Random Algorithms (RA-MIRI)
Assignment 2

Daniel Benedí

November 2022

1 QuickSelect

In this section, the algorithm QuickSelect designed by Hoare will be implemented and the number of comparisons made between array elements,$C_n^{(\alpha n}$, will be counted , so we can measure the average number of comparisons $\bar{C}_n^{(\alpha n)}$ and its standard deviation $\sigma_n^{(\alpha n)}$.

1.1 Implementation

From the `find` algorithm given by Hoare [2], we can define the following QuickSelect implementation:

The implementation provided uses the standard library of C++ (further detail in appendix A). To choose the pivot, we use a pseudo-random numbers engine provided by the standard library called `std::mt19937` (for implementation details check Makoto and Takuji paper [3]). We have chosen this engine instead of a linear congruential because with the right parameters it has the longest non-repeating sequence and we want to generate a lot of numbers, although it is slower and has greater state storage requirements. It makes use of `std::random_engine` which is a uniformly-distributed integer random number generator that produces non-deterministic random numbers or pseudo-random numbers if a non-deterministic source is not available. After generating the random (or pseudo-random) bits, they are passed to `std::uniform_int_distribution` so it can generate the index of the pivot that is going to be used. Because initializing the engine and the random device is highly cost task, we have used the `static` keyword, so it is only initialized once over all the experiments.

In order to get robust results from the experiments, the experiment has been realized 100 times shuffling the input vector every time so we could get a different permutation every time. Also, for each input size we have tried α values from 0 to 1 with a step of 0.01, so we could also get a fine-grained resolution of the costs. We have tried different input sizes in the range of $2 * 10^4$ to $2 * 10^6$, but not all the values in the range, just some of them. In order to use the whole

Algorithm 1 QuickSelect

```
procedure QUICKSELECT(A, j)        ▷ QuickSelect method to find the j-th element (α = j/n)
    min ← 1                                          ▷ First index of array
    max ← |A|                                        ▷ Last index of array
    C_n ← 0                                          ▷ Number of comparisons made
    k
    do
        a ← pick u.a.r a element from A
        l ← min, r ← max                             ▷ Partially ordered elements
        while l < r do        ▷ Order all the elements according to the pivot a
            while l ≤ max ∧ A[l] < a ∧ l < r do
                l ← l + 1, C_n ← C_n + 1
            end while
            C_n ← C_n + 1
            while r ≥ min ∧ A[r] > a ∧ l < r do
                r ← r − 1, C_n ← C_n + 1
            end while
            C_n ← C_n + 1
            SWAP(A[l], A[r])
        end while
        k = l
        if i < k then        ▷ If pivot is to the right of the looked for element
            max ← k − 1 i > k        ▷ If pivot is to the left of the looked for element
            min ← k + 1
        end if
    while i ≠ r
end procedure
```

CPU and decrease the time needed, we have made use of the library OpenMP to parallelize the experiments for each n.

We have also implemented the option to take $2 * t + 1$ elements and choose the pivot that falls closer to the j-th position in that sample. It is known that the expected number of comparisons is lower than the number of comparison of the previous method, but because of a lack of time we haven't been able to do the proper experimentation to add it in this report.

1.2 Results

In order to visualize our results, we have to take into account that we have two input variables and one output, the average cost $\bar{C}_n^{(\alpha n)}$. Theoretically, we know that $\frac{\bar{C}_n^{(\alpha n)}}{n} \approx 2-2(\alpha \ln \alpha+(1-\alpha)\ln(1-\alpha))$. The figure 1 shows this proportion depending on the input size, n and α

