106119029 , Algos Lab

Dipesh Kafle

Algos Lab 6

Remarks

- We can see from the graphs that randomized quick sort is very identical to the standard library implementation of sort. So the standard libray sort function is probably a randomized quicksort.
- Quick Sort(Non Randomized) : Worst case O(n^2) and Ω(nlogn). For reverse sorted array or sorted array , if we use the first element as pivot always, then we get a recurrence $T(n) = O(1) + T(n-1)$
- Quick Sort(Randomized) : Worst case O(n^2), amortized worst case O($nlogn$) and Ω(nlogn) . The Worst case can happen when every single time we generate a random number , it gives the first element and we have a sorted or reverse sorted array. In that case it's guaranteed to be O(n^2). This is a very rare case so the algorithm performs in amortized O(nlogn) time.

Code

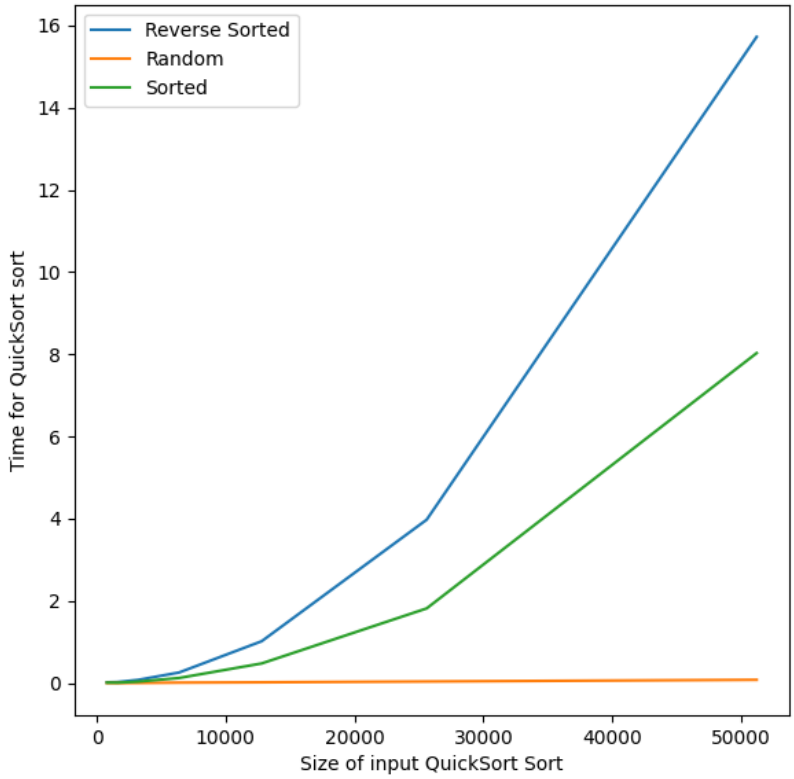
```cpp
#include <algorithm>
#include <cassert>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>
using namespace std;

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();
}
inline int qsort_partition(vector<int> &vec, int l, int r, int offset = 0) {
  int k = l;
  swap(vec[l + offset], vec[l]);
  for (int i = l + 1; i < r; i++) {
    if (vec[i] <= vec[l]) {
      swap(vec[++k], vec[i]);
    }
  }
  swap(vec[l], vec[k]);
  return k;
}

void qSort(vector<int> &vec, int l, int r, bool randomized = false) {
  srand(time(NULL));
  if (r - l <= 1) {
    return;
  }
  int off = randomized ? rand() % (r - l) : 0;
  int m = qsort_partition(vec, l, r, off);
  qSort(vec, l, m, randomized);
  qSort(vec, m + 1, r, randomized);
  return;
}

void stl_sort(vector<int> &vec) { sort(vec.begin(), vec.end()); }

void printDetails(ostream &os, const string &algo_name, const string &case_name,
                  double time_elapsed, size_t size) {
  os << algo_name << case_name << ":" << size << ": " << fixed
     << setprecision(30) << time_elapsed << endl;
}
```

