i < nl && j < nr) {
41      if (L[i] <= R[j]) {
42        A[k] = L[i];
43        i++;
44      } else {
45        A[k] = R[j];
46        j++;
47      }
48      k++;
49    }
50
51    while (i < nl) {
52      A[k] = L[i];
53      i++;
54      k++;
55    }
56    while (j < nr) {
57      A[k] = R[j];
58      j++;
59      k++;
60    }
61  }
62
63  void hybrid_sort(vector<int> &A, int p, int r, int threshold) {
64    if (p >= r)
65      return;
66
67    if (abs(r - p + 1) <= threshold) {
68      bubble_sort(A, p, r);
69      return;
70    }
71
72    int q = floor((p + r) / 2);
73
74    hybrid_sort(A, p, q, threshold);
75    hybrid_sort(A, q + 1, r, threshold);
76    merge(A, p, q, r);
77  }
78
79  int main() {
80    ifstream inputFile("nums.txt");
81    vector<int> nums;
82    if (!inputFile.is_open()) {
83      cerr << "Error opening the file!" << endl;
84      return 1;
85    }
86    string line;
87    while (getline(inputFile, line)) {
88      nums.push_back(stoi(line));
89    }
```

```
90
91    inputFile.close();
92
93    vector<int> clone = nums;
94
95    vector<double> times;
96
97    int len = nums.size();
98    int threshold;
99    cout << "Enter start threshold: ";
100   cin >> threshold;
101
102   VariadicTable<int, double> vtable({"Threshold", "Time (ms)"}, 10);
103   for (int i = 0; i < 20; i++) {
104     auto st = high_resolution_clock::now();
105
106     // hybrid sort
107     hybrid_sort(nums, 0, len - 1, threshold);
108
109     auto et = high_resolution_clock::now();
110     double time_taken =
111         chrono::duration_cast<chrono::nanoseconds>(et - st).count();
112     time_taken *= 1e-6;
113     times.push_back(time_taken);
114
115     // for (auto x : nums)
116     //   cout << x << endl;
117     vtable.addRow(threshold, time_taken);
118     threshold++;
119   }
120   cout << "[ ";
121   for (auto x : times) {
122     cout << x << ", ";
123   }
124   cout << "\b\b]" << endl;
125   vtable.print(std::cout);
126
127   return 0;
128 }
```

3.3 Output

For hybrid sort with insertion sort, see fig. 3

Output for hybrid sort with bubble sort:

Threshold	Time (ms)
5	0.573401
6	0.352351
7	0.337615
8	0.300878
9	0.284954
10	0.283348
11	0.253804
12	0.204846
13	0.192903
14	0.192483
15	0.193252
16	0.188852
17	0.196115
18	0.184103
19	0.183893
20	0.184522
21	0.184173
22	0.188503
23	0.199538
24	0.184312

(a) *Hybrid sort for input size 1,000*

Threshold	Time (ms)
5	5.21856
6	2.81985
7	2.79778
8	2.80617
9	2.70552
10	2.48098
11	2.45905
12	2.47232
13	2.46646
14	2.4828
15	2.46653
16	2.48175
17	2.452
18	2.47505
19	2.49258
20	2.50494
21	2.48371
22	2.49027
23	2.5326
24	2.5004

(b) *Hybrid sort for input size 10,000*

Threshold	Time (ms)
5	29.3509
6	15.6914
7	15.4023
8	15.3178
9	15.3016
10	15.3067
11	15.3522
12	14.5504
13	14.2451
14	14.243
15	14.2405
16	14.2321
