Algorithm Optimization

Seeking perfection

By

Fco Borja Lozano

Universidad Complutense de Madrid

September 22, 2019

Contents

1 Abstract

This paper includes some techniques and some advice about algorithm code optimization. The memory optimization section explores how to decrease cache misses and false sharing to increase the speed of memory accesses. The CPU optimization section explains many little tricks to decrease branching, which has a heavy toll on CPU speed, SIMD instructions and some advice on unnecessary operations. The last section compares the efficiency of data structures and shines some light on when each should be used. This section includes some test benches of some implementations of data structures found on Git Hub run on a Intel Core i7-7700K.

2 Introduction

Regardless of how good the pseudo-code is, the efficiency of the implementation can vary wildly depending on the data structures used and the internal structure of he code. The objective of this paper is to explore ways to optimize the implementation so it is close to optimal.

3 Efficient Memory Usage

Whether you are handling a massive amount of data or small amounts of data, doing so efficiently can be quite the challenge. In these cases, deciding the optimal data structures to use is crucial. But not only because of the available operations and their performance. The internal structure of the data type will determine whether the data will be stored contiguously or not, the size of the contiguous memory allocations, and the amount of cache misses per operation.

3.1 False Sharing

Whenever two processes are running simultaneously, memory conflicts arise. To prevent conflicts, if an object is being accessed by different processes, we have to keep the data updated so that no process uses outdated data. The process of updating the data of other CPU core when it is not necessary is called false sharing.

```
class A {
    char a,b,c;
public:
    void foo1(){
        for(int i = 0; i < 100; i++)
            a++;
    }
    void foo2(){
        for(int i = 0; i < 100; i++){
            b = c + 2;
            c++;
        }
    }
}
```

In this example, both functions can be run simultaneously independently. But without the proper compiler optimization's, it is very likely that there will be false sharing. To make the CPU know that neither is updating the data used in the other, we can make sure that a does not occupy the same cache line as b and c.

The common way to solve that problem is cache padding: doing some meaningless allocations between variables. That would force one variable to occupy a core's cache line alone, that way the CPU knows that each function uses different cache lines.

3.1.1 Compacting Structures

When we create classes, it can be worth taking into account how much memory it occupies and whether it is possible to decrease that space. All primitive types of a 64 bit systems occupy either 8, 16, 32 or 64 bits. And due to alignment required with primitive types, a

type that occupies 2 bytes has to start at an even byte, a byte that occupies 4 bytes has to start at a byte multiple of 4, and so on. In many languages, the compiler does not tamper with the order of the fields of classes, which leads to memory inefficiencies. For example, let's say we have the following class:

```
class A{
public:
    char c1;
    int i1;
    char c2;
    float f;
    double d;
    int i2;
    char a1,a2;

    void foo() {}
}
```

Memory in Bytes							
0	1	2	3	4	5	6	7
c1	Unused			i1			
c2	Unused			f			
d							
i2				a1	a2		

[1] Assuming that integers and floats occupy 4 bytes, and a double occupies 8 bytes.

As it can be seen, reordering the declarations would decrease the memory use by almost one fourth.

```
class A{
public:
    char c1,c2,a1,a2;
    int i1, i2;
    float f;
    double d;

    void foo() {}
}
```

Memory in Bytes							
0	1	2	3	4	5	6	7
c1	c2	a1	a2	i1			
i2				f			
d							

[1] Assuming that integers and floats occupy 4 bytes, and a double occupies 8 bytes.

Take into account that aside form chars, primitive types vary in size depending on the system, language and compiler. If you want complete control over the allocation of each field, you would have to work with bit fields or specific primitive types such as int_32 for a 32 bit integer.

3.1.2 Splitting Structures

It is fairly common in a class to have some fields that are only accessed under certain circumstances. There is a way to avoid having the operating system moving that data along every time one of its other operations are called. This is what splitting is. For example, if we had the following class:

```
class A{
    int a1,b1;
    char a2,b2;

public:
```

```
    int foo1(){
        return a1 + b1;
    }
    int foo2(){
        return a2 + b2;
    }
}
```

It might very well be the case that *foo1* is called a lot, while *foo2* is rarely called, or that there are parts of the code where one is more called and parts where the opposite happens. In those cases it would be more efficient to split the class.

```
class A{
    struct B{
        int a,b;
    }

    B *b_1, *b_2;
public:
    int foo1(){
        return b_1->a + b_1->b;
    }
    int foo2(){
        return b_2->a + b_2->b;
    }
}
```

That way, we don't move to the cache fields that are not necessary. This is also a common solution to false sharing.

3.1.3 Data Structures and Algorithms

The same data structure, with a different allocation layout have the same theoretical time complexity, but one can be considerably faster in practice. Let's see a very simple example, iteration over a bidimensional dynamic array. You might ask whether the iteration order matters or not. Theoretically, it shouldn't, but it does.

```
template<class T>
void func(bool isInverted, int sampleSize) {
    const int n = sampleSize;
    auto m = new T*[n];
    for (int i = 0; i < n; i++)
        m[i] = new T[n];

    if (isInverted)
        for (int j = 0; j < n; j++)
            for (int i = 0; i < n; i++)
                m[i][j]++;
    else
        for (int i = 0; i < n; i++)
```

```
            for (int j = 0; j < n; j++)
                m[i][j]++;

    for (int i = 0; i < n; i++)
        delete[] m[i];
    delete[] m;
}
```

With a simple benchmark we have the following outputs for $sampleSize = 10000$.

```
Long:     0.88 times faster.
Double:   0.81 times faster.
Integers: 0.85 times faster.
Char:     1.40 times faster.
Boolean:  1.37 times faster.
```

As you can see, cache misses caused by the layout of the data or the order of accesses do matter. The reason why the previous test showed that going column by column was slower than row by row is simple: the rows of the bidimensional array are contiguous in memory, and the columns are not.

To decrease cache misses we can implement a simple principles:

- Data that is usually accessed together should be allocated as close as possible to decrease cache misses.

Here is where data oblivious data structures come into play. A cache oblivious data structure is a data structure that works well on CPU's of all cache sizes. These structures try to obey the previous principle to create a data structure with the minimal amount of cache misses in each operation. It is important to note that what makes the data structure cache-oblivious is the implementation of the operations and data structure, and thus, any data structure can be made cache-oblivious. For example, there are multiple implementations of a binary search tree (BST) with different layouts. Let's compare some of them.

