Chapter 1

Problems on arrays

Arrays are the simplest data structures. An array A of length n is a sequence of n elements, $A[0], \dots, A[n-1]$. If each element is also an array, the array is called a *matrix*. An array of n arrays, in which each $A[i]$ is an array of length m, is called an $n \times m$ *matrix*.

Notation For an array A, $A[i \dots j]$ denotes the subarray $A[i] \dots A[j]$. For two arrays A and B, $A + B$ denotes the concatenation of A and B.

1.1 Find specified element(s)

In this section, we focus on the problems of finding specified element(s) in a given array or matrix.

Problem 1

Let A be an array of n integers, in which all integers are between 0 and $n-2$. Moreover, all integers between 0 and $n-2$ appear at least once in A; that is, there is only one duplicate integer in A. Design an algorithm to find the duplicate elements in A without modifying the content of A.

Answer of exercise 1

Since each integer in A is between 0 and $n-2$, each element in A can be considered as a node in a linked list; that is, the i-th element points to the $A[i]$-th element. Finding the duplicate integer, $k = A[i] = A[j]$ for some indices i and j, is equivalent to finding the k-th element, pointed by exactly two elements: the i-th element and the j-th element. Since $n-1$ is not in A, the $(n-1)$-th element can be considered as the head of the linked list. Since two elements are pointing to the k-th element, a cycle exists in the linked list, and the head of the cycle must be the k-th element. Thus, we can use the cycle detection algorithm to find the head of the cycle, which is the duplicate integer.

Complexity analysis The time complexity of this algorithm is linear, and the space complexity is $O(1)$.

Remark. This solution is discussed in Giant's paper [9]. This problem appears on LeetCode.

Problem 2 Majority

Given an integer $k > 1$ and an array A of n integers, design an algorithm to find all integers occurring more than $\frac{n}{k}$ times in A if such an integer exists.

Answer of exercise 2

The key observation for this problem is as follows: let c be an integer occurring more than $\frac{n}{k}$ times in A. If k distinct integers are removed from A, then c still occurs more than $\frac{n-k}{k}$ times among the remaining integers.

Based on this observation, we maintain a set S of at most $k-1$ candidates. For each candidate, we also maintain its frequency so far. Initially, S is an empty set. The algorithm reads the integers in A one by one. Let x be the integer read by the algorithm. If x is already in S, increase the frequency of x by one. If x is not in S and $|S|$ is smaller than k, then insert x into S and set x's frequency to one. Otherwise, decrement all frequencies of all candidates in S by one; this step is equal to removing k distinct integers from A.

After processing all integers in A, all integers occurring more than $\frac{n}{k}$ times in A should be in S, and each candidate can be verified in linear time.

Complexity analysis The time complexity of this algorithm is $O(nk)$.

Remark. This solution is discussed in Misra's paper [15]. When $k = 2$, this problem is called the *majority finding* problem [3]. For any positive integer k, all integers occurring more than $\frac{n}{k}$ times can be found in linear time independent of k [11]. This problem appears on LintCode and LeetCode.

Problem 3 Saddleback search

Given an integer x and an $n \times m$ matrix M of integers, $m \leq n$, in which all rows and columns are in non-decreasing order, design an algorithm to determine whether x in M.

Answer of exercise 3

The key observation for this problem is as follows: for any pair of indices , i and j, if $M[i][j] > x$, then $M[k][j]$ is larger than x for all $i \leq k < n$. Similarly, if $M[i][j] < x$, then $M[i][k]$ is smaller than x for all $0 \leq k \leq j$.

Based on this observation, the algorithm can eliminate either a row or a column after comparing an entry with x.

We always compare x of the top-right element of the remaining submatrix. If this entry equals x, then we found the answer. If this entry is smaller x, then the top row can be discarded. If this entry is larger than x, the the rightmost column can be discarded. This technique is called the *saddleback search*.

Complexity analysis Since M has $O(n+m)$ rows and columns, the time complexity is $O(n+m)$.

A better solution Let $c = \lfloor m/2 \rfloor$. First, use binary search to find an index i in the middle column such that $M[i][c] \leq x \leq M[i+1][c]$. If one of $M[i][c]$ and $M[i+1][c]$ equals x, then we found the answer. Otherwise, the submatrix in the top-left of $M[i][c]$ cannot contain x. Similarly, the submatrix in the bottom-right of $M[i+1][c]$ cannot contain x. We can recursively search for x in the remaining two submatrices.

Complexity analysis The time complexity is $O(m \lg \frac{n}{m})$.

Remark. This solution is from Bird's paper [1]. This problem appears on LeetCode.

Problem 4 Selection in a sorted matrix

Given a positive integer k and an $n \times m$ matrix M of integers, in which all rows and columns are in non-decreasing order, design an algorithm to find the k-th smallest integer in M.

Answer of exercise 4

Since the k-th smallest integer must be between $min = \min_{i,j} M[i][j]$ and $max = \max_{i,j} M[i][j]$, the k-th smallest integer can be found by using the binary searching on the integers $[min, \ldots, max]$.

Note that for any integer x, we can determine the rank of x in $O(n+m)$ time by using the saddleback search. Let $[l, \ldots, h]$ be the range of integers to be searched. If the rank of $mid = \lfloor (l+h)/2 \rfloor$ is k, then we found the answer. If the rank of mid is larger than k, then the search range can be reduced to $[l, \ldots, mid-1]$. Otherwise, the search range can be reduced to $[mid+1, \ldots, h]$.

Complexity analysis Since the maximum number of iterations is $\lg(max - min)$, the time complexity is $O((n+m)\lg(max - min))$.

This solution has two drawbacks: one minor drawback is that $\lg(max - min)$ is in fact linear in the length of the input. Theoretically, this method is not better than applying linear time selection algorithm on M.

Another drawback is that this method can only be applied on numbers or data like numbers, since this method needs to find the middle of two elements in M, even if the middle element is not in M. Thus, this method is not general.

A better solution We use a prune-and-search method to iteratively eliminate elements that cannot be the answer until only one element remains.

In each iteration, let m_i be the median of the remaining elements in the i-th column. Let m^* be the median of the set $\{m_i \mid 0 \le i < m\}$ and let r be the rank of m^* in M. If r is k, then we found the answer. If r is larger than k, then for each column i such that $m_i \ge m^*$, all elements at least m_i can be eliminated. If r is smaller than k, then for each column i such that $m_i \le m^*$, all elements at most m_i can be eliminated.

Complexity analysis In each iteration, m^* can be determined in $O(m)$ time, and the rank of m^* can be computed in $O(m+n)$ time. Since half of columns are halved in each iteration, the number of iterations is $O(\lg n)$. Thus, the time complexity is $O((m+n)\lg n)$.