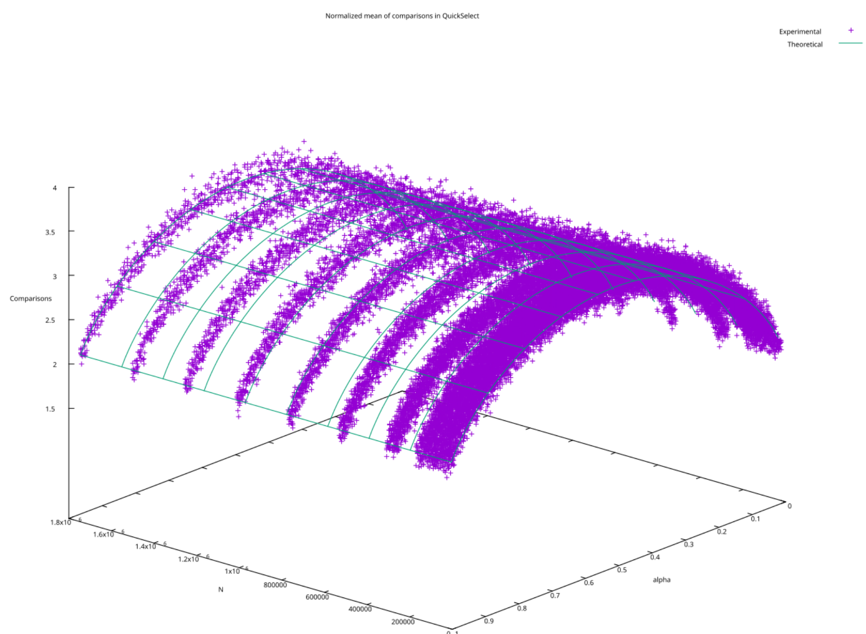


Figure 1: Average comparisons divided by the case, depending on the input size and α

We can observe that apparently the cost experimentally fits the cost theoretically. When we collapse the input size, n, dimmension, we obtain the figure 2. We can observe in it that it seems to follow the theoretical approximation quite nice, but moving around it.

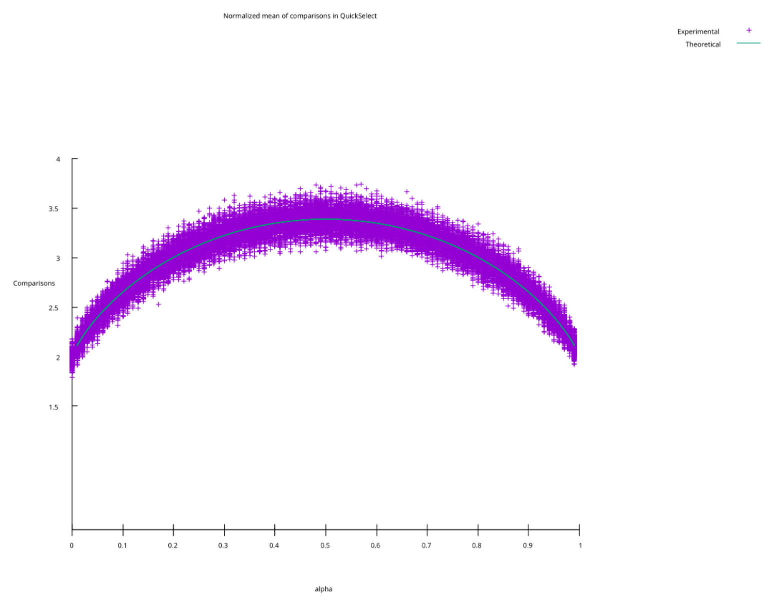
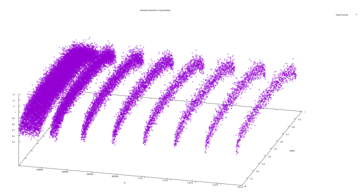
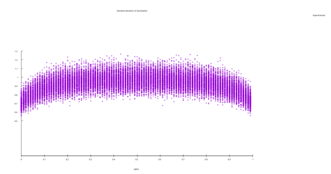


Figure 2: Average comparisons divided by the case, only depending on α



(a) Standard error of mean comparisons divided by input size over both dimensions



(b) Standard error of mean comparisons divided by input size over α

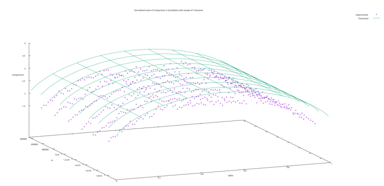
Figure 3: Standard error of QuickSelect

Previous figures showed us that the number of comparisons are quite close to the theoretical approximation. In figure 3 we analyzed the standard deviation as $\frac{\sigma_n^{(\alpha n)}}{n}$. We observe that it has a high standard deviation. This means that although it is not exactly the theoretical approximation, we know that the obtained values could be the theoretical value because the high standard deviation (We should do a statistical test, t-test, to determine if the value could be the theoretical or not).