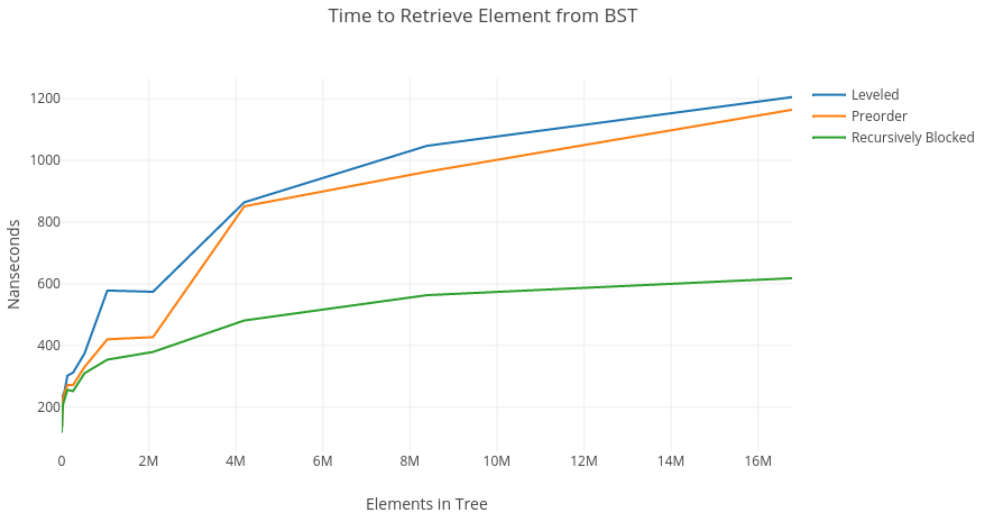


Figure 1: Comparison of the three layouts

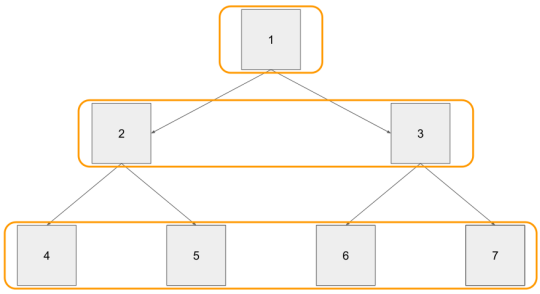


Figure 2: Breathfirst Layout

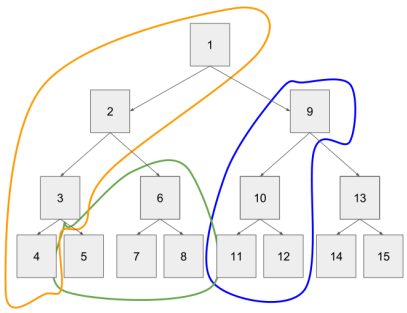


Figure 3: Deapthfirst Layout

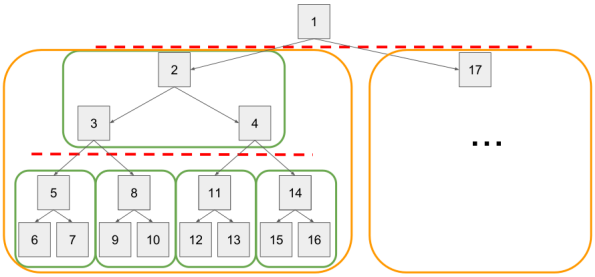


Figure 4: van Emde Boas Layout

Just as a data structure can be implemented to take advantage of the cache and decrease memory use, so can an algorithm. For example, let's explore the following dynamic programming algorithm:

$$M \in \mathcal{M}_{n\times n}(\mathbb{N})$$

$$M(i,j) = \begin{cases} 0 & \text{if } i = j \\ \max(M(i+1,j)+1, M(i,j-1)) & \text{if } j > i \end{cases}$$

This is very simple to code:

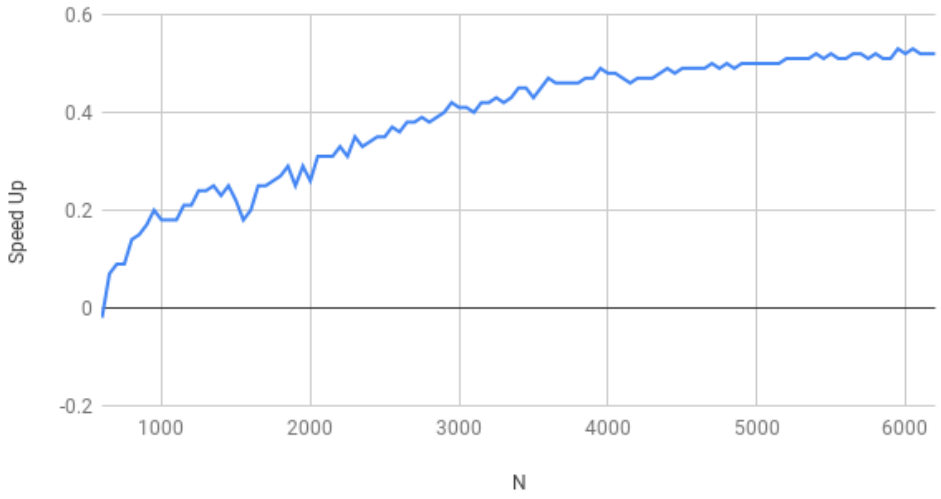
```
auto m = new int*[n];
for (int i = 0; i < n; i++)
    m[i] = new int[n];

for (int i = 0; i < n; i++)
    m[i][i] = 0;

for (int d = 1; d < n; d++)
    for (int i = 0; i + d < n; i++) {
        const int j = d + i;
        m[i][j] = std::max(m[i + 1][j] + 1, m[i][j - 1]);
    }

// do something

for (int i = 0; i < n; i++)
    delete[] m[i];
delete[] m;
```

But there is another non obvious approach to this. Using the fact that only positions of the upper triangular matrix are used and that we iterate over the diagonals we can do the following:

```
auto c = new T*[n];
for (int i = 0; i < n; i++)
    c[i] = new T[n - i];
for (int i = 0; i < n; i++)
    c[0][i] = 0;

for (int d = 1; d < n; d++)
    for (int i = 0; i+d < n; i++) {
        c[d][i] = std::max(c[d - 1][i], c[d - 1][i+1] + i);
    }

// do something

for (int i = 0; i < n; i++)
    delete[] c[i];
delete[] c;
```

If we compare them both, we have the following graph:

You might ask if that is caused by having less memory allocations, but even if we stored an entire matrix, and thus, $n-1$ diagonals instead of storing only the upper triangular matrix, the speed up is still considerable as n increases.