Remark. This problem can be solved in $O(n)$ time [6, 5, 14]. This problem appears on LintCode.

1.1.1 Other problems

Here are some other problems:

1. 3-sum: given an array A of n integers, three elements in A whose sum equals zero can be found in $o(n^2)$ time [10]. This problem appears on LeetCode.
2. all nearest smaller values: given an array A of n integers, an array S, in which $S[i]$ is the largest index $j < i$ subject to $A[j] < A[i]$, can be computed in linear time. That is, for all $0 \le i < n$, $S[i] = \max\{j \mid j < i \text{ and } A[j] < A[i]\}$ can be computed in linear time. The solution of this problem can be used to solve the largest rectangle in histogram problem on LeetCode. The solution of the largest rectangle in histogram problem can be used to solve the maximal rectangle problem on Leetcode.
3. inversion counting: given an array A of n integers, the number of inversions can be computed in $O(n \lg n)$ time. This problem appears on LeetCode.

1.2 Find subarray(s) or subsequence(s)

In this section, we focus on the problems of finding specified subarray(s) or subsequences in a given array.

Problem 5 Path partition

Given a positive integer k and an array A of n integers, design an algorithm to partition A into k subarrays so that the maximum subarray sum among these k subarrays is minimized.
That is, find a sequence of k pairs of indices $\langle (b_1, e_1) \cdots (b_k, e_k) \rangle$ minimizing $\max_{1 \le i \le k} \sum_{j=b_i}^{e_i} A[j]$ subject to

1. $b_1 = 0$ and $e_k = n - 1$,
2. $b_i \le e_i$ for all $1 \le i \le k$,
3. and $b_{i+1} = e_i + 1$ for all $1 \le i < k$.

Answer of exercise 5

For a number d, whether these n integers can be partitioned into k parts with the maximum subarray sum at most d can be verified in linear time by using a greedy approach. Let $S = \sum_{i=0}^{n-1} A[i]$ and let d^* be the minimum integer such that A can be partitioned into k parts with the maximum subarray sum at most d^*. Since d^* must be between $\lceil \frac{S}{k} \rceil$ and S, d^* can be found by using a binary search on integers between $\lceil \frac{S}{k} \rceil$ and S.

Complexity analysis The time complexity is $O(n \lg \frac{S(k-1)}{k})$.

A better solution Note that d^* must be $\sum_{k=i}^{j} A[k]$ for some i and j. Create a matrix M, such that $M[i][j] = \sum_{k=i}^{j} A[k]$, in which all rows and columns in M are in increasing order. Finding a pair of indices, i and j, such that $M[i][j]$ is d^*, is our goal.

We use a prune-and-search method to iteratively eliminate elements that cannot be the answer until only one element remains. Since creating M takes $O(n^2)$ time, the algorithm should not explicitly create M. Instead, the algorithm maintains a lower bound and an upper bound for each column to represent the current remaining elements in each column.

In each iteration, let m_i be the median of the remaining elements in the i-th column. Let m^* be the median of the set $C = \{m_i \mid 0 \leq i < m\}$. If A can be partitioned into k parts with the maximum subarray sum at most m^*, then d^* is at most m^*. Otherwise, d^* should be greater than m^*. All candidates that cannot be the answer can be pruned by using a saddle back search.

Complexity analysis In each iteration, m^* can be determined in $O(n)$ time. Since the saddleback search also takes $O(n)$ time, each iteration takes $O(n)$ time as well. Because at least a quarter of candidates are eliminated in each iteration, the time complexity is $O(n \lg n)$.

Remark. This solution is from the paper of Khanna *et al.* [12]. This problem can be solved in linear time [4]. This problem appears on LintCode.

Problem 6 k-covers

Given a positive integer k and an array A of n non-zero integers, design an algorithm to find k disjoints and non-empty subarrays, called a k-cover, with the maximum sum.

Answer of exercise 6

Without loss of generality, all consecutive positive/negative integers can be merged into one positive/negative integer. Thus, we assume that A has integers with alternating signs, and the integers at the both ends are positive; that is $\text{sgn}(A[i]) \neq \text{sgn}(A[i+1])$ for all $0 \leq i < n-1$, and both $A[0]$ and $A[n-1]$ are positive, where $n \geq 2k-1$ is an odd integer.

Creating a prefix sum array P, $P[i] = \sum_{j=0}^{i} A[i]$, for all $0 \leq i < n$ is a common trick for solving problems involving subarray sum, since the sum of $A[b \dots e] = \sum_{i=b}^{e-1} A[q]$ becomes $P[e] - P[b-1]$ assuming $P[-1] = 0$. The problem then becomes

$$\boxed{\text{find a sequence of disjoint index intervals } \langle [b_0, e_0], \dots, [b_{k-1}, e_{k-1}] \rangle \text{ maximizing } \sum_{i=0}^{k-1} P[e_i] - P[b_i].}$$

Since the signs of integers alternate in A, $P[i-1]$ is smaller than $P[i]$ for all even $0 \leq i \leq n-1$ and $P[i]$ is larger than $P[i+1]$ for all even $0 \leq i < n-1$. Thus, in an optimal solution, each interval begins at an odd index and ends at an even index. For convenience, the *value* of an index interval $[b, e]$ is defined as $P[e] - P[b]$, and its *value interval* is defined as $[P[b], P[e]]$. An index interval $[b, e]$ with a positive value satisfying b is odd and e is even is called a *useful* interval. Any optimal solution only contains useful intervals.

Now, consider two useful intervals $[b, e]$ and $[b', e']$, where $e < b'$. For these two intervals, four possibilities exist in an optimal solution. An optimal solution may take none of these intervals, may take both intervals, may take exactly one of the intervals, or may take the interval $[b, e']$. We can rule out some possibilities based on the relative order of the value intervals of $[b, e]$ and $[b', e']$.

Let $[l, h]$ and $[l', h']$ be the value intervals of $[b, e]$ and $[b', e']$ respectively. If $l \geq l'$, for any solution containing $[b, e'']$ where $e' \leq e''$, another solution can be constructed by replacing the interval $[b, e'']$ by $[b', e'']$, and the resulting solution is no worse than the original solution. Thus, combining the interval $[b, e]$ with other intervals after $[b', e']$ is not necessary.

Based on this observation, we maintain a stack S of value intervals so that

1. let $[l_0, h_0] \dots [l_t, h_t]$ be their value intervals from the bottom to the top. The value interval $[l_i, h_i]$ strictly includes $[l_{i+1}, h_{i+1}]$ for all $0 \leq i < t-1$.

