INVESTIGATING SORTING OF MANY SIMILAR LISTS USING MEMOIZATION TECHNIQUES

Gonzalo Urroz
Master in Computer Science
School of Computer Science and Information Technology, RMIT University
Melbourne, Australia

Supervisor: Geoff Leach

April 2019

Abstract

TODO

Contents

1 Introduction

Sorting data is one of the most important and studied problems in computer science, producing optimised algorithms, data structures and heuristics to solve different range of problems where sorting is involved. This is mostly the case nowadays when gigabytes of information is being produced, stored and analysed in real time and the need for fast sorting is crucial, mainly in sorting big size sets. In general, the algorithms considered for sorting data are the theoretical fastest ones which in an average scenario have a time complexity of O(n logn), but successfully applying this families of algorithms like Timsort[Pet] or Introsort[Mus97] (which are implemented in the standard libraries of Java and C++ languages correspondingly) depends greatly in the nature of the data to be sorted, then it is always important to analyse the data to be able to identify which algorithm or approach is best suited. In some cases these algorithms are not sufficient, so different optimising techniques have been produced, for example the usage of parallel programming or using GPU for faster results [SHG09].

Industry uses this kind of approach as part of external sorting, which handles big sets sorting by using external memory and involves reading the data not in a linear stream, but in segments that needs to be sorted separately in memory and later merged, which can prove to be faster than sorting the whole data as one big set. So applying algorithms that have worse time complexity to each of the segments, like Insertionsort, performs better than applying Mergesort or other general average case faster sorting algorithm. This big data set sorting optimised algorithms are being researched and published in an international competition [Gra]

A related kind of problem arises in the area of computer graphics[Arc15] , where on the display of computer generated scenes each object or polygon to be rendered requires multiple computations to be made as quickly as possible to achieve faster graphic rendering, preferably at real time rates. One of these calculation is the transparency value for each polygon in each pixel that screen is displaying. If we considered that nowadays a screen usually holds millions of pixels, therefore millions of calculations need to take place to run a technique called order independent transparency or OIT, which for each pixel must sort each list in depth order before the transparency operation can be performed. The current approach is to sort them by brute force, meaning that all of the lists are sorted independently.

This problem has been studied, but not in great length, where most of the solutions involves parallel sorting with multi-threading in CPU [HT02] and in GPU [HLWF17], implementing hybrid algorithms combining the merging power of Mergesort with the speed of comparison based algorithms, like Insertionsort.

This research then aims to explore if there is other approaches that could aid in the performance in sorting multiple similar list by taking advantage of the repetition of elements by using memoization, which is a technique to utilise pre-calculated results, in this case the sorted lists. This involves generating a signature to uniquely identify a repeated list, by using the best suited hash function to produce it.

Due to time constraint during the design and development of this study, an abstraction of the original problem is being studied with synthetic data in a way that it allows to simulate the behaviour of this techniques in different scenarios of the similarity between lists. In order to accomplish this, a testbed was developed that allowed the generation of different sets of data, according to the case that needed to be studied to outline the cases in which the potential of memoization on this problem could be of an advantage over the brute force approach.

Research Questions

Memoization is a optimisation method used in divide and conquer that consists in the storage of the result of a computation for a later utilisation when the same sub problem arises. This gives better performance because it saves the computation of the same sub problem. To achieve this the result must be stored, generally in a lookup table. Dynamic programming uses this kind of memoization to solve problems based in previous results. For our study, we will use a hash table that will hold the signature of a single list as the key and the reference to the sorted list as the value. The aim therefore is not to determine if this techniques optimises the problem, just because it is always faster performing the sorting operation only once, but to determine when does this technique has a better performance given the overhead of calculating the signature and looking up for the presence of the pre calculated value.