3.2 Conclusion

The time complexity does not take into account the cache misses. The same data structure can be exponentially slower due to cache misses if we don't allocate it's data correctly. Take into accounts the previous principle, that is a simplification of the principles of data locality, and prevent and false sharing and unnecessary memory accesses. Also try to

avoid making large contiguous allocations, because these are harder to make as the size increases.

4 Efficient CPU Usage

4.1 Decreasing Branching

One of the things that most affect the efficiency of the CPU is branching. When there is a conditional jump, depending on the predictions and whether the jump is taken or not, the CPU might have to undo or clear a lot of operation that it was doing simultaneously. Even tho the branch prediction is becoming better and better, it is still better to try to avoid conditional jumps when we can.

4.1.1 Unnecessary Conditional Jumps

let's see the next example of a very common example of IF statement. Whether it is expressed with tertiary operators or not, jumps will occur. But they are not necessary. We can make the same instruction as follows:

```
if (b > a)
    b = a;
```

```
bool if_else = b > a;
b = a * if_else + b * !if_else;
```

Another example:

```
bool a,b;
...
if (a) {
  if (b) {
    result = q;
  } else {
    result = r;
  }
} else {
  if (b) {
    result = s;
  } else {
    result = t;
  }
}
```

```
bool a,b;
...
static const table[] = {t, s, r, q};
unsigned index = (a << 1) | b;
result = table[index];
```

Nevertheless, I'll point out that CPU's branching predictions have come very far, and it is very hard to increase the efficiency of the code to a large extent by doing this; and that the compiler might do this optimizations like this already.

4.1.2 Loop Optimization

4.1.2.1 Unrolling

Sometimes, we can decrease the amount on jumps of a loop by unfolding it. For example, let's say we have the following code:

```
for(int i = 0; i < n; i++)
    foo();
```

If we know previously that n is a multiple of $k \in \mathbb{N}$. Then we can do the following:

```
for(int i = 0; i < n; i++){
    foo(i);
    i++;
    ... // k-2 foo() calls followed by i++
    foo(i);
}
```

And in doing so, we have decreased the amounts of jumps by k times. This is a very easy strategy to implement and a good practice.

4.1.2.2 Vectorization or Automatic Parallelization

Vectorization is the process of unrolling the loop into concurrent operations. For example, let's say you have the next piece of code:

```
for(int i = 0; i < n; i++)
    v[i] = std::sqrt(v[i]);
```

Because each iteration can be done independently, the CPU can do various iterations of the loop at once. When we talk about vectorization of loops, we normally mean automatic vectorization, which is done by the compiler. The CPU not always vectorices a loop when it can, this is why it is good practice to help the compiler notice that autovectorization is possible. For example, in the previous example we should do the following:

```
for(std::vector<float>::iterator it = v.begin(), e = v.end(); it != e; ++it)
    *it = std::sqrt(*it);
```

Most CPU's and compilers support automatic vectorization, but not all of them.

4.1.2.3 Loop Skewing

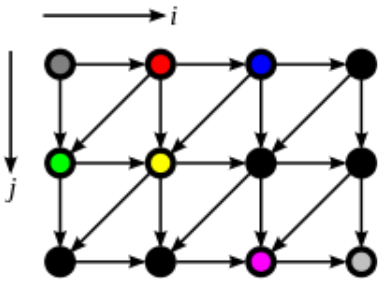
In the case of nested loops, we sometimes can increase the number of consecutive independent iterations. This makes the compiler able to optimize the code further and, maybe, even autovectorize the loop.

```
int a[n][n];
//...
for(int i = 1;  i < n; i++)
    for(int j = 1;j < n; j++)
        a[i][j] = a[i][j] + a[i][j-1];
```

```
int a[n][n];
//...
for(int i = 1;  i < n; i++)
    for(int j = 1;j < n; j++)
        a[i][j] = a[i-1][j] + a[i][j-1];
```

```
int a[n][n];
//...
for(int j = 1;  j < n; j++)
    for(int i = 1; i < n; i++)
        a[i][j] = a[i][j] + a[i][j-1];
```

```
int a[n][n];
//...
for (i = 1; i < n; i++) {
    for (j = 1; j < (i + 2) && j < n; j++)
      a[i][j] = a[i - 1][j] + a[i][j - 1];
```



It is easier to see it with a graph. In this last example, because we need the upper and left positions of the matrix to update each position, we go through the matrix using the diagonals in graph, from left to right. That way, every iteration over the same diagonal is independent to each other.

4.1.2.4 Repeated Wasted Operations

Most loops used nowadays are *for* loops. And it is regrettably common seeing something like:

```
const auto n = 15, k = 2;
//...
for(int i = 0; i < n/k; i++)
    // do stuff
```

```
const auto v = std::vector<int>({1,2,3,4});
//...
for(auto it = v.begin(); i != v.end(); i++)
    // do stuff
```

As you can clearly see, the *for* condition requires operations in every loop that have the return the same thing every call. It is much better practice avoiding this. In these examples, it would be easily solved by doing the following:

```
const auto n = 15, k = 2;
//...
const auto end = n/k;
for(int i = 0; i < end; i++)
    // do stuff
```

```
const auto v = std::vector<int>({1,2,3,4});
//...
auto end = v.end();
for(auto it = v.begin(); i != end; i++)
    // do stuff
```

4.1.2.5 Pre-increment or Post-increment

In most cases $i++$ and $++i$ are equivalent, but they are not the same. $i++$ returns the value before it is incremented and $++i$ returns the value after it is incremented. Because of this $i++$ requires extra copying and can slowly, but steadily, decrease the speed of your program. Nevertheless, if the functions are inlined, then the compiler will probably find whether this copying is or is not necessary and optimize it. Thus, the difference is only noticeable when the function is not inlined or the compiler does not optimize it.

4.2 Conclusion

The compilers have come a long way, but they are not perfect. To take advantage of all its optimization techniques the code must already be very optimized.

- Make constant all the variables that won't change its value.
- Declare the variables locally where they will be used, whether it is inside a loop or a function.
- Avoid calling the same function repeatedly when the returned value won't change.
- Unroll the loops as much as possible.
- Avoid placing consecutive dependant operations.