2. let $[b_0, e_0] \dots [b_t, e_t]$ be their corresponding index intervals from the bottom to the top. These intervals are non-empty and satisfy $e_i < b_{i+1}$ for all $0 \leq i < t-1$.

All useful index intervals having the following form $[b, b+1]$ are processed in order. Let $[l, h]$ be the left end of the top value interval in the stack. For each value interval $[l', h']$, three cases are possible:

1. $l \geq l'$: since the top value interval will never be combined with any interval after the current interval, the top interval is popped from S, and the value of the top interval, $h_t - l_t$, is stored in a list L.

2. $l < l'$ and $h > h'$: push the current interval to S.

3. $l < l'$ and $h \leq h'$: merge the top interval into the current interval. That is, the top interval is popped, and change the current interval to $[l, h']$. Since an optimal solution may need to use these two intervals separately if $h - l' > 0$, an additional value $h - l'$ is stored in L.

In the end, the k largest values in L and S are selected, which will be the maximum sum of k disjoint subarrays.

Complexity analysis Since each index interval can be pushed and popped at most once, the time complexity is $O(n)$.

Remark. Bengtsson and Chen designed the first linear time algorithm for this problem [8]. This problem appears on LintCode and LeetCode. When $k = 1$, the problem can be solved by using Kadane's algorithm. This special case appears on LeetCode.

1.2.1 Other problems

Here are some other problems:

1. longest increasing subsequence: this problem can be solved in $n \lg n + O(n)$ time [7]. This problem appears on LeetCode.

1.3 Manipulate

In this section, we focus on the problems of manipulating a given array.

Problem 7

Given a positive integer k and an array A of n integers, design an algorithm to modify A into another array B of n integers in which $|B[i] - B[i+1]| \leq k$ for all $0 \leq i < n-1$, while minimizing $\sum_{i=0}^{n-1} |A[i] - B[i]|$.

Answer of exercise 7

This problem can be formulated as a linear programming problem. Two set of variables, X and Y, are used to represent the result $B[i] = A[i] + X[i] - Y[i]$. The objective function is the summation of X and Y. By the difference constraint, $-k \leq B[i] - B[i+1] \leq k$ is required, and this can be formulated as $-k \leq A[i] + X[i] - Y[i] - A[i+1] - X[i+1] + Y[i+1] \leq k$. After algebraic manipulation and transform into standard form by introducing slack variables S and T, the following two equality constraints can be obtained:

$$X[i] - Y[i] - X[i+1] + Y[i+1] + S[i] = k - P[i] \text{ and } X[i] - Y[i] - X[i+1] + Y[i+1] - T[i] = -k - P[i],$$

where $P[i] = A[i] - A[i+1]$.

From these two equality constraints, $S[i] + T[i]$ must be equal to $2k$. Thus, one of the equality constraint is removed, and an upper bound $2k$ to $S[i]$ or $T[i]$ is introduced. If one one redundant constraint by summing all $n-1$ equality constraints is introduced and all constraints are negated on both sides, the formulation becomes a min-cost circulation problem. Since the underlying graph is a series-parallel graph and min-cost circulation on series-parallel graphs can be solved in $O(n \lg n)$ time, this problem can be solved in $O(n \lg n)$ time as well.

Remark. This solution is from here. Booth and Tarjan designed an $O(n \lg n)$-time algorithm for finding a min-cost circulation on series-parallel graphs [2]. This problem appears on LintCode.

Problem 8 Block interchange

Given two indices a and b and an array A of n integers, design an algorithm to interchange $A[0 \dots a]$ with $A[b \dots n-1]$. That is, initially A equals $P + M + S$, where $P = A[0 \dots a]$, $M = A[a+1 \dots b-1]$, and $S = A[b \dots n-1]$, but A equals $S + M + P$ in the end.

Answer of exercise 8

One easy solution is as follows: First, reverse the whole string so that the string becomes $A^R = S^R + M^R + P^R$. Then, reverse S^R, M^R, and P^R respectively. However, this algorithm is not optimal. For example, when $|P| = |S|$, only $|P|$ swaps are required, but the algorithm uses n swaps.

A better solution The idea is to move characters to the desired locations directly. If $|P| \le |S|$, then first swap P with the last $|P|$ characters of S. The intermediate result is $S'' + M + S' + P$, where $S = S' + S''$ and $|S''| = |P|$. Since the desired result is $S + M + P$, the algorithm recursively interchanges $S'' + M$ with S' in the subarray $S'' + M + S'$.

If $|P| > |S|$, then first swap the first $|S|$ characters of P with S. The intermediate result is $S + P'' + M + P'$, where $P = P' + P''$ and $|P'| = |S|$. Since the desired result is $S + M + P$, the algorithm recursively interchanges P'' with $M + P'$ in the subarray $P'' + M + P'$.

Complexity analysis The algorithms uses $n - \gcd(|P| + |M|, |M| + |S|)$ swaps.

Remark. This solution is from Mohammed and Subi's paper [16]. One special case, right shift circularly, appears on LeetCode.

1.3.1 Other problems

Here are some other problems:

1. matching nuts and bolts: given n nuts and n bolts, every bolt matches exactly one nuts. You can try to match a nut and a bolt and see which one is bigger or they can be matched. However, you cannot compare two bolts or two nuts. All bolts and nuts can be matched in $O(n \lg n)$ time deterministically [13]. This problem can also be solved easily by using the randomized quick sort.[1] This problem appears on LintCode.

2. partition: given an integer k and an array A of n integers, A can be rearranged in linear time so that all integers smaller than k are in front of all integers at least k by using Hoare's partition, which is more efficient than Lomuto partition.[2] This problem appears on Lintcode.

3. Dutch national flag problem: this problem appears on LeetCode.

1.4 Range query

In this section, we focus on the problems of building data structures to support range queries.

1. range sum query: given an array A of n integers, build a data structure to represent A supporting the following two queries:
   (a) given a pair of indices $i < j$, compute the subarray sum of $A[i \dots j]$.
   (b) given an index i and a value v, set $A[i]$ to v.

   This problem can be solved by using a Fenwick tree. This problem appears on LeetCode.

2. range minimum query problem: this problem appears on LintCode.

[1] Using Lomuto partition is easier.
[2] http://cs.stackexchange.com/questions/11458/quicksort-partitioning-hoare-vs-lomuto.

References

[1] Richard S. Bird. “Improving Saddleback Search: A Lesson in Algorithm Design”. In: *Mathematics of Program Construction, MPC*. Volume 4014. Lecture Notes in Computer Science. Springer, 2006, pages 82–89. ISBN: 978-3-540-35631-8. DOI: 10.1007/11783596_8. URL: http://dx.doi.org/10.1007/11783596_8 (cited on page 2).

