EECE7205 – Fundamentals of Computer Engineering

Homework # 1

Name(Last, First): Li, Zhengnan

Grade:

Question 1

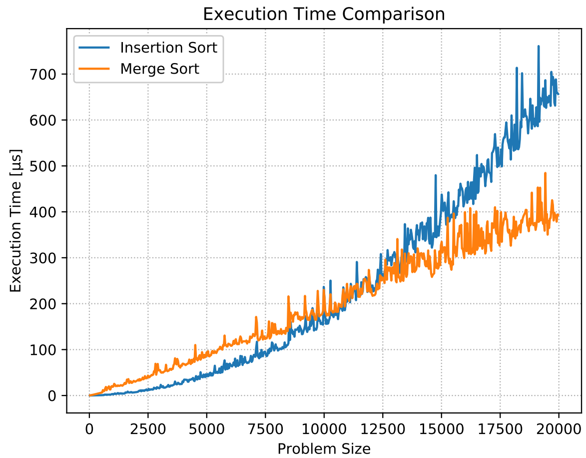

Figure 1: Memory not aligned, problem size increases linearly with step 32.

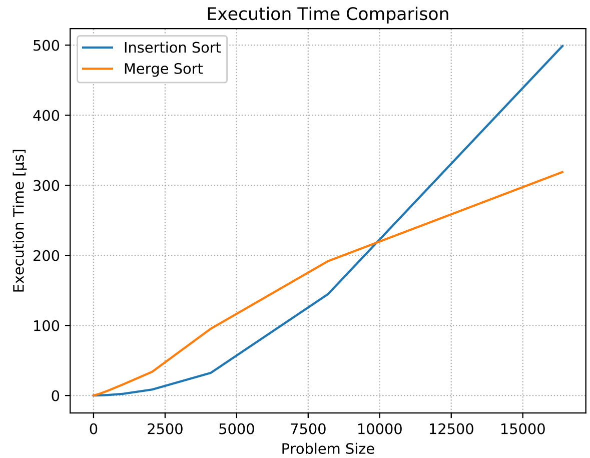

Figure 2: Memory aligned, problem size doubles starting with size 2.

Fig. 1 and Fig. 2 present the execution times of insertion sort and merge sort. Clearly we see when problem size approaching $n = 10000$, merge sort starts to outperform the insertion sort in terms of execution time.
Note that to compile and execute the code listing, execute on macOS

```
clang++ -O3 --std=c++17 main.cpp && ./a.out > results.csv
```

Question 2

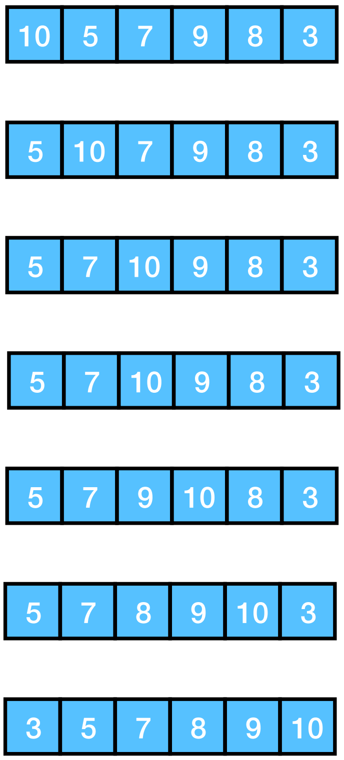

Figure 3: Insertion Sort

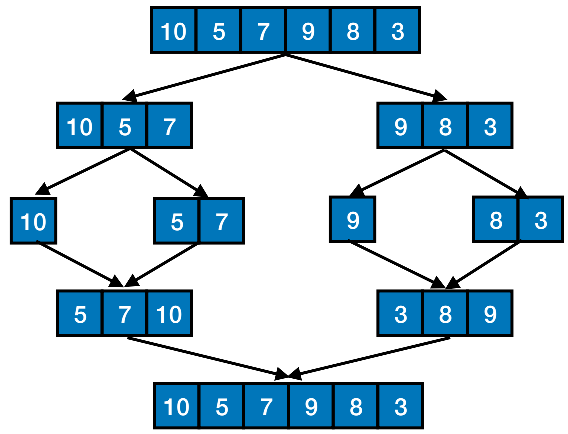

Figure 4: Merge Sort

Question 3

True, True, False, False, True

Question 4

Method from *Algorithms*[1] by Jeff Erickson

Master method tells us the running time is, if given $T(n) = rT(n/c) + f(n)$, we have,

$$T(n) = \sum_{i=0}^{\log_c n} r^i f(n/c^i) \tag{1}$$

There are three common cases where level-by-level sum is easy to evaluate,

(a) **Decreasing**, if the sum is a decreasing geometric series – every term is a constant factor smaller than the previous term, then

$$T(n) = O(f(n))$$

In this case, the sum is dominated by the value at the root of recursion tree.