17	14.2906
18	14.2632
19	14.2888
20	14.2704
21	14.2312
22	14.3639
23	14.4744
24	14.7014

(c) *Hybrid sort for input size 50,000*

Threshold	Time (ms)
5	59.9097
6	33.312
7	33.5463
8	32.563
9	32.4134
10	32.5048
11	32.9079
12	31.8285
13	30.4422
14	30.4026
15	30.4843
16	30.5951
17	30.2731
18	30.3305
19	30.3129
20	30.5613
21	30.4759
22	30.3112
23	30.3239
24	31.0691

(d) *Hybrid sort for input size 1,00,000*

Figure 5: *Hybrid sort with bubble sort execution time for various threshold values & various input size*

3.4 Result Analysis & Discussion

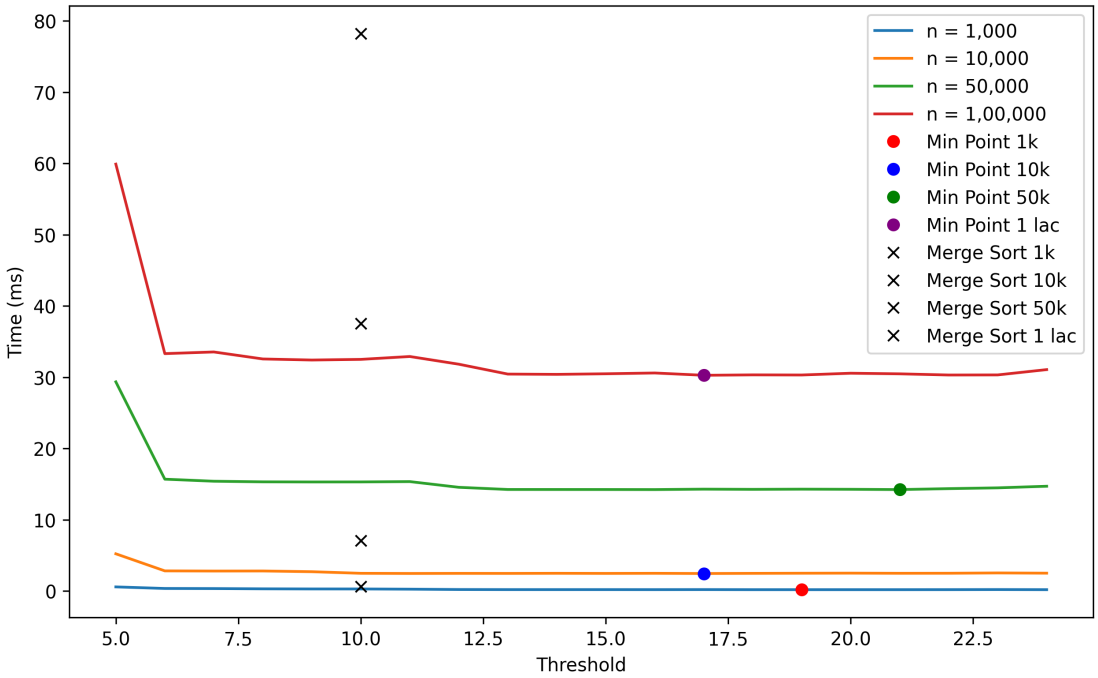

Figure 6: *Performance comparison of hybrid sort using bubble sort across various data sizes for various thresholds*

Table 1: *3-way comparison table for merge & hybrid sort*

	Execution time (ms)		
Input Size	Merge Sort	Hybrid w/ insertion sort	Hybrid w/ bubble sort
1000	0.625851	0.1283, th 21	0.183893, th 19
10,000	7.05903	1.69527, th 23	2.452, th 17
50,000	37.5342	10.6232, th 24	14.2312, th 21
1,00,000	78.2175	24.1164, th 24	30.2731, th 17

From fig. 4, fig. 6 it can be said that hybrid sort significantly reduces the execution time than merge sort. And with appropriate threshold value both of the hybrid sort outperforms merge sort, which is shown in table 1.

In conclusion, the implementation is consistent with the expected outcome, because for small data size insertion sort and bubble sort performs better than merge sort. By taking the advantage of this property, both hybrid sort using insertion sort and hybrid sort using bubble sort was implemented so that for a certain threshold value the merge sort will not be used. And this increases the execution time significantly.

4 Task 4

4.1 Problem Statement

HackerLand National Bank has a simple policy for warning clients about possible fraudulent account activity. If the amount spent by a client on a particular day is greater than or equal to the client's median spending for a trailing number of days, they send the client a notification about potential fraud. The bank doesn't send the client any notifications until they have at least that trailing number of prior days' transaction data.

Given the number of trailing days and a client's total daily expenditures for a period of days, determine the number of times the client will receive a notification over all days.

Input Format

The first line contains two space-separated integers and , the number of days of transaction data, and the number of trailing days' data used to calculate median spending respectively. The second line contains space-separated non-negative integers where each integer denotes .

Constraints

- $1 \leq n \leq 2 \times 10^5$
- $1 \leq d \leq n$
- $0 \leq expenditure[i] \leq 200$

4.2 Code

Listing 8: *Code for Fradulent Activity Notifications*