[2] Heather Booth and Robert Endre Tarjan. “Finding the Minimum-Cost Maximum Flow in a Series-Parallel Network”. In: *Journal of Algorithms* 15.3 (November 1993), pages 416–446. ISSN: 0196-6774. DOI: 10.1006/jagm.1993.1048. URL: http://dx.doi.org/10.1006/jagm.1993.1048 (cited on page 6).

[3] Robert S. Boyer and J. Strother Moore. “MJRTY: A Fast Majority Vote Algorithm”. In: *Automated Reasoning: Essays in Honor of Woody Bledsoe*. 1991, pages 105–118. URL: http://www.cs.utexas.edu/users/moore/best-ideas/mjrty/index.html (cited on page 2).

[4] Greg N. Frederickson. “Optimal Algorithms for Tree Partitioning”. In: *ACM/SIGACT-SIAM Symposium on Discrete Algorithms, SODA*. January 1991, pages 168–177. URL: http://dl.acm.org/citation.cfm?id=127787.127822 (cited on page 4).

[5] Greg N. Frederickson and Donald B. Johnson. “Generalized Selection and Ranking: Sorted Matrices”. In: *SIAM Journal on Computing* 13.1 (1984), pages 14–30. ISSN: 0097-5397. DOI: 10.1137/0213002. URL: http://dx.doi.org/10.1137/0213002 (cited on page 3).

[6] Greg N. Frederickson and Donald B. Johnson. “The Complexity of Selection and Ranking in X+Y and Matrices with Sorted Columns”. In: *Journal of Computer and System Sciences* 24.2 (April 1982), pages 197–208. ISSN: 0022-0000. DOI: 10.1016/0022-0000(82)90048-4. URL: http://dx.doi.org/10.1016/0022-0000(82)90048-4 (cited on page 3).

[7] Michael L. Fredman. “On computing the length of longest increasing subsequences”. In: *Discrete Mathematics* 11.1 (1975), pages 29–35. ISSN: 0012-365X. DOI: 10.1016/0012-365X(75)90103-X. URL: http://dx.doi.org/10.1016/0012-365X(75)90103-X (cited on page 5).

[8] Pawel Gawrychowski and Patrick K. Nicholson. “Encodings of Range Maximum-Sum Segment Queries and Applications”. In: *Combinatorial Pattern Matching, CPM*. Volume 9133. Lecture Notes in Computer Science. June 2015, pages 196–206. ISBN: 978-3-319-19929-0. DOI: 10.1007/978-3-319-19929-0_17. URL: http://dx.doi.org/10.1007/978-3-319-19929-0_17 (cited on page 5).

[9] David Ginat. “Insight Tasks for Examining Student Illuminations”. In: *Olympiads in Informatics* 6 (2012), pages 44–52. ISSN: 1822-7732. URL: http://www.ioinformatics.org/oi/shtm/INFOL106.shtml (cited on page 1).

[10] Allan Grønlund Jørgensen and Seth Pettie. “Threesomes, Degenerates, and Love Triangles”. In: *IEEE Annual Symposium on Foundations of Computer Science, FOCS*. October 2014, pages 621–630. DOI: 10.1109/FOCS.2014.72. URL: http://dx.doi.org/10.1109/FOCS.2014.72 (cited on page 3).

[11] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. “A simple algorithm for finding frequent elements in streams and bags”. In: *ACM Transactions on Database Systems* 28 (March 2003), pages 51–55. ISSN: 0362-5915. DOI: 10.1145/762471.762473. URL: http://doi.acm.org/10.1145/762471.762473 (cited on page 2).

[12] Sanjeev Khanna, S. Muthukrishnan, and Steven Skiena. “Efficient Array Partitioning”. In: *International Colloquium Automata, Languages and Programming, ICALP*. Volume 1256. Lecture Notes in Computer Science. July 1997, pages 616–626. ISBN: 978-3-540-63165-1. DOI: 10.1007/3-540-63165-8_216. URL: http://dx.doi.org/10.1007/3-540-63165-8_216 (cited on page 4).

[13] János Komlós, Yuan Ma, and Endre Szemerédi. “Matching Nuts and Bolts in O(n log n) Time”. In: *SIAM Journal on Discrete Mathematics* 11.3 (1998), pages 347–372. ISSN: 0895-4801. DOI: 10.1137/S0895480196304982. URL: http://dx.doi.org/10.1137/S0895480196304982 (cited on page 6).

[14] Andranik Mirzaian and Eshrat Arjomandi. “Selection in X+Y and Matrices With Sorted Rows and Columns”. In: *Information Processing Letters* 20.1 (January 1985), pages 13–17. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90123-1. URL: http://dx.doi.org/10.1016/0020-0190(85)90123-1 (cited on page 3).

[15] Jayadev Misra and David Gries. “Finding Repeated Elements”. In: *Science of Computer Programming* 2.2 (November 1982), pages 143–152. ISSN: 0167-6423. DOI: `10.1016/0167-6423(82)90012-0`. URL: `http://dx.doi.org/10.1016/0167-6423(82)90012-0` (cited on page 2).

[16] John L. Mohammed and Carlos S. Subi. “An Improved Block-Interchange Algorithm”. In: *Journal of Algorithms* 8.1 (March 1987), pages 113–121. ISSN: 0196-6774. DOI: `10.1016/0196-6774(87)90031-9`. URL: `http://dx.doi.org/10.1016/0196-6774(87)90031-9` (cited on page 6).

Chapter 2

Problems on strings

A string is a sequence of characters.

Notation For a string S of length n, $S[0], \ldots, S[n-1]$ denotes S's characters in order, and $S[i \ldots j]$ denotes the substring starting at i ending at j. For two strings S and T, $S + T$ denotes the concatenation of S and T.

2.1 Determine properties

In this section, we focus on the problems of determining properties of a given string.

Problem 9 Cyclic equality

Two strings S and T are *cyclic-equivalent* if S can be transformed into T by shift circularly. That is, for some index i, $S[i \ldots n-1] + S[0 \ldots i-1]$ equals T. Given two strings S and T of the same length n, design an algorithm to determine whether S and T are cyclic-equivalent.

Answer of exercise 9

For a string A, we use A^i to denote the string of rotating A to the left by i positions circularly; that is $A^i = A[i \ldots n-1] + A[0 \ldots i-1]$. The *lexicographically minimal string* of a string A is $\min_i A^i$ with respect to the lexicographical order.

Let S^* and T^* be the lexicographically minimal strings of S and T respectively. The algorithm is based on the following observation: strings S and T are cyclic-equivalent if and only if $S^* = T^*$.