This work addresses the following research questions:

- **RQ1:** What are the benefits of utilising memoization techniques to improve repeated list sorting? When does it happen?
- **RQ2:** What are the benefit of utilising memoization techniques to improve list sorting when there is repeated blocks within similar lists? When does it happen?
- **RQ3:** Which hash function fits better to produce a signature over a list of integers?
- **RQ4:** What are some posibles enhancement techniques that can be applied to improve memoization in sorting?

2 Background

The problem that is being explored in this research is not the focus of many studies, as there is no direct study on the problem of sorting similar small list. This might be because the problem is not that common or there has not been a need to optimise this process.

For this study we do consider the different areas that are involved in the objectives or that could be of interest to address the questions.

2.1 Sorting Lists

The kind of algorithms can be grouped in different categories, with one of them being the internal sort. These are the kind of algorithms that work with all the data within the main memory through out the whole process.

- **Insertionsort:** In this category, *Insertionsort* is a comparison sorting algorithm that has an average and worse case time complexity of O(n^2), but its best case scenario is O(n). Being a stable algorithm, mainly because it sorts the elements in place, using only one auxiliary space in memory[Knu97a]. Compared to other comparison sorting, like Selectionsort and Bubblesort, all have the same time complexity but in practice the efficiency of this algorithm in the average case is superior to the other quadratic algorithms because there are less comparisons steps. Optimisations over this algorithms have been made, such as Shellsort[Knu97c], Librarysort[BFCM06] and also is part of hybrid algorithms as Timsort[Pet]. It is recommended to sort small list.

- **Mergesort:** The approach of this stable algorithm is sorting by divide and conquer, this by merging the result of the subsequent division of the input list into sub lists and recursively dividing and merging them. The standard implementation uses O(n) in space complexity and has average, best and worst case of O(n logn) in time complexity [Knu97b]. It is suitable for medium and large data set. Multiple optimisations has been made, mainly to reduce the space complexity and the cost of copying [HL88]. Other optimisation takes advantage of the nature of the divide and conquer process, by parallelising the execution with multiple threads [CNL$^+$08] on different architectures and cache optimised versions. Also it is extensible to be used in external sorting.

Even though these are some of the most used algorithms, they do not set the lower boundary for time complexity. Theoretically the lower bound of sorting in the best case scenario is O(n), given by the fact that each element of the input of size n has to be read at least once, but there are some internal algorithm that achieve almost linear time $O(n\sqrt{loglogn})$ [HT02] for average cases.

2.2 Segmented Sort

External sorting is used when the data to be sorted is too big to fit in main memory so it must be read from the source (network, hard drive) in separate chunks or segments that fits in memory. This kind of solution generally apply a hybrid approach in the sorting of the data in memory and merging the final result.

- **Segmented sort in GPU:** In the paper [HLWF17] a new solution is discussed using the advantage of parallel programming using GPU architecture in order to speed the sorting of segments. The solution proposed is based on data that is naturally segmented, and also the reason why it is interesting for us, the distribution of lengths of this segments tends to smaller ones. The authors then debate over the overhead that using multi threading in GPU and the best way to sort each segment individually. The process that they use is finally clustering each segments in buckets of same size and then sorting each one of them using Bitonic sort [Bat68] so it can be parallelised in GPU, finally to merge them continuously in memory using GPU registers and warp instructions to speed up the process.

2.3 Memory

As much of the literature reviewed exposed, when studying an algorithm not only the time complexity must be taken to account for the performance in speed, because the memory and the operations over it have a great impact on the practical implementation. As Figure 1 shows, memory on a modern end user computer follows a hierarchy, where the CPU register is the fastest one, even for multi core architectures. After that comes in the hierarchy the different levels of Cache in the CPU which gradually have more capacity but they are more expensive to read and write to, comparing in speed. When the CPU needs to load a particular memory location, it first checks if the required address is already in the cache (checking starts from the lowest level and continues to the highest one). If the required address is absent in the cache, then it must be loaded from the main memory. Such situation is called a cache miss. Algorithms should be constructed then to avoid cache misses whenever possible.

