Algorithm Design Topics

8

This chapter discusses a selection of algorithm design topics.

Section 8.1 focuses on bit-parallel algorithms that use bit operations to efficiently process data. Typically, we can replace a for loop with bit operations, which may remarkably improve the running time of the algorithm.

Section 8.2 presents the amortized analysis technique, which can be used to estimate the time needed for a sequence of operations in an algorithm. Using the technique, we can analyze algorithms for determining nearest smaller elements and sliding window minima.

Section 8.3 discusses ternary search and other techniques for efficiently calculating minimum values of certain functions.

8.1 Bit-Parallel Algorithms

Bit-parallel algorithms are based on the fact that individual bits of numbers can be manipulated in parallel using bit operations. Thus, a way to design an efficient algorithm is to represent the steps of the algorithm so that they can be efficiently implemented using bit operations.

8.1.1 Hamming Distances

The *Hamming distance* $\texttt{hamming}(a, b)$ between two strings a and b of equal length is the number of positions where the strings differ. For example,

$$\texttt{hamming}(01101, 11001) = 2.$$

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_8

Consider the following problem: Given n bit strings, each of length k, calculate the minimum Hamming distance between two strings. For example, the answer for [00111, 01101, 11110] is 2, because

- hamming(00111, 01101) = 2,
- hamming(00111, 11110) = 3, and
- hamming(01101, 11110) = 3.

A straightforward way to solve the problem is to go through all pairs of strings and calculate their Hamming distances, which yields an $O(n^2k)$ time algorithm. The following function calculates the distance between strings a and b:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```

However, since the strings consist of bits, we can optimize the solution by storing the strings as integers and calculating distances using bit operations. In particular, if $k \leq 32$, we can just store the strings as `int` values and use the following function to calculate distances:

```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

In the above function, the *xor* operation constructs a string that has one bits in positions where a and b differ. Then, the number of one bits is calculated using the `__builtin_popcount` function.

Table 8.1 shows a comparison of running times of the original algorithm and the bit-parallel algorithm on a modern computer. In this problem, the bit-parallel algorithm is about 20 times faster than the original algorithm.

8.1.2 Counting Subgrids

As another example, consider the following problem: Given an $n \times n$ grid whose each square is either black (1) or white (0), calculate the number of subgrids whose all corners are black. For example, Fig. 8.1 shows two such subgrids in a grid.

Table 8.1 The running times of the algorithms when calculating minimum Hamming distances of n bit strings of length $k = 30$

Size n	Original algorithm (s)	Bit-parallel algorithm (s)
5000	0.84	0.06
10000	3.24	0.18
15000	7.23	0.37
20000	12.79	0.63
25000	19.99	0.97

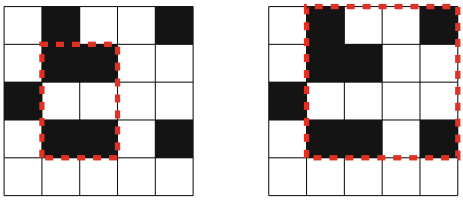


Fig. 8.1 This grid contains two subgrids with black corners

There is an $O(n^3)$ time algorithm for solving the problem: go through all $O(n^2)$ pairs of rows, and for each pair (a, b) calculate, in $O(n)$ time, the number of columns that contain a black square in both rows a and b. The following code assumes that `color`$[y][x]$ denotes the color in row y and column x:

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) {
        count++;
    }
}
```

Then, after finding out that there are `count` columns where both squares are black, we can use the formula `count(count` $- 1)/2$ to calculate the number of subgrids whose first row is a and last row is b.

To create a bit-parallel algorithm, we represent each row k as an n-bit bitset `row`$[k]$ where one bits denote black squares. Then, we can calculate the number of columns where rows a and b both have black squares using an *and* operation and counting the number of one bits. This can be conveniently done as follows using `bitset` structures:

```
int count = (row[a]&row[b]).count();
```

Table 8.2 shows a comparison of the original algorithm and the bit-parallel algorithm for different grid sizes. The comparison shows that the bit-parallel algorithm can be up to 30 times faster than the original algorithm.

