Manual do Programador Competitivo

Antti Laaksonen

Rascunho de 28 de abril de 2024

Sumário

Prefácio **v**

1 Ordenação **1**
1.1 Teoria da ordenação 1
1.2 Ordenação em C++ 6
1.3 Busca binária 8

Bibliografia **13**

Índice Remissivo **19**

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

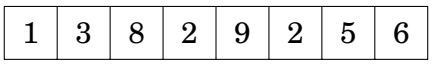
algoritmo de ordenação em tempo $O(n^2)$ é o **bubble sort** onde os elementos "flutuam"no array de acordo com seus valores.

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:

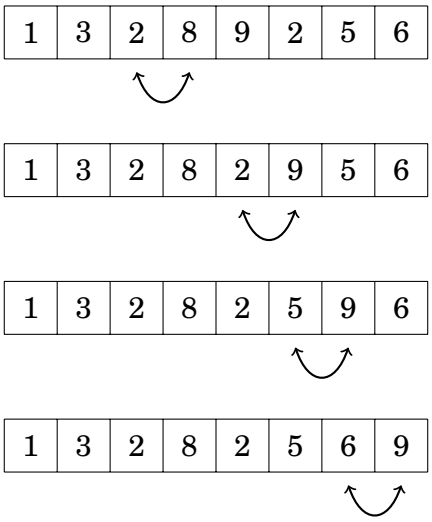
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j],array[j+1]);
        }
    }
}
```

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



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



Inversões

O Bubble sort é um exemplo de um algoritmo de ordenação que sempre troca elementos *consecutivos* no array. Acontece que a complexidade de tempo de tal algoritmo é *sempre* pelo menos $O(n^2)$, porque no pior caso, são necessárias, $O(n^2)$ trocas para ordenar o array.

Um conceito útil ao analisar algoritmos de ordenação é uma **inversão**: um par de elementos de array $(\texttt{array}[a], \texttt{array}[b])$ tal que $a < b$ and $\texttt{array}[a] > \texttt{array}[b]$, ou seja, os elementos estão na ordem errada. Por exemplo, o array

1	2	2	6	3	5	9	8

tem três inversões: (6,3), (6,5) and (9,8). O número de inversões indica o quanto de trabalho é necessário para ordenar o array. Um array está completamente ordenado quando não há inversões. Por outro lado, se os elementos do array estiverem em ordem reversa, o número de inversões é o máximo possível:

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

A troca de um par de elementos consecutivos que estão na ordem errada remove exatamente uma inversão do array. Portanto, se um algoritmo de ordenação só pode trocar elementos consecutivos, cada troca remove no máximo uma inversão, e a complexidade de tempo do algoritmo é pelo menos $O(n^2)$.

Algoritmos $O(n \log n)$

É possível ordenar um array de forma eficiente em tempo $O(n \log n)$ usando algoritmos que não estão limitados a trocar elementos consecutivos. Um desses algoritmos é o **merge sort**[1], que é baseado em recursão.

Merge sort ordena um subarray $\texttt{array}[a \ldots b]$ da seguinte forma:

1. Se $a = b$, não faça nada, pois o subarray já está ordenado..
2. Calcule a posição do elemento do meio: $k = \lfloor (a+b)/2 \rfloor$.
3. Ordene recursivamente o subarray $\texttt{array}[a \ldots k]$.
4. Ordene recursivamente o subarray $\texttt{array}[k+1 \ldots b]$.
5. *Junte* os subarrays ordenados $\texttt{array}[a \ldots k]$ e $\texttt{array}[k+1 \ldots b]$ em um subarray ordenado $\texttt{array}[a \ldots b]$.

O merge sort é um algoritmo eficiente porque ele reduz pela metade o tamanho do subarray a cada passo. A recursão consiste em $O(\log n)$ níveis, e processar cada nível leva tempo $O(n)$. Juntar os subarrays $\texttt{array}[a \ldots k]$ e $\texttt{array}[k+1 \ldots b]$ é possível em tempo linear, porque eles já estão ordenados.

Por exemplo, considere ordenar o seguinte array:

1	3	6	2	8	2	5	9

O array será dividido em dois subarrays da seguinte forma:

1	3	6	2

8	2	5	9

Então, os subarrays serão ordenados recursivamente da seguinte forma:

[1] De acordo com [47], o merge sort foi inventado por J. von Neumann em 1945.

1	2	3	6

2	5	8	9

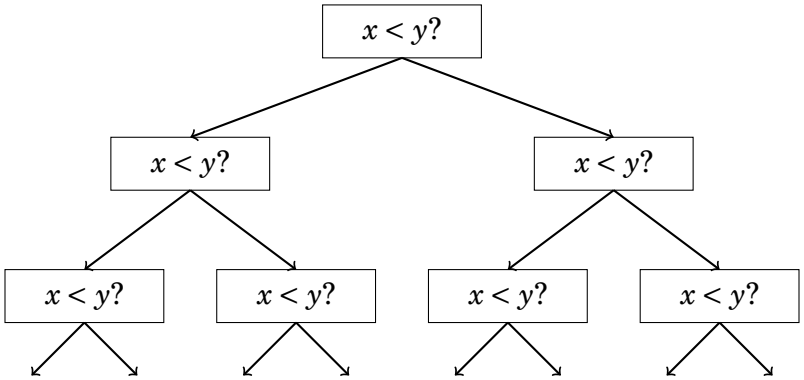
Finalmente, o algoritmo junta os subarrays ordenados e cria o array final ordenado:

1	2	2	3	5	6	8	9

Limite Inferior de Ordenação

É possível ordenar um array mais rápido do que em tempo $O(n \log n)$? Acontece que isso *não* é possível quando nos limitamos a algoritmos de ordenação baseados na comparação de elementos do array.

O limite inferior para a complexidade temporal pode ser demonstrado considerando a ordenação como um processo no qual cada comparação de dois elementos fornece mais informações sobre o conteúdo do array. O processo cria a seguinte árvore:



Aqui "$x < y$?" significa que alguns elementos x e y são comparados. Se $x < y$, o processo continua para a esquerda e, caso contrário, para a direita. Os resultados do processo são as possíveis maneiras de ordenar o array, um total de $n!$ maneiras. Por essa razão, a altura da árvore deve ser pelo menos

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

Obtemos um limite inferior para esta soma escolhendo os últimos $n/2$ elementos e alterando o valor de cada elemento para $\log_2(n/2)$. Isso nos dá uma estimativa

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

portanto, a altura da árvore e o número mínimo possível de etapas em um algoritmo de ordenação no pior caso é pelo menos $n \log n$.

Counting sort

O limite inferior $n \log n$ não se aplica a algoritmos que não comparam elementos de array, mas usam alguma outra informação. Um exemplo de tal algoritmo

é o **counting sort** que ordena um array em tempo $O(n)$ assumindo que cada elemento no array é um inteiro entre $0 \ldots c$ e $c = O(n)$.

O algoritmo cria um *array de contagem*, cujos índices são elementos do array original. O algoritmo itera pelo array original e calcula quantas vezes cada elemento aparece no array.

Por exemplo, o array

1	3	6	9	9	3	5	9

corresponde ao array de contagem a seguir:

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

Por exemplo, o valor na posição 3 no array de contagem é 2, porque o elemento 3 aparece 2 vezes no array original.

A construção do array de contagem leva tempo $O(n)$. Depois disso, o array ordenado pode ser criado em tempo $O(n)$ porque o número de ocorrências de cada elemento pode ser recuperado do array de contagem. Portanto, a complexidade temporal total do counting sort é $O(n)$.

O counting sort é um algoritmo muito eficiente, mas só pode ser usado quando a constante c é pequena o suficiente, de modo que os elementos do array possam ser usados como índices no array de contagem.

1.2 Ordenação em C++

Quase nunca é uma boa ideia usar um algoritmo de ordenação feito em casa em uma competição, porque existem boas implementações disponíveis em linguagens de programação. Por exemplo, a biblioteca padrão de C++ contém a função `sort` que pode ser facilmente usada para ordenar arrays e outras estruturas de dados.

Há muitos benefícios em usar uma função de biblioteca. Primeiro, isso economiza tempo porque não há necessidade de implementar a função. Segundo, a implementação da biblioteca é certamente correta e eficiente: é improvável que uma função de ordenação feita em casa seja melhor.

Nesta seção, veremos como usar a função `sort` em C++. O código a seguir ordena um vetor em ordem crescente:

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

Após a ordenação, o conteúdo do vetor será: $[2,3,3,4,5,5,8]$. A ordem de classificação padrão é crescente, mas uma ordem reversa é possível da seguinte forma:

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

Um array comum pode ser ordenado da seguinte forma:

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

O seguinte código ordena a string s:

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

Ordenar uma string significa que os caracteres da string são ordenados. Por exemplo, a string ”monkey” se torna ”ekmnoy”.

Operadores de comparação

A função `sort` requer que um **operador de comparação** seja definido para o tipo de dados dos elementos a serem ordenados. Ao ordenar, esse operador será usado sempre que for necessário determinar a ordem de dois elementos.

A maioria dos tipos de dados em C++ tem um operador de comparação integrado, e elementos desses tipos podem ser ordenados automaticamente. Por exemplo, números são ordenados de acordo com seus valores e strings são ordenadas em ordem alfabética.

Pares (`pair`) são ordenados principalmente de acordo com seus primeiros elementos (`first`). No entanto, se os primeiros elementos de dois pares forem iguais, eles são ordenados de acordo com seus segundos elementos (`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());
```