5 Efficient Decisions

5.1 Comparison of sorting Algorithms

The majority of algorithms require to have the data sorted. Sorting is a very time consuming process, choosing the best sorting method can have major effects on performance. Let's see a table of some of the most famous sorting algorithms and its time and space complexities.

	Time			
sort	Average	Best	Worst	Space
Mergesort	nlog(n)	nlog(n)	nlog(n)	1
Heapsort	nlog(n)	nlog(n)	nlog(n)	1
Quicksort	nlog(n)	nlog(n)	n^2	log(n)
Introsort	nlog(n)	nlog(n)	nlog(n)	log(n)
Smoothsort	nlog(n)	n	nlog(n)	1
Blocksort	nlog(n)	n	nlog(n)	1
Timsort	nlog(n)	n	nlog(n)	n
Bucketsort	n+k	n+k	n^2	n
Radixsort	nk	nk	nk	n+k
Countingsort	n+k	n+k	n+k	k

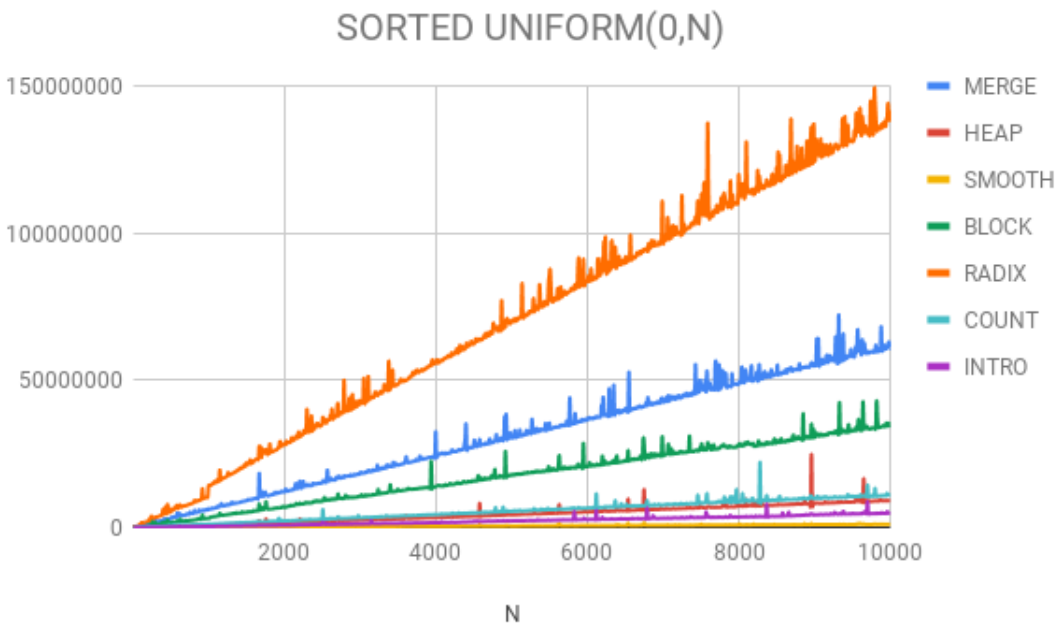
You probably have heard of the first ones, which have been the most used for a long time (introsort being the most used). But the last ones are less known, even though their time complexities look incredibly promising.