Let D_S be the set of indices i that S^i is larger than T^j for some j; that is, $D_S = \{i \mid \exists_j T^j < S^i\}$. Similarly, D_T can be defined. Note that S and T are not cyclic-equivalent, if and only if both D_S or D_T contains $\{0, \ldots, n-1\}$. The idea is to compute D_S and D_T incrementally and test whether D_S or D_T contains $\{0, \ldots, n-1\}$.

The computation of D_S is based on the following fact: if $S[i+l] = T[j+l]$ for all $0 \le l < k$ and $S[i+k] < T[i+k]$ for some k, then $S^{i+l} < T^{i+l}$ for all $0 \le l \le k$. Thus, for a give pair of indices (i, j), the algorithm finds the largest k such that $S[i+l] = T[j+l]$ for all $0 \le l < k$ and $T[i+k] < S[i+k]$, then D_S must contain $\{i \ldots i+k\}$. Repeatedly apply this rule to compute both D_S and D_T.

Complexity analysis The time complexity is $O(n)$ and the space complexity is $O(1)$.

Remark. This solution is from Shiloach's paper [5].[1] Another approach is to find the lexicographically minimal string rotation directly. Since lexicographically minimal string rotation can be found in linear time using $O(1)$ space, this problem can be solved in linear time using $O(1)$ space. This problem appears on CodeEval.

[1] `http://stackoverflow.com/a/31001317/1260984`.

2.1.1 Other problems

Here are some other problems:

1. string matching: this problem can be solved in linear time using constant space [4]. This problem appears on LeetCode.

2. credit card number validation: a credit card can be verified by using Luhn algorithm. This problem appears CodeEval.

3. pangram: this problem appears on CodeEval.

4. anagram: this problem appears on LeetCode.

2.2 Find substring(s)

1. minimum palindromic factorization: given a string S of length n, partition S into the minimum number of substrings such that each substring is a palindrome. That is, find the minimum number of substrings $S_1 \dots S_k$ such that $S = S_1 + \cdots + S_k$ and each S_i is a palindrome. This problem can be solved in $O(n \lg n)$ time [2, 3]. This problem appears on LeetCode.

2. longest palindromic substring: given a string S of length n, the longest palindromic substring can be found in linear time using Manacher's algorithm. This problem appears on LeetCode.

3. longest common substring: given two strings S and T, the longest common substring can be found in linear time using a generalized suffix tree. This problem appears on LintCode.

4. rolling hash: this problem appears on LeetCode.

2.3 Approximate string matching

Problem 10 Wildcard matching

Let '.' be a wildcard character that can match any single character and '*' be a wildcard character that can match any sequence of characters. Given a string T of length n and a pattern P of length m in which P may contain wildcard characters, design an algorithm to determine whether P matches T.

Answer of exercise 10

Since the wildcard '*' can match any sequence of characters, our idea is to partition the pattern into several sub-patterns separated by '*', and match each sub-pattern independently.

Without loss of generality, we assume that $P[0]$ and $P[m-1]$ are '*''s. Suppose that P contains $k+1$ '*''s. The string P can be partitioned into k sub-patterns, $S_1, \dots, S_k$, and $k-1$ $*$'s such that $P = * + S_1 + * + \cdots + * + S_k + *$, and each sub-pattern S_i has no '*' wildcard.

Consider the first two sub-patterns S_1 and S_2. Let i_1 be the smallest index that $T[i_1 \dots i_1 + |S_1| - 1]$ matches S_1. Since match S_1 and S_2 should be matched in order, S_2 can only be matched to a substring starting after $i_1 + |S_1| - 1$. Moreover, since a '*' wildcard appears between S_2 and S_3, it suffices to find the smallest $i_2 \geq i_1 + |S|$ that $T[i_2 \dots i_2 + |S_2| - 1]$ matches S_2. Let i_j be the smallest index at least $i_{j-1} + |S_{j-1}|$ that $T[i_j \dots i_j + |S_j| - 1]$ matches S_j. If all i_j's are well-defines, then we found a match. Thus, the crucial part of this problem is as follows: given a pattern S without a '*' wildcard, find the first match of S in T.

Let S be a pattern without a '*' wildcard. Suppose that all characters in the alphabet are represented by positive integers, and the wildcard '.' is represented by zero. Then, for each start position i in T, the expression $\sum_{j=0}^{m-1} S[j]T[i+j](S[j] - T[i+j])^2$ is zero if and only if S matches $T[i \dots |S|]$. The expressions for all start positions can be evaluated in $O(n \lg n)$ time by using the fast Fourier transform.

In order to reduce the time complexity to $O(n \lg |S|)$, we create $h = \frac{n}{|S|}$ overlapping substrings of T, $T_1, \dots, T_h$, where $|T_a| = 2m$ for all $1 \leq a \leq h$, T_1 is a prefix of T, and the first m characters of T_a is the same as the last m characters of T_{a-1} for all $1 < a \leq h$. We evaluate the expressions in the order of

$T_1, \ldots, T_h$ and the time complexity is $O(n \lg|S|)$. Moreover, the index i of the first match can be computed in $O((i + |S|) \lg|S|)$.

Complexity analysis The time complexity is $O(n \lg|S|)$.

Remark. This solution is from Clifford and Clifford's paper [1]. This problem appears on LeetCode.

2.3.1 Other problems

Here are other problems:

1. edit distance: this problem can be solved in $O(\frac{nm}{\lg m})$ time by using four Russians method. This problem appears on LeetCode.
2. longest common subsequence: this problem appears on LintCode.

2.4 Regular expression

1. regular expression: given a string S and a regular expression R, test whether S can be matched to R. This problem can be solved by using Thompson's algorithm. Special case of this problem appears on LeetCode.
2. e-mail validation: given a string S, test whether S is a valid e-mail address. A complete solution can be found here. This problem appears on CodeEval.

2.5 Decode

1. inverse Burrows-Wheeler transform: this problem appears on CodeEval.
2. decode Gronsfeld cipher: this problem appears on CodeEval.

References

[1] Peter Clifford and Raphaël Clifford. "Simple deterministic wildcard matching". In: *Information Processing Letters* 101.2 (January 2007), pages 53–54. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2006.08.002. URL: http://dx.doi.org/10.1016/j.ipl.2006.08.002 (cited on page 11).

[2] Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. "A subquadratic algorithm for minimum palindromic factorization". In: *Journal of Discrete Algorithms* 28 (September 2014), pages 41–48. ISSN: 1570-8667. DOI: 10.1016/j.jda.2014.08.001. URL: http://dx.doi.org/10.1016/j.jda.2014.08.001 (cited on page 10).