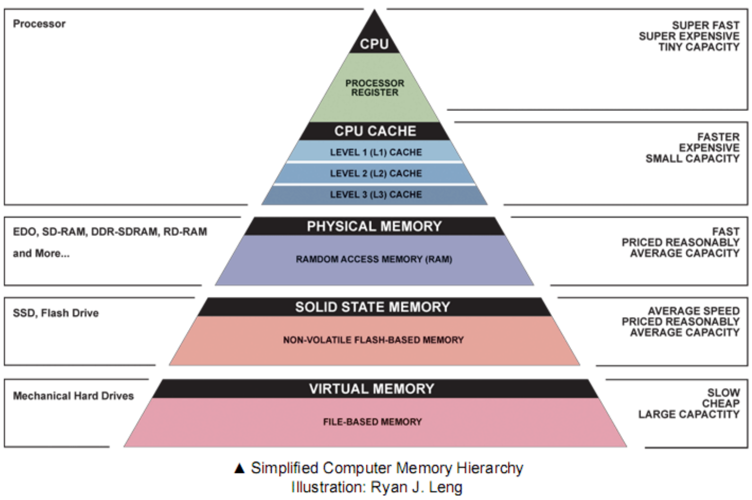


Figure 1: Memory hierarchy. (https://sites.google.com/site/cachememory2011/memory-hierarchy)

Several modification on known algorithms as Mergesort and Quicksort have been proposed to be *cache friendly*, improving the practical performance of such algorithms [LL99], [XZK00].

2.4 Memoization and Hashing

Memoization is a concept that has been considered for a long time and has been implemented in different situations [ABH03] such as dynamic programming and incremental computation. Most of the applications of this techniques are used to prevent recursive repetition within a function call, to prevent repetition over time, and to make persistent lookup tables [HM97]. This last approach consists in storing a pair of key and value, and being able to quickly fetch the value given the key if it is present in this table. This property is useful when the calculations to be stored as values are uniquely identified by some key.

It is then important to study the creation of this identity key that could allow us to uniquely map a complete list or a block of the list to be sorted as the key and store the sorted version as the value.

Hash functions allows mapping between an undefined set of data to a fixed sized data. Hash functions has many uses, from cryptography, file verification to finding duplicates records. A hash function often needs some type of properties depending on the application of it, because what defines a good hash function could be different if it is used to calculate indexes in a database or to use it as a cryptographic signature.

A hash function can be defined as universal if the function itself belong to a family that holds the property that the unique value that it evaluates to, h(x), has a probability of 1/M to have two different values x and y to evaluate to the same value, that is h(x) = h(y), which is called a collision. M is the size of all of the possible values the function can return [CW79].

The classic universal hash function defined in [CW79] is based on a prime number p bigger or equal to M. The constants a and b should be picked from {1, p-1} and {0, p-1} respectively, so to define:

$$ha, b(x) = ((ax + b) * modp) * modM \tag{1}$$

Furthermore a hash function h is considered strongly universal if the probability of having two different keys, x and y and the hash function for the event h(x) and h(y) is $1/M^2$. In other words, strong universality is that each key is hashed uniformly into M, and that every two distinct keys are hashed independently.

When the size of the universe of keys is smaller or equals to the posible outcomes (M), there are ways to construct what is known as a perfect hash function [Spr77], where there is no collisions for any given key. This might be the best solution to construct a lookup table, or hash table, to be able to apply the memoization technique. Given that the universe of the problem we are exploring consists of a pre defined length of lists and the range of the elements in the lists are limited, there should be a hash function that is perfect for this problem, but the size of M might be too big or the performance of it could be worse compared to a sufficiently strongly universal or even only universal hash function in this particular study.

3 Design and development

Given the exploratory objective of the goals of this research on the feasibility of using memoization on sorting lists, which by itself is a generalisation of a specific problem, the evaluation considers running each sorting algorithm on the same synthetic data set generated specially to understand the behaviour of it in specific cases. The goal is to measure the amount of time taken to sort the full data set, also considering the amount of memory needed to accomplish it.

