

Manual do Programador Competitivo

Antti Laaksonen

Rascunho de 27 de abril de 2024

Sumário

Prefácio	v
1 Ordenação	1
1.1 Teoria da ordenação	1
1.2 Sorting in C++	5
1.3 Binary search	7
Bibliografia	11
Índice Remissivo	17

Prefácio

O objetivo deste livro é oferecer uma introdução completa à programação competitiva. É necessário que você já conheça os conceitos básicos de programação, mas não é preciso ter experiência prévia com programação competitiva.

O livro é especialmente destinado a estudantes que desejam aprender algoritmos e, possivelmente, participar da *International Olympiad in Informatics* (IOI) ou do *International Collegiate Programming Contest* (ICPC). No Brasil, a Olimpíada Brasileira de Informática (OBI) classifica para a IOI, e a Maratona de Programação da Sociedade Brasileira de Computação é a fase regional do ICPC. É claro que o livro também é adequado para qualquer pessoa interessada em programação competitiva.

Leva muito tempo para se tornar um bom programador competitivo, mas também é uma oportunidade para aprender muito. Você pode ter certeza de que o seu entendimento geral sobre algoritmos ficará muito melhor se dedicar um tempo para ler este livro, resolver problemas e participar de competições.

Esta tradução e o livro em si estão em constante desenvolvimento. Você pode enviar seu *feedback* da versão original do livro para ahslaaks@cs.helsinki.fi, ou enviar um *pull request* diretamente para fazer correções na tradução do livro.

Helsinki, agosto de 2019

Antti Laaksonen

Capítulo 1

Ordenação

Ordenação é um problema fundamental no design de algoritmos. Muitos algoritmos eficientes utilizam a ordenação como uma sub-rotina, pois frequentemente é mais fácil processar os dados quando os elementos estão ordenados.

Por exemplo, o problema "um array contém dois elementos iguais?" é fácil de resolver usando ordenação. Se o array contiver dois elementos iguais, eles estarão um ao lado do outro após a ordenação, então é fácil encontrá-los. Além disso, o problema "qual é o elemento mais frequente em um array?" pode ser resolvido de forma semelhante.

Existem muitos algoritmos para ordenação, e eles também são bons exemplos de como aplicar diferentes técnicas de design de algoritmos. Os algoritmos de ordenação eficientes funcionam em tempo $O(n \log n)$, e muitos algoritmos que usam a ordenação como sub-rotina também têm essa complexidade de tempo.

1.1 Teoria da ordenação

O problema básico na ordenação é o seguinte:

Dado um array que contém n elementos, sua tarefa é ordenar os elementos em ordem crescente.

Por exemplo, o array

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

ficará da seguinte forma após a ordenação:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Algoritmos $O(n^2)$

Algoritmos simples para ordenar um array operam em tempo $O(n^2)$. Tais algoritmos são curtos e geralmente consistem em dois loops aninhados. Um famoso

algoritmo de ordenação em tempo $O(n^2)$ é o **bubble sort** onde os elementos "flutuam" no array de acordo com seus valores.

O Bubble sort consiste em n rodadas. Em cada rodada, o algoritmo percorre os elementos do array. Sempre que dois elementos consecutivos são encontrados que não estão na ordem correta, o algoritmo os troca. O algoritmo pode ser implementado da seguinte forma:

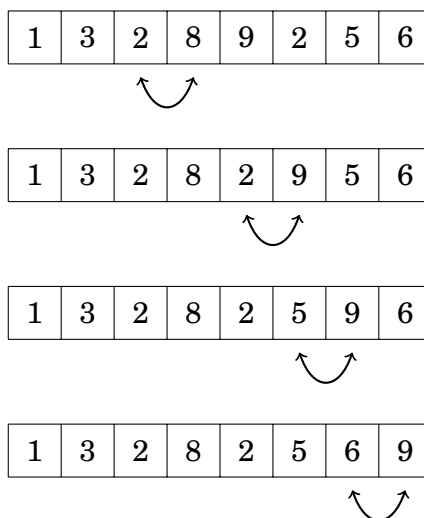
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n-1; j++) {  
        if (array[j] > array[j+1]) {  
            swap(array[j], array[j+1]);  
        }  
    }  
}
```

Após a primeira rodada do algoritmo, o maior elemento estará na posição correta e, em geral, após k rodadas, os k maiores elementos estarão nas posições corretas. Portanto, após n rodadas, o array inteiro estará ordenado.

Por exemplo, no array

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

na primeira rodada do bubble sort, os elementos são trocados da seguinte forma:



Inversões

Bubble sort is an example of a sorting algorithm that always swaps *consecutive* elements in the array. It turns out that the time complexity of such an algorithm is *always* at least $O(n^2)$, because in the worst case, $O(n^2)$ swaps are required for sorting the array.

A useful concept when analyzing sorting algorithms is an **inversion**: a pair of array elements ($\text{array}[a], \text{array}[b]$) such that $a < b$ and $\text{array}[a] > \text{array}[b]$, i.e., the elements are in the wrong order. For example, the array

1	2	2	6	3	5	9	8
---	---	---	---	---	---	---	---

has three inversions: (6,3), (6,5) and (9,8). The number of inversions indicates how much work is needed to sort the array. An array is completely sorted when there are no inversions. On the other hand, if the array elements are in the reverse order, the number of inversions is the largest possible:

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

Swapping a pair of consecutive elements that are in the wrong order removes exactly one inversion from the array. Hence, if a sorting algorithm can only swap consecutive elements, each swap removes at most one inversion, and the time complexity of the algorithm is at least $O(n^2)$.

$O(n \log n)$ algorithms

It is possible to sort an array efficiently in $O(n \log n)$ time using algorithms that are not limited to swapping consecutive elements. One such algorithm is **merge sort**¹, which is based on recursion.

Merge sort sorts a subarray $\text{array}[a \dots b]$ as follows:

1. If $a = b$, do not do anything, because the subarray is already sorted.
2. Calculate the position of the middle element: $k = \lfloor (a + b)/2 \rfloor$.
3. Recursively sort the subarray $\text{array}[a \dots k]$.
4. Recursively sort the subarray $\text{array}[k + 1 \dots b]$.
5. *Merge* the sorted subarrays $\text{array}[a \dots k]$ and $\text{array}[k + 1 \dots b]$ into a sorted subarray $\text{array}[a \dots b]$.

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step. The recursion consists of $O(\log n)$ levels, and processing each level takes $O(n)$ time. Merging the subarrays $\text{array}[a \dots k]$ and $\text{array}[k + 1 \dots b]$ is possible in linear time, because they are already sorted.

For example, consider sorting the following array:

1	3	6	2	8	2	5	9
---	---	---	---	---	---	---	---

The array will be divided into two subarrays as follows:

1	3	6	2
---	---	---	---

8	2	5	9
---	---	---	---

Then, the subarrays will be sorted recursively as follows:

1	2	3	6
---	---	---	---

2	5	8	9
---	---	---	---

¹According to [47], merge sort was invented by J. von Neumann in 1945.

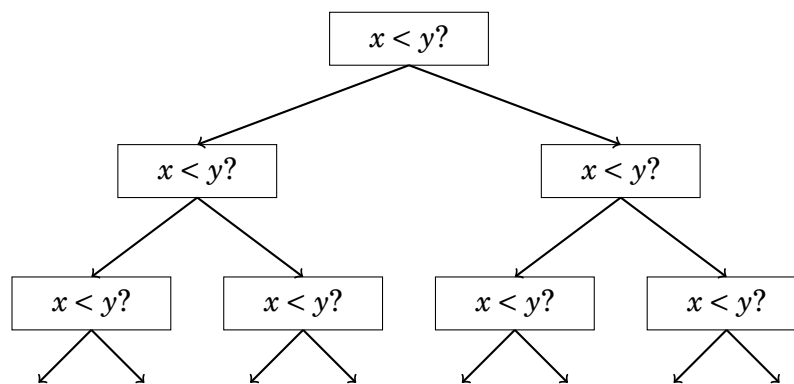
Finally, the algorithm merges the sorted subarrays and creates the final sorted array:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Sorting lower bound

Is it possible to sort an array faster than in $O(n \log n)$ time? It turns out that this is *not* possible when we restrict ourselves to sorting algorithms that are based on comparing array elements.

The lower bound for the time complexity can be proved by considering sorting as a process where each comparison of two elements gives more information about the contents of the array. The process creates the following tree:



Here " $x < y$?" means that some elements x and y are compared. If $x < y$, the process continues to the left, and otherwise to the right. The results of the process are the possible ways to sort the array, a total of $n!$ ways. For this reason, the height of the tree must be at least

$$\log_2(n!) = \log_2(1) + \log_2(2) + \dots + \log_2(n).$$

We get a lower bound for this sum by choosing the last $n/2$ elements and changing the value of each element to $\log_2(n/2)$. This yields an estimate

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

so the height of the tree and the minimum possible number of steps in a sorting algorithm in the worst case is at least $n \log n$.

Counting sort

The lower bound $n \log n$ does not apply to algorithms that do not compare array elements but use some other information. An example of such an algorithm is **counting sort** that sorts an array in $O(n)$ time assuming that every element in the array is an integer between $0 \dots c$ and $c = O(n)$.

The algorithm creates a *bookkeeping* array, whose indices are elements of the original array. The algorithm iterates through the original array and calculates how many times each element appears in the array.

For example, the array

1	3	6	9	9	3	5	9
---	---	---	---	---	---	---	---

corresponds to the following bookkeeping array:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

For example, the value at position 3 in the bookkeeping array is 2, because the element 3 appears 2 times in the original array.

Construction of the bookkeeping array takes $O(n)$ time. After this, the sorted array can be created in $O(n)$ time because the number of occurrences of each element can be retrieved from the bookkeeping array. Thus, the total time complexity of counting sort is $O(n)$.

Counting sort is a very efficient algorithm but it can only be used when the constant c is small enough, so that the array elements can be used as indices in the bookkeeping array.

1.2 Sorting in C++

It is almost never a good idea to use a home-made sorting algorithm in a contest, because there are good implementations available in programming languages. For example, the C++ standard library contains the function `sort` that can be easily used for sorting arrays and other data structures.

There are many benefits in using a library function. First, it saves time because there is no need to implement the function. Second, the library implementation is certainly correct and efficient: it is not probable that a home-made sorting function would be better.

In this section we will see how to use the C++ `sort` function. The following code sorts a vector in increasing order:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

After the sorting, the contents of the vector will be `[2,3,3,4,5,5,8]`. The default sorting order is increasing, but a reverse order is possible as follows:

```
sort(v.rbegin(),v.rend());
```

An ordinary array can be sorted as follows:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

The following code sorts the string s:

```
string s = "monkey";  
sort(s.begin(), s.end());
```

Sorting a string means that the characters of the string are sorted. For example, the string "monkey" becomes "ekmnoy".

Comparison operators

The function sort requires that a **comparison operator** is defined for the data type of the elements to be sorted. When sorting, this operator will be used whenever it is necessary to find out the order of two elements.

Most C++ data types have a built-in comparison operator, and elements of those types can be sorted automatically. For example, numbers are sorted according to their values and strings are sorted in alphabetical order.

Pairs (pair) are sorted primarily according to their first elements (first). However, if the first elements of two pairs are equal, they are sorted according to their second elements (second):

```
vector<pair<int,int>> v;  
v.push_back({1,5});  
v.push_back({2,3});  
v.push_back({1,2});  
sort(v.begin(), v.end());
```

After this, the order of the pairs is (1,2), (1,5) and (2,3).

In a similar way, tuples (tuple) are sorted primarily by the first element, secondarily by the second element, etc.²:

```
vector<tuple<int,int,int>> v;  
v.push_back({2,1,4});  
v.push_back({1,5,3});  
v.push_back({2,1,3});  
sort(v.begin(), v.end());
```

After this, the order of the tuples is (1,5,3), (2,1,3) and (2,1,4).

User-defined structs

User-defined structs do not have a comparison operator automatically. The operator should be defined inside the struct as a function operator<, whose parameter is another element of the same type. The operator should return true if the element is smaller than the parameter, and false otherwise.

For example, the following struct P contains the x and y coordinates of a point. The comparison operator is defined so that the points are sorted primarily by the

²Note that in some older compilers, the function make_tuple has to be used to create a tuple instead of braces (for example, make_tuple(2,1,4) instead of {2,1,4}).

x coordinate and secondarily by the y coordinate.