(b) **Equal**, if all terms in the sum is equal, we have

$$T(n) = O(f(n) \cdot L) = O(f(n) \log n)$$

(c) **Increasing**, if the sum is an increasing geometric series – every term is a constant factor larger than the previous term, then

$$T(n) = O(n^{\log_c r})$$

In this case, the sum is dominated by the number of leaves in the recursion tree.

Therefore,

(a) $c = 2, r = 8, f(n) = n$,

$$T(n) = \sum_{i=0}^{\log_c n} r^i f(n/c^i) = \sum_{i=0}^{\log_2 n} 8^i \left(\frac{n}{2^i}\right) = \sum_{i=0}^{\log_2 n} 4^i n \leq n^3 = O(n^3)$$

This falls into case (c).

Furthermore, $\sum_{i=0}^{\log_2 n} 4^i n \geq n \log_2 n = \Omega(n \log_2 n)$ when taking into account $i = 0$ for all i and in total there are only $\log_2 n$ items.

So we can conclude that,

$$\Omega(n \log_2 n) \leq T(n) \leq O(n^3)$$

[1] http://algorithms.wtf

(b) $c = 2, r = 8, f(n) = n^2$,

$$T(n) = \sum_{i=0}^{\log_c n} r^i f(n/c^i) = \sum_{i=0}^{\log_2 n} 8^i \left(\frac{n}{2^i}\right)^2 = \sum_{i=0}^{\log_2 n} 2^i n^2 \leq n^3 = O(n^3)$$

This falls into case (c).

Furthermore, $\sum_{i=0}^{\log_2 n} 2^i n^2 \geq n^2 \log_2 n = \Omega(n^2 \log_2 n)$ when taking into account $i = 0$ for all i and in total there are only $\log_2 n$ items.

So we can conclude that,

$$\Omega(n \log_2 n) \leq T(n) \leq O(n^3)$$

(c) $c = 2, r = 8, f(n) = n^3$,

$$T(n) = \sum_{i=0}^{\log_c n} r^i f(n/c^i) = \sum_{i=0}^{\log_2 n} 8^i \left(\frac{n}{2^i}\right)^3 = \sum_{i=0}^{\log_2 n} n^3 = n^3 \log_2 n = \Theta(n^3 \log_2 n)$$

This falls into case (b).

(d) $c = 2, r = 8, f(n) = n^4$,

$$T(n) = \sum_{i=0}^{\log_c n} r^i f(n/c^i) = \sum_{i=0}^{\log_2 n} 8^i \left(\frac{n}{2^i}\right)^4 = \sum_{i=0}^{\log_2 n} \frac{n^4}{2^i} \geq n^4 = \Omega(n^4)$$

This falls into case (a).

Furthermore, $\sum_{i=0}^{\log_2 n} \frac{n^4}{2^i} \leq n^4 \log_2 n = O(n^4 \log_2 n)$ when taking into account $i = 0$ for all i and in total there are only $\log_2 n$ items.

So we can conclude that,

$$\Omega(n^4) \leq T(n) \leq O(n^4 \log_2 n)$$

Method from *Introduction to Algorithms* by Thomas H. Cormen et al.

The Theorem 4.1 (Master theorem) tells us, for recurrence $T(n) = rT(n/c) + f(n)$,

(a) If $f(n) = O(n^{\log_c r - \epsilon})$ for $\epsilon > 0$, then $T(n) = \Theta(n^{\log_c r})$.

(b) If $f(n) = \Theta(n^{\log_c r})$, then $T(n) = \Theta(n^{\log_c r} \log_c n)$.

(c) If $f(n) = \Omega(n^{\log_c r} + \epsilon)$ for $\epsilon > 0$, and if $rf(n/c) \leq af(n)$ for some constant $a < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Therefore,

(a) $c = 2, r = 8, f(n) = n$, this falls into case (a) when $\epsilon = 2$, so $T(n) = \Theta(n^3)$.

(b) $c = 2, r = 8, f(n) = n^2$, this falls into case (a) when $\epsilon = 1$, so $T(n) = \Theta(n^3)$.

(c) $c = 2, r = 8, f(n) = n^3$, this falls into case (b), so $T(n) = \Theta(n^3 \log_2 n)$.

(d) $c = 2, r = 8, f(n) = n^4$, this falls into case (c) when $\epsilon = 1$ and $a \leq 0.5$, so $T(n) = \Theta(n^4)$.