3.1 Synthetic Data

In order for the data set to be representative of the original problem that this study took inspiration from, the amount of lists correspond to half of what could be the expected amount of data to be processed. This comes from the fact that the sorting of transparency vectors needs to be done for each pixel shown in a physical screen, so considering that nowadays most screen have a HD or Full HD resolution, which display a resolution of 1,920 x 1,080, we have around 2 millions pixels to sort. For the purpose of this study we will use half of the amount of available pixels (approximately 1 million) because it provides a good enough representation of a real case scenario but it still allows to run test in a environment without exceeding system memory or CPU specification.

Each of the million lists to be tested is generated to hold normal distributed random generated integer numbers that range between 0 and RAND_MAX in C++ (value depends on the implementation, but normally is 2.147.483.647), just to have a big enough pool to distribute it through the different lengths that are tested, which vary between cases. To generate the baseline on run time between sorting algorithms and also compare them using the new algorithms to use memoization in order to answer RQ1, the length of arrays considered were from 16 integers to 512. For RQ2 to be answered, a larger length of the arrays is needed because of the expected overhead that reading and merging blocks or sublist, so for smaller lists no better performance is expected given that it is already fast enough, so increasing the length size permits for an actual appreciation of what could be the benefits of applying this technique.

Finally, each array's data is generated accordingly to the scenario that it is being tested. Given that for RQ1 and RQ2 the behaviour that it is studied is on repeating list and similar list, it is then necessary to state how much repeating lists or how similar are them.

3.2 Test Cases

For RQ1 different scenarios will be tested on each array length size. Each scenario is as follow:

Repetition	*Description*
100%	Only one original list
75%	250.000 originals list
50%	500.000 originals list
25%	750.000 originals list
0%	All lists are different

For RQ2 the different scenarios are the following:

Repetition	*Description*
100%	Only one original block, repeated for the rest of the data
75%	250.000 originals blocks
50%	500.000 originals blocks
25%	750.000 originals list
0%	All blocks are different

For each of this scenarios, different length of blocks will be tested, still to be defined but given the preliminary results, choosing small length blocks will produce an overhead in memory and in operations and choosing a higher block will make convert it into a RQ1 scenario.

3.3 Hash Function

In order to consider which hash function to implement the universe of posible inputs has to be considered. We are only working with sets of integers with elements between 0 and RAND_MAX, and each list (or a sublist) is a key or input for the hash function. Then the size of the universe of keys is the combination of the range of integers over the size of the lists $\binom{RAND\_MAX}{Maxlistlength}$, which in the environment where the test bed will be run, where max list length is 512, it is in the order of 10^{3611}.

Given this universe size, considering a perfect universal hash function is nearly imposible because the potential hash table need 10^{3611} elements to map to. In consequence, a strongly universal hash function will be considered.

As reviewed by [Tho19], one of the fastest technique to construct a strongly universal hash function for an integer set is to multiply and shift pairs of elements, given the following formula:

$$h_{a_0,a_{d-1},b}(x_0, x_{d-1}) = \left(\left(\sum_{i=0}^{d/2}(a_{2i} + x_{2i+1})(a_{2i+1} + x_{2i})\right) + b\right)[W - l, W). \quad (2)$$

Where d is the length of the set and it is assumed to be even. The set a is a set of uniformly random distributed integers in the range $[0, 2^w]$ where w is the amount of bits of the keys,b is just a constant in the range $[0, 2^w]$. The seed l is the bits of the hash value. W is at least the sum of w + l -1 or higher.

For a 32 bits keys, the values utilised are W= 64, w=32 and l = 32. Which considers the overflow of the multiplication of 32 bits integer.