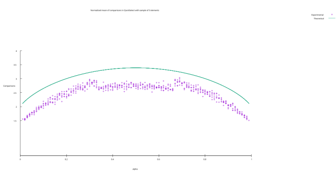
1.3 Pivot with sample of $(2t + 1)$ elements

Instead of choosing randomly the pivot, we also provide an implementation which sample $2t+1$ elements u.a.r and choose the element α in the sample. The key with this method is that t is given in compilation time so it can be implemented in constant time and it improves the number of comparisons needed.

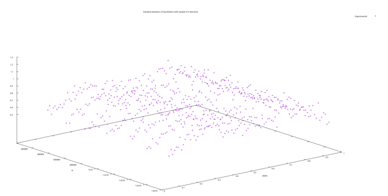
In figure 4, we show the performance for $t = 1$ for this algorithm. We have done not too many experiments because of a lack of time, but we can still observe some significant results. Further experimentation should be realized to analyze how the parameter t affects the number of comparisons. For instance, the subfigure 4a and subfigure 4b show that it have significantly less comparisons than QuickSelect (figure 1 and figure 2). Also it is noticeable that the standard error is also lower with the sample than the other one.



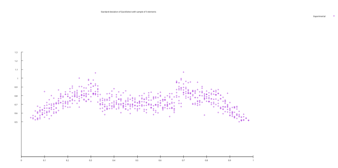
(a) Mean comparisons divided by input size over both dimensions. Compared with the theoretical cost of the simple QuickSelect.



(b) Mean comparisons divided by input size over α. Compared with the theoretical cost of the simple QuickSelect.



(c) Standard error of mean comparisons divided by input size over both dimensions



(d) Standard error of mean comparisons divided by input size over α

Figure 4: Standard error of QuickSelect

2 Randomized Selection

In this section, the algorithm Randomized Selection designned by Floy and Rivest [1] with the version studied in the lectures. This version will be used to obtain the median, but can be easily modified to get any element.

2.1 Implementation

The algorithm showed the lectures is:

Algorithm 2 Randomized Selection

1: **procedure** RANSELECT(A) ▷ Randomized Selection method to find the median
2: $R \leftarrow$ Sample $\lceil n^{\frac{3}{4}} \rceil$ u.a.r with replacement from A
3: SORT(R)
4: $d \leftarrow R[\lfloor \frac{1}{2} n^{\frac{3}{4}} - \sqrt{n} \rfloor]$
5: $u \leftarrow R[\lfloor \frac{1}{2} n^{\frac{3}{4}} + \sqrt{n} \rfloor]$
6: $C \leftarrow x \in A | d <= x <= u$
7: $l_d \leftarrow |x \in A | x < d|$
8: $l_u \leftarrow |x \in A | u < x|$
9: **if** $l_d > \frac{n}{2} \wedge l_u > \frac{n}{2}$ **then**
10: **FAIL**
11: **end if**
12: **if** $|C| > 4n^{\frac{3}{4}}$ **then**
13: **FAIL**
14: **end if**
15: SORT(C)
16: **return** $C[\lceil \frac{n}{2} \rceil - l_d + 1]$
17: **end procedure**

If instead of choosing the median, we want a α element then we need to change in line 4 and 6 the $\frac{1}{2}$ with α, the line 9 by $l_d > \alpha n \wedge l_u > n(1 - \alpha)$ and inn the line 16 by $C[\lceil \alpha n \rceil - l_d + 1]$.

The implementation provided uses the standard library of C++ (further detail in appendix A). To sample uniformly at random with replacement, we use a pseudo-random numbers engine provided by the standard library called `std::mt19937` (for implementation details check Makoto and Takuji paper [3]). We have chosen this engine instead of a linear congruential because with the right parameters it has the longest non-repeating sequence and we want to generate a lot of numbers, although it is slower and has greater state storage requirements. It makes use of `std::random_engine` which is a uniformly-distributed integer random number generator that produces non-deterministic random numbers or pseudo-random numbers if a non-deterministic source is not available. After generating the random (or pseudo-random) bits, they are passed to `std::uniform_int_distribution` so it can generate the index of the sampled element that is going to be used. Because initializing the engine and the random device is a highly cost task, we have used the `static` keyword, so it is only initialized once over all the experiments.

