ANALYSIS OF ALGORITHMS I

ASSIGNMENT I

BUĞRA EKUKLU

150120016

ekuklu@icloud.com

ISTANBUL TECHNICAL UNIVERSITY

FACULTY OF COMPUTER AND INFORMATICS

ASYMPTOTIC UPPER BOUNDS OF THE ALGORITHMS

Insertion Sort	O(n2)
Merge Sort	O(n log(n))
Pseudo-sorted Linear Search	O(n k)

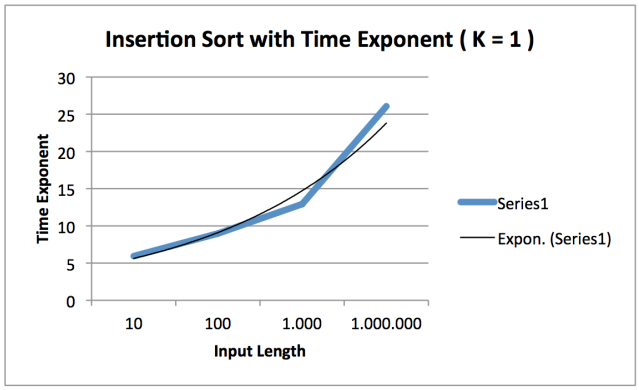
CAVEATS

The source code should be compiled with latest GNU C++ compiler or Clang compiler in order to make optimization efforts work. The code is fully-compliant with GNU C++11 (14) standard, it could be flawlessly optimized using -Ofast (or -Ounchecked option in Clang) argument option of the compiler. The program asserts the result array after sort methods work, to ensure the input data is properly sorted. In the essence of the optimization, the executable size may be inefficient.

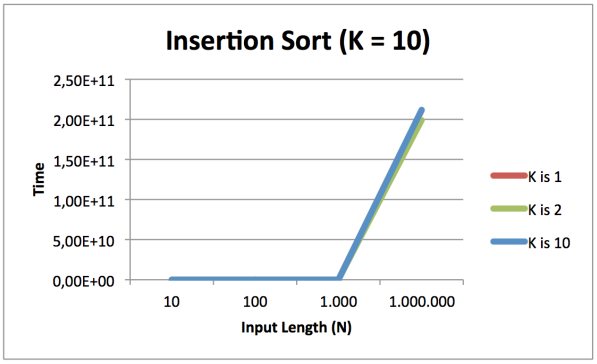
I. IMPLEMENTATION OF INSERTION SORT

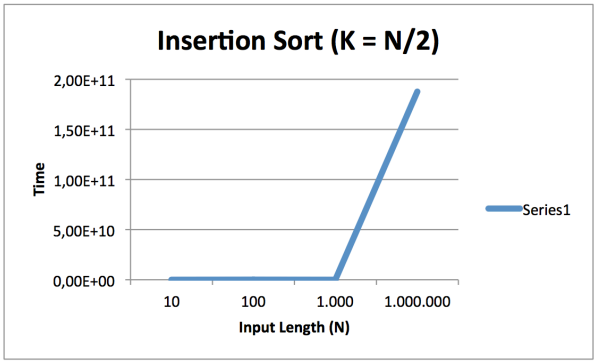
```
inline void sort() noexcept {
    for (FastUInt index = 0; index < length; ++index) {
        const T object = data[index];

        FastUInt reverseIndex = index - 1;

        while (reverseIndex >= 0 && data[reverseIndex] > object) {
            data[reverseIndex + 1] = data[reverseIndex];

            reverseIndex -= 1;
        }

        data[reverseIndex + 1] = object;
    }
}
```

The implementation of insertion sort is based on inserting each element to the proper locations. Since the loop will execute for each element in the array, it will execute *n* times. The block executed by the loop will also execute a while loop, which iterates through an linear iterable/ enumerable from reverse index, may execute between *0* to *n* times. Therefore, the best case will be *O(n)*.



However, the inner loop may execute *n* times at its worst, which makes the worst case scenario $O(n^2)$. Henceforth, the average case performance will be $O(n^2)$ swaps/ comparisons.