Após isso, a ordem dos pares é: $(1,2)$, $(1,5)$ and $(2,3)$.

De forma semelhante, tuplas (`tuple`) são ordenadas principalmente pelo primeiro elemento, secundariamente pelo segundo elemento, 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());
```

Após isso, a ordem das tuplas é: $(1,5,3)$, $(2,1,3)$ e $(2,1,4)$.

Structs definidas pelo usuário

As structs definidas pelo usuário não possuem um operador de comparação automaticamente. O operador deve ser definido dentro da struct como uma função `operator<`, cujo parâmetro é outro elemento do mesmo tipo. O operador deve retornar `true` se o elemento for menor que o parâmetro, e `false` caso contrário.

[2]Note que em alguns compiladores mais antigos, a função `make_tuple` deve ser usada para criar uma tupla em vez de chaves (por exemplo, `make_tuple(2,1,4)` em vez de `{2,1,4}`).

Por exemplo, a seguinte struct `P` contém as coordenadas x e y de um ponto. O operador de comparação é definido de forma que os pontos sejam ordenados principalmente pela coordenada x e secundariamente pela coordenada y.

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

Funções de comparação

Também é possível fornecer uma **função de comparação** externa para a função `sort` como uma função de callback. Por exemplo, a seguinte função de comparação `comp` ordena strings principalmente por comprimento e secundariamente por ordem alfabética:

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

Agora um vetor de strings pode ser ordenado da seguinte forma:

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

1.3 Busca binária

Um método geral para buscar um elemento em um array é usar um loop `for` que itera pelos elementos do array. Por exemplo, o seguinte código busca por um elemento x no array:

```
for (int i = 0; i < n; i++) {
    if (array[i] == x) {
        // x encontrado no indice i
    }
}
```

A complexidade temporal desta abordagem é $O(n)$, porque no pior caso é necessário verificar todos os elementos do array. Se a ordem dos elementos for arbitrária, esta também é a melhor abordagem possível, pois não há informações adicionais disponíveis sobre onde no array devemos procurar pelo elemento x.

No entanto, se o array estiver *ordenado*, a situação é diferente. Neste caso, é possível realizar a busca muito mais rapidamente, porque a ordem dos elementos

no array orienta a busca. O seguinte algoritmo de **busca binária** efetua a busca por um elemento em um array ordenado de forma eficiente em tempo $O(\log n)$.

Método 1

A maneira usual de implementar a busca binária se assemelha a procurar uma palavra em um dicionário. A busca mantém uma região ativa no array, que inicialmente contém todos os elementos do array. Em seguida, um número de passos é executado, cada um dos quais divide pela metade o tamanho da região.

Em cada etapa, a busca verifica o elemento do meio da região ativa. Se o elemento do meio for o elemento alvo, a busca termina. Caso contrário, a busca continua recursivamente para a metade esquerda ou direita da região, dependendo do valor do elemento do meio.

A ideia acima pode ser implementada da seguinte forma:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x encontrado no indice k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Nesta implementação, a região ativa é $a \ldots b$, e inicialmente a região é $0 \ldots n-1$. O algoritmo divide o tamanho da região pela metade a cada etapa, então a complexidade temporal é $O(\log n)$.

Método 2

Um método alternativo para implementar a busca binária é baseado em uma maneira eficiente de iterar pelos elementos do array. A ideia é fazer saltos e diminuir a velocidade quando estivermos mais perto do elemento alvo.

busca percorre o array da esquerda para a direita, e o comprimento inicial do salto é $n/2$. Em cada etapa, o comprimento do salto será dividido pela metade: primeiro $n/4$, depois $n/8$, $n/16$, etc., até que finalmente o comprimento seja 1. Após os saltos, ou o elemento alvo foi encontrado ou sabemos que ele não aparece no array.

O código a seguir implementa a ideia acima:

```
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 encontrado no indice k
}
```

Durante a busca, a variável b contém o comprimento atual do salto.. A complexidade temporal do algoritmo é $O(\log n)$, porque o código no loop `while` é executado no máximo duas vezes para cada comprimento de salto.

Funções em C++

A biblioteca padrão de C++ contém as seguintes funções que são baseadas em busca binária e funcionam em tempo logarítmico:

- `lower_bound` retorna um ponteiro para o primeiro elemento do array cujo valor é pelo menos x.
- `upper_bound` retorna um ponteiro para o primeiro elemento do array cujo valor é maior do que x.
- `equal_range` retorna ambos os ponteiros acima.

As funções assumem que o array está ordenado. Se não houver tal elemento, o ponteiro aponta para o elemento após o último elemento do array. Por exemplo, o seguinte código verifica se um array contém um elemento com valor x:

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x encontrado no indice k
}
```

Então, o seguinte código conta o número de elementos cujo valor é x:

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

Usando `equal_range`, o código fica mais curto:

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

Encontrando a menor solução

Um uso importante para a busca binária é encontrar a posição onde o valor de uma *função* muda. Suponha que desejamos encontrar o menor valor k que é uma solução válida para um problema. emos uma função ok(x) que retorna `true` se x é uma solução válida e `false` caso contrário. Além disso, sabemos que ok(x) é `false` quando $x < k$ e `true` quando $x \geq k$. A situação é a seguinte:

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

Agora, o valor de k pode ser encontrado usando busca binária

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

A busca encontra o maior valor de x para o qual ok(x) é `false`. Assim, o próximo valor $k = x+1$ é o menor valor possível para o qual ok(k) é `true`. O comprimento inicial do salto z deve ser grande o suficiente, por exemplo, algum valor para o qual sabemos de antemão que ok(z) é `true`.

O algoritmo chama a função ok $O(\log z)$ vezes, então a complexidade temporal total depende da função ok. Por exemplo, se a função funciona em tempo $O(n)$, a complexidade temporal total é $O(n \log z)$.

Encontrando o valor máximo

A busca binária também pode ser usada para encontrar o valor máximo de uma função que é primeiro crescente e depois decrescente. Nossa tarefa é encontrar uma posição k tal que

- $f(x) < f(x+1)$ quando $x < k$, e
- $f(x) > f(x+1)$ quando $x \geq k$.

A ideia é usar busca binária para encontrar o maior valor de x para o qual $f(x) < f(x+1)$. Isso implica que $k = x+1$ porque $f(x+1) > f(x+2)$. O seguinte código implementa a busca:

```
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 que, ao contrário da busca binária comum, aqui não é permitido que valores consecutivos da função sejam iguais. Nesse caso, não seria possível saber como continuar a busca.

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 multicommodity 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 componemts 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

bubble sort, 1
busca binária, 8

counting sort, 4

função de comparação, 8

inversão, 2

merge sort, 3

operador de comparação, 7
Ordenação, 1

`pair`, 7

`sort`, 6

`tuple`, 7