[3] Mikhail Rubinchik and Arseny M. Shur. "Eertree: An Efficient Data Structure for Processing Palindromes in Strings". In: *CoRR* abs/1506.04862 (2015). URL: http://arxiv.org/abs/1506.04862 (cited on page 10).

[4] Wojciech Rytter. "On maximal suffixes, constant-space linear-time versions of KMP algorithm". In: *Theoretical Computer Science* 1-3.299 (April 2003), pages 763–774. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(02)00590-X. URL: http://dx.doi.org/10.1016/S0304-3975(02)00590-X (cited on page 10).

[5] Yossi Shiloach. "A Fast Equivalence-Checking Algorithm for Circular Lists". In: *Information Processing Letters* 8.5 (June 1979), pages 236–238. ISSN: 0020-0190. DOI: 10.1016/0020-0190(79)90114-5. URL: http://dx.doi.org/10.1016/0020-0190(79)90114-5 (cited on page 9).

Chapter 3

Data structures

3.1 Stack & queue

Problem 11 Implement a stack using queues

Implement a stack by using queues.

Answer of exercise 11

Use two queues. The first queue stores all top elements in the stack with size at most $O(\sqrt{n})$. The second queue stores all other elements. Both queues maintain stack order.

For the pop operation, if the first queue is not empty, dequeue from the first queue. Otherwise, dequeue from the second queue.

For push operation, enqueue to the first queue. Then, use queue operations to reorder the elements in the first queue to maintain stack order. If the size of the first queue is larger than $\sqrt{n}$, then insert all elements in the second queue to the first queue and swap first queue with the second queue.

Complexity analysis The pop operation takes $O(1)$ time in the worst case. The push operation takes $O(\sqrt{n})$ amortized time. In general, a stack can be simulated by k queues so that each of push and pop operations takes $O(n^{\frac{1}{k}})$ amortized time.

Remark. The above solution is from Hühne's paper [4].[1] This problem appears on LeetCode.

3.1.1 Other problems

1. implement a queue using stacks: this problem has been used in Ph.D. qualify exam in Princeton.[2] A queue can be simulated by stacks so that each operation takes $O(1)$ time in the worst case [3].[3] A deque can be simulated by stacks so that each operation takes $O(1)$ time in the worst case [1]. This problem appears on LeetCode.

2. implement a deque with max operation: an implementation of deque with max operation for which the worst-case time per operation is $O(1)$ exists [2]. The special case of implementing a queue with max operation appears on LeetCode.

3. evaluate reverse Polish notation: this problem appears on LeetCode.

4. evaluate infix expression: this problem can be solved by using the shunting-yard algorithm. This problem appears on LintCode.

[1]`http://cstheory.stackexchange.com/questions/2562/one-stack-two-queues`.
[2]`https://rjlipton.wordpress.com/2010/04/25/natural-natural-and-natural/`.
[3]`http://stackoverflow.com/a/5573398/1260984`.

3.2 Heap

1. median filter: one special case of finding the running medians appears on LeetCode.

3.3 Tree

1. Morris traversal: traverse a binary tree without using a stack. One application is to invert a binary tree, which appears on LeetCode.
2. encode binary tree: this problem appears on LeetCode.

3.4 Graph

1. Euler path: the problem of finding the smallest Euler path in the lexical order appears on LeetCode.
2. connected-component labeling: this problem appears on LeetCode.
3. topological sort: this problem appears on LintCode.

References

[1] Tyng-Ruey Chuang and Benjamin Goldberg. "Real-Time Deques, Multihead Thring Machines, and Purely Functional Programming". In: *Functional programming languages and computer architecture, FPCA*. 1993, pages 289–298. ISBN: 0-89791-595-X. DOI: `10.1145/165180.165225`. URL: `http://doi.acm.org/10.1145/165180.165225` (cited on page 13).

[2] Hania Gajewska and Robert Endre Tarjan. "Deques with Heap Order". In: *Information Processing Letters* 22.4 (April 1986), pages 197–200. ISSN: 0020-0190. DOI: `10.1016/0020-0190(86)90028-1`. URL: `http://dx.doi.org/10.1016/0020-0190(86)90028-1` (cited on page 13).

[3] Robert Hood and Robert Melville. "Real-Time Queue Operation in Pure LISP". In: *Information Processing Letters* 13.2 (November 1981), pages 50–54. ISSN: 0020-0190. DOI: `10.1016/0020-0190(81)90030-2`. URL: `http://dx.doi.org/10.1016/0020-0190(81)90030-2` (cited on page 13).

[4] Martin Hühne. "On the Power of Several Queues". In: *Theoretical Computer Science* 113.1 (May 1993), pages 75–91. ISSN: 0304-3975. DOI: `10.1016/0304-3975(93)90211-B`. URL: `http://dx.doi.org/10.1016/0304-3975(93)90211-B` (cited on page 13).

Chapter 4

Algorithms design

4.1 Greedy

Problem 12

Given an array A of n integers, design an algorithm to construct an array B of n integers minimizing $\sum_{i=0}^{n-1} B[i]$ subject to:

1. $B[i] > 0$ for all $0 \leq i < n$.
2. $B[i] > B[i+1]$ if $A[i] > A[i+1]$ for all $0 \leq i < n-1$.
3. $B[i] < B[i+1]$ if $A[i] < A[i+1]$ for all $0 \leq i < n-1$.

Answer of exercise 12

This problem can be considered as a constraint satisfaction problem in which all $B[i]$'s are variables, and corresponding constraints can be modeled. Since the constraint graph is a path, the problem can be solved in linear time by enforcing directional consistency first and then assign values.

For each i, let D_i be the domain of $B[i]$. The algorithm is as follows: initially, $D[i]$ is the set of all positive integers for all $0 \leq i < n$. Then, process $A[i]$ for $i = 0$ to $i = n-1$ in order. If $A[i] < A[i+1]$, then the domain of $B[i+1]$ is changed to the set of all positive integers that are larger than $\min D_i$. After all $A[i]$ are processed, all domains are directional consistent in the constraint graph. Finally, assign all $B[i]$ from $i = n-1$ to 0 to the smallest integer in D_i that are consistent with the previous variables.

Remark. This problem appears on LeetCode.

Problem 13 Gasoline puzzle

There are n gas station around a circle and station i has a finite amount of gas $g[i]$. You can only drive car in increasing order of i and wrapping around in the n-th station. The fuel consumption from the station i to the station $(i+1) \bmod n$ is $c[i]$. Design an algorithm to find a feasible stating location so that you can finish a round trip.

