Additional Topics

15

This final chapter presents a selection of advanced algorithms and data structures. Mastering the techniques of this chapter may sometimes help you to solve the most difficult problem in a programming contest.

Section 15.1 discusses square root techniques for creating data structures and algorithms. Such solutions are often based on the idea of dividing a sequence of n elements into $O(\sqrt{n})$ blocks, each of which consists of $O(\sqrt{n})$ elements.

Section 15.2 further explores the possibilities of segment trees. For example, we will see how to create a segment tree that supports both range queries and range updates at the same time.

Section 15.3 presents the treap data structure which allows us to efficiently split an array into two parts and combine two arrays into a single array.

Section 15.4 focuses on optimizing dynamic programming solutions. First we will learn the convex hull trick which is used with linear functions, and after this we will discuss the divide and conquer optimization and Knuth's optimization.

Section 15.5 deals with miscellaneous algorithm design techniques, such as meet in the middle and parallel binary search.

15.1 Square Root Techniques

A square root can be seen as a "poor man's logarithm": the complexity $O(\sqrt{n})$ is better than $O(n)$ but worse than $O(\log n)$. In any case, many data structures and algorithms involving square roots are fast and usable in practice. This section shows some examples of how square roots can be used in algorithm design.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_15

15.1.1 Data Structures

Sometimes we can create an efficient data structure by dividing an array into *blocks* of size $\sqrt{n}$ and maintaining information about array values inside each block. For example, suppose that we should process two types of queries: modifying array values and finding minimum values in ranges. We have previously seen that a segment tree can support both operations in $O(\log n)$ time, but next we will solve the problem in another simpler way where the operations take $O(\sqrt{n})$ time.

We divide the array into blocks of $\sqrt{n}$ elements, and maintain for each block the minimum value inside it. For example, Fig. 15.1 shows an array of 16 elements that is divided into blocks of 4 elements. When an array value changes, the corresponding block needs to be updated. This can be done in $O(\sqrt{n})$ time by going through the values inside the block, as shown in Fig. 15.2. Then, to calculate the minimum value in a range, we divide the range into three parts such that the range consists of single values and blocks between them. Figure 15.3 shows an example of such a division. The answer to the query is either a single value or the minimum value inside a block. Since the number of single elements is $O(\sqrt{n})$ and the number of blocks is also $O(\sqrt{n})$, the query takes $O(\sqrt{n})$ time.

How efficient is the resulting structure in practice? To find this out, we conducted an experiment where we created an array of n random `int` values and then processed n random minimum queries. We implemented three data structures: a segment tree with $O(\log n)$ time queries, the square root structure described above with $O(\sqrt{n})$ time queries, and a plain array with $O(n)$ time queries. Table 15.1 shows the results of the experiment. It turns out that in this problem, the square root structure is quite efficient up to $n = 2^{18}$; however, after this, it requires clearly more time than a segment tree.

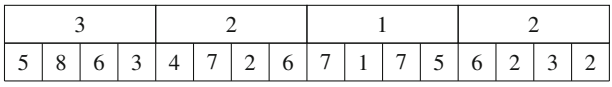


Fig. 15.1 A square root structure for finding minimum values in ranges

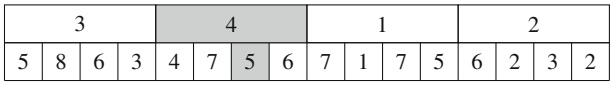


Fig. 15.2 When an array value is updated, the value in the corresponding block has to be also updated

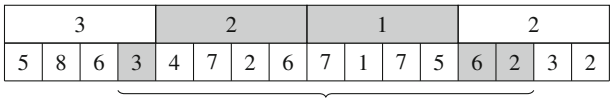


Fig. 15.3 To determine the minimum value in a range, the range is divided into single values and blocks

Table 15.1 The running times of three data structures for range minimum queries: a segment tree ($O(\log n)$), a square root structure ($O(\sqrt{n})$), and a plain array ($O(n)$)

Input size n	$O(\log n)$ Queries (s)	$O(\sqrt{n})$ Queries (s)	$O(n)$ Queries (s)
2^{16}	0.02	0.05	1.50
2^{17}	0.03	0.16	6.02
2^{18}	0.07	0.28	24.82
2^{19}	0.14	1.14	> 60
2^{20}	0.31	2.11	> 60
2^{21}	0.66	9.27	> 60

Fig. 15.4 An instance of the letter distance problem

A	C	E	A
B	D	F	D
E	A	B	C
C	F	E	A

15.1.2 Subalgorithms

Next we discuss two problems that can be efficiently solved by creating two *subalgorithms* that are specialized for different kinds of situations during the algorithm. While either of the subalgorithms could be used to solve the problem without the other, we get an efficient algorithm by combining them.

Letter Distances Our first problem is as follows: We are given an $n \times n$ grid whose each square is assigned a letter. What is the minimum Manhattan distance between two squares that have the same letter? For example, in Fig. 15.4 the minimum distance is 2 between the two squares with letter "D."

To solve the problem, we can go through all letters that appear in the grid, and for each letter c, determine the minimum distance between two squares with letter c. Consider two algorithms for processing a fixed letter c:

Algorithm 1: Go through all pairs of squares that contain the letter c and determine the minimum distance pair among them. This algorithm works in $O(k^2)$ time, where k is the number of squares with letter c.

Algorithm 2: Perform a breadth-first search that simultaneously begins at each square with letter c. The search takes $O(n^2)$ time.

Both algorithms have certain worst-case situations. The worst case for Algorithm 1 is a grid where each square has the same color, in which case $k = n^2$ and the algorithm takes $O(n^4)$ time. Then, the worst case for Algorithm 2 is a grid where each square has a distinct color. In this case, the algorithm is performed $O(n^2)$ times, which takes $O(n^4)$ time.

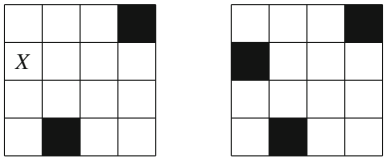


Fig. 15.5 A turn in the black squares game. The minimum distance from X to a black square is 3

However, we can *combine* the algorithms so that they function as subalgorithms of a single algorithm. The idea is to decide for each color c separately which algorithm to use. Clearly, Algorithm 1 works well if k is small, and Algorithm 2 is best suited for cases where k is large. Thus, we can fix a constant x and use Algorithm 1 if k is at most x, and otherwise use Algorithm 2.

In particular, by choosing $x = \sqrt{n^2} = n$, we get an algorithm that works in $O(n^3)$ time. First, each square that is processed using Algorithm 1 is compared with at most n other squares, so processing those squares takes $O(n^3)$ time. Then, since there are at most n colors that appear in more than n squares, Algorithm 2 is performed at most n times, and its total running time is also $O(n^3)$.

Black Squares As another example, consider the following game: We are given an $n \times n$ grid where exactly one square is black and all other squares are white. On each turn, one white square is chosen, and we should calculate the minimum Manhattan distance between this square and a black square. After this, the white square is painted black. This process continues for $n^2 - 1$ turns, after which all squares have been painted black.

For example, Fig. 15.5 shows a turn in the game. The minimum distance from the chosen square X to a black square is 3 (by going two steps down and one step right). After this, the square is painted black.

We can solve the problem by processing the turns in *batches* of k turns. Before each batch, we calculate for each square of the grid the minimum distance to a black square. This can be done in $O(n^2)$ time using breadth-first search. Then, when processing a batch, we keep a list of all squares that have been painted black during the current batch. Thus, the minimum distance to a black square is either the precalculated distance or a distance to one of the squares on the list. Since the list contains at most k values, it takes $O(k)$ time to go through the list.

Then, by choosing $k = \sqrt{n^2} = n$, we get an algorithm that works in $O(n^3)$ time. First, there are $O(n)$ batches, so the total time used for breadth-first searches is $O(n^3)$. Then, the list of squares in a batch contains $O(n)$ values, so calculating minimum distances for $O(n^2)$ squares also takes $O(n^3)$ time.