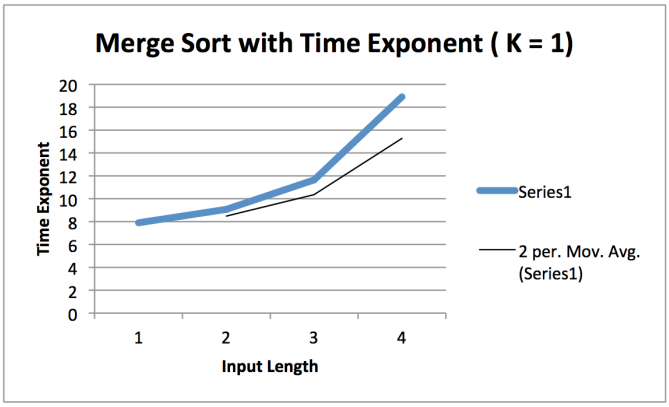


Insertion sort algorithm works on same address space with the input algorithm, no additional space is needed. Therefore, the optimal use case of this algorithm would be memory improficiency or lack of heap abstraction. In contrast of other algorithms compared in the article, this algorithm would use only stack to execute, without additional memory upfront.

II. IMPLEMENTATION OF MERGE SORT

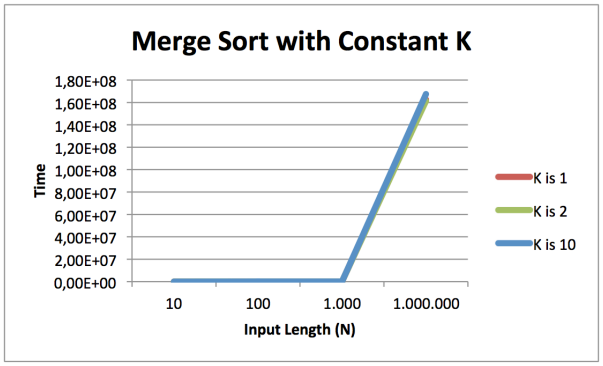
```
template <typename T>
class MergeSortableArray : public Array<T> {
    using Array<T>::data;
    using Array<T>::length;

    T *cache = nullptr;

    inline void mergeRange(UInt start, UInt end) noexcept {
        __unlikely (end == start + 1) return;

        UInt i = 0;
        UInt length = end - start;
        UInt middle = length / 2;

        UInt leftIterator = start, rightIterator = start + middle;

        mergeRange(start, start + middle);
        mergeRange(start + middle, end);

        for (i = 0; i < length; ++i) {
            if (leftIterator < start + middle && (rightIterator == end || data[leftIterator] > data[rightIterator])) {
                cache[i] = data[leftIterator++];
            } else {
                cache[i] = data[rightIterator++];
            }
        }

        for (i = start; i < end; ++i) {
            data[i] = cache[i - start];
        }
    }
```

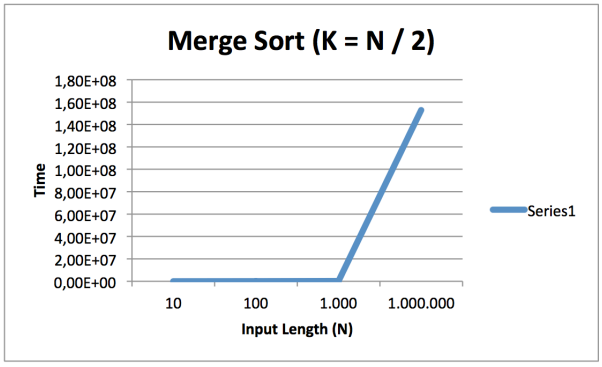
The merge sort implementation merges the input data to the pieces, and revokes itself on those dividents. The algorithm has $O(n\ log(n))$ complexity, since it divides the input data for $n\ /\ 2$ times and recurres itself on that data, and merges them with the n complexity.



The merge sort implementation utilizes divide and conquer technique, where the input data is divided and transformed by the algorithm. It recurses to the input data, each invocation affecting to the divisions. Therefore, this algorithm could also be implemented as parallelized, with multi-threading.