Answer of exercise 13

Let $x[i] = g[i] - c[i]$. If the summation of $x[i]$ is less than zero, then round-trip is impossible. Let $P[i]$ be the summation of $x[0 \ldots i]$. Than pick the index i that minimizes $P[i-1]$ as the starting point. For any station $j > i$, the amount of gas that we will have at station j is $P[j-1] - P[i-1]$, which is never negative. Similarly, it is feasible to arrive all stations $j < i$ and back to i.

Remark. The solution can be used to solve combinatorial optimization problem [1]. This problem appears on LeetCode.

4.1.1 Other problems

1. change-making problem: whether the greedy method is applicable can be tested in polynomial time [2]. This problem appears on LeetCode.

2. interval scheduling: this problem appears on Codility.

4.2 Backtracking

1. Sudoku: this problem appears on LeetCode.

2. n-queens: this problem appears on LeetCode.

4.3 Dynamic programming

1. subset sum problem: this problem appears on LintCode.

2. knapsack problem: this problem appears on LintCode.

4.4 Computational geometry

1. collinear points: given a set of points in a plane, collinear points can be identified in $O(n^2)$ time in the worst case using $O(n)$ space by using arrangement of lines. One special case of finding the maximum number of collinear points appears on LeetCode.

References

[1] André Berger, Vincenzo Bonifaci, Fabrizio Grandoni, and Guido Schäfer. "Budgeted matching and budgeted matroid intersection via the gasoline puzzle". In: *Mathematical Programming* 128.1-2 (June 2011), pages 355–372. ISSN: 0025-5610. DOI: 10.1007/s10107-009-0307-4. URL: http://dx.doi.org/10.1007/s10107-009-0307-4 (cited on page 15).

[2] David Pearson. "A polynomial-time algorithm for the change-making problem". In: *Operations Research Letters* 33.3 (May 2005), pages 231–234. ISSN: 0167-6377. DOI: 10.1016/j.orl.2004.06.001. URL: http://dx.doi.org/10.1016/j.orl.2004.06.001 (cited on page 16).

Chapter 5

Math problems

5.1 Combinatorics

Problem 14

Given four positive integers, n, k, l, and h, enumerate all possible k-tuple $(x_1, \ldots, x_k)$ such that

1. $x_i < x_{i+1}$ for all $0 \leq i < k$.
2. $l \leq x_i \leq h$ for all $0 \leq i \leq k$.
3. $n = \sum_{i=1}^{k} x_i$.

Answer of exercise 14

One simple solution is to use backtracking directly. However, since lower-bound and upper-bound constraints exist, the backtracking may encounter dead end and this may slow down the enumeration.

A method is called *backtrack-free* if whenever we assign a value to a variable, a feasible solution always exists. A backtrack-free method can be designed based on the following fact: for four positive integers n, k, l, and h, a feasible partition of n into k parts exists if and only if

$$k \cdot l + \binom{k-1}{2} \leq n \leq k \cdot h - \binom{k-1}{2}.$$

Remark. This solution is from Riha and James' paper [8]. One special case of this problem appears on LeetCode.

5.1.1 Subset

1. generating the power set: This problem appears on LeetCode.

5.1.2 Combination

1. combination: $\binom{n}{k}$ can be computed efficiently [4]. This problem appears on LeetCode.
2. all combinations: all combinations can be generated efficiently in lexical order by using Gosper's Hack. Loopless algorithm exists (not in lexical order) [10]. This problem appears on LeetCode.

5.1.3 Permutation

1. all permutations: all permutations can be generated efficiently (not in lexical order) by using Heaps' algorithm. This problem appears on LeetCode.
2. next permutation: this problem appears on LeetCode. The problem of finding the previous permutation appears on LintCode.
3. rank of a permutation in lexical order: this problem can be solved by using Lehmer code. Linear time algorithm exists [5]. This problem appears on LintCode.
4. unrank to a permutation in lexical order: this problem appears on LeetCode.
5. all permutations for a multi-set: loopless algorithm with constant space usage exists (not in lexical order) [12]. This problem appears on LeetCode.
6. rank of a permutation for a multi-set in lexical order: this problem appears on LintCode.

5.1.4 Catalan number

1. all parentheses: loopless algorithm with constant space usage exists [9]. This problem appears on LeetCode.
2. all binary search trees: this problem appears on LeetCode.
3. Catalan number: this problem appears on LeetCode.

5.2 Arithmetic

1. exponentiation by squaring: this problem appears on LeetCode.
2. integer multiplication: this problem appears on LeetCode.
3. digital root: this problem appears on LeetCode.

5.3 Numerical

1. Horner's method: this problem appears on LintCode.
2. integer square root: this problem appears on LeetCode.
3. compute the digits of π: digits of π can be computed by using Spigot algorithm [2]. This problem appears on CodeEval.

5.4 Number theory

1. sieve of Eratosthenes: this problem appears on LeetCode.
2. repeating decimal: this problem appears on LeetCode.
3. regular number: this problem appears on LeetCode.
4. happy number: this problem appears on LeetCode.
5. Lagrange's four-square theorem: this problem appears on LeetCode.
6. palindromic number: this problem appears on LeetCode.
7. narcissistic number: this problem appears on CodeEval.

8. Lychrel number: this problem appears on CodeEval.
9. palindromic prime: this problem appears on CodeEval.
10. self-descriptive number: this problem appears on CodeEval.

5.5 Bitwise operations

Problem 15

Given an array A of distinct $n-k$ positive integers, where $A[i] < n$ for all $0 \leq i < n-k-1$, design an algorithm to find the k positive integers that are smaller than n but not in A.

Answer of exercise 15

Let $x_1, \ldots, x_k$ be the missing numbers. By scanning through A, $b_h = \sum_{i=1}^{k} x_i^h$ can be computed, for all $1 \leq h \leq k$. Then, using Newton's identities, a polynomial $P(x)$ can be constructed, such that the roots of P are $x_1, \ldots, x_k$. Finally, the roots can be obtained by either factoring P or evaluating P on all integers between zero and $n-1$ In order to reduce the space usage, all operations are performed in the ring Z_p, where p is a prime, $n \leq p \leq 2n$. The resulting algorithm uses only one-pass.

Remark. This solution is from the paper of Minsky *et al.* [6]. This problem is mentioned in Muthukrishnan's textbook of streaming algorithms [7].[1] The special case of $k = 1$ appears on LeetCode.

Problem 16

Suppose that all integers are represented by w bits in a computer. All integer arithmetics $(+, -, \times, /)$ and all bitwise operations $\text{and}, \text{or}, \text{xor}, \text{nor}$ can be done in $O(1)$ time. Given an integer x, compute $\lfloor \lg x \rfloor$.

