106129029 , Algos Lab

Dipesh Kafle

Code

```cpp
#include <algorithm>
#include <cassert>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std;

template <typename Func, typename... Args>
double timeMyFunction(Func func, Args &...args) {
  auto start_time = std::chrono::steady_clock::now();
  func(args...);
  auto end_time = std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_time =
      std::chrono::duration_cast<std::chrono::duration<double>>(end_time -
                                                                start_time);
  return elapsed_time.count();
}

double time_stl_sort(vector<int> &vec) {
  auto start_time = std::chrono::steady_clock::now();
  std::sort(vec.begin(), vec.end());
  auto end_time = std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_time =
      std::chrono::duration_cast<std::chrono::duration<double>>(end_time -
                                                                start_time);
  return elapsed_time.count();
}
```

```cpp
bool heapify_at_index(vector<int> &hp, int index, int size) {
  int br1 = 2 * index + 1;
  int br2 = br1 + 1;
  bool swapped = false;
  if (br1 < size && br2 < size) {
    int i = min(hp[br1], hp[br2]) == hp[br1] ? br1 : br2;
    if (hp[i] < hp[index]) {
      std::swap(hp[i], hp[index]);
      heapify_at_index(hp, i, size);
      swapped = true;
    }
  } else if (br1 < size && hp[br1] < hp[index]) {
    std::swap(hp[br1], hp[index]);
    heapify_at_index(hp, br1, size);
    swapped = true;
  }
  return swapped;
}

inline void extract_min_to_back(vector<int> &vec, int size) {
  std::swap(vec[size - 1], vec[0]);
  heapify_at_index(vec, 0, size - 1);
}

void heapSort(vector<int> &vec) {
  for (int i = vec.size() / 2; i >= 0; i--) {
    heapify_at_index(vec, i, vec.size());
  }
  for (int i = vec.size(); i >= 1; i--) {
    extract_min_to_back(vec, i);
  }
  reverse(vec.begin(), vec.end());
}
```

```cpp
void merge_sorted_arrays(vector<int> &vec, int l, int m, int r) {
  vector<int> left_arr(m - l);
  vector<int> right_arr(r - m);
  copy(vec.begin() + l, vec.begin() + m, left_arr.begin());
  copy(vec.begin() + m, vec.begin() + m, right_arr.begin());
  int i = l;
  int l_i = 0;
  int r_i = 0;
  while (i < r) {
    if (l_i < left_arr.size() && r_i < right_arr.size()) {
      if (left_arr[l_i] <= right_arr[r_i]) {
        vec[i] = left_arr[l_i];
        l_i++;
      } else {
        vec[i] = right_arr[r_i];
        r_i++;
      }
      i++;
    } else if (l_i < left_arr.size()) {
      copy(left_arr.begin() + l_i, left_arr.end(), vec.begin() + i);
      break;
    } else {
      copy(right_arr.begin() + r_i, right_arr.end(), vec.begin() + i);
      break;
    }
  }
  return;
}
```

```cpp
void mergesort(vector<int> &vec, int l, int r) {
  if (r - l == 1) {
    return;
  }
  int m = (l + r) / 2;
  mergesort(vec, l, m);
  mergesort(vec, m, r);
  merge_sorted_arrays(vec, l, m, r);
}
```