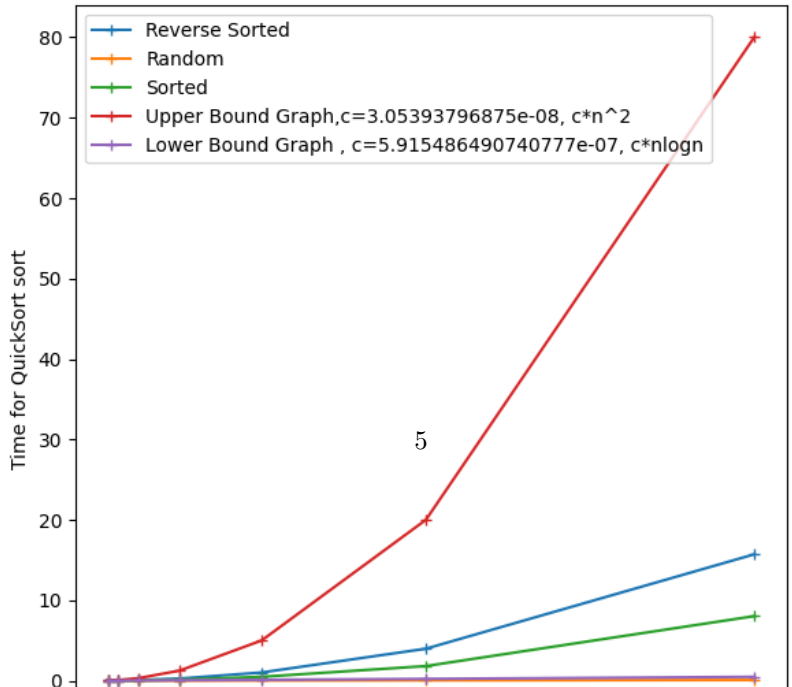
```cpp
int main() {
  srand(time(0));
  ofstream q_sort("QuickSort.txt");
  ofstream q_sort_rand("QuickSortRandom.txt");
  ofstream stlsort("StlSort.txt");
  double time_elapsed;

  for (int size : {800, 1600, 3200, 6400, 12800, 25600, 51200}) {
    int l = 0;
    vector<int> reverseSorteda(size);
    iota(reverseSorteda.begin(), reverseSorteda.end(), 1);
    reverse(reverseSorteda.begin(), reverseSorteda.end());
    vector<int> reverseSortedb = reverseSorteda;
    vector<int> reverseSortedc = 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> randomArrB = randomArrA;
    vector<int> randomArrC = randomArrA;

    // Random pivot quick sort
    // Reverse Sorted
    time_elapsed = timeMyFunction(qSort, reverseSorteda, 0, size, true);
    printDetails(q_sort_rand, "", "Worst", time_elapsed, size);
    printDetails(cout, "QuickSortRand ", "Worst", time_elapsed, size);
    assert(is_sorted(reverseSorteda.begin(), reverseSorteda.end()));

    // Random
    time_elapsed = timeMyFunction(qSort, randomArrA, 0, size, true);
    printDetails(q_sort_rand, "", "Random", time_elapsed, size);
    printDetails(cout, "QuickSortRand ", "Random", time_elapsed, size);
    assert(is_sorted(randomArrA.begin(), randomArrA.end()));
```

```cpp
    // Sorted
    time_elapsed = timeMyFunction(qSort, randomArrA, 0, size, true);
    printDetails(q_sort_rand, "", "Sorted", time_elapsed, size);
    printDetails(cout, "QuickSortRand ", "Sorted", time_elapsed, size);
    assert(is_sorted(randomArrA.begin(), randomArrA.end()));

    // pivot is first elem quick sort
    // Reverse Sorted
    time_elapsed = timeMyFunction(qSort, reverseSortedb, 0, size, false);
    printDetails(q_sort, "", "Worst", time_elapsed, size);
    printDetails(cout, "QuickSort ", "Worst", time_elapsed, size);
    assert(is_sorted(reverseSortedb.begin(), reverseSortedb.end()));

    // Random
    time_elapsed = timeMyFunction(qSort, randomArrB, 0, size, false);
    printDetails(q_sort, "", "Random", time_elapsed, size);
    printDetails(cout, "QuickSort ", "Random", time_elapsed, size);
    assert(is_sorted(randomArrB.begin(), randomArrB.end()));

    // Sorted
    time_elapsed = timeMyFunction(qSort, randomArrB, 0, size, false);
    printDetails(q_sort, "", "Sorted", time_elapsed, size);
    printDetails(cout, "QuickSort ", "Sorted", time_elapsed, size);

    // Built in Sort
    // Reverse Sorted
    time_elapsed = timeMyFunction(stl_sort, reverseSortedc);
    printDetails(stlsort, "", "Worst", time_elapsed, size);
    printDetails(cout, "StlSort ", "Worst", time_elapsed, size);
    assert(is_sorted(reverseSortedc.begin(), reverseSortedc.end()));

    // Random
    time_elapsed = timeMyFunction(stl_sort, randomArrC);
    printDetails(stlsort, "", "Random", time_elapsed, size);
    printDetails(cout, "StlSort ", "Random", time_elapsed, size);
    assert(is_sorted(randomArrC.begin(), randomArrC.end()));
```