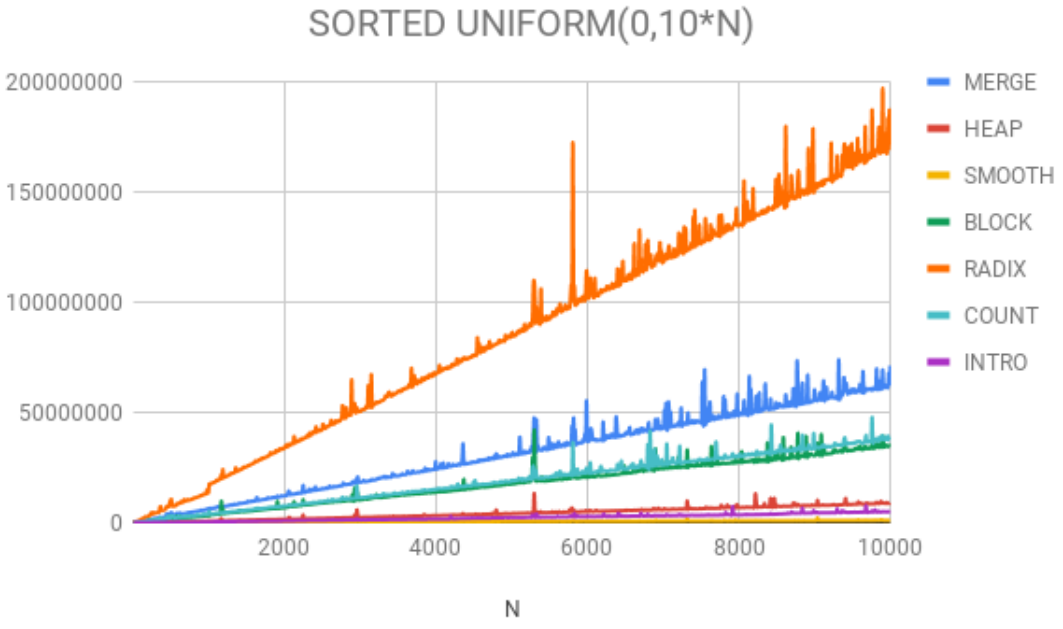
5.1.1 Why introsort is the most used

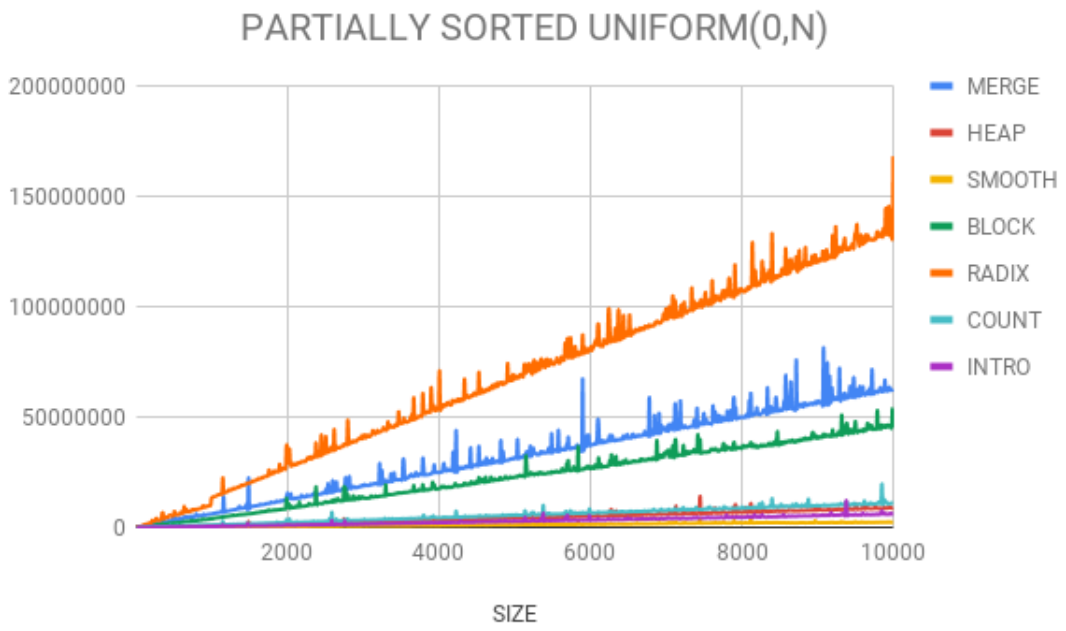
Introsort is basically and algorithm that starts quicksort and changes to heapsort halfway through if it is not going fast enough. Quicksort has been the algorithm of choice for unstable, in-place, in-memory sorting for so long because we can implement its inner loop very efficiently and it is very cache-friendly. But due to the fact that it's worse case scenario is $O(n^2)$, it switches to an heapsort when it detects that quicksort is taking too long.

5.1.2 Test Benches

This are some tests of the sorting algorithms on random integers. As you can see, cache misses and hidden constants have a heavy toll on the expected theoretical speed. In this examples, $c++$ implementations are used, where the implementation of introsort is in fact the *unbeatable* std::sort.


SORTED UNIFORM(0,N)
150000000
100000000
50000000
0
2000
4000
6000
8000
10000
N
MERGE
HEAP
SMOOTH
BLOCK
RADIX
COUNT
INTRO


SORTED UNIFORM(0,10*N)
200000000
150000000
100000000
50000000
0
2000
4000
6000
8000
10000
N
MERGE
HEAP
SMOOTH
BLOCK
RADIX
COUNT
INTRO


PARTIALLY SORTED UNIFORM(0,N)
200000000
150000000
100000000
50000000
0
2000
4000
6000
8000
10000
SIZE
MERGE
HEAP
SMOOTH
BLOCK
RADIX
COUNT
INTRO

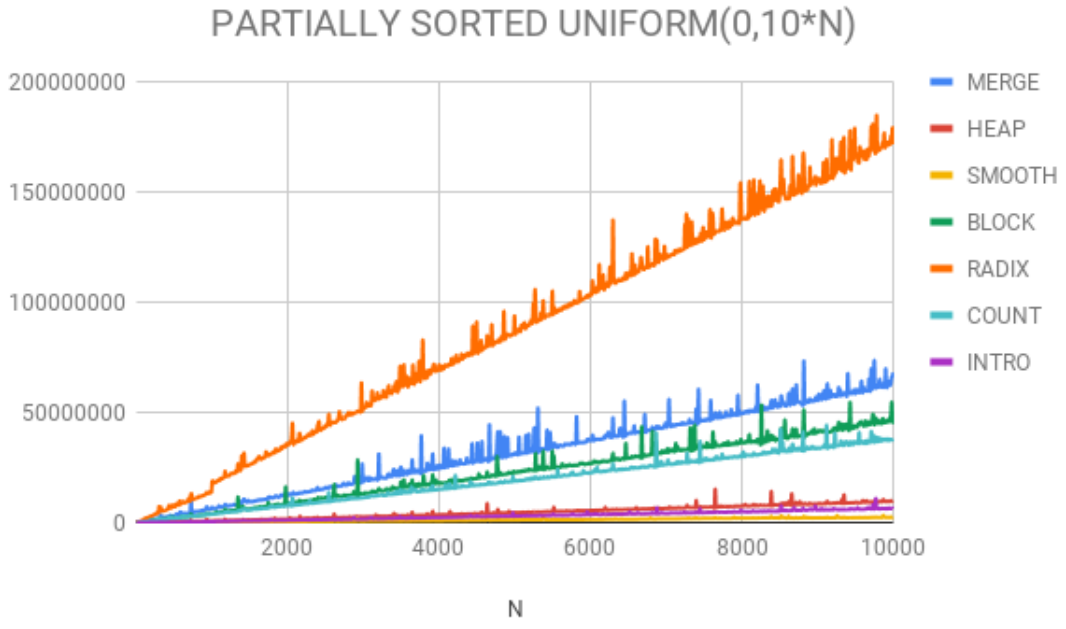

PARTIALLY SORTED UNIFORM(0,10*N)
200000000
150000000
100000000
50000000
0
2000
4000
6000
8000
10000
N
MERGE
HEAP
SMOOTH
BLOCK
RADIX
COUNT
INTRO

Figure 5

Table 1: Example of cases in which some algorithms beat the pre-existing and heavily optimized functions.

Test	std::sort (Introsort)	Timsort
Randomized (int)	978	1576
Randomized (double)	1068	1677
Reversed (int)	213	19
Reversed (double)	224	23
Sorted (int)	167	13
Sorted (double)	179	17
Partially sorted (int) 0	961	1478
Partially sorted (double) 0	1049	1594
Partially sorted (int) 1	713	634
Partially sorted (double) 1	761	732

[Partially sorted 0: each subarray of size 10 in 100,000 sequence is sorted.]
[Partially sorted 1: each subarray of size 1000 in 100,000 sequence is sorted]

5.1.3 Why you haven't heard about the theoretically fastest ones

Sometimes, the hidden constants and the memory accesses makes the asymptotically best algorithms, the slowest for relatively small amounts of elements. This is the case for smoothsort, in which the cache miss-rate and the hidden constants makes it slower than introsort in most real world scenarios.

5.1.4 When to use radixsort or countingsort

Radix sort is used to sort numeric keys. Its time complexity is $O(nk)$, where n refers to thee number of keys, and k to the length, in bits, of the largest number in the set of keys. This is why, radixsort is useful to sort, short keys (k is low). We also have to take into account, that as it requires more memory than other sorting algorithms, as n goes higher, the amount of cache misses will escalate.