The implementation utilizes compiler based branch prediction hinting to the processor in first clause, where the condition only works at the end of the runtime of the algorithm. Lastly, it copies the data from the temporary data structure to main data structure (if it wants to mutates it). Therefore, the algorithm itself will occupy heap block similar to input data.





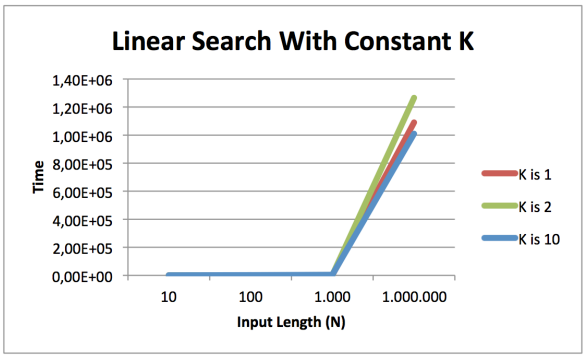
The optimal use case would be multi-threaded systems with large RAMs, especially with core unlockers or multipliers (hyper-threading). Since the algorithm does not utilize ALU of the CPU, a hyper-threaded system would dispatch each instruction to the execution cores without asynchronous locking. However, one drawback of this algorithm is it depends on the hardware it is utilized by, to illustrate, a floating-point vector could be sorted faster with a GPU or an array coprocessor rather than a CPU. Since it could be also flatly recursed, a GPU shader may optimize it to achieve large pipeline performance. Integer sorting operations with a CPU could be optimized by a memory allocation library (i.e. jemalloc, tcmalloc) and a proper integrated memory controller.

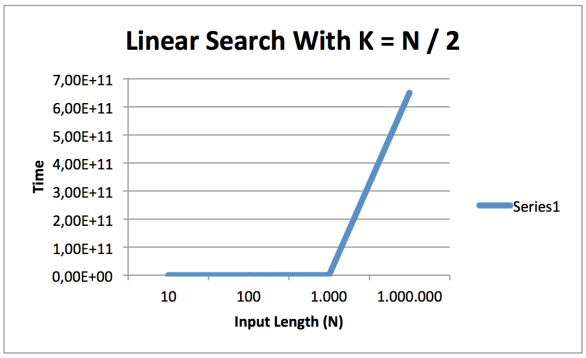
III. IMPLEMENTATION OF PSEUDO-SORTED LINEAR SEARCH

```
for (UInt i = 0; i < numberOfWarehouses; ++i) {
    buffer[i] = distances[i];

    if (buffer[i] > buffer[currentMaximumIndex] || i == 0) {
        currentMaximumIndex = i;
    }
}

for (UInt a = numberOfWarehouses; a < distances.getLength(); ++a) {
    if (distances[a] < buffer[currentMaximumIndex]) {
        buffer.insertAtIndex(distances[a], currentMaximumIndex);

        currentMaximumIndex = 0;

        for (UInt b = 0; b < numberOfWarehouses; ++b) {
            if (buffer[b] > buffer[currentMaximumIndex] || b == 0) {
                currentMaximumIndex = b;
            }
        }
    }
}
```

The linear search algorithm is a search algorithm, obviously, but it can used to sort with additional data structures. The implementation uses a buffer, which holds the pseudo-sorted output, where it uses this data structure as an cache of the minimum indexes.

The algorithm performs linear search on the input data in each iteration, therefore it will execute the search operation n times. The output of the search algorithm will be compared with the values in the temporary buffer, therefore there will be k comparisons in worst scenario, which yields the average performance of the algorithm as $O(n * k)$.





An optimal use case of this algorithm would be queries where the number of output is small. The output decreases, algorithm will perform up to *O (n)* average performance. The algorithm could be furtherly optimized using prefetch utilities of the compiler, reducing cache miss opportunity.