```
struct P {
    int x, y;
    bool operator<(const P &p) {
        if (x != p.x) return x < p.x;
        else return y < p.y;
    }
};
```

Comparison functions

It is also possible to give an external **comparison function** to the sort function as a callback function. For example, the following comparison function comp sorts strings primarily by length and secondarily by alphabetical order:

```
bool comp(string a, string b) {
    if (a.size() != b.size()) return a.size() < b.size();
    return a < b;
}
```

Now a vector of strings can be sorted as follows:

```
sort(v.begin(), v.end(), comp);
```

1.3 Binary search

A general method for searching for an element in an array is to use a for loop that iterates through the elements of the array. For example, the following code searches for an element x in an array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x found at index i
    }
}
```

The time complexity of this approach is $O(n)$, because in the worst case, it is necessary to check all elements of the array. If the order of the elements is arbitrary, this is also the best possible approach, because there is no additional information available where in the array we should search for the element x .

However, if the array is *sorted*, the situation is different. In this case it is possible to perform the search much faster, because the order of the elements in the array guides the search. The following **binary search** algorithm efficiently searches for an element in a sorted array in $O(\log n)$ time.

Method 1

The usual way to implement binary search resembles looking for a word in a dictionary. The search maintains an active region in the array, which initially contains all array elements. Then, a number of steps is performed, each of which halves the size of the region.

At each step, the search checks the middle element of the active region. If the middle element is the target element, the search terminates. Otherwise, the search recursively continues to the left or right half of the region, depending on the value of the middle element.

The above idea can be implemented as follows:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

In this implementation, the active region is $a \dots b$, and initially the region is $0 \dots n-1$. The algorithm halves the size of the region at each step, so the time complexity is $O(\log n)$.

Method 2

An alternative method to implement binary search is based on an efficient way to iterate through the elements of the array. The idea is to make jumps and slow the speed when we get closer to the target element.

The search goes through the array from left to right, and the initial jump length is $n/2$. At each step, the jump length will be halved: first $n/4$, then $n/8$, $n/16$, etc., until finally the length is 1. After the jumps, either the target element has been found or we know that it does not appear in the array.

The following code implements the above idea:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

During the search, the variable b contains the current jump length. The time complexity of the algorithm is $O(\log n)$, because the code in the while loop is performed at most twice for each jump length.

C++ functions

The C++ standard library contains the following functions that are based on binary search and work in logarithmic time:

- `lower_bound` returns a pointer to the first array element whose value is at least x .
- `upper_bound` returns a pointer to the first array element whose value is larger than x .
- `equal_range` returns both above pointers.

The functions assume that the array is sorted. If there is no such element, the pointer points to the element after the last array element. For example, the following code finds out whether an array contains an element with value x :

```
auto k = lower_bound(array, array+n, x) - array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Then, the following code counts the number of elements whose value is x :

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

Using `equal_range`, the code becomes shorter:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

Finding the smallest solution

An important use for binary search is to find the position where the value of a *function* changes. Suppose that we wish to find the smallest value k that is a valid solution for a problem. We are given a function $ok(x)$ that returns true if x is a valid solution and false otherwise. In addition, we know that $ok(x)$ is false when $x < k$ and true when $x \geq k$. The situation looks as follows:

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	false	false	...	false	true	true	...

Now, the value of k can be found using binary search:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;
```

The search finds the largest value of x for which $ok(x)$ is false. Thus, the next value $k = x + 1$ is the smallest possible value for which $ok(k)$ is true. The initial jump length z has to be large enough, for example some value for which we know beforehand that $ok(z)$ is true.

The algorithm calls the function ok $O(\log z)$ times, so the total time complexity depends on the function ok . For example, if the function works in $O(n)$ time, the total time complexity is $O(n \log z)$.

Finding the maximum value

Binary search can also be used to find the maximum value for a function that is first increasing and then decreasing. Our task is to find a position k such that

- $f(x) < f(x + 1)$ when $x < k$, and
- $f(x) > f(x + 1)$ when $x \geq k$.

The idea is to use binary search for finding the largest value of x for which $f(x) < f(x + 1)$. This implies that $k = x + 1$ because $f(x + 1) > f(x + 2)$. The following code implements the search:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Note that unlike in the ordinary binary search, here it is not allowed that consecutive values of the function are equal. In this case it would not be possible to know how to continue the search.

Referências Bibliográficas

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).

- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>

- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).

- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).
- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

Índice Remissivo

binary search, 7	merge sort, 3
bubble sort, 1	Ordenação, 1
comparison function, 7	pair, 6
comparison operator, 6	sort, 5
counting sort, 4	tuple, 6
inversão, 2	