Tuning Parameters In practice, it is not necessary to use the exact square root value as the parameter, but rather we can fine-tune the performance of an algorithm by experimenting with different parameters and choosing the parameter that works best. Of course, the optimal parameter depends on the algorithm and also on the properties of the test data.

Table 15.2 Optimizing the value of the parameter k in the black squares algorithm

Parameter k	Running time (s)
200	5.74
500	2.41
1000	1.32
2000	1.02
5000	1.28
10000	2.13
20000	3.97

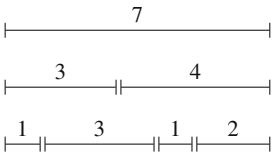


Fig. 15.6 Some integer partitions of a stick of length 7

Table 15.2 shows the results of an experiment where the $O(n^3)$ time algorithm for the black squares game was performed for different values of k when $n = 500$. The order in which the squares were painted black was randomly selected. In this case, the optimal parameter seems to be about $k = 2000$.

15.1.3 Integer Partitions

Suppose that there is a stick whose length is n, and it is divided into some parts whose lengths are integers. For example, Fig. 15.6 shows some possible partitions for $n = 7$. What is the maximum number of *distinct* lengths in such a partition?

It turns out that there are at most $O(\sqrt{n})$ distinct lengths. Namely an optimal way to produce as many distinct lengths as possible is to include lengths $1, 2, \ldots, k$. Then, since

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2},$$

we can conclude that k can be at most $O(\sqrt{n})$. Next, we will see how this observation can be used when designing algorithms.

Knapsack Problem Consider a knapsack problem where we are given a list of integer weights $[w_1, w_2, \ldots, w_k]$ such that $w_1 + w_2 + \cdots + w_k = n$, and our task is to determine all possible weight sums that can be created. For example, Fig. 15.7 shows the possible sums using the weights [3, 3, 4].

Fig. 15.7 The possible sums using the weights [3, 3, 4]

Using a standard knapsack algorithm (Sect. 6.2.3), we can solve the problem in $O(nk)$ time, so if $k = O(n)$, the time complexity becomes $O(n^2)$. However, since there are at most $O(\sqrt{n})$ distinct weights, we can actually solve the problem more efficiently by simultaneously processing all weights of a certain value. For example, if the weights are [3, 3, 4], we first process the two weights of value 3 and then the weight of value 4. It is not difficult to modify the standard knapsack algorithm so that processing each group of equal weights only takes $O(n)$ time, which yields an $O(n\sqrt{n})$ time algorithm.

String Construction As another example, suppose that we are given a string of length n and a dictionary of words whose total length is m. Our task is to count the number of ways we can construct the string using the words. For example, there are four ways to construct the string `ABAB` using the words {`A`, `B`, `AB`}:

- `A` + `B` + `A` + `B`
- `AB` + `A` + `B`
- `A` + `B` + `AB`
- `AB` + `AB`

Using dynamic programming, we can calculate for each $k = 0, 1, \ldots, n$ the number of ways to construct a prefix of length k of the string. One way to do this is to use a trie that contains reverses of all the words in the dictionary, which yields an $O(n^2 + m)$ time algorithm. However, another approach is to use string hashing and the fact that there are at most $O(\sqrt{m})$ distinct word lengths. Thus, we can restrict ourselves to word lengths that actually exist. This can be done by creating a set that contains all hash values of words, which results in an algorithm whose running time is $O(n\sqrt{m} + m)$ (using `unordered_set`).

15.1.4 Mo's Algorithm

Mo's algorithm[1] processes a set of range queries on a *static* array (i.e., the array values do not change between the queries). Each query requires us to calculate something based on the array values in a range $[a, b]$. Since the array is static, the queries can be processed in any order, and the trick in Mo's algorithm is to use a special order which guarantees that the algorithm works efficiently.

The algorithm maintains an *active range* in the array, and the answer to a query concerning the active range is known at each moment. The algorithm processes the

[1] According to [5], Mo's algorithm is named after Mo Tao, a Chinese competitive programmer.

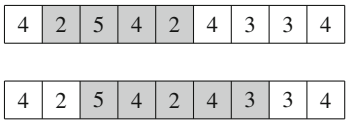


Fig. 15.8 Moving between two ranges in Mo's algorithm

queries one by one and always moves the endpoints of the active range by inserting and removing elements. The array is divided into blocks of $k = O(\sqrt{n})$ elements, and a query $[a_1, b_1]$ is always processed before a query $[a_2, b_2]$ if

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ or
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ and $b_1 < b_2$.

Thus, all queries whose left endpoints are in a certain block are processed one after another sorted according to their right endpoints. Using this order, the algorithm only performs $O(n\sqrt{n})$ operations, because the left endpoint moves $O(n)$ times $O(\sqrt{n})$ steps, and the right endpoint moves $O(\sqrt{n})$ times $O(n)$ steps. Thus, both endpoints move a total of $O(n\sqrt{n})$ steps during the algorithm.

Example Consider a problem where we are given a set of array ranges, and we are asked to calculate the number of *distinct* values in each range. In Mo's algorithm, the queries are always sorted in the same way, but the way the answer to the query is maintained depends on the problem.

To solve the problem, we maintain an array `count` where `count`$[x]$ indicates the number of times an element x occurs in the active range. When we move from one query to another query, the active range changes. For example, consider the two ranges in Fig. 15.8. When we move from the first range to the second range, there will be three steps: the left endpoint moves one step to the right, and the right endpoint moves two steps to the right.

After each step, the array `count` needs to be updated. After adding an element x, we increase the value of `count`$[x]$ by 1, and if `count`$[x] = 1$ after this, we also increase the answer to the query by 1. Similarly, after removing an element x, we decrease the value of `count`$[x]$ by 1, and if `count`$[x] = 0$ after this, we also decrease the answer to the query by 1. Since each step requires $O(1)$ time, the algorithm works in $O(n\sqrt{n})$ time.

15.2 Segment Trees Revisited

A segment tree is a versatile data structure that can be used to solve a large number of problems. However, so far we have only seen a small part of the possibilities of segment trees. Now is time to discuss some more advanced variants of segment trees that allow us to solve more advanced problems.

Until now, we have implemented the operations of a segment tree by walking *from bottom to top* in the tree. For example, we have used the following function (Sect. 9.2.2) to calculate the sum of values in a range $[a, b]$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

However, in advanced segment trees, it is often necessary to implement the operations *from top to bottom* as follows:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

Using this function, we can calculate the sum in a range $[a, b]$ as follows:

```
int s = sum(a,b,1,0,n-1);
```

The parameter k indicates the current position in `tree`. Initially k equals 1, because we begin at the root of the tree. The range $[x, y]$ corresponds to k and is initially $[0, n-1]$. When calculating the sum, if $[x, y]$ is outside $[a, b]$, the sum is 0, and if $[x, y]$ is completely inside $[a, b]$, the sum can be found in `tree`. If $[x, y]$ is partially inside $[a, b]$, the search continues recursively to the left and right half of $[x, y]$. The left half is $[x, d]$, and the right half is $[d+1, y]$, where $d = \lfloor \frac{x+y}{2} \rfloor$.

Figure 15.9 shows how the search proceeds when calculating the value of $\texttt{sum}_q(a, b)$. The gray nodes indicate nodes where the recursion stops and the sum can be found in `tree`. Also in this implementation, operations take $O(\log n)$ time, because the total number of visited nodes is $O(\log n)$.

15.2.1 Lazy Propagation

Using *lazy propagation*, we can build a segment tree that supports *both* range updates and range queries in $O(\log n)$ time. The idea is to perform updates and queries from top to bottom and perform updates *lazily* so that they are propagated down the tree only when it is necessary.