Countingsort complexity is $O(n+k)$, where k is the range of the elements, which have to be non-negative. This means that it will be worse as the elements become more disperse.

5.1.5 Conclusion

There is no jack-of-all-trades sorting algorithm. Take into account the type of data you are sorting, whether it is already partially sorted or reversed, and the size of it, and decide

which algorithm suits it best. Having said that, keep in mind that in every language, there are preexisting and heavily optimized sorting functions that will probably run much faster than a normal implementation of any of these algorithms.

5.2 Comparison of Data Structures

5.2.1 Priority Queues

Priority queues are used a lot in algorithms and resource management, and have tremendous effects on their performance. Let's have a look at some of the possible implementations:

	Operations				
Heap	Min	Delete Min	Insert	Decrease Key	Merge
Binary	1	log(n)	log(n)	log(n)	n
Binomial	log(n)	log(n)	log(n)	1*	log(n)*
Fibonacci	1	log(n)*	1	1*	1
Strict Fibonacci	1	log(n)	1	1	1
Brodal	1	log(n)	1	1	1

* - Amortized time.

Binary heaps are by far the most used among them due to their efficiency and their simplicity, but they are surpassed asymptotically by other heap implementations, such as Fibonacci and Brodal heaps. Having said that, the number of elements required for these heaps to outperform binary heaps is enormous and that is why they are mainly used to execute algorithms over colossal graphs, such as Dijkstra's algorithm (in theory).

The implementations used in the test benches of these structures are common implementations that are not heavily optimized, excepting the binary heap implementation, that is the priority queue found in the STL.

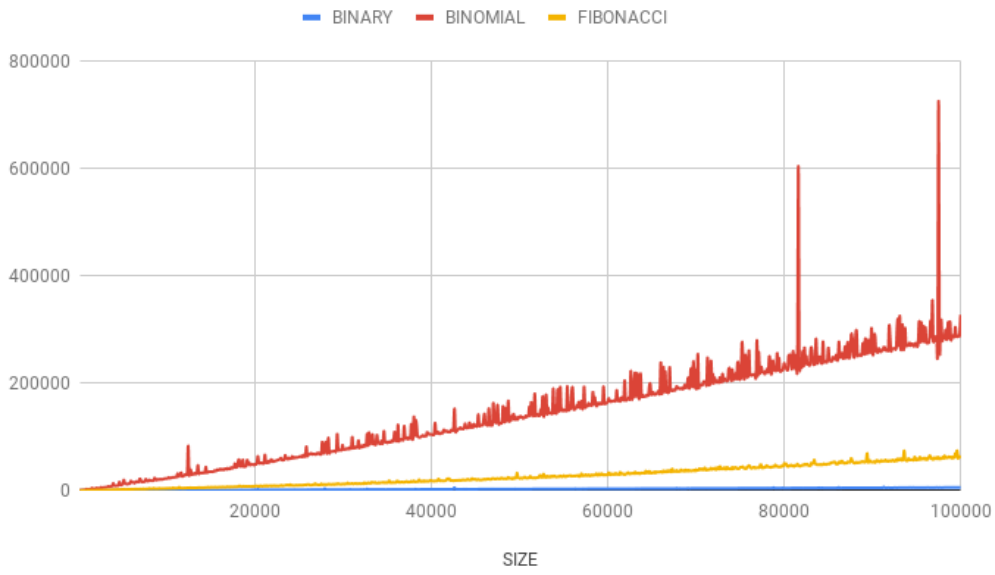


Figure 6: Pop (Microseconds)

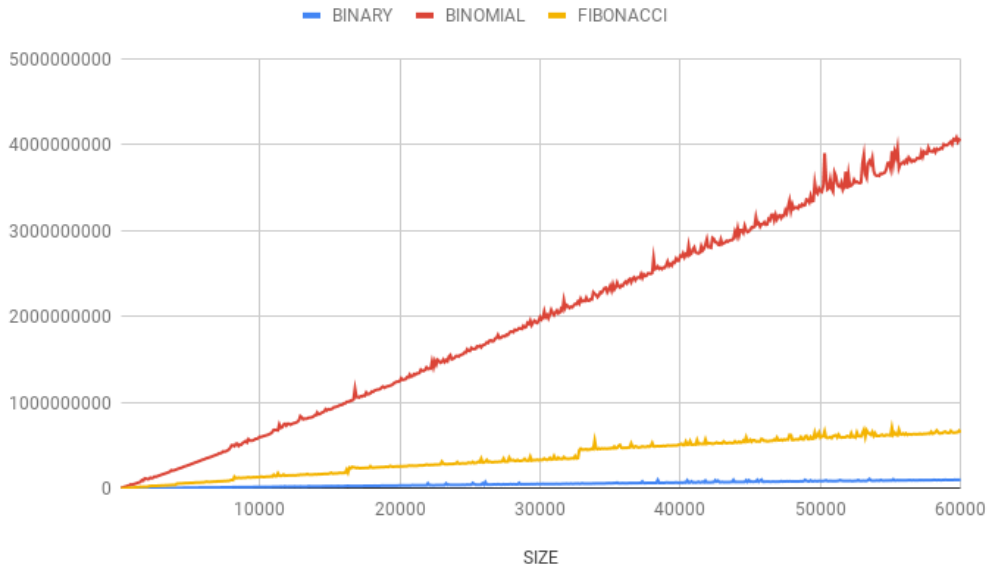


Figure 7: Insertion (Nanoseconds)