Answer of exercise 16

$\lfloor \lg x \rfloor$ can be computed in $O(\lg w)$ time by using a binary search, but this method involves $O(\lg w)$ branches. Since comparison may be expensive for some architectures, using branches is not desirable.

A better solution The idea is to normalize x into an integer that has exactly one set bit in the binary representation, and the answer can found by table lookup. Specifically, an integer $z = 2^{\lfloor \lg x \rfloor + 1}$ will be created.

First, set all bits that are on the right of the most significant bit to one. This can be done by using the following operations (in C language) assuming $w = 32$:

```
y = x;
y |= y >> 1;
y |= y >> 2;
y |= y >> 4;
y |= y >> 8;
y |= y >> 16;
```

In the end, $y = 2^{\lfloor \lg x \rfloor + 1} - 1$. Let $z = y + 1 = 2^{\lfloor \lg x \rfloor + 1}$. Since z has only one set bit in the binary representation, we can find the number of trailing zeros of z, which is $\lfloor \lg x \rfloor$ by using a De Bruijn sequence as follows:

```
static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 9, 1, 10, 13, 21, 2, 29, 11, 14, 16, 18, 22, 25, 3, 30,
  8, 12, 20, 28, 15, 17, 24, 7, 19, 27, 23, 6, 26, 5, 4, 31
};
r = MultiplyDeBruijnBitPosition[(z * 0x07C4ACDDU) >> 27];
```

[1]http://stackoverflow.com/questions/3492302/easy-interview-question-got-harder-given-numbers-1-100-find-the-missing-numbe.

Complexity analysis This method uses $O(\lg w)$ operations without any branch.

Remark. This solution is from Bit Twiddling Hacks. Leiserson *et al.* design an algorithm to find the index of the most significant bit by using a De Bruijn sequence [3]. Find the number of trailing zeros of an integer can be done in $O(1)$ time [1]. This problem appears on LeetCode.

5.5.1 Other problems

1. Hamming weight: this problem appears on LeetCode.
2. reverse bits: this problem can be solved in $O(\lg w)$ time.[2] This problem appears on LeetCode.

5.6 Other problems

1. Fibonacci number: this problem can be solved in $O(\lg n)$ time [11]. The problem of finding the $(n+1)$-th Fibonacci number appears on LeetCode.
2. Gray code: this problem appears on LeetCode.
3. look-and-say sequence: $O(\lg n)$-time algorithm exists.[3] This problem appears on LeetCode.
4. Thue-Morse sequence: this problem appears on CodeEval.
5. Josephus problem: this problem appears on CodeEval.

References

[1] Michael L. Fredman and Dan E. Willard. "BLASTING through the Information Theoretic Barrier with FUSION TREES". In: *ACM Symposium on Theory of Computing, STOC*. May 1990, pages 1–7. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100217. URL: http://doi.acm.org/10.1145/100216.100217 (cited on page 20).

[2] Jeremy Gibbons. "Unbounded Spigot Algorithms for the Digits of Pi". In: *The American Mathematical Monthly* 113.4 (April 2006), pages 318–328. ISSN: 0002-9890. URL: http://www.jstor.org/stable/27641917 (cited on page 18).

[3] Charles E. Leiserson, Harald Prokop, and Keith H. Randall. *Using de Bruijn Sequences to Index a 1 in a Computer Word*. Technical report. Department of Computer Science, Massachusetts Institute of Technology, 1998 (cited on page 20).

[4] Yannis Manolopoulos. "Binomial coefficient computation: recursion or iteration?" In: *SIGCSE Bulletin* 34.4 (December 2002), pages 65–67. ISSN: 0097-8418. DOI: 10.1145/820127.820168. URL: http://doi.acm.org/10.1145/820127.820168 (cited on page 17).

[5] Martin Mares and Milan Straka. "Linear-Time Ranking of Permutations". In: *Proceedings of European Symposium on Algorithms, ESA*. Volume 4698. Lecture Notes in Computer Science. October 2007, pages 187–193. ISBN: 978-3-540-75519-7. DOI: 10.1007/978-3-540-75520-3_18. URL: http://dx.doi.org/10.1007/978-3-540-75520-3_18 (cited on page 18).

[6] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. "Set reconciliation with nearly optimal communication complexity". In: *IEEE Transactions on Information Theory* 49.9 (September 2003), pages 2213–2218. ISSN: 0018-9448. DOI: 10.1109/TIT.2003.815784. URL: http://dx.doi.org/10.1109/TIT.2003.815784 (cited on page 19).

[7] Muthu S. Muthukrishnan. "Data Streams: Algorithms and Applications". In: *Foundations and Trends in Theoretical Computer Science* 1.2 (September 2005). DOI: 10.1561/0400000002. URL: http://dx.doi.org/10.1561/0400000002 (cited on page 19).

[2] https://graphics.stanford.edu/~seander/bithacks.html#ReverseParallel.
[3] http://www.njohnston.ca/2010/10/a-derivation-of-conways-degree-71-look-and-say-polynomial/.

[8] W. Riha and K. R. James. “Algorithm 29 efficient algorithms for doubly and multiply restricted partitions”. In: *Computing* 16.1-2 (March 1976), pages 163–168. ISSN: 0010-485X. DOI: 10.1007/BF02241987. URL: http://dx.doi.org/10.1007/BF02241987 (cited on page 17).

[9] Frank Ruskey and Aaron Williams. “Generating Balanced Parentheses and Binary Trees by Prefix Shifts”. In: *Computing: The Australasian Theory Symposium (CATS 2008)*. January 2008, pages 107–115. URL: http://crpit.com/abstracts/CRPITV77Ruskey.html (cited on page 18).

[10] Frank Ruskey and Aaron Williams. “The coolest way to generate combinations”. In: *Discrete Mathematics* 309.17 (September 2009), pages 5305–5320. ISSN: 0012-365X. DOI: 10.1016/j.disc.2007.11.048. URL: http://dx.doi.org/10.1016/j.disc.2007.11.048 (cited on page 17).

[11] Joseph Shortt. “An Iterative Program to Calculate Fibonacci Numbers in O(log n) Arithmetic Operations”. In: *Information Processing Letters* 7.6 (October 1978), pages 299–303. ISSN: 0020-0190. DOI: 10.1016/0020-0190(78)90022-4. URL: http://dx.doi.org/10.1016/0020-0190(78)90022-4 (cited on page 20).

[12] Aaron Williams. “Loopless generation of multiset permutations using a constant number of variables by prefix shifts”. In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*. January 2009, pages 987–996. URL: http://dl.acm.org/citation.cfm?id=1496770.1496877 (cited on page 18).