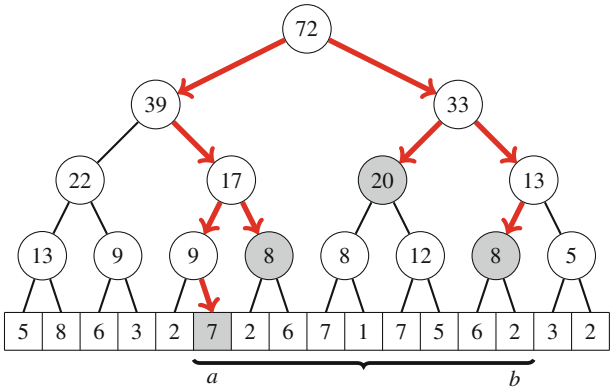


Fig. 15.9 Traversing a segment tree from top to bottom

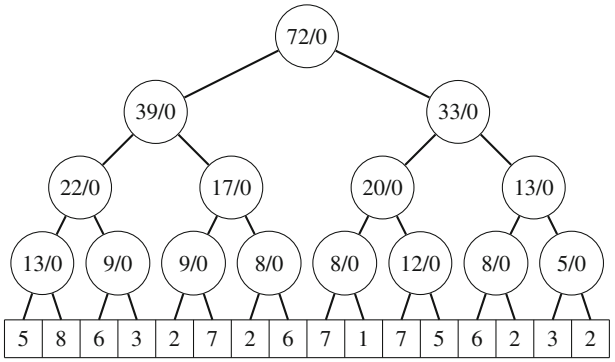


Fig. 15.10 A lazy segment tree for range updates and queries

The nodes of a lazy segment tree contain two types of information. Like in an ordinary segment tree, each node contains the sum, minimum value, or some other value related to the corresponding subarray. In addition, a node may contain information about a lazy update which has not been propagated to its children. Lazy segment trees can support two types of range updates: each array value in the range is either *increased* by some value or *assigned* some value. Both operations can be implemented using similar ideas, and it is even possible to construct a tree that supports both operations at the same time.

Let us consider an example where our goal is to construct a segment tree that supports two operations: increasing each value in $[a, b]$ by a constant and calculating the sum of values in $[a, b]$. To achieve this goal, we construct a tree where each node has two values s/z: s denotes the sum of values in the range, and z denotes the value of a lazy update, which means that all values in the range should be increased by z. Figure 15.10 shows an example of such a tree, where $z = 0$ in all nodes, meaning that there are no ongoing lazy updates.

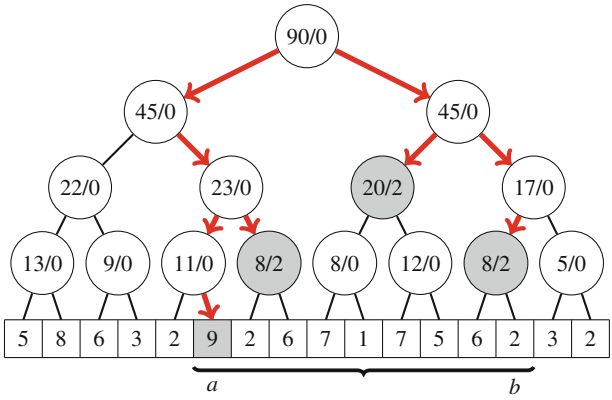


Fig. 15.11 Increasing the values in the range $[a, b]$ by 2

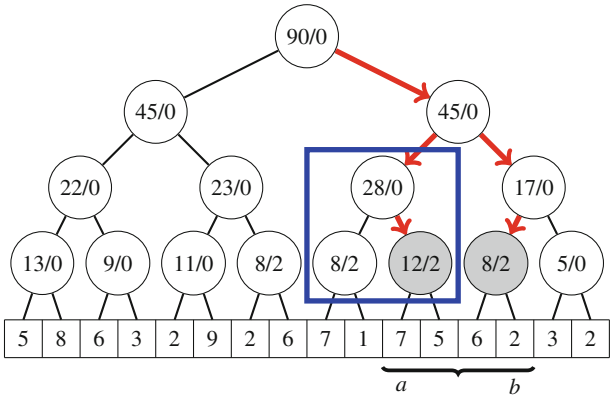


Fig. 15.12 Calculating the sum of values in the range $[a, b]$

We implement the tree operations from top to bottom. To increase the values in a range $[a, b]$ by u, we modify the nodes as follows: If the range $[x, y]$ of a node is completely inside $[a, b]$, we increase the z value of the node by u and stop. Then, if $[x, y]$ partially belongs to $[a, b]$, we continue our walk recursively in the tree, and after this calculate the new s value for the node. As an example, Fig. 15.11 shows our tree after increasing the range $[a, b]$ by 2.

In both updates and queries, lazy updates are propagated downwards when we move in the tree. Always before accessing a node, we check if it has an ongoing lazy update. If it has, we update its s value, propagate the update to its children, and then clear its z value. For example, Fig. 15.12 shows how our tree changes when we calculate the value of $\texttt{sum}_a(a, b)$. The rectangle contains the nodes whose values change when a lazy update is propagated downwards.

Polynomial Updates We can generalize the above segment tree so that it is possible to update ranges using polynomials of the form

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \cdots + t_0.$$

In this case, the update for a value at position i in $[a, b]$ is $p(i - a)$. For example, adding the polynomial $p(u) = u + 1$ to $[a, b]$ means that the value at position a increases by 1, the value at position $a + 1$ increases by 2, and so on.

To support polynomial updates, each node is assigned $k + 2$ values, where k equals the degree of the polynomial. The value s is the sum of the elements in the range, and the values $z_0, z_1, \ldots, z_k$ are the coefficients of a polynomial that corresponds to a lazy update. Now, the sum of values in a range $[x, y]$ equals

$$s + \sum_{u=0}^{y-x} (z_k u^k + z_{k-1} u^{k-1} + \cdots + z_1 u + z_0),$$

and the value of such a sum can be efficiently calculated using sum formulas. For example, the term z_0 corresponds to the sum $z_0(y - x + 1)$, and the term $z_1 u$ corresponds to the sum

$$z_1(0 + 1 + \cdots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}.$$

When propagating an update in the tree, the indices of $p(u)$ change, because in each range $[x, y]$, the values are calculated for $u = 0, 1, \ldots, y - x$. However, we can easily handle this, because $p'(u) = p(u + h)$ is a polynomial of equal degree as $p(u)$. For example, if $p(u) = t_2 u^2 + t_1 u + t_0$, then

$$p'(u) = t_2(u + h)^2 + t_1(u + h) + t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h + t_0.$$

15.2.2 Dynamic Trees

An ordinary segment tree is static, which means that each node has a fixed position in the segment tree array and the structure requires a fixed amount of memory. In a *dynamic segment tree*, memory is allocated only for nodes that are actually accessed during the algorithm, which can save a large amount of memory.

The nodes of a dynamic tree can be represented as structs:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

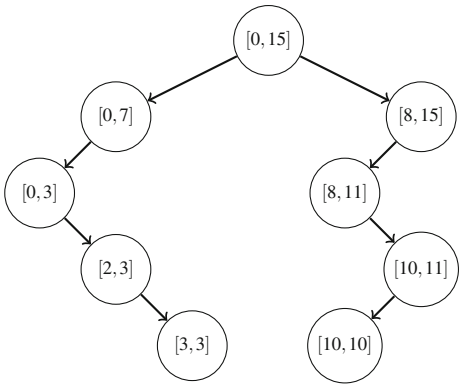


Fig. 15.13 A sparse segment tree where the elements at positions 3 and 10 have been modified

Here `value` is the value of the node, [`x`, `y`] is the corresponding range, and `left` and `right` point to the left and right subtree. Nodes can be created as follows:

```
// create a node with value 2 and range [0,7]
node *x = new node(2,0,7);
// change value
x->value = 5;
```

Sparse Segment Trees A dynamic segment tree is a useful structure when the underlying array is *sparse*, i.e., the range $[0, n-1]$ of allowed indices is large, but most array values are zeros. While an ordinary segment tree would use $O(n)$ memory, a dynamic segment tree only uses $O(k \log n)$ memory, where k is the number of operations performed.