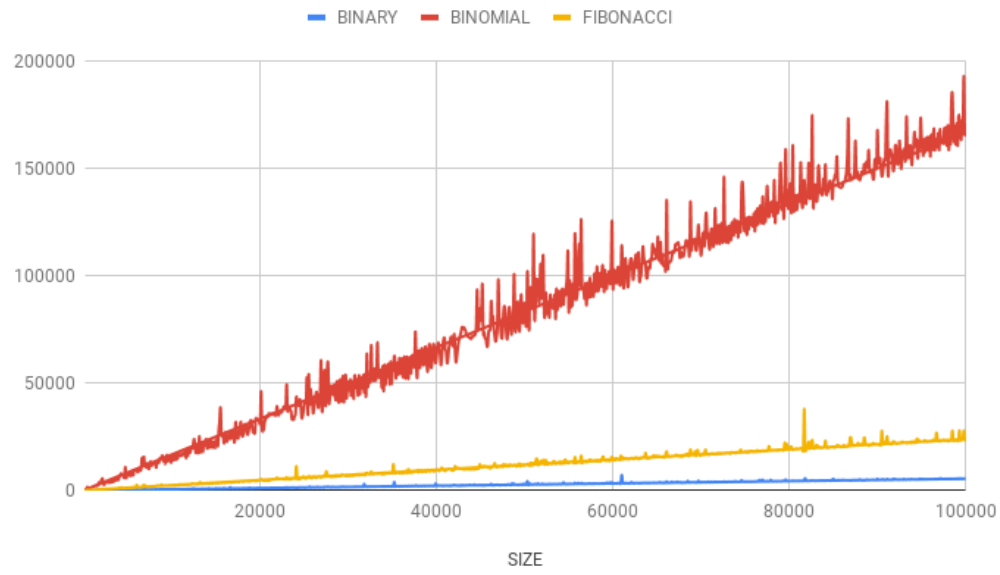


Figure 8: Random Insertions and Pops (Microseconds)

As expected, a well optimized binary heap implemented with am array, in this case the STL priority queue, due to fewer cache misses and its simplicity is faster than the other heaps. In this case, it would require a very optimized Fibonacci or binomial heap for it to be able to match the speed of the STL heap, and it would still need a large number of insertions and pops for it to happen.

5.2.2 String Storage

Text editors are very used to read data and store it. Let's see the most used data structures to do this in the next table.

	Operations					
D.S.	Insert	Remove	Concatenate	Index	Iteration	Split
Array	n	n	n	1	n	1
Gap Buffer	n[1]	n[1]	n	1	n	1
Rope	log(n)	log(n)	1	log(n)	n[2]	log(n)

[1] - Fixed insertion is O(1)
[2] - Not all implementations of ropes have linear iteration.

As the table suggests, deciding between a gap buffer and a rope is depends on which operations will be used the most. In normal circumstances, ropes' fast concatenation and the fact that it doesn't require large contiguous memory spaces makes them the best choice, even though their implementation is more complex and they use more overall memory space.

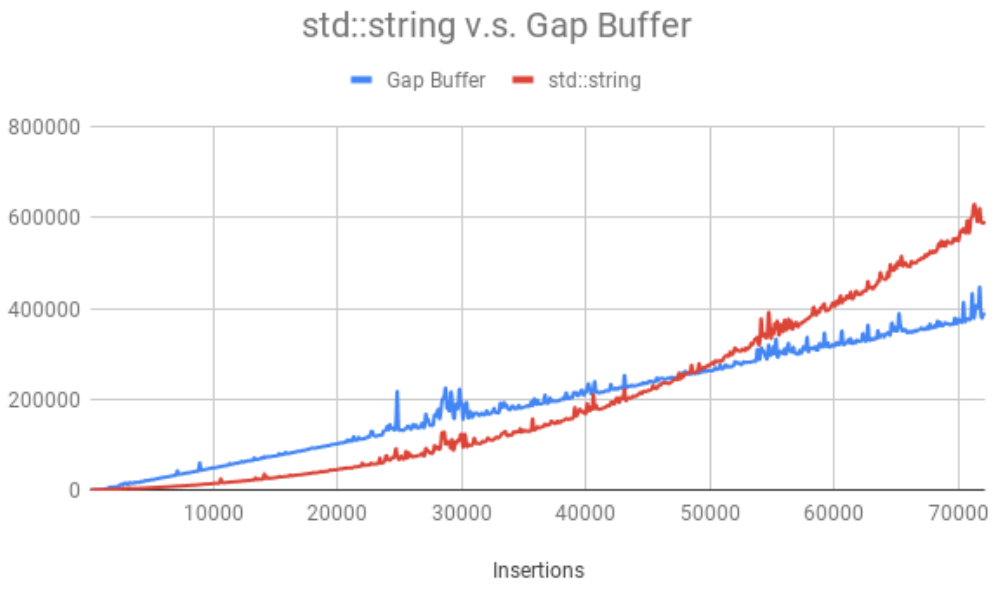


Figure 9: Repeated insertions at specific locations (microseconds)

As one can see, even if the internal implementation of a gap buffer is quite close to that of a char array, a simple implementation of a gap buffer using two std::strings can outperform a std::string when there are large insertions in specific positions.

5.2.3 Ordered Elements

There aren't that many data structures to keep elements ordered, in fact, only two are frequently used: ordered arrays and AVL trees. But there exists another data structure that rivals AVL trees: skiplists.

	Operations					
D.S.	Insert	Search	Remove	Index	Min	Max
Ordered Array	n	log(n)	n	1	1	1
AVL Tree	log(n)	log(n)	log(n)	log(n)	log(n)	log(n)
SkipList	log(n)	log(n)	log(n)	log(n)	1	log(n)

Skiplists work virtually the same as AVL trees, but the elements are kept in layered linked lists instead of a tree. Due to that, they can be easily iterated over and we dont have to keep it balanced. But not only that, they are also much better at handling concurrent accesses to the data. This is why it is highly recommended to use them instead of AVL trees. Keep in mind that sometimes, when you need to perform an algorithm over sorted data, keeping the elements unordered and ordering them once before executing the algorithm can perform faster than keeping the elements sorted from the get go.