Table 8.2 The running times of the algorithms for counting the subgrids

Grid size n	Original algorithm (s)	Bit-parallel algorithm (s)
1000	0.65	0.05
1500	2.17	0.14
2000	5.51	0.30
2500	12.67	0.52
3000	26.36	0.87

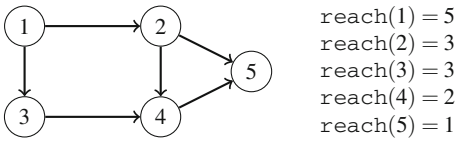


Fig. 8.2 A graph and its `reach` values. For example, $\texttt{reach}(2) = 3$, because nodes 2, 4, and 5 can be reached from node 2

8.1.3 Reachability in Graphs

Given a directed acyclic graph of n nodes, consider the problem of calculating for each node x a value $\texttt{reach}(x)$: the number of nodes that can be reached from node x. For example, Fig. 8.2 shows a graph and its `reach` values.

The problem can be solved using dynamic programming in $O(n^2)$ time by constructing for each node a list of nodes that can be reached from it. Then, to create a bit-parallel algorithm, we represent each list as a bitset of n bits. This permits us to efficiently calculate the union of two such lists using an *or* operation. Assuming that `reach` is an array of `bitset` structures and the graph is stored as adjacency lists in `adj`, the calculation for node x can be done as follows:

```
reach[x][x] = 1;
for (auto u : adj[x]) {
    reach[x] |= reach[u];
}
```

Table 8.3 shows some running times for the bit-parallel algorithm. In each test, the graph has n nodes and $2n$ random edges $a \to b$ where $a < b$. Note that the

Table 8.3 The running times of the algorithms when counting reachable nodes in a graph

Graph size n	Running time (s)	Memory usage (MB)
$2 \cdot 10^4$	0.06	50
$4 \cdot 10^4$	0.17	200
$6 \cdot 10^4$	0.32	450
$8 \cdot 10^4$	0.51	800
10^5	0.78	1250

algorithm uses a great amount of memory for large values of n. In many contests, the memory limit may be 512 MB or lower.

8.2 Amortized Analysis

The structure of an algorithm often directly tells us its time complexity, but sometimes a straightforward analysis does not give a true picture of the efficiency. *Amortized analysis* can be used to analyze a sequence of operations whose time complexity varies. The idea is to estimate the total time used to all such operations during the algorithm, instead of focusing on individual operations.

8.2.1 Two Pointers Method

In the *two pointers method*, two pointers walk through an array. Both pointers move to one direction only, which ensures that the algorithm works efficiently. As a first example of how to apply the technique, consider a problem where we are given an array of n positive integers and a target sum x, and we want to find a subarray whose sum is x or report that there is no such subarray.

The problem can be solved in $O(n)$ time by using the two pointers method. The idea is to maintain pointers that point to the first and last value of a subarray. On each turn, the left pointer moves one step to the right, and the right pointer moves to the right as long as the resulting subarray sum is at most x. If the sum becomes exactly x, a solution has been found.

For example, Fig. 8.3 shows how the algorithm processes an array when the target sum is $x = 8$. The initial subarray contains the values 1, 3, and 2, whose sum is 6. Then, the left pointer moves one step right, and the right pointer does not move, because otherwise the sum would exceed x. Finally, the left pointer moves one step right, and the right pointer moves two steps right. The sum of the subarray is $2 + 5 + 1 = 8$, so the desired subarray has been found.

The running time of the algorithm depends on the number of steps the right pointer moves. While there is no useful upper bound on how many steps the pointer can move

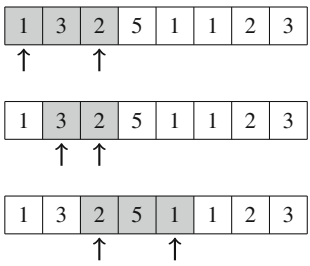


Fig. 8.3 Finding a subarray with sum 8 using the two pointers method

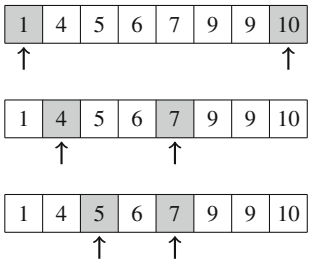


Fig. 8.4 Solving the 2SUM problem using the two pointers method

on a *single* turn, we know that the pointer moves *a total of* $O(n)$ steps during the algorithm, because it only moves to the right. Since both the left and right pointer move $O(n)$ steps, the algorithm works in $O(n)$ time.