A *sparse segment tree* initially has only one node $[0, n-1]$ whose value is zero, which means that every array value is zero. After updates, new nodes are dynamically added to the tree. Any path from the root node to a leaf contains $O(\log n)$ nodes, so each segment tree operation adds at most $O(\log n)$ new nodes to the tree. Thus, after k operations, the tree contains $O(k \log n)$ nodes. For example, Fig. 15.13 shows a sparse segment tree where $n = 16$, and the elements at positions 3 and 10 have been modified.

Note that if we know all elements that will be updated during the algorithm beforehand, a dynamic segment tree is not necessary, because we can use an ordinary segment tree with index compression (Sect. 9.2.3). However, this is not possible when the indices are generated during the algorithm.

Persistent Segment Trees Using a dynamic implementation, we can also create a *persistent segment tree* that stores the *modification history* of the tree. In such an implementation, we can efficiently access all versions of the tree that have existed during the algorithm. When the modification history is available, we can perform queries in any previous tree like in an ordinary segment tree, because the full structure

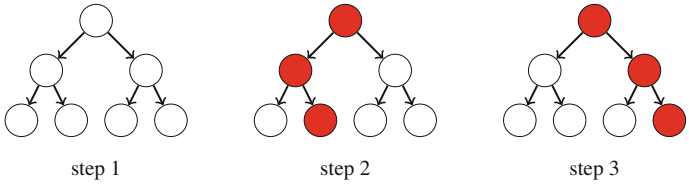


Fig. 15.14 A modification history of a segment tree: the initial tree and two updates

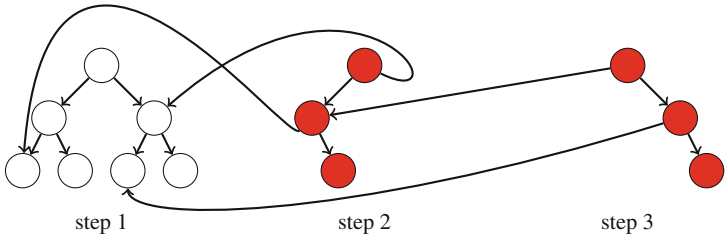


Fig. 15.15 A compact way to store the modification history

of each tree is stored. We can also create new trees based on previous trees and modify them independently.

Consider the sequence of updates in Fig. 15.14, where marked nodes change and other nodes remain the same. After each update, most nodes of the tree remain the same, so a compact way to store the modification history is to represent each historical tree as a combination of new nodes and subtrees of previous trees. Figure 15.15 shows how the modification history can be stored. The structure of each previous tree can be reconstructed by following the pointers starting at the corresponding root node. Since each operation adds only $O(\log n)$ new nodes to the tree, it is possible to store the full modification history of the tree.

15.2.3 Data Structures in Nodes

Instead of single values, the nodes of a segment tree can also contain *data structures* that maintain information about the corresponding ranges. As an example, suppose that we should be able to efficiently count the number of occurrences of an element x in a range $[a, b]$. To do this, we can create a segment tree where each node is assigned a data structure that can be asked how many times any element x appears in the corresponding range. After this, the answer to a query can be calculated by combining the results from nodes that belong to the range.

The remaining task is to choose a suitable data structure for the problem. A good choice is a `map` structure whose keys are array elements and values indicate how many times each element occurs in a range. Figure 15.16 shows an array and the corresponding segment tree. For example, the root node of the tree tells us that element 1 appears 4 times in the array.

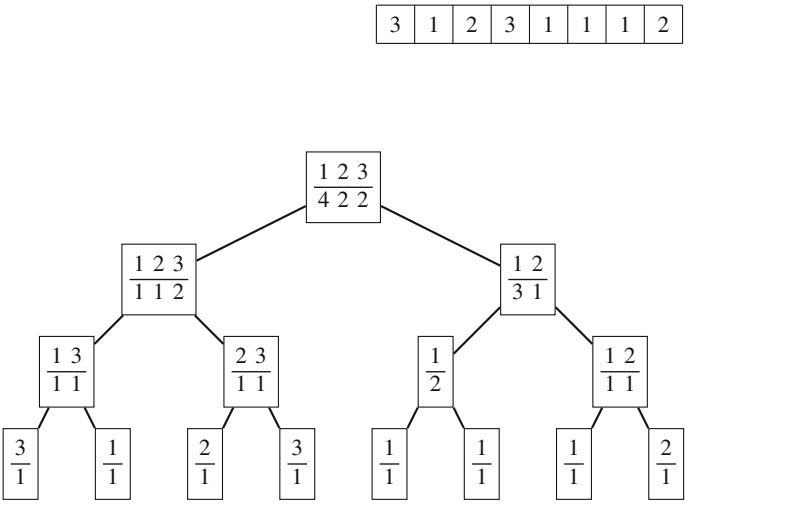


Fig. 15.16 A segment tree for calculating the number of occurrences of an element in an array range

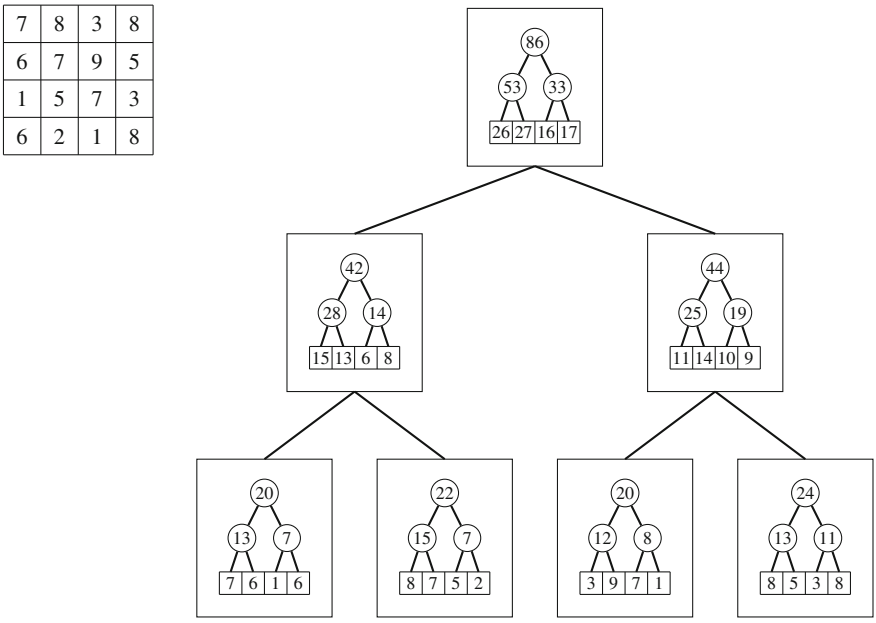


Fig. 15.17 A two-dimensional array and the corresponding segment tree for calculating sums of rectangular subarrays

Each query in the above segment tree works in $O(\log^2 n)$ time, because each node has a `map` structure whose operations take $O(\log n)$ time. The tree uses $O(n \log n)$ memory, because it has $O(\log n)$ levels, and each level contains n elements that have been distributed in the `map` structures.

15.2.4 Two-Dimensional Trees

A *two-dimensional segment tree* allows us to process queries related to rectangular subarrays on a two-dimensional array. The idea is to create a segment tree that corresponds to the columns of the array and then assign each node of this structure a segment tree that corresponds to the rows of the array.

For example, Fig. 15.17 shows a two-dimensional segment tree that supports two queries: calculating the sum of values in a subarray and updating a single array value. Both the queries take $O(\log^2 n)$ time, because $O(\log n)$ nodes in the main segment tree are accessed, and processing each node takes $O(\log n)$ time. The structure uses a total of $O(n^2)$ memory, because the main segment tree has $O(n)$ nodes, and each node has a segment tree of $O(n)$ nodes.

15.3 Treaps

A *treap* is a binary tree that can store the contents of an array in such a way that we can efficiently split an array into two arrays and merge two arrays into an array. Each node in a treap has two values: a *weight* and a *value*. Each node's weight is smaller or equal than the weights of its children, and the node is located in the array *after* all nodes in its left subtree and *before* all nodes in its right subtree.

Figure 15.18 shows an example of an array and the corresponding treap. For example, the root node has weight 1 and value D. Since its left subtree contains three nodes, this means that the array element at position 3 has value D.