2.2 Results and comparison

Similarly, to the previous algorithm we have realized 100 times for each input size and shuffling the input vector each time so we could get a different permutation. We have tried different input size in the range of $2 * 10^4$ to $2 * 10^6$ with a step of 1000. In this case, we have been able to try all of them because we weren't trying different α, only the median. To be able to compare this algorithm with QuickSelect, we had to slice the results in the dimension with $\alpha = 0.5$. To have a comparable metric with QuickSelect, we have also divided the average number of comparisons by the input size.

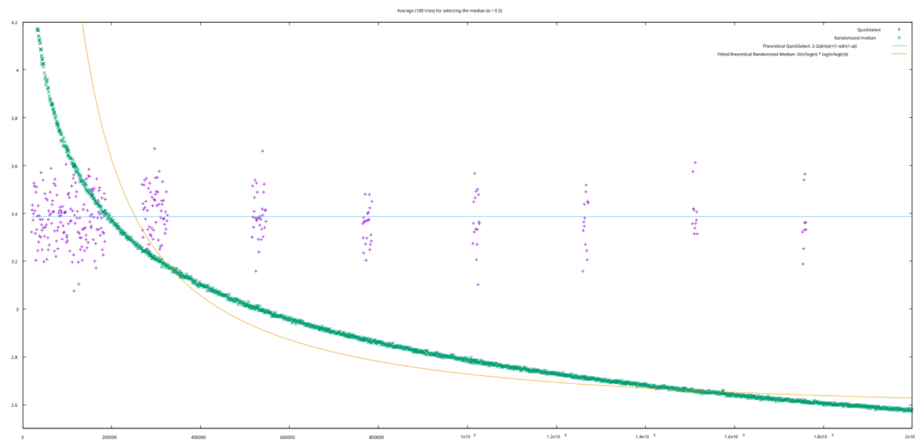


Figure 5: Comparison of mean comparisons in QuickSelect and Randomized Selection

In the figure 5, we observe the proportion number of comparisons decrease with the input size. Compared with QuickSelect this means Randomized Selection performs much better than QuickSelect. We have fitted the theoretical curves and we see that for QuickSelect it fits nicely, for Randomized Selection we observe that the theoretical curve is an upper-bound but not tight which fits with the small-o notation used in the lectures.

The figure 6 shows that not only Randomized Selection has a lower cost in terms of comparisons, but also has a lower standard error which means that the total number of comparisons is much more stable. A drawback of Randomized Selection is that it can fail which it doesn't happen in QuickSelect, but the probability of fail is really low, $P[\mathbf{FAIL}] \leq \frac{1}{n^{1/4}}$, and in all the experiments realized we never got a failure from Randomized Selection.

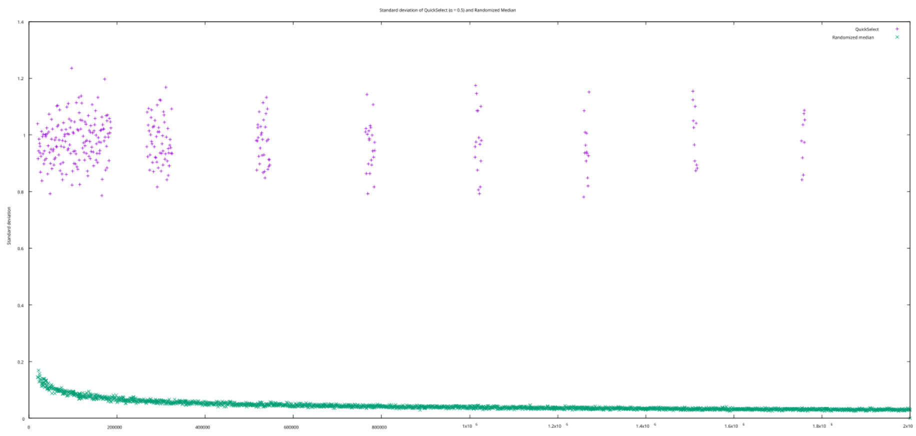


Figure 6: Comparison of standard deviation in comparisons in QuickSelect and Randomized Selection