Then the hash function proposed is named **PairMultiplyShift**, and has the following pseudo-code:

```
int hash(int[] data, int l, int[] a, int b, int w, int d) {
    hash = 0;
    for(int i = 0; i < d/2; i++) {
        pairResult =  (a[2i] + data[2i+1]) * (a[2i+1] + data[2i]);
        hash +=  (pairResult + b) >> (w-l);
    }
    return hash;
}
```

It is then stablished that this kind of hash function has a time complexity of O(n) and space complexity of O(l). For educational purpose and to understand the impact of choosing the correct hash function, other hash functions will be tested, even though it is known before hand that some of them are not the best solution to the problem.

The following hash function are compared:

- **MD5:** This is a cryptographic function [Riv92] which is no longer secure [WY05]. The implementation is given by the standard C++ library *CommonCrypto*.
- **Sum:** The naive approach of just adding each element of the input list. The objective is to study the collisions behaviour on simple approach.
- **MurMur3:** A non-cryptographic hash function suitable for general hash-based lookup. Implemented based on the source code available online [App].

3.4 Implementation of Insertionsort and Mergesort

It is important to discuss the implementation of the sorting algorithms as it impacts directly on the performance and the extensibility of it.

For Insertionsort, the implementation used is the straight forward iterative approach, comparing tuples starting from the second position and moving to the end of the list.

The case for Mergesort is more interesting, because the natural approach is to have depth first recursive calls. The way that was implemented in this study is by an iterative double loop that operates in place and merges the arrays in pair, doubling the range of each tuple in each loop. The advantages for this choice of implementation is to avoid the overhead of the stack usage when recursive calling the methods and the ability to access in place to all elements, which gives the opportunity for further modifications when considering memoization over the algorithm.

3.5 Repeating lists

3.5.1 Constructing hash table with sorting

The theoretical behaviour for each of the sorting algorithm gives us an idea on the amount of overhead that the construction and execution of the look up in the hash table has to take.

For Insertionsort, the average time complexity is O(n^2), being n the length of the list and if M is the total amount of lists to sort in brute force case, and M′ the amount of unique lists to sort when using memoization, then it follows that:

$$M * O_{sorting}(n^2) \geq M' * O_{sorting}(n^2) + Overhead \tag{3}$$

Which implies

$$(M - M') * O_{sorting}(n^2) \geq Overhead \tag{4}$$

As the memoization technique involves the operation for each list of calculating the signature, looking up each, and insert only the ones that are unique, then:

$$Overhead = M * O_{hashing} + M * O_{lookup} + M' * O_{insert} \tag{5}$$

Accordingly the implementation of the hash table then has to be constant for both the search of a key and in the insertion of a new tuple. Other operations as delete or edit are of no concern because they do not apply for the use of this memoization technique. Then constructing the hash table is a matter of using the correct data structure. For this case the choice was to use an unordered map, which is a structure that holds the pair (key, value) and it has O(1) for the search, insert and delete operation and is implemented using the hash table class in C++. Other options involves the usage of trees, which have O(n logn) operations.

In consequence for the memoization technique to be beneficial it must stand that:

$$(M - M') * n^2 \geq M * O_{hashing} + M' \tag{6}$$

From this, the following cases can be defined:

- **All lists are the same:** This means M′ = 1, holds clearly that (M - 1) * $O_{sorting}(n^2)$ $\geq M * O_{hashing}(n) + 1$

- **None of the lists are the same:** This means M′ = M. Then there is no benefit as $0 \geq M * O_{hashing}(n)$ + M is not true if M > 0

The same analysis can be made for the case of Mergesort where sorting is O(n logn). This theoretical analysis implies that at least in the best case there is a benefit, then in practice must be established the real benefit of using the following pseudo algorithm to apply the memoization using a hash table:

```
for(j = 0; j < M; j++) {
    signature = hashFunc.hash(data[j]); -> Generates signature for list
    sortedListIndex = hashtable.find(signature) -> Looks signature in hash table

    if(sortedListIndex not found)  -> New list to be sorted
        sortFunction.sort(data[j]);
        hashtable.insert(signature, j); -> Stores the reference to the sorted list
     } else  // List already sorted
        data[j] = data[sortedListIndex] -> Copy sorted list
}
```

3.5.2 Constructing optimised sorting

The intuition to this approach comes from the fact that when sorting there is the need to visit each element at least once, as in the current approach of calculating the signature, so merging both process may result in a faster approach. For each case the hash table holding the current calculated signatures need to be passed as a reference to the sorting method and queried as soon as the signature is calculated. If the signature is not present, the signature is stored in the hash table with the reference to the ordered list.