2SUM Problem Another problem that can be solved using the two pointers method is the *2SUM problem*: given an array of n numbers and a target sum x, find two array values such that their sum is x, or report that no such values exist.

To solve the problem, we first sort the array values in increasing order. After that, we iterate through the array using two pointers. The left pointer starts at the first value and moves one step to the right on each turn. The right pointer starts at the last value and always moves to the left until the sum of the left and right value is at most x. If the sum is exactly x, a solution has been found.

For example, Fig. 8.4 shows how the algorithm processes an array when the target sum is $x = 12$. In the initial position, the sum of the values is $1 + 10 = 11$ which is smaller than x. Then the left pointer moves one step right, and the right pointer moves three steps left, and the sum becomes $4 + 7 = 11$. After this, the left pointer moves one step right again. The right pointer does not move, and a solution $5 + 7 = 12$ has been found.

The running time of the algorithm is $O(n \log n)$, because it first sorts the array in $O(n \log n)$ time, and then both pointers move $O(n)$ steps.

Note that it is also possible to solve the problem in another way in $O(n \log n)$ time using binary search. In such a solution, we first sort the array and then iterate through the array values and for each value binary search for another value that yields the sum x. In fact, many problems that can be solved using the two pointers method can also be solved using sorting or set structures, sometimes with an additional logarithmic factor.

The more general kSUM problem is also interesting. In this problem we have to find k elements such that their sum is x. It turns out that we can solve the 3SUM problem in $O(n^2)$ time by extending the above 2SUM algorithm. Can you see how we can do it? For a long time, it was actually thought that $O(n^2)$ would be the best possible time complexity for the 3SUM problem. However, in 2014, Grønlund and Pettie [12] showed that this is not the case.

8.2.2 Nearest Smaller Elements

Amortized analysis is often used to estimate the number of operations performed on a data structure. The operations may be distributed unevenly so that most operations occur during a certain phase of the algorithm, but the total number of the operations is limited.

As an example, suppose that we want to find for each array element the *nearest smaller element*, i.e., the first smaller element that precedes the element in the array. It is possible that no such element exists, in which case the algorithm should report this. Next we will efficiently solve the problem using a stack structure.

We go through the array from left to right and maintain a stack of array elements. At each array position, we remove elements from the stack until the top element is smaller than the current element, or the stack is empty. Then, we report that the top element is the nearest smaller element of the current element, or if the stack is empty, there is no such element. Finally, we add the current element to the stack.

Figure 8.5 shows how the algorithm processes an array. First, the element 1 is added to the stack. Since it is the first element in the array, it clearly does not have a nearest smaller element. After this, the elements 3 and 4 are added to the stack. The nearest smaller element of 4 is 3, and the nearest smaller element of 3 is 1. Then, the next element 2 is smaller than the two top elements in the stack, so the elements 3 and 4 are removed from the stack. Thus, the nearest smaller element of 2 is 1. After this, the element 2 is added to the stack. The algorithm continues like this, until the entire array has been processed.

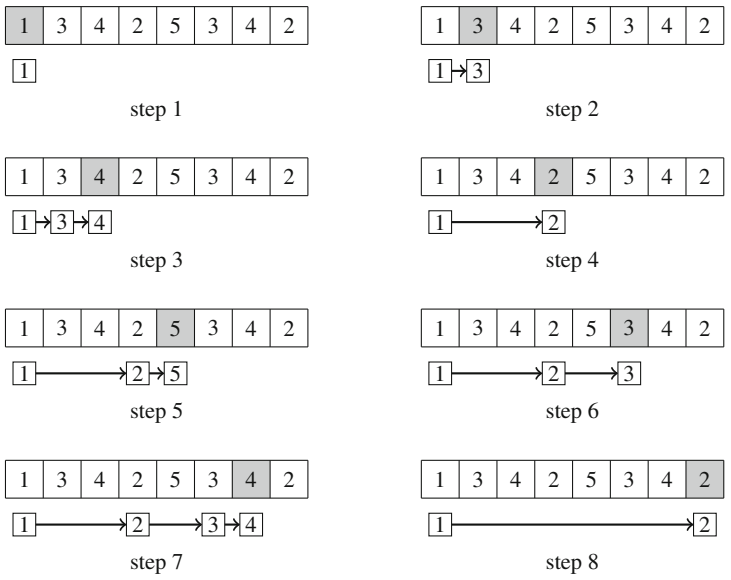


Fig. 8.5 Finding the nearest smaller elements in linear time using a stack

The efficiency of the algorithm depends on the total number of stack operations. If the current element is larger than the top element in the stack, it is directly added to the stack, which is efficient. However, sometimes the stack can contain several larger elements and it takes time to remove them. Still, each element is added *exactly once* to the stack and removed *at most once* from the stack. Thus, each element causes $O(1)$ stack operations, and the algorithm works in $O(n)$ time.