15.3.1 Splitting and Merging

When a new node is added to the treap, it is assigned a *random* weight. This guarantees that the tree is balanced (its height is $O(\log n)$) with high probability, and its operations can be performed efficiently.

Splitting The splitting operation of a treap creates two treaps which divide the array into two arrays so that the first k elements belong to the first array and the rest of the elements belong to the second array. To do this, we create two new treaps that are initially empty and traverse the original treap starting at the root node. At each

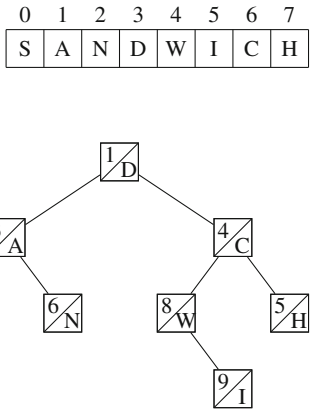


Fig. 15.18 An array and the corresponding treap

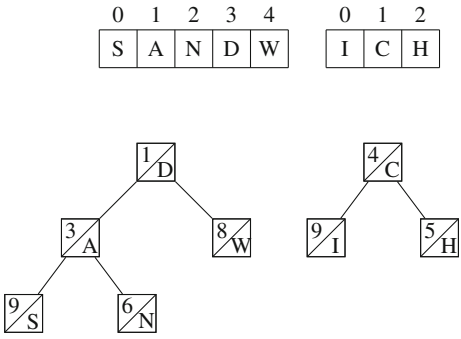


Fig. 15.19 Splitting an array into two arrays

step, if the current node belongs to the left treap, the node and its left subtree are added to the left treap and we recursively process its right subtree. Similarly, if the current node belongs to the right treap, the node and its right subtree are added to the right treap and we recursively process its left subtree. Since the height of the treap is $O(\log n)$, this operation works in $O(\log n)$ time.

For example, Fig. 15.19 shows how to divide our example array into two arrays so that the first array contains the first five elements of the original array and the second array contains the last three elements. First, node D belongs to the left treap, so we add node D and its left subtree to the left treap. Then, node C belongs to the right treap, and we add node C and its right subtree to the right treap. Finally, we add node W to the left treap and node I to the right treap.

Merging The merging operation of two treaps creates a single treap that concatenates the arrays. The two treaps are processed simultaneously, and at each step, the treap whose root has the smallest weight is selected. If the root of the left treap has the smallest weight, the root and its left subtree are moved to the new treap and its right subtree becomes the new root of the left treap. Similarly, if the root of the right treap

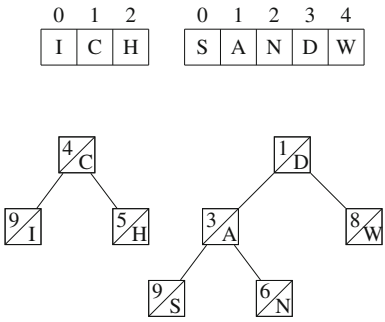


Fig. 15.20 Merging two arrays into an array, before merging

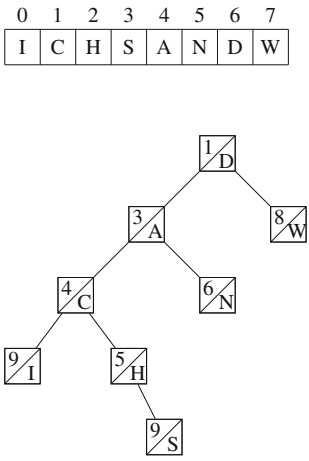


Fig. 15.21 Merging two arrays into an array, after merging

has the smallest weight, the root and its right subtree are moved to the new treap and its left subtree becomes the new root of the right treap. Since the height of the treap is $O(\log n)$, this operation works in $O(\log n)$ time.

For example, we may now swap the order of the two arrays in our example scenario and then concatenate the arrays again. Figure 15.20 shows the arrays before merging, and Fig. 15.21 shows the final result. First, node D and its right subtree is added to the new treap. Then, node A and its right subtree become the left subtree of node D. After this, node C and its left subtree become the left subtree of node A. Finally, node H and node S are added to the new treap.

15.3.2 Implementation

Next we will learn a convenient way to implement a treap. First, here is a struct that stores a treap node:

```
struct node {
    node *left, *right;
    int weight, size, value;
    node(int v) {
        left = right = NULL;
        weight = rand();
        size = 1;
        value = v;
    }
};
```

The field `size` contains the size of the subtree of the node. Since a node can be `NULL`, the following function is useful:

```
int size(node *treap) {
    if (treap == NULL) return 0;
    return treap->size;
}
```

The following function `split` implements the splitting operation. The function recursively splits the treap `treap` into treaps `left` and `right` so that the left treap contains the first k nodes and the right treap contains the remaining nodes.

```
void split(node *treap, node *&left, node *&right, int k) {
    if (treap == NULL) {
        left = right = NULL;
    } else {
        if (size(treap->left) < k) {
            split(treap->right, treap->right, right,
                  k-size(treap->left)-1);
            left = treap;
        } else {
            split(treap->left, left, treap->left, k);
            right = treap;
        }
        treap->size = size(treap->left)+size(treap->right)+1;
    }
}
```

Then, the following function `merge` implements the merging operation. This function creates a treap `treap` that contains first the nodes of the treap `left` and then the nodes of the treap `right`.

```
void merge(node *&treap, node *left, node *right) {
    if (left == NULL) treap = right;
    else if(right == NULL) treap = left;
    else {
        if (left->weight < right->weight) {
            merge(left->right, left->right, right);
            treap = left;
        } else {
            merge(right->left, left, right->left);
            treap = right;
        }
        treap->size = size(treap->left)+size(treap->right)+1;
    }
}
```

For example, the following code creates a treap that corresponds to the array [1, 2, 3, 4]. Then it divides it into two treaps of size 2 and swaps their order to create a new treap that corresponds to the array [3, 4, 1, 2].

```
node *treap = NULL;
merge(treap, treap, new node(1));
merge(treap, treap, new node(2));
merge(treap, treap, new node(3));
merge(treap, treap, new node(4));
node *left, *right;
split(treap, left, right, 2);
merge(treap, right, left);
```

15.3.3 Additional Techniques

The splitting and merging operations of treaps are very powerful, because we can freely "cut and paste" arrays in logarithmic time using them. Treaps can be also extended so that they work almost like segment trees. For example, in addition to maintaining the size of each subtree, we can also maintain the sum of its values, the minimum value, and so on.

One special trick related to treaps is that we can efficiently *reverse* an array. This can be done by swapping the left and right child of each node in the treap. For example, Fig. 15.22 shows the result after reversing the array in Fig. 15.18. To do this efficiently, we can introduce a field that indicates if we should reverse the subtree of the node, and process swapping operations lazily.

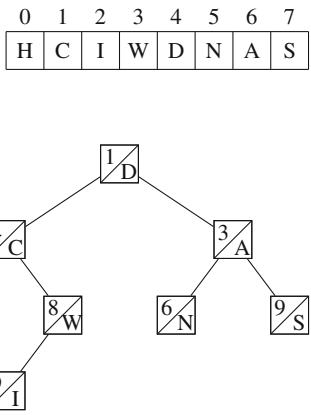


Fig. 15.22 Reversing an array using a treap

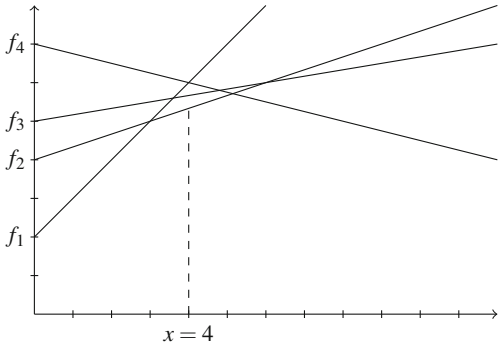


Fig. 15.23 The minimum function value at point $x = 4$ is $f_2(4) = 16/3$

15.4 Dynamic Programming Optimization

This section discusses techniques for optimizing dynamic programming solutions. First, we focus on the convex hull trick, which can be used to efficiently find minimum values of linear functions. After this, we discuss two other techniques that are based on properties of cost functions.