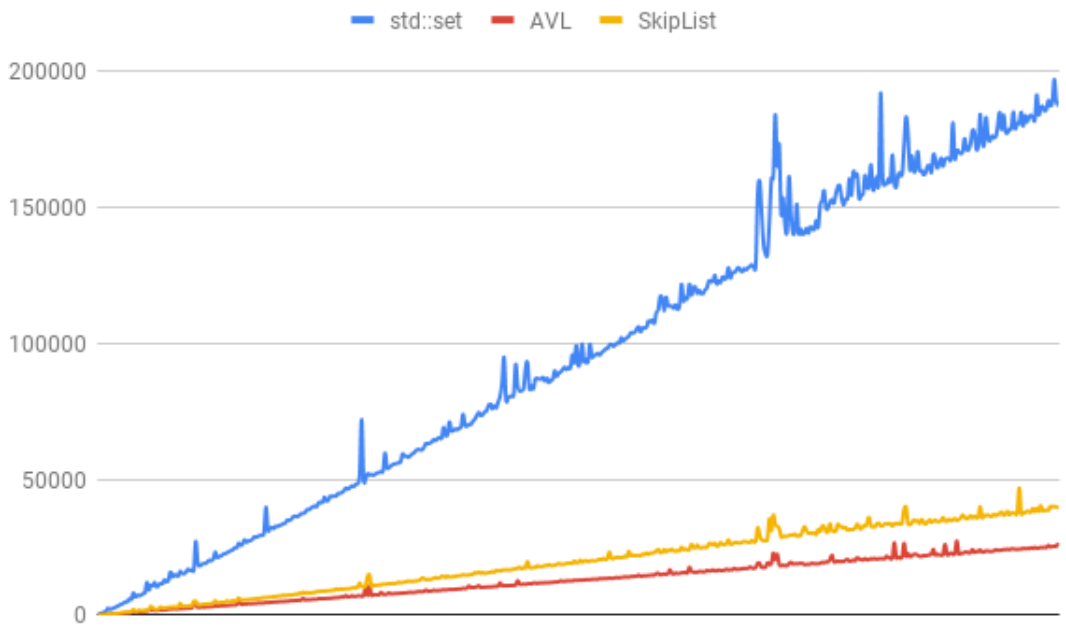


Figure 10: Insertions (microseconds)

5.2.4 Unordered Elements

When keeping the elements ordered is unnecessary, hash tables are by far the fastest data structures to store and retrieve elements. But their efficiency is strictly dependant on the hash function used, its speed and the amount of collisions between keys.

Let's explore the performance of some of of the most used hash functions in three tests.

Table 2: Speed and number of collisions of various hash functions.

	Tests		
Hash	Lowercase[1]	UUID[2]	Numbers[3]
Murmur	145ns 6	259ns 5	92ns 0
FNV-1a	152ns 4	504ns 4	86ns 0
FNV-1	184ns 1	730ns 5	92ns 0
DBJ2a	158ns 5	443ns 6	91ns 0
DJB2	156ns 7	437ns 6	93ns 0
SDBM	148ns 4	484ns 6	90ns 0
SuperFastHash	164ns 85	344ns 4	118ns 18742
CRC3	250ns 2	946ns 0	130ns 0
LoseLose	338ns 215178		

Even if unordered sets have loop up and insertion in O(1) and ordered ones in O(logn), when the amount of elements in the set is low, ordered sets are faster.

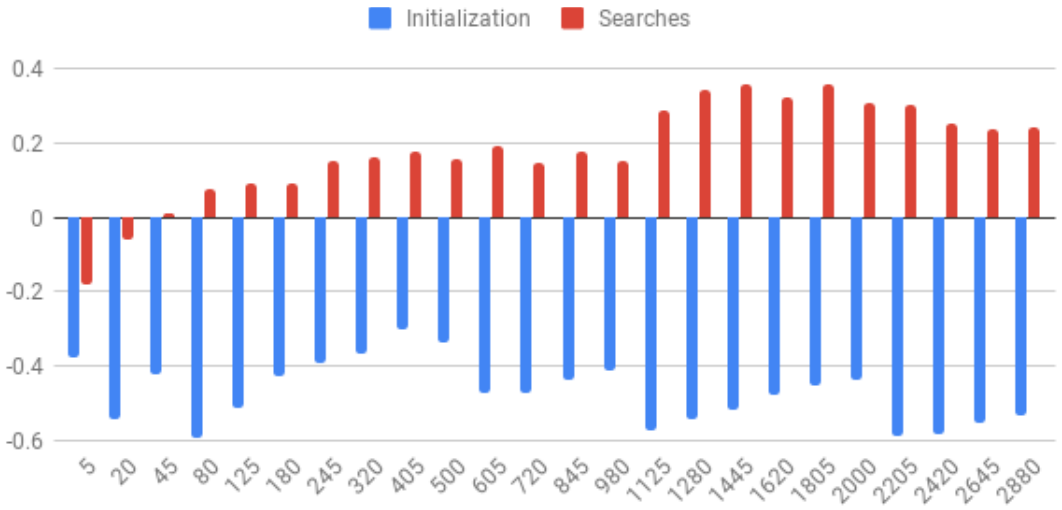


Figure 11: Speed up obtained using std::unorderd_sets instead of std::sets.

Creating and inserting the elements is, as you can see, much faster in a set, but searches are much faster in the unordered set. This is why we have to keep in mind what we need the set for and the number of elements it will have to decide the best data structure.

5.3 Conclusion

The speed of data structures depends heavily on the amount of cache misses and jumps. Do not use a data structure only because of the time complexity it has. Use the previous sections to implement efficiently data structures and keep yourself informed of the latest data structures and algorithms and to make informed decisions to keep your code optimized.

References

[1] False Sharing
https://parallelcomputing2017.wordpress.com/2017/03/17/understanding-false-sharing/

[2] What's false sharing and how to solve it
https://medium.com/@genchilu/whats-false-sharing-and-how-to-solve-it-using-golang-as-example-ef978a305e10

[3] Cache Oblivious Data Structures
https://en.wikipedia.org/wiki/Oblivious_data_structure

[4] Cache Oblivious Data Structures (Examples)
https://rcoh.me/posts/cache-oblivious-datastructures/

[5] Cache Oblivious Data Structures (Brief)
https://bryanpendleton.blogspot.com/2009/06/cache-oblivious-data-structures.html

[6] Cache Oblivious Algorithm
https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

[7] MIT Memory Hierarchy Models
https://www.youtube.com/watch?v=V3omVLzI0WE

[8] Comparison of Hash Functions
https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed#145633

[9] Tim Sort vs Introsort
https://www.gamasutra.com/view/news/172542/Indepth_Smoothsort_vs_Timsort.php

[10] SmoothSort
http://www.keithschwarz.com/smoothsort/

[11] Introducing Tim Sort
https://medium.com/@george.seif94/this-is-the-fastest-sorting-algorithm-ever-b5cee86b559c

[12] Wikisort
https://github.com/BonzaiThePenguin/WikiSort

[13] Counting Sort performance and Code
https://gist.github.com/jcomo/8214743

[14] Rope
https://en.wikipedia.org/wiki/Rope_(data_structure)

[15] Rope Performance
https://www.ibm.com/developerworks/library/j-ropes/index.html

[16] Gap Buffer Performance
https://www.codeproject.com/articles/20910/generic-gap-buffer

[17] Reduce IF Statements
https://www.captechconsulting.com/blogs/eliminate-branching-if-statements-to-produce-better-code

[18] Multiprocessing
http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/csci360/lecture_notes/chapter_07.pdf