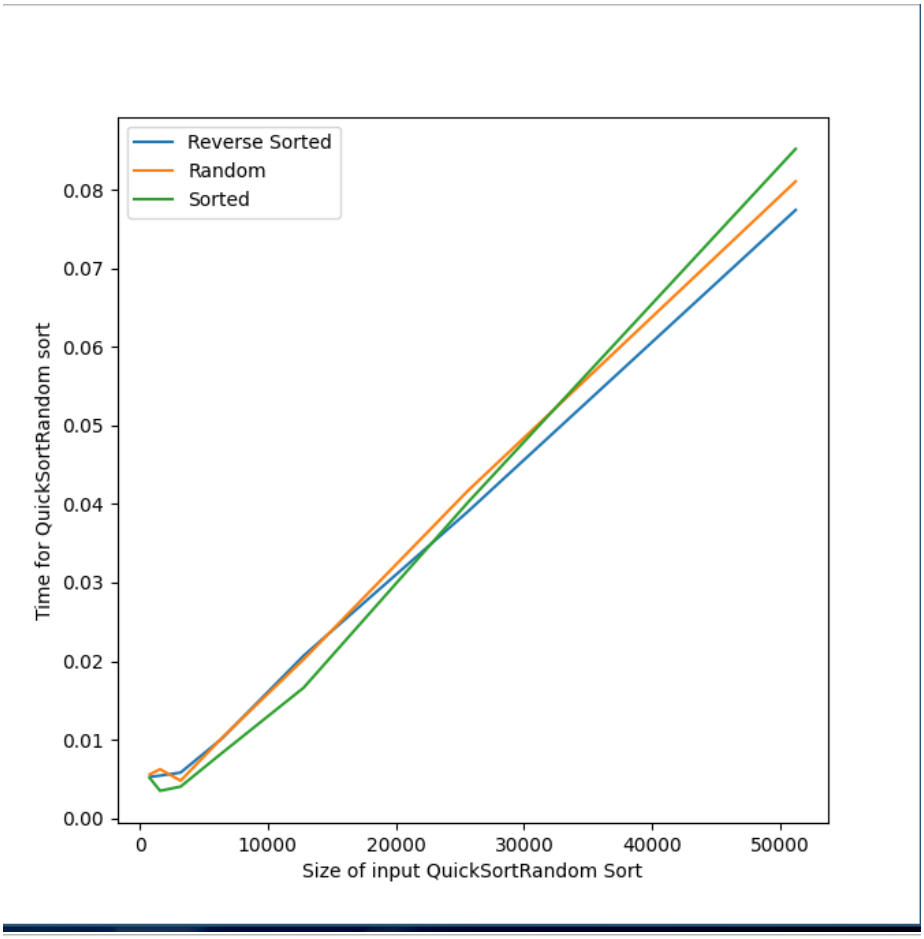
```
128
  1     // Sorted
  2     time_elapsed = timeMyFunction(stl_sort, randomArrC);
  3     printDetails(stlsort, "", "Sorted", time_elapsed, size);
  4     printDetails(cout, "StlSort ", "Sorted", time_elapsed, size);
  5     assert(is_sorted(randomArrC.begin(), randomArrC.end()));
  6
  7     cout << endl << endl;
  8   }
  9 }
~
~
```