15.4.1 Convex Hull Trick

The *convex hull trick* allows us to efficiently find the minimum function value at a given point x among a set of n linear functions of the form $f(x) = ax + b$. For example, Fig. 15.23 shows functions $f_1(x) = x + 2$, $f_2(x) = x/3 + 4$, $f_3(x) = x/6 + 5$, and $f_4(x) = -x/4 + 7$. The minimum value at point $x = 4$ is $f_2(4) = 16/3$.

The idea is to divide the x-axis into ranges where a certain function has the minimum value. It turns out that each function will have at most one range, and we can store the ranges in a sorted list that will contain at most n ranges. For example,

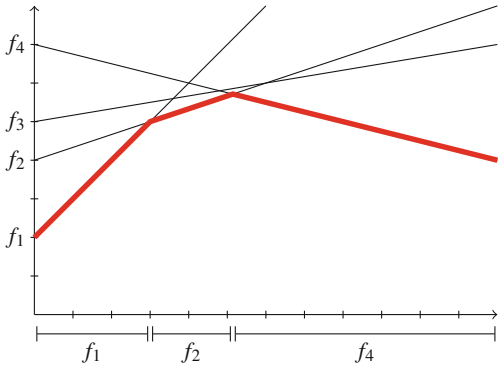


Fig. 15.24 The ranges where f_1, f_2, and f_4 have the minimum value

Fig. 15.24 shows the ranges in our example scenario. First, f_1 has the minimum value, then f_2 has the minimum value, and finally f_4 has the minimum value. Note that f_3 never has the minimum value.

Given a list of ranges, we can find the minimum function value at point x in $O(\log n)$ time using binary search. For example, since point $x = 4$ belongs to the range of f_2 in Fig. 15.24, we immediately know that the minimum function value at point $x = 4$ is $f_2(4) = 16/3$. Thus, we can process a set of k queries in $O(k \log n)$ time. Moreover, if the queries are given in increasing order, we can process them in $O(k)$ time by just iterating through the ranges from left to right.

Then, how to determine the ranges? If the functions are given in decreasing order of their slopes, we can easily find the ranges, because we can maintain a stack that contains the ranges, and the amortized cost for processing each function is $O(1)$. If the functions are given in an arbitrary order, we need to use a more sophisticated set structure and processing each function takes $O(\log n)$ time.

Example Suppose that there are n consecutive concerts. The ticket for concert i costs p_i euros, and if we attend the concert, we get a discount coupon whose value is d_i ($0 < d_i < 1$). We can later use the coupon to buy a ticket for $d_i\, p$ euros where p is the original price. It is also known that $d_i \geq d_{i+1}$ for all consecutive concerts i and $i + 1$. We definitely want to attend the last concert, and we can also attend other concerts. What is the minimum total price for this?

We can easily solve the problem using dynamic programming by calculating for each concert i a value u_i: the minimum price for attending concert i and possibly some previous concerts. A simple way to find the optimal choice for the previous concert is to go through all previous concerts in $O(n)$ time, which results in an $O(n^2)$ time algorithm. However, we can use the convex hull trick to find the optimal choice in $O(\log n)$ time and get an $O(n \log n)$ time algorithm.

The idea is to maintain a set of linear functions, which initially only contains the function $f(x) = x$, which means that we do not have a discount coupon. To calculate the value u_i for a concert, we find a function f in our set that minimizes the value of $f(p_i)$, which can be done in $O(\log n)$ time using the convex hull trick. Then, we add a function $f(x) = d_i x + u_i$ to our set, and we can use it to attend another concert later. The resulting algorithm works in $O(n \log n)$ time.

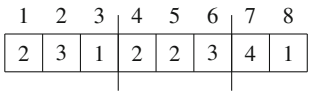


Fig. 15.25 An optimal way to divide a sequence into three blocks

Note that if it is additionally known that $p_i \leq p_{i+1}$ for all consecutive concerts i and $i+1$, we can solve the problem more efficiently in $O(n)$ time, because we can process the ranges from left to right and find each optimal choice in amortized constant time instead of using binary search.

15.4.2 Divide and Conquer Optimization

The *divide and conquer optimization* can be applied to certain dynamic programming problems where a sequence $s_1, s_2, \ldots, s_n$ of n elements has to be divided into k subsequences of consecutive elements. A cost function $\texttt{cost}(a, b)$ is given, which determines the cost of creating a subsequence $s_a, s_{a+1}, \ldots, s_b$. The total cost of a division is the sum of the individual costs of the subsequences, and our task is to find a division that minimizes the total cost.

As an example, suppose that we have a sequence of positive integers and $\texttt{cost}(a, b) = (s_a + s_{a+1} + \cdots + s_b)^2$. Figure 15.25 shows an optimal way to divide a sequence into three subsequences using this cost function. The total cost of the division is $(2+3+1)^2 + (2+2+3)^2 + (4+1)^2 = 110$.

We can solve the problem by defining a function $\texttt{solve}(i, j)$ which gives the minimum total cost of dividing the first i elements $s_1, s_2, \ldots, s_i$ into j subsequences. Clearly, $\texttt{solve}(n, k)$ equals the answer to the problem. To calculate a value of $\texttt{solve}(i, j)$, we have to find a position $1 \leq p \leq i$ that minimizes the value of

$$\texttt{solve}(p-1, j-1) + \texttt{cost}(p, i).$$

For example, in Fig. 15.25, an optimal choice for $\texttt{solve}(8, 3)$ is $p = 7$. A simple way to find an optimal position is to check all positions $1, 2, \ldots, i$, which takes $O(n)$ time. By calculating all values of $\texttt{solve}(i, j)$ like this, we get a dynamic programming algorithm that works in $O(n^2 k)$ time. However, using the divide and conquer optimization, we can improve the time complexity to $O(nk \log n)$.

The divide and conquer optimization can be used if the cost function satisfies the *quadrangle inequality*

$$\texttt{cost}(a, c) + \texttt{cost}(b, d) \leq \texttt{cost}(a, d) + \texttt{cost}(b, c)$$

for all $a \leq b \leq c \leq d$. Let $\texttt{pos}(i, j)$ denote the smallest position p that minimizes the cost of a division for $\texttt{solve}(i, j)$. If the above inequality holds, it is guaranteed that $\texttt{pos}(i, j) \leq \texttt{pos}(i+1, j)$ for all values of i and j, which allows us to calculate the values of $\texttt{solve}(i, j)$ more efficiently.

The idea is to create a function $\texttt{calc}(j, a, b, x, y)$ that calculates all values of $\texttt{solve}(i, j)$ for $a \leq i \leq b$ and a fixed j using the information that $x \leq \texttt{pos}(i, j) \leq y$. The function first calculates the value of $\texttt{solve}(z, j)$ where $z = \lfloor (a+b)/2 \rfloor$. Then it performs recursive calls $\texttt{calc}(j, a, z-1, x, p)$ and $\texttt{calc}(j, z+1, b, p, y)$ where $p = \texttt{pos}(z, j)$. Here the fact that $\texttt{pos}(i, j) \leq \texttt{pos}(i+1, j)$ is used to limit the search range. To calculate all values of $\texttt{solve}(i, j)$, we perform a function call $\texttt{calc}(j, 1, n, 1, n)$ for each $j = 1, 2, \ldots, k$. Since each such function call takes $O(n \log n)$ time, the resulting algorithm works in $O(nk \log n)$ time.

Finally, let us prove that the squared sum cost function in our example satisfies the quadrangle inequality. Let $\texttt{sum}(a, b)$ denote the sum of values in range $[a, b]$, and let $x = \texttt{sum}(b, c)$, $y = \texttt{sum}(a, c) - \texttt{sum}(b, c)$, and $z = \texttt{sum}(b, d) - \texttt{sum}(b, c)$. Using this notation, the quadrangle inequality becomes

$$(x+y)^2 + (x+z)^2 \leq (x+y+z)^2 + x^2,$$

which is equal to

$$0 \leq 2yz.$$

Since y and z are nonnegative values, this completes the proof.