Appendices

A C++ Implementation

```
//*****************************************************
// File:    assignment2.cpp
// Author: Daniel Bened  Garc a
// Date:
// Coms:
//*****************************************************

#include <iostream>
#include <random>
#include <fstream>
#include <numeric>
#include <vector>
#include <algorithm>
#include <sstream>
#include <optional>
#include <omp.h>

/*
```

```
 * Recursive implementation
 *       int r = RanPartition(A,p,q);

 *       if(r == j){
 *               return A[r];
 *       }else if( i < r){
 *               return QuickSelect(A,i,min,r-1);
 *       }else{
 *               return QuickSelect(A,i-r,r+1, max);
 *       }
 *
 *
 * Note: It returns only the number of comparisons, but it can also be return
 *        the i-th element
 */
int QuickSelect(int* A, int i, int min, int max){
        // Random device and Mersenne twistter are static so they are stored
        // generated only once
        static std::random_device rd;
        static std::mt19937_64 g(rd());

        std::shuffle(A+min, A+max,g);

        max--;
        int r;
        int Cj = 0;
        do{
                // Create distribution for range min, max in that iteration
                std::uniform_int_distribution<int> distrib(min,max);

                // Choose an element a in A uniformly at random
                int a = A[distrib(g)];

                // Order according the pivot element a
                int left = min, right = max;
                while(left < right){
                        while(left <= max && A[left]  < a && left < right){
                                left++;
                                Cj++;
                        }
                        Cj++;
                        while(right >= min && A[right] > a && left < right){
                                right--;
                                Cj++;
                        }
                        Cj++;
```

```
                        int temp = A[left];
                        A[left] = A[right];
                        A[right] = temp;
                        left++; right--;
                }

                r = left;
                if(i < r){
                        max = r - 1;
                }else if(i > r){
                        min = r + 1;
                }
        }while(i != r);

        return Cj;
}

void task1(){
        const int MIN = 20000, MAX = 2000000;
        std::ofstream comparisons("quickselect_comparisons.csv");

        #pragma omp parallel for
        for(int N = MIN; N < MAX; N+=1000){
                int* vec = new int[N];
                std::iota(vec, vec+N,1);

                std::stringstream buff;
                for(double p = 0.00; p <= 1; p += 0.01){
                        int j = int(p*(N-1));
                        buff << N << ", " << j;
                        for(int attempt = 0; attempt < 100; attempt++){ //Do
                                int Cj = QuickSelect(vec, j, 0, N);
                                buff << ", " << Cj;
                        }
                        buff << std::endl;
                }
                comparisons << buff.str();
                delete[] vec;
        }
}

/*
 * Same implementation of QuickSelect but now the pivot will depend on the
 * sample u.a.r of (2t+1) elements
 */
template<int t>
```

```
int QuickSelect_t(int* A, int i, int min, int max){
        // Random device and Mersenne twistter are static so they are stored
        // generated only once
        static std::random_device rd;
        static std::mt19937_64 g(rd());

        std::shuffle(A+min, A+max,g);

        max--;
        int r;
        int Cj = 0;
        do{
                // Create distribution for range min, max in that iteration
                std::uniform_int_distribution<int> distrib(min,max);

                // Choose 2t+1 elements in A uniformly at random
                int a;
                if( (max+1-min) > (2*t+1) ){
                        int S[2*t+1];
                        for(int k = 0; k < 2*t+1; k++)
                                S[k] = A[distrib(g)];
                        std::sort(S,S+2*t+1);
                        // Choose pivot depending on the sample
                        a = S[(i-min)/((max+1-min)/(2*t+1))];
                }else{
                        a = A[distrib(g)];
                }

                // Order according the pivot element a
                int left = min, right = max;
                while(left < right){
                        while(left <= max && A[left] < a && left < right){
                                left++;
                                Cj++;
                        }
                        Cj++;
                        while(right >= min && A[right] > a && left < right){
                                right--;
                                Cj++;
                        }
                        Cj++;
                        int temp = A[left];
                        A[left] = A[right];
                        A[right] = temp;
                        left++; right--;
                }
```

```
        r = left;
        if(i < r){
                max = r - 1;
        }else if(i > r){
                min = r + 1;
        }
    }while(i != r);

    return Cj;
}

void task1_extra(){
    const int MIN = 20000, MAX = 2000000;
    std::ofstream comparisons("quickselect_sample_comparisons.csv");

    #pragma omp parallel for
    for(int N = MIN; N < MAX; N+=1000){
        int* vec = new int[N];
        std::iota(vec, vec+N,1);

        std::stringstream buff;
        for(double p = 0.00; p <= 1; p += 0.01){
            int j = int(p*(N-1));
            buff << N << ", " << j;
            for(int attempt = 0; attempt < 100; attempt++){ //Do
                int Cj = QuickSelect_t<1>(vec, j, 0, N);
                buff << ", " << Cj;
            }
            buff << std::endl;
        }
        comparisons << buff.str();
        delete[] vec;
    }
}



/*
 *      @return The median element of A[min..max]
 *
 *      R := Sample ceil(n**0.75) from A uniform and w/replacement
 *      Sort(R)
 *      d := floor(0.5*n**0.75-n**0.5) element in R
 *      u := floor(0.5*n**0.75+n**0.5) element in R
 *
```

```
 *        C    := elements between d and u
 *        l_d  := num elements lower than d
 *        l_u  := num elements bigger than u
 *
 *        if l_d > n/2 or l_u > n/2
 *                FAIL
 *
 *        if |C| > 4n**0.75
 *                FAIL
 *
 *        Sort(C)
 *        return floor(0.5*n) - l_d + 1 element in C
 */
std::optional<std::pair<int,int>> RanSelect(std::vector<int> A){
        static std::random_device rd;
        static std::mt19937_64 g(rd());

        std::shuffle(A.begin(), A.end(),g);

        int C_n = 0;
        auto comparator = [&C_n](int a, int b){C_n++; return a < b;};

        int n = A.size();
        double n_3_4 = std::pow(n, 0.75);
        double n_1_2 = std::sqrt(n);

        std::vector<int> R(std::ceil(n_3_4));

        std::sample(A.begin(), A.end(),
                                R.begin(),
                                int(std::ceil(n_3_4)), g);

        std::sort(R.begin(), R.end(), comparator);

        int d = R[std::floor(0.5*n_3_4-n_1_2)],
                u = R[std::floor(0.5*n_3_4+n_1_2)];

        std::vector<int> C;
        int l_d = 0, l_u = 0;
        for(int elem : A){
            if(elem < d) l_d++;
            else if(elem > u) l_u++;
            else C.push_back(elem);
            C_n++;
        }