Insertion Sort vs. Merge Sort vs. Pseudo-sorted Linear Search

Algorithm	Complexity	Benchmark nr.	Number of Benchmarks	Input Length	Output Length	Runtime			Runtime Difference Relative To Base Run	Theoretical Time Difference Relative To Base Run	Deviation of Theoretical vs. Practical	Logarithmic Success Exponent
Units: Sub-SI	N/A	N/A	N/A	N/A	N/A	ns	s	min	N/A	N/A	N/A	N/A
Units: SI	N/A	N/A	N/A	N/A	N/A	s * 10^-9	s	s * 60^-1	N/A	N/A	N/A	N/A
Insertion Sort	*O(n^2)*	1	10	10	1	3,78E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		2	10	100	1	7,86E+03	0,000008	0,000000	2,08E+01	3,78E+02	5,50E-02	-1,26E+00
		4	10	1.000	1	4,15E+05	0,000415	0,000007	1,10E+03	1,43E+05	7,68E-03	-2,11E+00
		5	10	1.000.000	1	2,11E+11	211,208307	3,520138	5,59E+08	5,40E+07	1,03E+01	1,01E+00
		6	10	10	2	3,46E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		7	10	100	2	7,62E+03	0,000008	0,000000	4,54E-02	3,46E+02	1,31E-04	-3,88E+00
		8	10	1.000	2	3,24E+05	0,000324	0,000005	9,36E+02	1,20E+05	7,81E-03	-2,11E+00
		9	10	1.000.000	2	1,99E+11	199,045578	3,317426	5,75E+08	4,14E+07	1,39E+01	1,14E+00
		10	10	10	10	3,91E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		11	10	100	10	6,99E+03	0,000007	0,000000	1,79E+01	3,91E+02	4,57E-02	-1,34E+00
		12	10	1.000	10	2,09E+05	0,000209	0,000003	5,35E+02	1,53E+05	3,50E-03	-2,46E+00
		13	10	1.000.000	10	2,12E+11	211,821157	3,530353	5,42E+08	5,98E+07	9,06E+00	9,57E-01
		14	10	1.000	100	2,31E+05	0,000231	0,000004	N/A	N/A	N/A	N/A
		15	10	10.000	100	1,94E+07	0,019440	0,000324	8,42E+01	2,31E+05	3,65E-04	-3,44E+00
		16	10	100.000	100	1,55E+09	1,545199	0,025753	6,69E+03	5,33E+10	1,26E-07	-6,90E+00
		17	10	1.000.000	100	1,88E+11	187,675890	3,127932	8,13E+05	1,23E+16	6,60E-11	-1,02E+01
		18	10	10	5	2,31E+05	0,000231	0,000004	N/A	N/A	N/A	N/A
		19	10	100	50	1,94E+07	0,019440	0,000324	8,42E+01	2,31E+05	3,65E-04	-3,44E+00
		20	10	1.000	500	2,31E+05	0,000231	0,000004	N/A	N/A	N/A	N/A
		21	10	1.000.000	500.000	1,88E+11	187,675890	3,127932	8,13E+05	1,23E+16	6,60E-11	-1,02E+01
Merge Sort	*O(n.log(n))*	22	10	10	1	2,68E+03	0,000003	0,000000	N/A	N/A	N/A	N/A
		23	10	100	1	8,72E+03	0,000009	0,000000	3,25E+00	6,18E+04	5,27E-05	-4,28E+00
		24	10	1.000	1	1,14E+05	0,000114	0,000002	4,23E+01	1,24E+06	3,43E-05	-4,47E+00
		25	10	1.000.000	1	1,63E+08	0,162669	0,002711	6,26E+06	3,09E+09	2,03E-03	-2,69E+00
		26	10	10	2	2,27E+03	0,000002	0,000000	N/A	N/A	N/A	N/A
		27	10	100	2	1,03E+04	0,000010	0,000000	4,55E+00	5,24E+04	8,69E-05	-4,06E+00
		28	10	1.000	2	1,26E+05	0,000126	0,000002	1,21E+01	1,05E+06	1,16E-05	-4,94E+00