In Insertion sort, only after half of the array is processed the signature can be calculated, so it saves at least half of the operations. Different is the case of Mergesort where after the first pass of the merge step all elements have been visited and the signature can be calculated, thus saving al the subsequent merging steps.

3.6 Similar lists

3.6.1 Segmented Sorting

* Define pseudo algorithm

3.6.2 K Way Merging

* Define pseudo algorithm

3.6.3 2 Way Merging

* Define pseudo algorithm

3.6.4 4 Way Merging

* Define pseudo algorithm

3.6.5 Memory Optimised

* Define ways to improve the code to use less memory

3.6.6 Optimising for ordered segments

* Define pseudo algorithm

4 Results

4.1 Benchmarking

The first activity that was developed was to design and build a software framework that could guarantee correct and trustworthy calculation of the execution of certain algorithms. The first approach was to develop the software in Java language version 8, but it proved to be a not so reliable language for benchmarking given that the Java Virtual Machine environment on which the program runs, uses JIT compilation and handles memory through a garbage collector in a unpredictable way, thus when running the experiments the results vary in a big percentage. The first measure to mitigate this was to run each test ten times and calculate an average on the result. Also, to prevent the compilation of code throughout the test, the code had to be *warmed up* by executing it a number of times so it would be forcefully compiled.

Even after taking this safeguarding, the tests took a long time to run, and still the results were not as solid as it is required to confidently compare algorithms. For that reason, the code was re written in C++. With this new code the test where executed and it prove to give more concise and uniform results. This allowed to execute the test a couple of times without having a big difference (below 0.5%). For this reason the results presented are the best of three execution for each tests.

4.2 Mergesort vs Insertionsort

The first experiment executed was to compare both algorithm Insertionsort and Mergesort given the synthetic data set created, which comprehends all three scenarios (worst, best and average) using the same parameters for each one. As the literature confirms, Insertionsort has a better performance on smaller length list. By using the parameter ˝o2 to compile the code, better results are obtained and Insertionsort sorts in less time until length of 256 elements. After this length of list, Mergesort has a better performance.

The time taken to sort the lists in milliseconds is displayed in Figure 2. The average, best and worst scenario (all sorted and inverted sorted) for both algorithms are displayed and they exhibit the expected behaviour, this is O(n) for sorted array in Insertionsort and $O(n^2)$ for the average and worst case. Mergesort maintains the O(n logn) time complexity for all cases.