Appendix

CPP Code

```cpp
#include <chrono>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

template <class Iterator, class T>
inline auto populate_numbers(Iterator first, Iterator last, T value,
                             bool ascending = false) {
  while (first != last) {
    *first++ = value;
    ascending ? value++ : --value;
  }
}

template <typename T>
auto insertion_sort(std::vector<T> &vec) {
  for (auto j = vec.begin() + 1; j != vec.end(); j++) {
    auto key = *j;
    auto i = j - 1;
    while (i >= vec.begin() && *i > key) {
      *(i + 1) = *i;
      i--;
    }
    *(i + 1) = key;
    // std::cout << std::distance(vec.begin(), j) << ' ';
    // for (auto it : vec) std::cout << it << '\t';
    // std::cout << std::endl;
  }
}

template <class T1, class T2, class T3>
auto merge_vec(T1 first1, T1 last1, T2 first2, T2 last2, T3 d_first) {
  for (; first1 != last1; ++d_first) {
    if (first2 == last2) {
      return std::copy(first1, last1, d_first);
    }
    if (*first2 < *first1) {
      *d_first = *(first2++);
    } else {
      *d_first = *(first1++);
    }
  }
  return std::copy(first2, last2, d_first);
}

```

```cpp
template <typename T1, typename T2>
auto merge_sort(T1 source_begin, T1 source_end, T2 target_begin,
                T2 target_end) {
  auto range_length = std::distance(source_begin, source_end);
  if (range_length < 2) {
    return;
  }

  auto left_chunk_length = range_length >> 1;
  auto source_left_chunk_end = source_begin;
  auto target_left_chunk_end = target_begin;

  std::advance(source_left_chunk_end, left_chunk_length);
  std::advance(target_left_chunk_end, left_chunk_length);

  // for (auto dummy_begin = target_begin; dummy_begin != target_end;
  // ++dummy_begin) std::cout << *dummy_begin << '\t';
  // std::cout << std::endl;

  merge_sort(target_begin, target_left_chunk_end, source_begin,
             source_left_chunk_end);

  merge_sort(target_left_chunk_end, target_end, source_left_chunk_end,
             source_end);

  merge_vec(source_begin, source_left_chunk_end, source_left_chunk_end,
            source_end, target_begin);
}

template <typename T>
auto merge_sort(std::vector<T> &vec) {
  auto aux = vec;
  merge_sort(aux.begin(), aux.end(), vec.begin(), vec.end());
}

inline auto nanoseconds() {
  std::chrono::high_resolution_clock clock;
  return std::chrono::duration_cast<std::chrono::nanoseconds>(
             clock.now().time_since_epoch())
      .count();
}

int main() {
  auto n_repeat = 100;
  auto stepin = 2;
  for (auto size = stepin; size < 20000; size *= stepin) {
    std::vector<uint64_t> insertion_sort_runtimes(n_repeat);
    std::vector<uint64_t> merge_sort_runtimes(n_repeat);
    std::vector<int> vec(size);

```

```
    populate_numbers(vec.begin(), vec.end(), vec.size(), false);

    for (auto repeat = 0; repeat < n_repeat; repeat++) {
      auto vec_for_insertion_sort = vec;
      auto vec_for_merge_sort = vec;

      // insertion sort
      auto start_time = nanoseconds();
      insertion_sort(vec_for_insertion_sort);
      auto end_time = nanoseconds();
      insertion_sort_runtimes.push_back(end_time - start_time);

      // merge sort
      start_time = nanoseconds();
      auto aux = vec_for_merge_sort;
      merge_sort(aux.begin(), aux.end(), vec.begin(), vec.end());
      end_time = nanoseconds();
      merge_sort_runtimes.push_back(end_time - start_time);
    }

    std::cout << size << ','
              << std::accumulate(insertion_sort_runtimes.begin(),
                                 insertion_sort_runtimes.end(), 0.0) /
                     n_repeat
              << ','
              << std::accumulate(merge_sort_runtimes.begin(),
                                 merge_sort_runtimes.end(), 0.0) /
                     n_repeat
              << std::endl;
  }

  // std::vector<int> vec{10, 5, 7, 9, 8, 3};
  // auto vec1 = vec;
  // auto vec2 = vec;
  // insertion_sort(vec1);
  // std::cout << std::endl;
  // merge_sort(vec2);

  return 0;
}
```

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    results = np.genfromtxt('results_nano.csv', delimiter=',')

    plt.figure()
    plt.plot(results[:,0], results[:,1]/1e3, label='Insertion Sort')
    plt.plot(results[:,0], results[:,2]/1e3, label='Merge Sort')
    # plt.yscale('log')
    plt.legend()
    plt.grid(which="both", linestyle='dotted')
    plt.xlabel('Problem Size')
    plt.ylabel('Execution Time [µs]')
    plt.title('Execution Time Comparison')
    plt.show()
```