15.4.3 Knuth's Optimization

Knuth's optimization[2] can be used in certain dynamic programming problems where we are asked to divide a sequence $s_1, s_2, \ldots, s_n$ of n elements into single elements using splitting operations. A cost function $\texttt{cost}(a, b)$ gives the cost of processing a sequence $s_a, s_{a+1}, \ldots, s_b$, and our task is to find a solution that minimizes the total sum of the splitting costs.

For example, suppose that $\texttt{cost}(a, b) = s_a + s_{a+1} + \cdots + s_b$. Figure 15.26 shows an optimal way to process a sequence in this case. The total cost of this solution is $19 + 9 + 10 + 5 = 43$.

We can solve the problem by defining a function $\texttt{solve}(i, j)$ which gives the minimum cost of dividing the sequence $s_i, s_{i+1}, \ldots, s_j$ into single elements. Then, $\texttt{solve}(1, n)$ gives the answer to the problem. To determine a value of $\texttt{solve}(i, j)$, we have to find a position $i \leq p < j$ that minimizes the value of

$$\texttt{cost}(i, j) + \texttt{solve}(i, p) + \texttt{solve}(p+1, j).$$

If we check all positions between i and j, we get a dynamic programming algorithm that works in $O(n^3)$ time. However, using Knuth's optimization, we can calculate the values of $\texttt{solve}(i, j)$ more efficiently in $O(n^2)$ time.

[2] Knuth [20] used his optimization to construct optimal binary search trees; later, Yao [32] generalized the optimization to other similar problems.

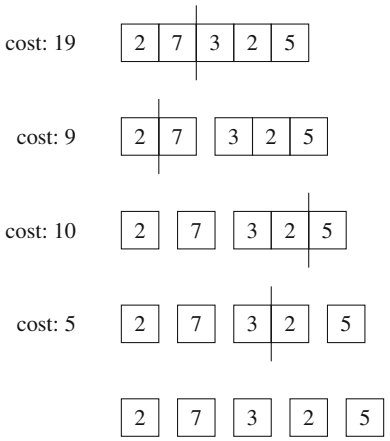


Fig. 15.26 An optimal way to divide an array into single elements

Knuth's optimization is applicable if

$$\texttt{cost}(b, c) \leq \texttt{cost}(a, d)$$

and

$$\texttt{cost}(a, c) + \texttt{cost}(b, d) \leq \texttt{cost}(a, d) + \texttt{cost}(b, c)$$

for all values of $a \leq b \leq c \leq d$. Note that the latter inequality is the quadrangle inequality that was also used in the divide and conquer optimization. Let $\texttt{pos}(i, j)$ denote the smallest position p that minimizes the cost for $\texttt{solve}(i, j)$. If the above inequalities hold, we know that

$$\texttt{pos}(i, j-1) \leq \texttt{pos}(i, j) \leq \texttt{pos}(i+1, j).$$

Now we can perform n rounds $1, 2, \ldots, n$, and on round k calculate the values of $\texttt{solve}(i, j)$ where $j - i + 1 = k$, i.e., we process the subsequences in increasing order of length. Since we know that $\texttt{pos}(i, j)$ has to be between $\texttt{pos}(i, j-1)$ and $\texttt{pos}(i+1, j)$, we can perform each round in $O(n)$ time, and the total time complexity of the algorithm becomes $O(n^2)$.

15.5 Miscellaneous

This section presents a selection of miscellaneous algorithm design techniques. We discuss the meet in the middle technique, a dynamic programming algorithm for counting subsets, the parallel binary search technique, and an offline solution to the dynamic connectivity problem.

15.5.1 Meet in the Middle

The *meet in the middle* technique divides the search space into two parts of about equal size, performs a separate search for both of the parts, and finally combines the results of the searches. Meet in the middle allows us to speed up certain $O(2^n)$ time algorithms so that they work in only $O(2^{n/2})$ time. Note that $O(2^{n/2})$ is much faster than $O(2^n)$, because $2^{n/2} = \sqrt{2^n}$. Using an $O(2^n)$ algorithm we can process inputs where $n \approx 20$, but using an $O(2^{n/2})$ algorithm the bound is $n \approx 40$.

Suppose that we are given a set of n integers and our task is to determine whether the set has a subset with sum x. For example, given the set $\{2, 4, 5, 9\}$ and $x = 15$, we can choose the subset $\{2, 4, 9\}$, because $2 + 4 + 9 = 15$. We can easily solve the problem in $O(2^n)$ time by going through every possible subset, but next we will solve the problem more efficiently in $O(2^{n/2})$ time using meet in the middle.

The idea is to divide our set into two sets A and B such that both sets contain about half of the numbers. We perform two searches: the first search generates all subsets of A and stores their sums to a list S_A, and the second search creates a similar list S_B for B. After this, it suffices to check if we can choose one element from S_A and another element from S_B such that their sum is x, which is possible exactly when the original set contains a subset with sum x.

For example, let us see how the set $\{2, 4, 5, 9\}$ is processed. First, we divide the set into sets $A = \{2, 4\}$ and $B = \{5, 9\}$. After this, we create lists $S_A = [0, 2, 4, 6]$ and $S_B = [0, 5, 9, 14]$. Since S_A contains the sum 6 and S_B contains the sum 9, we conclude that the original set has a subset with sum $6 + 9 = 15$.

With a good implementation, we can create the lists S_A and S_B in $O(2^{n/2})$ time in such a way that the lists are sorted. After this, we can use a two pointers algorithm to check in $O(2^{n/2})$ time if the sum x can be created from S_A and S_B. Thus, the total time complexity of the algorithm is $O(2^{n/2})$.

15.5.2 Counting Subsets

Let $X = \{0 \ldots n-1\}$, and each subset $S \subset X$ is assigned an integer $\texttt{value}[S]$. Our task is to calculate for each S

$$\texttt{sum}(S) = \sum_{A \subset S} \texttt{value}[A],$$

i.e., the sum of values of subsets of S.

For example, suppose that $n = 3$ and the values are as follows:

- $\texttt{value}[\emptyset] = 3$
- $\texttt{value}[\{0\}] = 1$
- $\texttt{value}[\{1\}] = 4$
- $\texttt{value}[\{0, 1\}] = 5$
- $\texttt{value}[\{2\}] = 5$
- $\texttt{value}[\{0, 2\}] = 1$
- $\texttt{value}[\{1, 2\}] = 3$
- $\texttt{value}[\{0, 1, 2\}] = 3$

In this case, for example,

$$\begin{aligned}\texttt{sum}(\{0,2\}) &= \texttt{value}[\emptyset] + \texttt{value}[\{0\}] + \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}] \\ &= 3+1+5+1 = 10.\end{aligned}$$

Next we will see how to solve the problem in $O(2^n n)$ time using dynamic programming and bit operations. The idea is to consider subproblems where it is limited which elements may be removed from S.

Let $\texttt{partial}(S,k)$ denote the sum of values of subsets of S with the restriction that only elements $0 \ldots k$ may be removed from S. For example,

$$\texttt{partial}(\{0,2\},1) = \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}],$$

because we only may remove elements $0 \ldots 1$. Note that we can calculate any value of $\texttt{sum}(S)$ using $\texttt{partial}$, because

$$\texttt{sum}(S) = \texttt{partial}(S, n-1).$$

To use dynamic programming, we have to find a recurrence for $\texttt{partial}$. First, the base cases are

$$\texttt{partial}(S,-1) = \texttt{value}[S],$$

because no elements can be removed from S. Then, in the general case we can calculate the values as follows:

$$\texttt{partial}(S,k) = \begin{cases} \texttt{partial}(S,k-1) & k \notin S \\ \texttt{partial}(S,k-1) + \texttt{partial}(S \setminus \{k\}, k-1) & k \in S \end{cases}$$

Here we focus on the element k. If $k \in S$, there are two options: we can either keep k in the subset or remove it from the subset.