8.2.3 Sliding Window Minimum

A *sliding window* is a constant-size subarray that moves from left to right through an array. At each window position, we want to calculate some information about the elements inside the window. Next we will focus on the problem of maintaining the *sliding window minimum*, which means that we want to report the smallest value inside each window.

The sliding window minima can be calculated using a similar idea that we used to calculate the nearest smaller elements. This time we maintain a queue where each element is larger than the previous element, and the first element always corresponds to the minimum element inside the window. After each window move, we remove elements from the end of the queue until the last queue element is smaller than the new window element, or the queue becomes empty. We also remove the first queue element if it is not inside the window anymore. Finally, we add the new window element to the queue.

Figure 8.6 shows how the algorithm processes an array when the sliding window size is 4. At the first window position, the smallest value is 1. Then the window moves one step right. The new element 3 is smaller than the elements 4 and 5 in the queue, so the elements 4 and 5 are removed from the queue and the element 3

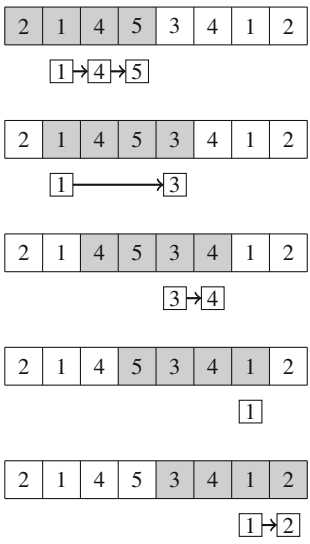


Fig. 8.6 Finding sliding window minima in linear time

is added to the queue. The smallest value is still 1. After this, the window moves again, and the smallest element 1 does not belong to the window anymore. Thus, it is removed from the queue, and the smallest value is now 3. Also the new element 4 is added to the queue. The next new element 1 is smaller than all elements in the queue, so all elements are removed from the queue, and it only contains the element 1. Finally, the window reaches its last position. The element 2 is added to the queue, but the smallest value inside the window is still 1.

Since each array element is added to the queue exactly once and removed from the queue at most once, the algorithm works in $O(n)$ time.

8.3 Finding Minimum Values

Suppose that there is a function $f(x)$ that first only decreases, then attains its minimum value, and then only increases. For example, Fig. 8.7 shows such a function whose minimum value is marked with an arrow. If we know that our function has this property, we can efficiently find its minimum value.

8.3.1 Ternary Search

Ternary search provides an efficient way to find the minimum value of a function that first decreases and then increases. Assume that we know that the value of x that minimizes $f(x)$ is in a range $[x_L, x_R]$. The idea is to divide the range into three equal-size parts $[x_L, a]$, $[a, b]$, and $[b, x_R]$ by choosing

$$a = \frac{2x_L + x_R}{3} \quad \text{and} \quad b = \frac{x_L + 2x_R}{3}.$$

Then, if $f(a) < f(b)$, we conclude that the minimum must be in range $[x_L, b]$, and otherwise it must be in range $[a, x_R]$. After this, we recursively continue the search, until the size of the active range is small enough.

As an example, Fig. 8.8 shows the first step of ternary search in our example scenario. Since $f(a) > f(b)$, the new range becomes $[a, x_R]$.