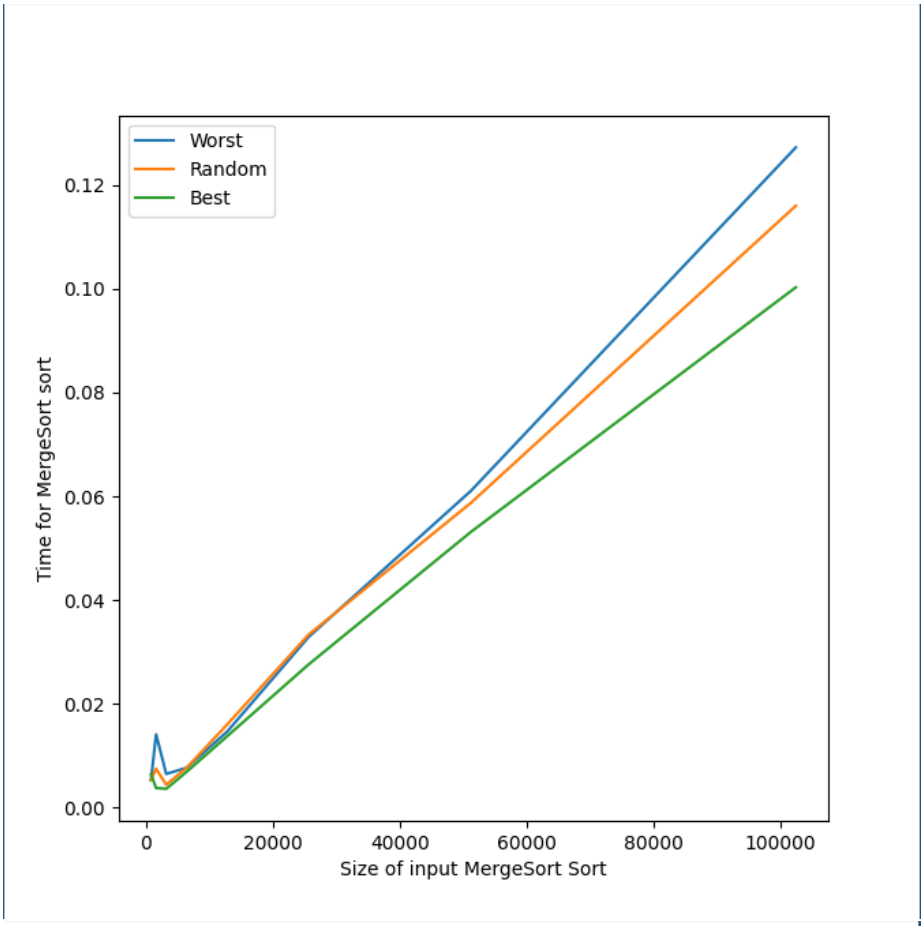
```cpp
int main() {
  srand(time(0));
  ofstream merge_sort("MergeSort.txt");
  ofstream heap_sort("HeapSort.txt");
  ofstream stl_sort("StlSort.txt");
  double time_elapsed;
  for (int size : {800, 1600, 3200, 6400, 12800, 25600, 51200, 102400}) {
    int l = 0;
    vector<int> reverseSorteda(size);
    iota(reverseSorteda.begin(), reverseSorteda.end(), 1);
    reverse(reverseSorteda.begin(), reverseSorteda.end());

    vector<int> reverseSorteda2 = reverseSorteda;
    vector<int> reverseSorteda3 = reverseSorteda;

    // will generate random numbers and put it in array of size =size
    vector<int> randomArrA(size);
    generate(randomArrA.begin(), randomArrA.end(), []() { return rand(); });
    vector<int> randomArrA2 = randomArrA;
    vector<int> randomArrA3 = randomArrA;

    //
    //
    // worst cases
    time_elapsed = timeMyFunction(mergesort, reverseSorteda, l, size);
    cout << "Merge Sort Worst for  "
         << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
         << endl;
    merge_sort << "Worst:" << size << ":" << fixed << setprecision(30)
               << time_elapsed << endl;

    assert(is_sorted(reverseSorteda.begin(), reverseSorteda.end()));

    time_elapsed = timeMyFunction(heapSort, reverseSorteda2);
    cout << "Heap Sort Worst for  "
         << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
         << endl;
```