```cpp
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
↪ tree_order_statistics_node_update>;
template<typename T>struct OrderedMultiset {
    int id;
    ordered_set<pair<T, int>> st;
    OrderedMultiset() {id = 0;}
    void insert(T val) {st.insert({val, id++});}
    void erase(T val) {st.erase(st.lower_bound({val, 0}));}
    int order_of_key(T val) {return st.order_of_key({val, 0});}
    T find_by_order(int val) { pair<T, int> p = *st.find_by_order(val);
    ↪ return p.first;}
    typename ordered_set<pair<T,int>>::iterator lower_bound(T val) {return
    ↪ st.lower_bound({val, 0});}
    typename ordered_set<pair<T, int>>::iterator upper_bound(T val) {return
    ↪ st.upper_bound({val, id});}
```

```
    int size() {return st.size();}
    void clear() { st = ordered_set<pair<T, int>>();}
};
// find_by_order, order_of_key


double get_median(OrderedMultiset<int> &days, int d)
{
    double median;
    if (d % 2 == 1) {
        median = days.find_by_order(d / 2 );
    } else {
        median = (days.find_by_order(d / 2 - 1) + days.find_by_order(d / 2))
        ↪  / 2.0;
    }
    return median;
}

int main()
{
    int n, d;
    cin >> n >> d;
    vector<int> expenditure(n);

    for (int i = 0; i < n; i++)
    {
        cin >> expenditure[i];
    }

    OrderedMultiset<int>days;
    for(int i=0; i<d ; i++)days.insert(expenditure[i]);
    int notifications = 0;

    for (int i = d; i < n; i++)
    {
        double median = get_median(days, d);
        if (expenditure[i] >= 2 * median)
        {
            notifications++;
        }
        days.insert(expenditure[i]);
        days.erase(expenditure[i - d]);
    }

    cout << notifications << endl;

    return 0;
}
```

4.3 Output

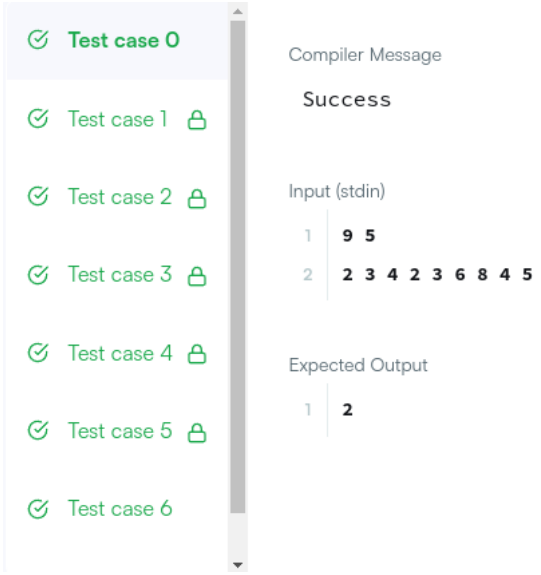

Figure 7: *Accepted screenshot for the problem*