Implementation There is a particularly clever way to implement a dynamic programming solution using bit operations. Namely we can declare an array

```
int sum[1<<N];
```

that will contain the sum of each subset. The array is initialized as follows:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

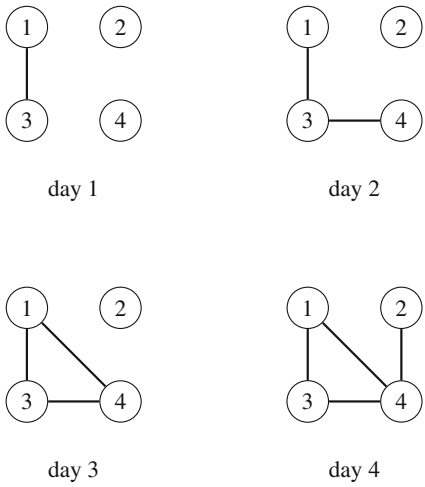


Fig. 15.27 An instance of the road building problem

Then, we can fill the array as follows:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];
    }
}
```

This code calculates the values of `partial`(S, k) for $k = 0 \ldots n-1$ to the array `sum`. Since `partial`(S, k) is always based on `partial`$(S, k-1)$, we can reuse the array `sum`, which yields a very efficient implementation.

15.5.3 Parallel Binary Search

Parallel binary search is a technique that allows us to make some binary search based algorithms more efficient. The general idea is to perform several binary searches simultaneously, instead of doing the searches separately.

As an example, consider the following problem: There are n cities numbered $1, 2, \ldots, n$. Initially there are no roads between the cities. Then, during m days, each day a new road is built between two cities. Finally, we are given k queries of the form (a, b), and our task is to determine for each query the earliest moment when cities a and b are connected. We can assume that all requested pairs of cities are connected after m days.

Figure 15.27 shows an example scenario where there are four cities. Suppose that the queries are $q_1 = (1, 4)$ and $q_2 = (2, 3)$. The answer for q_1 is 2, because cities 1 and 4 are connected after day 2, and the answer for q_2 is 4, because cities 2 and 3 are connected after day 4.

Let us first consider an easier problem where we have only one query (a, b). In this case, we can use a union-find structure to simulate the process of adding roads to the network. After each new road, we check if cities a and b are connected and stop the search if they are. Both adding a road and checking if cities are connected take $O(\log n)$ time, so the algorithm works in $O(m \log n)$ time.

How could we generalize this solution to k queries? Of course we could process each query separately, but such an algorithm would take $O(km \log n)$ time, which would be slow if both k and m are large. Next we will see how we can solve the problem more efficiently using parallel binary search.

The idea is to assign each query a range $[x, y]$ which means that the cities are connected for the first time no earlier than after x days and no later than after y days. Initially, each range is $[1, m]$. Then, we simulate $\log m$ times the process of adding all roads to the network using a union-find structure. For each query, we check at moment $u = \lfloor (x + y)/2 \rfloor$ if the cities are connected. If they are, the new range becomes $[x, u]$, and otherwise the range becomes $[u + 1, y]$. After $\log m$ rounds, each range only contains a single moment which is the answer to the query.

During each round, we add m roads to the network in $O(m \log n)$ time and check whether k pairs of cities are connected in $O(k \log n)$ time. Thus, since there are $\log m$ rounds, the resulting algorithm works in $O((m + k) \log n \log m)$ time.

15.5.4 Dynamic Connectivity

Suppose that there is a graph of n nodes and m edges. Then, we are given q queries, each of which is either "add an edge between nodes a and b" or "remove the edge between nodes a and b." Our task is to efficiently report the number of connected components in the graph after each query.

Figure 15.28 shows an example of the process. Initially, the graph has three components. Then, the edge 2–4 is added, which joins two components. After this, the edge 4–5 is added and the edge 2–5 is removed, but the number of components remains the same. Then, the edge 1–3 is added, which joins two components, and finally, the edge 2–4 is removed, which divides a component into two components.

If edges would only be added to the graph, the problem would be easy to solve using a union-find data structure, but the removal operations make the problem much more difficult. Next we will discuss a divide and conquer algorithm for solving the offline version of the problem where all queries are known beforehand, and we are allowed to report the results in any order. The algorithm presented here is based on the work by Kopeliovich [21].

The idea is to create a timeline where each edge is represented by an interval that shows the insertion and removal time of the edge. The timeline spans a range $[0, q + 1]$, and an edge that is added on step a and removed on step b is represented by an interval $[a, b]$. If an edge belongs to the initial graph, $a = 0$, and if an edge is never removed, $b = q + 1$. Figure 15.29 shows the timeline in our example scenario.

To process the intervals, we create a graph that has n nodes and no edges, and use a recursive function that is called with range $[0, q + 1]$. The function works as

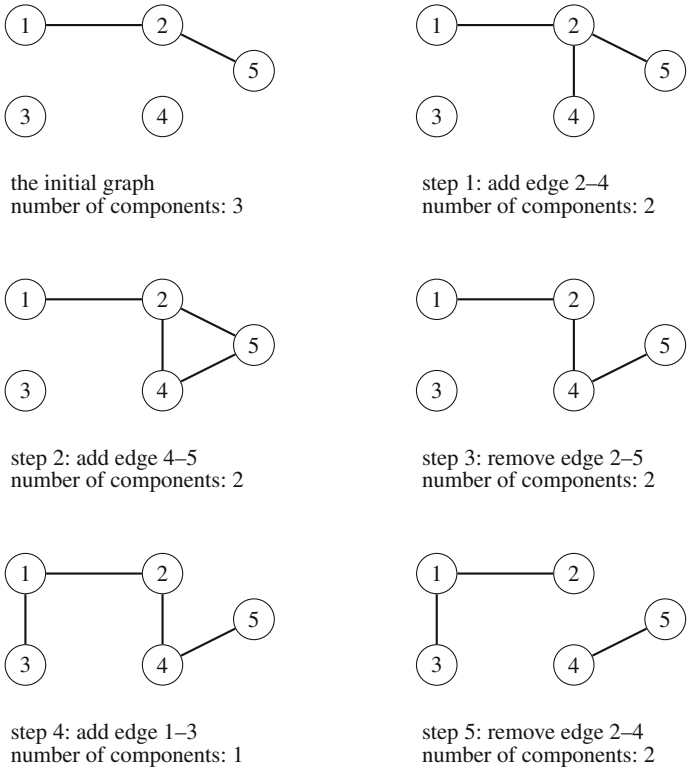
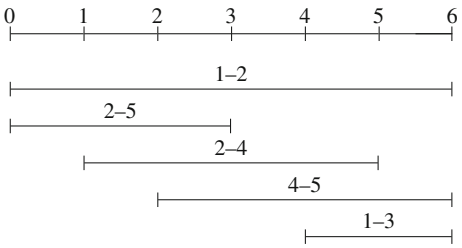


Fig. 15.28 The dynamic connectivity problem

Fig. 15.29 Timeline of edge insertions and removals



follows for a range $[a, b]$: First, if $[a, b]$ is completely inside the interval of an edge, and the edge does not belong to the graph, it is added to the graph. Then, if the size of $[a, b]$ is 1, we report the number of connected components, and otherwise we recursively process ranges $[a, k]$ and $[k, b]$ where $k = \lfloor (a + b)/2 \rfloor$. Finally, we remove all edges that were added at the beginning of processing the range $[a, b]$.

Always when an edge is added or removed, we also update the number of components. This can be done using a union-find data structure, because we always remove the edge that was added last. Thus, it suffices to implement an *undo* oper-

ation for the union-find structure, which is possible by storing information about operations in a stack. Since each edge is added and removed at most $O(\log q)$ times and each operation works in $O(\log n)$ time, the total running time of the algorithm is $O((m+q)\log q \log n)$.

Note that in addition to counting the number of components, we may maintain any information that can combined with the union-find data structure. For example, we may maintain the number of nodes in the largest component or the bipartiteness of each component. The technique can also be generalized to other data structures that support insertion and undo operations.