		29	10	1.000.000	2	1,61E+08	0,161223	0,002687	1,28E+03	2,62E+09	4,90E-07	-6,31E+00
		30	10	10	10	2,98E+03	0,000003	0,000000	N/A	N/A	N/A	N/A
		31	10	100	10	1,13E+04	0,000011	0,000000	3,78E+00	6,86E+04	5,52E-05	-4,26E+00
		32	10	1.000	10	1,13E+05	0,000113	0,000002	1,00E+01	1,37E+06	7,30E-06	-5,14E+00
		33	10	1.000.000	10	1,68E+08	0,167504	0,002792	1,48E+03	3,43E+09	4,33E-07	-6,36E+00
		34	10	1.000	100	9,06E+04	0,000091	0,000002	N/A	N/A	N/A	N/A
		35	10	10.000	100	1,82E+05	0,000182	0,000003	2,01E+00	2,09E+06	9,65E-07	-6,02E+00
		36	10	100.000	100	3,93E+05	0,000393	0,000007	4,34E+00	4,17E+07	1,04E-07	-6,98E+00
		37	10	1.000.000	100	1,59E+06	0,001591	0,000027	1,76E+01	6,26E+08	2,81E-08	-7,55E+00
		38	10	10	5	4,59E+03	0,000005	0,000000	N/A	N/A	N/A	N/A
		39	10	100	50	7,97E+03	0,000008	0,000000	1,74E+00	1,06E+05	1,64E-05	-4,78E+00
		40	10	1.000	500	2,15E+05	0,000215	0,000004	4,68E+01	2,11E+06	2,22E-05	-4,65E+00
		41	10	1.000.000	500.000	1,53E+08	0,152884	0,002548	3,33E+04	5,29E+09	6,30E-06	-5,20E+00
Pseudo-sorted Linear Search	*O(n*k)*	42	10	10	1	3,40E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		43	10	100	1	6,39E+02	0,000001	0,000000	1,88E+00	3,40E+04	5,53E-05	-4,26E+00
		44	10	1.000	1	1,80E+03	0,000002	0,000000	5,28E+00	3,40E+05	1,55E-05	-4,81E+00
		45	10	1.000.000	1	1,09E+06	0,001091	0,000018	3,21E+03	3,40E+08	9,43E-06	-5,03E+00
		46	10	10	2	4,46E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		47	10	100	2	6,50E+02	0,000001	0,000000	1,46E+00	8,92E+04	1,63E-05	-4,79E+00
		48	10	1.000	2	1,86E+03	0,000002	0,000000	4,18E+00	8,92E+05	4,69E-06	-5,33E+00
		49	10	1.000.000	2	1,27E+06	0,001266	0,000021	2,84E+03	8,92E+08	3,18E-06	-5,50E+00
		50	10	10	10	3,16E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		51	10	100	10	1,45E+03	0,000001	0,000000	4,57E+00	3,16E+05	1,45E-05	-4,84E+00
		52	10	1.000	10	3,62E+03	0,000004	0,000000	1,15E+01	3,16E+06	3,63E-06	-5,44E+00
		53	10	1.000.000	10	1,01E+06	0,001010	0,000017	3,20E+03	3,16E+09	1,01E-06	-6,00E+00
		54	10	1.000	100	1,06E+05	0,000106	0,000002	N/A	N/A	N/A	N/A
		55	10	10.000	100	1,41E+05	0,000141	0,000002	1,33E+00	1,06E+11	1,25E-11	-1,09E+01
		56	10	100.000	100	1,39E+07	0,013895	0,000232	1,31E+02	1,06E+12	1,24E-10	-9,91E+00
		57	10	1.000.000	100	1,67E+08	0,166956	0,002783	1,58E+03	1,06E+13	1,49E-10	-9,83E+00
		58	10	10	500	4,65E+02	0,000000	0,000000	N/A	N/A	N/A	N/A
		59	10	100	5.000	7,44E+05	0,000744	0,000012	1,60E+03	2,33E+08	6,88E-06	-5,16E+00
		60	10	1.000	50.000	7,85E+07	0,078517	0,001309	1,69E+05	2,33E+10	7,26E-06	-5,14E+00
		61	10	1.000.000	500.000	6,51E+11	650,500262	10,841671	1,40E+09	2,33E+14	6,02E-06	-5,22E+00