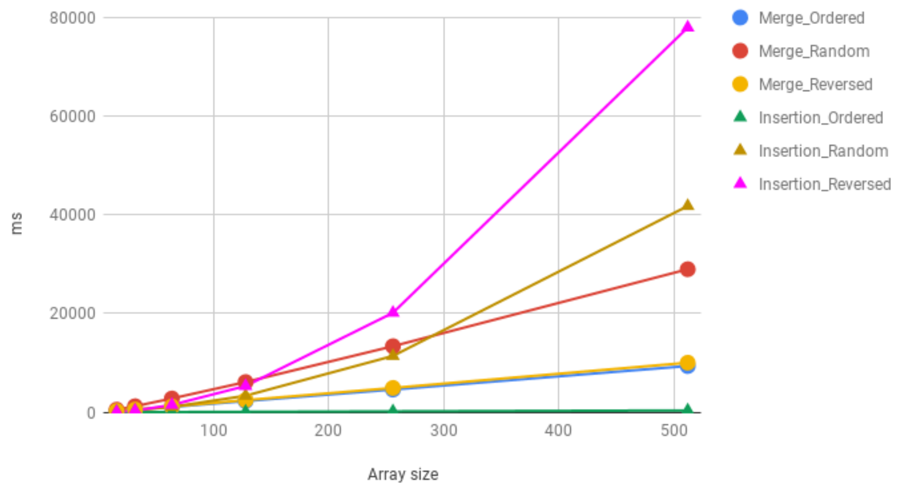


Figure 2: Insertionsort v/s Mergesort

4.3 Hash functions

TODO

- Table of time
- Table of memory.
- Table of collisions.

4.4 Memoization in repeating lists

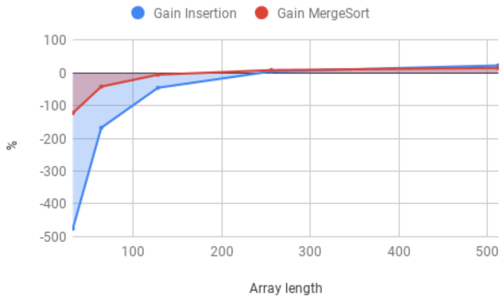
TODO Revisit with new tests that includes new integer range and new hash function. Update figures

The following result represent the test over normally distributed random data after applying the memoization technique using a hash table, and using the selected hash function to generate the signature of each of the unique list.
The results displayed in Figure 3 shows the performance comparison in a ratio to the average case of brute force sorting all list for each algorithm, where it can be seen that there is a severe penalty for using this approach in small length lists for Insertionsort. There starts to be an improved performance after lists with length of 128, mainly because the sorting performance below this length surpass the overhead of calculating and looking up

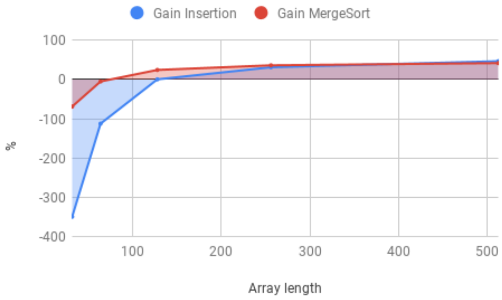
the signature of the list.
As it is expected, when there is no repetition, there is only overhead in applying this approach. The trend tends to be that the bigger the list length, the more gain there is. Can be concluded the that in a general case, there is a gain when there is 50% or more list repetition.

The same behaviour is shown for Mergesort, but the gain is not given much by the length of the array, as only for the amount of repeated lists given that the increase or decrease is not so drastic as with Insertionsort.

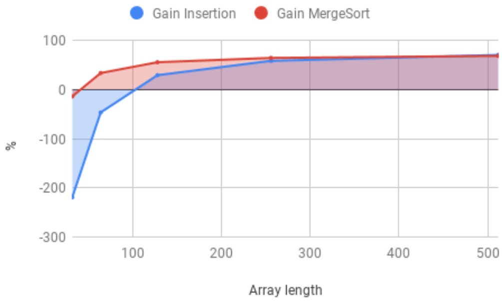
(a) 0 repetition



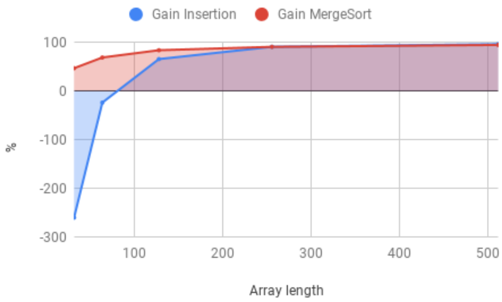
(b) 25% repetition



(c) 50% repetition



(d) 75% repetition



(e) 100% repetition

Figure 3: Memoization performance

TODO * Show figures of optimised sort.

4.5 Memoization in similar lists

TODO

- Graph of time for each approach
- Table of memory.