```

```
        if(l_d > A.size()/2 || l_u > A.size()/2){
                std::cerr << "FAIL: l_d or l_u too big" << std::endl;
                return std::nullopt;
        }

        if(C.size() > 4*n_3_4){
                std::cerr << "FAIL: C is too big" << std::endl;
            return std::nullopt;
        }

        std::sort(C.begin(), C.end(), comparator);

        return std::optional<std::pair<int,int>>(std::make_pair(C[n/2 - l_d +
}

void task2(){
        const int MIN = 20000, MAX = 2000000;
        std::ofstream comparisons("randselect_comparisons.csv");

        #pragma omp parallel for
        for(int N = MIN; N < MAX; N+=1000){
                std::vector<int> vec(N);
                std::iota(vec.begin(), vec.end(),1);


                std::stringstream buff;
                int tries = 0;
                for(int attempt = 0; attempt < 100; attempt++){ //Do 100 rand
                        std::optional<std::pair<int,int>> res = std::nullopt;
                        while(!res){
                                tries++;
                                res = RanSelect(vec);
                        }
                        buff << ", " << res.value().second;
                }

                std::stringstream temp;
                temp << N << ", " << tries - 100 << buff.rdbuf() << std::endl
                buff = std::move(temp);
                comparisons << buff.str();
        }
}
int main(int argc, char* argv[]){

        //task1();
        task1_extra();
```

```
            //task2();
}
```

References

[1] Robert W. Floyd and Ronald L. Rivest. "Algorithm 489: The Algorithm SELECT—for Finding the Ith Smallest of n Elements [M1]". In: *Commun. ACM* 18.3 (Mar. 1975), p. 173. ISSN: 0001-0782. DOI: `10.1145/360680.360694`. URL: `https://doi.org/10.1145/360680.360694`.

[2] C. A. R. Hoare. "Algorithm 65: Find". In: *Commun. ACM* 4.7 (July 1961), pp. 321–322. ISSN: 0001-0782. DOI: `10.1145/366622.366647`. URL: `https://doi.org/10.1145/366622.366647`.

[3] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: `10.1145/272991.272995`. URL: `https://doi.org/10.1145/272991.272995`.