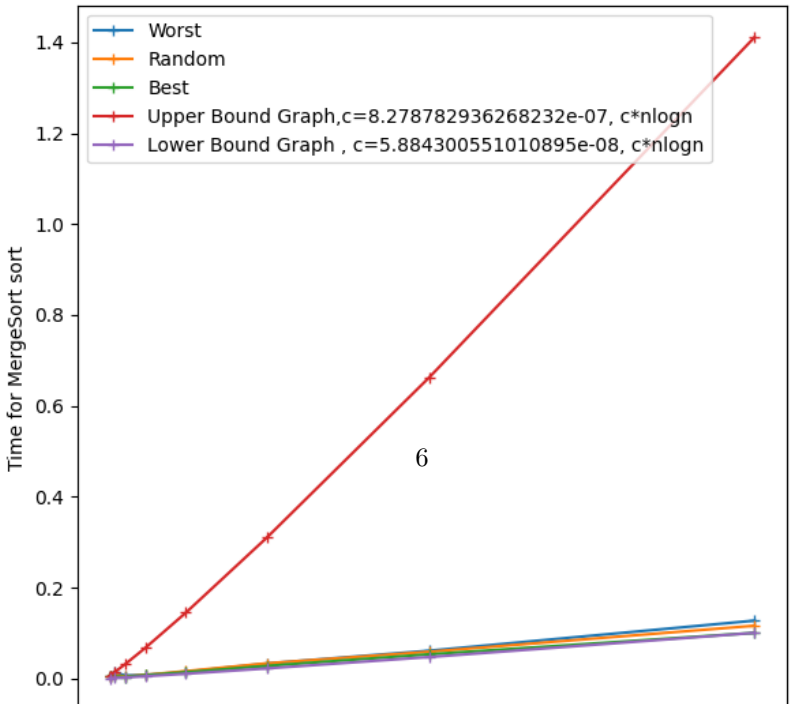
```
148     heap_sort << "Worst:" << size << ":" << fixed << setprecision(30)
1               << time_elapsed << endl;
2
3       assert(is_sorted(reverseSorteda2.begin(), reverseSorteda2.end()));
4
5       time_elapsed = time_stl_sort(reverseSorteda3);
6       cout << "Built in Sort Worst for  "
7            << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
8            << endl;
9       stl_sort << "Worst:" << size << ":" << fixed << setprecision(30)
10              << time_elapsed << endl;
11
12      assert(is_sorted(reverseSorteda3.begin(), reverseSorteda3.end()));
13
14      //
15      //
16      //
17      //
18      //
19      //
20      // Random cases
21      time_elapsed = timeMyFunction(mergesort, randomArrA, l, size);
22      cout << "Merge Sort Random Case for  "
23           << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
24           << endl;
25      merge_sort << "Random:" << size << ":" << fixed << setprecision(30)
26                 << time_elapsed << endl;
27      assert(is_sorted(randomArrA.begin(), randomArrA.end()));
28
29      time_elapsed = timeMyFunction(heapSort, randomArrA2);
30      cout << "Heap Sort Random for  "
31           << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
32           << endl;
33      heap_sort << "Random:" << size << ":" << fixed << setprecision(30)
34                << time_elapsed << endl;
35
36      assert(is_sorted(randomArrA2.begin(), randomArrA2.end()));
```

```
185
1       time_elapsed = time_stl_sort(randomArrA3);
2       cout << "Built in Sort Random for  "
3            << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
4            << endl;
5       stl_sort << "Random:" << size << ":" << fixed << setprecision(30)
6                << time_elapsed << endl;
7
8       assert(is_sorted(randomArrA3.begin(), randomArrA3.end()));
9
10      //
11      //
12      //
13      // Best case
14      // I'll just use the arrays from last sort, as they'll be sorted and  hence
15      // it should be best case
16      time_elapsed = timeMyFunction(mergesort, randomArrA, l, size);
17      cout << "Merge Sort Best Case for  "
18           << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
19           << endl;
20      merge_sort << "Best:" << size << ":" << fixed << setprecision(30)
21                 << time_elapsed << endl;
22      assert(is_sorted(randomArrA.begin(), randomArrA.end()));
23
24      time_elapsed = timeMyFunction(heapSort, randomArrA2);
25      cout << "Heap Sort Best for "
26           << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
27           << endl;
28      heap_sort << "Best:" << size << ":" << fixed << setprecision(30)
29                << time_elapsed << endl;
30
31      assert(is_sorted(randomArrA2.begin(), randomArrA2.end()));
32
33      time_elapsed = time_stl_sort(randomArrA3);
34      cout << "Built in Sort Best for  "
35           << "size " << size << ": " << fixed << setprecision(30) << time_elapsed
36           << endl;
```