5 Conclusions

TODO

- Conclusion on hash functions (it depends on the data, even on the architecture of the computer) but using generic but proven universal hash function is a better approach if the data is not known before hand, where a perfect hash function is preferred.
- Conclusions of Memoization in repeating lists (It has gain starting from 50% repetition), compared to optimised sort is better.
- Conclusions of Memoization in similar lists ?.

6 Future Work

While this study has exploratory objectives reached by constructing the data to run the test cases in a synthetic way, there is a need to find a suitable problem where the nature of the data fits the conclusions on when there is benefit of applying the explored memoization techniques. The problem that served as inspiration [Arc15] is a possible candidate but some changes need to be done to fit the data of this problem, further more, some preliminary probing suggest that the data is not so repetitive to be able to benefit from the techniques studied.

Because the limitation on time of this study, there are multiple different improves or alterations to test different scenarios, like exploring the behaviour with optimised version of the sorting algorithms, or others like Radixsort or Shellsort. Also other memoization techniques can be constructed, like a probabilistic heuristic that samples of elements of the lists instead of reading the whole list to match for previous sorted list.

For the specific problem on choosing the correct hash function, there may be faster hash function that are not universal but still have no collisions within the problems data and makes the memoization techniques more beneficial.

Other approaches to determine the similarities between list can be explored other than using a hash table, like local sensitive hashing, which can cluster similar sets with a percentage of error.

References

[ABH03] Umut A Acar, Guy E Blelloch, and Robert Harper. *Selective memoization*, volume 38. ACM, 2003.

[App] Austin Appleby. MurMur3 murmur3. `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`. Accessed: 2019-05-01.

[Arc15] Jesse Archer. Sorting many similar small lists for faster order independent transparency. 2015.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968.

[BFCM06] Michael A Bender, Martin Farach-Colton, and Miguel A Mosteiro. Insertion sort is o (n log n). *Theory of Computing Systems*, 39(3):391–397, 2006.

[CNL+08] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[CW79] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[Gra] Jim Gray. SortBenchmark sortbenchmark.org. `http://sortbenchmark.org`. Accessed: 2019-04-15.

[HL88] Bing-Chao Huang and Michael A. Langston. Practical in-place merging. *Commun. ACM*, 31(3):348–352, March 1988.

[HLWF17] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, page 12. ACM, 2017.

[HM97] Marty Hall and J Paul McNamee. Improving software performance with automatic memoization. *Johns Hopkins APL Technical Digest*, 18(2):255, 1997.

[HT02] Yijie Han and Mikkel Thorup. Integer sorting in o (n/spl radic/(log log n)) expected time and linear space. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 135–144. IEEE, 2002.

[Knu97a] Donald Ervin Knuth. *The art of computer programming: sorting and searching. 5.2.1: Sorting by Insertion*, volume 3. Pearson Education, 1997.

[Knu97b] Donald Ervin Knuth. *The art of computer programming: sorting and searching. 5.2.4: Sorting by Merging*, volume 3. Pearson Education, 1997.

[Knu97c] Donald Ervin Knuth. *The art of computer programming: sorting and searching. Shell's method*, volume 3. Pearson Education, 1997.

[LL99] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[Mus97] David R Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[Pet] Tim Peters. Timsort python-dev sorting. `https://mail.python.org/pipermail/python-dev/2002-July/026837.html`. Accessed: 2019-04-15.

[Riv92] Ronald Rivest. The md5 message-digest algorithm. Technical report, 1992.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[Spr77] Renzo Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[Tho19] Mikkel Thorup. High speed hashing for integers and strings. *arXiv preprint arXiv:1504.06804v6*, 2019.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer, 2005.

[XZK00] Li Xiao, Xiaodong Zhang, and Stefan A Kubricht. Improving memory performance of sorting algorithms. *Journal of Experimental Algorithmics (JEA)*, 5:3, 2000.

7 Apendix