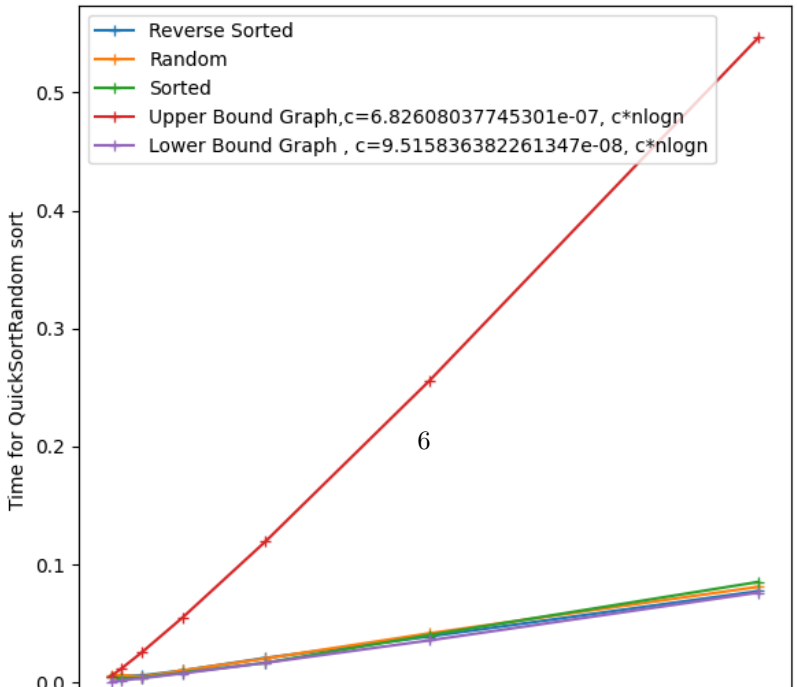
Quick Sort (Non Randomized) Plots



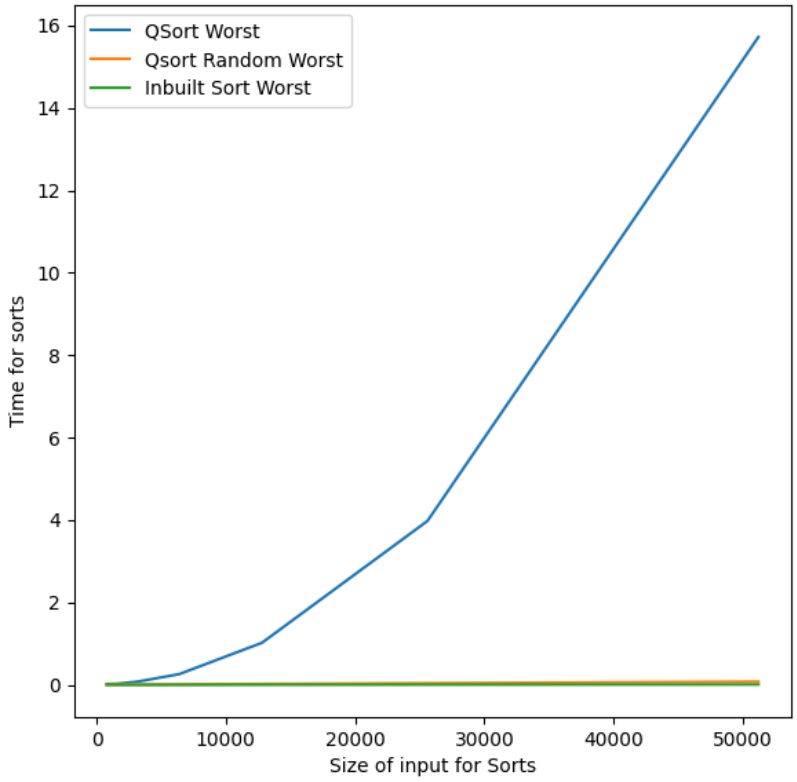


Quick Sort (Randomized) Plots

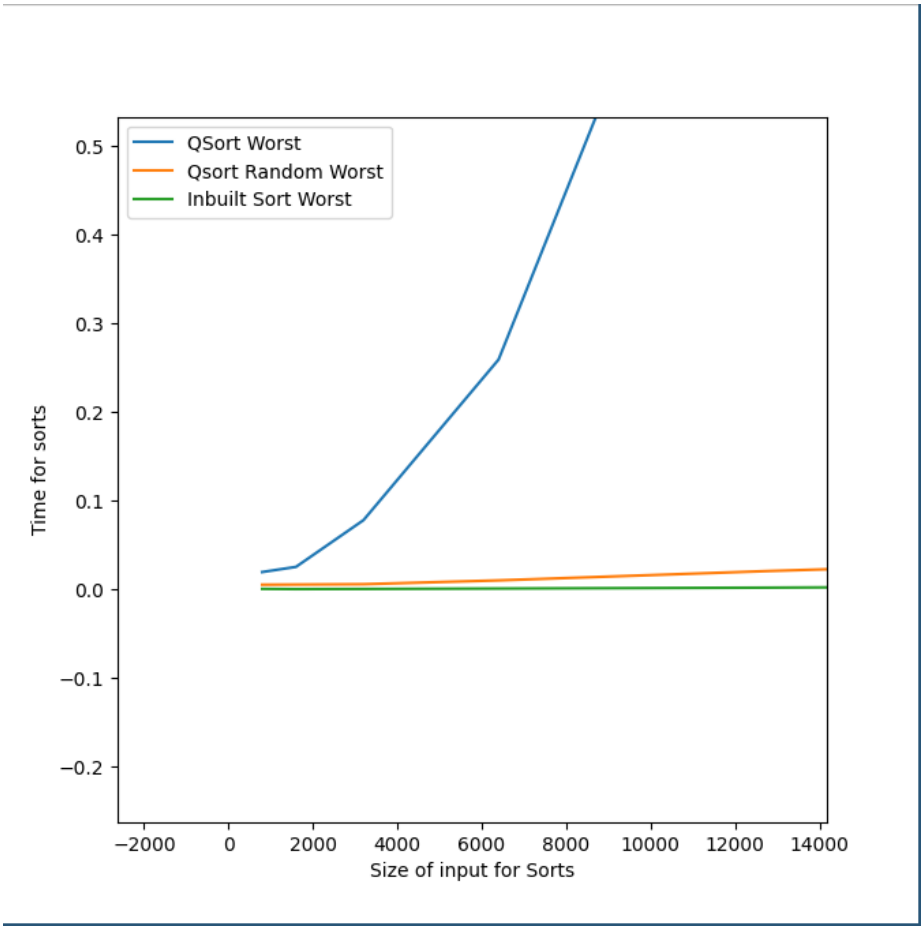




Comparisons between Quick Sort(Randomized), Quick Sort(Classical) and built in sort



- Zooomed


QSort Worst
Qsort Random Worst
Inbuilt Sort Worst
Time for sorts
Size of input for Sorts
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−2000
0
2000
4000
6000
8000
10000
12000
14000