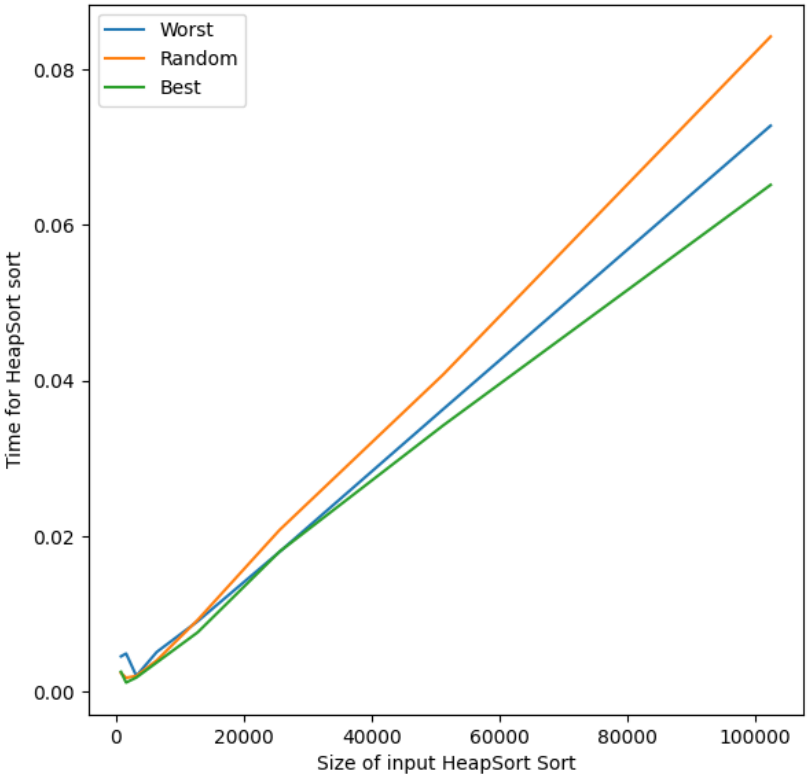
```
222     stl_sort << "Best:" << size << ":" << fixed << setprecision(30)
  1              << time_elapsed << endl;
  2
  3     assert(is_sorted(randomArrA3.begin(), randomArrA3.end()));
  4     cout << endl;
  5     cout << endl;
  6   }
  7 }
~
~
~
```