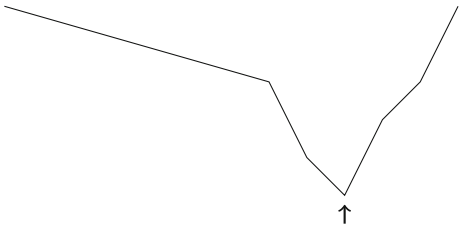


Fig. 8.7 A function and its minimum value

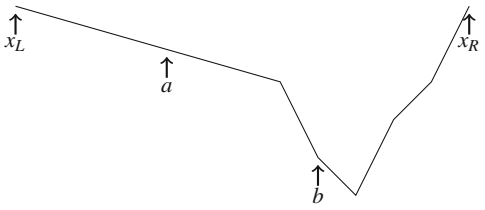
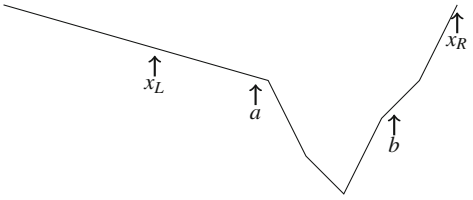


Fig. 8.8 Searching for the minimum using ternary search



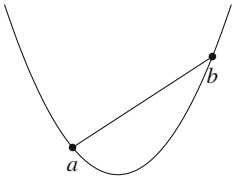


Fig. 8.9 Example of a convex function: $f(x) = x^2$

In practice, we often consider functions whose parameters are integers, and the search is terminated when the range only contains one element. Since the size of the new range is always $2/3$ of the previous range, the algorithm works in $O(\log n)$ time, where n is the number of elements in the original range.

Note that when working with integer parameters, we can also use *binary search* instead of ternary search, because it suffices to find the first position x for which $f(x) \leq f(x+1)$.

8.3.2 Convex Functions

A function is *convex* if a line segment between any two points on the graph of the function always lies above or on the graph. For example, Fig. 8.9 shows the graph of $f(x) = x^2$, which is a convex function. Indeed, the line segment between points a and b lies above the graph.

If we know that the minimum value of a convex function is in range $[x_L, x_R]$, we can use ternary search to find it. However, note that several points of a convex function may have the minimum value. For example, $f(x) = 0$ is convex and its minimum value is 0.

Convex functions have some useful properties: if $f(x)$ and $g(x)$ are convex functions, then also $f(x)+g(x)$ and $\max(f(x), g(x))$ are convex functions. For example,

if we have n convex functions $f_1, f_2, \ldots, f_n$, we immediately know that also the function $f_1 + f_2 + \ldots + f_n$ has to be convex and we can use ternary search to find its minimum value.

8.3.3 Minimizing Sums

Given n numbers $a_1, a_2, \ldots, a_n$, consider the problem of finding a value of x that minimizes the sum

$$|a_1 - x| + |a_2 - x| + \cdots + |a_n - x|.$$

For example, if the numbers are $[1, 2, 9, 2, 6]$, the optimal solution is to choose $x = 2$, which produces the sum

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

Since each function $|a_k - x|$ is convex, the sum is also convex, so we could use ternary search to find the optimal value of x. However, there is also an easier solution. It turns out that the optimal choice for x is always the *median* of the numbers, i.e., the middle element after sorting. For example, the list $[1, 2, 9, 2, 6]$ becomes $[1, 2, 2, 6, 9]$ after sorting, so the median is 2.

The median is always optimal, because if x is smaller than the median, the sum becomes smaller by increasing x, and if x is larger then the median, the sum becomes smaller by decreasing x. If n is even and there are two medians, both medians and all values between them are optimal choices.

Then, consider the problem of minimizing the function

$$(a_1 - x)^2 + (a_2 - x)^2 + \cdots + (a_n - x)^2.$$

For example, if the numbers are $[1, 2, 9, 2, 6]$, the best solution is to choose $x = 4$, which produces the sum

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

Again, this function is convex and we could use ternary search to solve the problem, but there is also a simple solution: the optimal choice for x is the *average* of the numbers. In the example the average is $(1 + 2 + 9 + 2 + 6)/5 = 4$. This can be proved by presenting the sum as follows:

$$nx^2 - 2x(a_1 + a_2 + \cdots + a_n) + (a_1^2 + a_2^2 + \cdots + a_n^2)$$

The last part does not depend on x, so we can ignore it. The remaining parts form a function $nx^2 - 2xs$ where $s = a_1 + a_2 + \cdots + a_n$. This is a parabola opening upwards with roots $x = 0$ and $x = 2s/n$, and the minimum value is the average of the roots $x = s/n$, i.e., the average of the numbers $a_1, a_2, \ldots, a_n$.