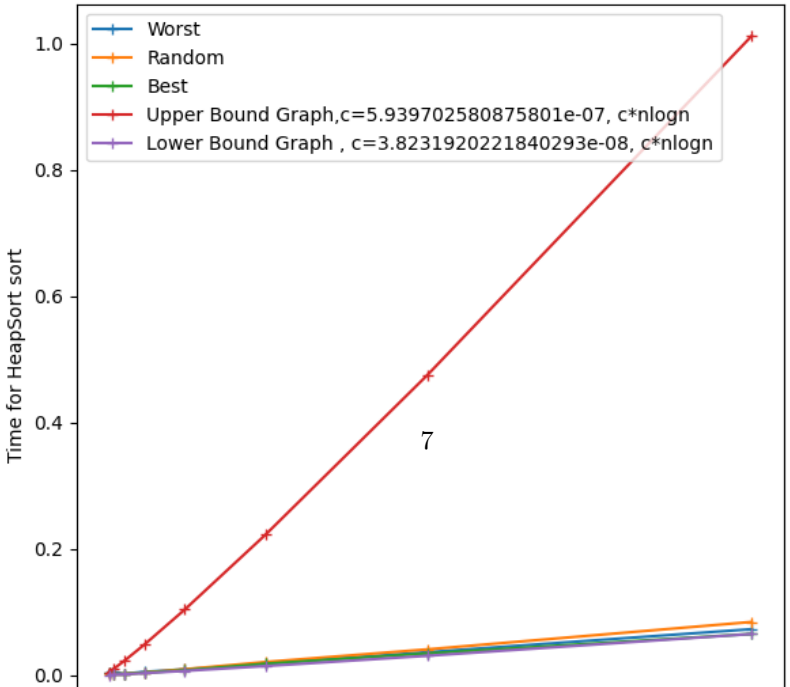
Merge Sort Plots



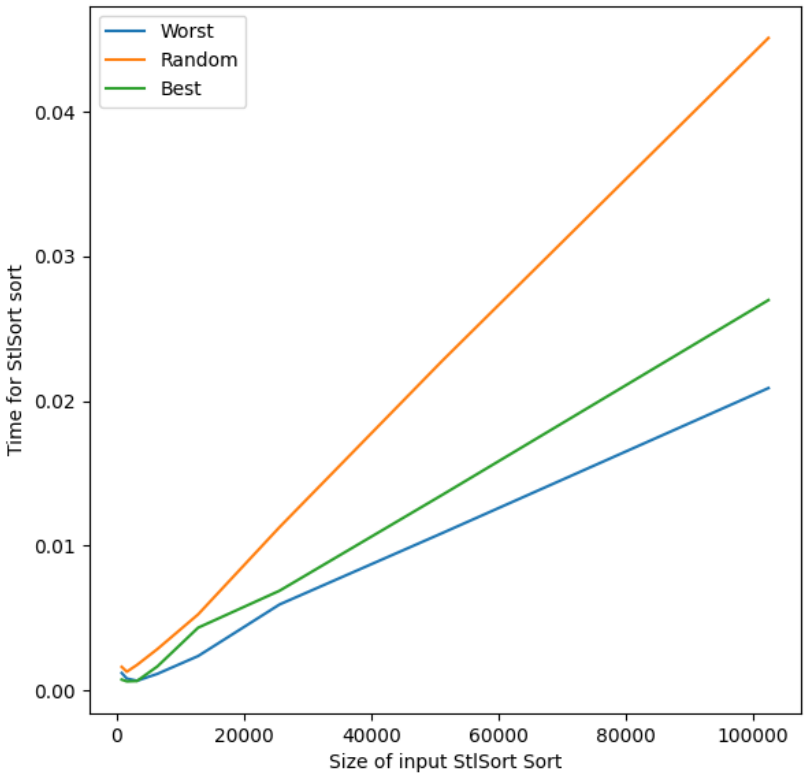


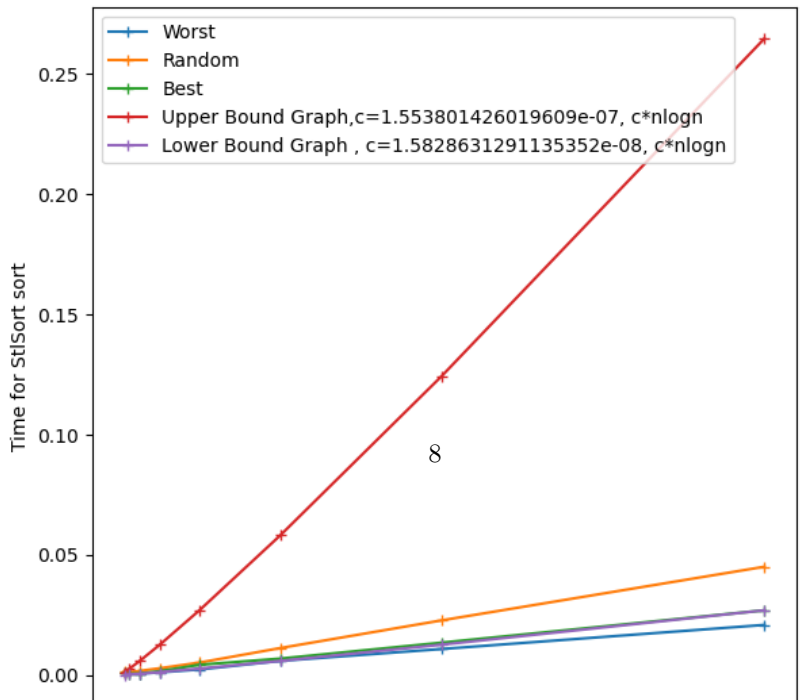
Heap Sort Plots





Built In Sort Plots





Comparision of three