FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Integer data structures
Student:	Bc. Miloslav Brožek
Supervisor:	RNDr. Tomáš Valla, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

With the recent development of modern CPUs, their word size is steadily increasing.
There is an ongoing research of integer data structures, which are optimised for large word sizes.
This thesis aims to contribute to this topic. Its goals are:
1) survey existing results
2) design the integer segment tree with a nontrivial set of supported operations
3) study the applications of integer data structures for optimizing various integer algorithms
This is a research-oriented thesis, the expected results are theoretical designs of algorithms.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 14, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Integer data structures

Miloslav Brožek

Department of Theoretical Informatics
Supervisor: Tomáš Valla

January 9, 2019

Acknowledgements

I would like to thank my supervisor Tom for many helpful discussion about the topics of this thesis. I would also like to express my thanks to blazewan for helping me with my poor English, stylization, and wording.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on January 9, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Brožek, Miloslav. *Integer data structures.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Cílem této práce je návrh a zrychlení datových struktur a algoritmů na modelu RAM s velkou šířkou slova. Zrychlení se děje zejména za pomoci bitových paralelních instrukcí.

V práci tuto techniku aplikujeme pro zlepšení asymptotické složitost vybraných algoritmů a struktur.

Studujeme zejména datové struktury, které se zaměřují na segmentové operace, kterými jsou například počet prvků na intervalu či maximum na intervalu. Algoritmy, na které se práce zaměřuje jsou pak největší společný dělitel, Sparse Table a Number Theoretic Transform.

Klíčová slova Integer, Datové Struktury, Algoritmy, Segmentový Strom, NTT, NSD, Bity, Paralelismus, RAM

Abstract

This thesis focuses on speedup and design of algorithms and data structures on RAM model with the limited size of integer. We use effective bitwise operations to achieve the speedup.

The thesis shows the impact of such operations on the asymptotic complexity of selected algorithms and data structures.

The data structures which are being speeded up focuses on segmental operations such as the number of elements on interval or biggest value on an interval. This thesis also focusing on greatest common divisor algorithm, Sparse Table and Number Theoretic Transform.

Keywords Integer, Data Structure, Algorithm, Segment Tree, NTT, GCD, Bits, Parallelism, RAM

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In this work, we will focus on the RAM model with large word size. We will take peek into possibilities to speed up multiple data structures and algorithms. The key to achieve the acceleration will be efficient usage of bit tricks and bit parallelism. To understand this more, this thesis will present several known techniques and bitwise functions.

We will introduce and discuss a couple of models similar to RAM model with limited size of integers, such as AC^0.

Our research also shows a few basic algorithms and data structures. It shows the impact of efficient bit tricks and bit parallelism on their asymptotic complexity. Our research is rather theoretical as it mostly relies on word size, which might hopefully increase in some time.

1.2 Results of this thesis

The goal of this thesis is to show asymptotic speedup in respect to the word size of the machine of data structures and algorithms by using bitwise techniques. We focus on the following problems:

The first problem is finding the number of elements from a given range. To solve this problem we propose a data structure called *Sum-element Tree* which combines the properties of *Segment Trees* and *Word-Encoded B-Trees*. Achieved complexity of operations of this data structure (insertion, deletion and counting elements) is $\mathcal{O}(\log_{W/\log n} n)$ (where W stands for the size of the word and n for the size of the universe) for each query. Afterwards, we propose an extension to this data structure which solves a slight generalization of the problem.

Next, we propose data structure called *Max-value Tree* for searching the maximum or minimum on a given range. Similarly to the previous data structure, it is derived from *Segment Trees* and *Word-Encoded B-Trees*. The

achieved complexity of operations of this data structure (update and maximum) is $\mathcal{O}(\log_W n)$ for each query. We also discuss this structure under AC^0.

In the next part of this work, we focus on bringing some persistence into the data structures. We also show a simple utilization of persistence of the *Sum-element Tree*.

The second half of the work mainly focuses on improving and analyzing the known algorithms. It is shown how to produce asymptotical speed-up by using bitwise parallelism.

First of these algorithms is binary GCD algorithm. We show how bitwise parallelization could be achieved for every operation while not changing the original asymptotic complexity. We also show how to use this in a simple *Sparse Table* data structure.

The last part pf this thesis focuses on bitwise parallelisation of NTT. We claim we achieved $\mathcal{O}\left(n \log(n) \log(m)/W + n + (\log m)^2\right)$ complexity, where m stands for modulus used in this algorithm.

1.3 Motivation

The results of this thesis are nowadays mainly theoretical. Yet this does not mean they cannot come to use later. Most of the work relies on the size of the word.

The most common size of integers is now 64 bits. Anyway, this number is slowly but steadily increasing.

We can also refer to **AVX** which are extensions to the x86 instruction set architecture. As it can be seen there is already an article by James Reinders [1], describing an instruction set for 512 bits sized integers, called **AVX-512**. It isn't fully implementet and consolidated, yet it is on good way to extend **AVX-256** (instruction set for 256 bits sized integers). The main purpose of **AVX-512** is vectorisation of operations over integers, yet it already includes multiple basic bitwise instructions (such as AND, OR and XOR) and some very interesting instructions, such as KUNPCK (instruction for UNPACK) or instruction for POPCOUNT.

1.4 RAM Model

Definition 1. ***Random Access Machine*** *as you could see in Mareš [2] is a family of models. The common attribute of this family is that it works with discrete numbers and the memory is indexed by integers. Instructions in program work with operands, which are either constants or memory cells which are directly or indirectly indexed.*

If we allow counting with arbitrarily large numbers in constant time, we get a very strong parallel computer which could count almost everything in

constant time (neglecting encoding of the input). We have to limit this model to make it more realistic.

Definition 2. *Model limitation could be done in multiple ways:*

- *Logarithmic price of instructions: The cost of an operation is equal to the number of bits it works with, including addresses to memory. This removes absurd situations, yet makes it very hard to estimate time complexity. It usually leads to time which is* $\Theta(\log n)$ *times bigger than in unlimited RAM.*

- *Limit the size of integers: This method limits the width of integers in bits to some* W *and the operations remain* $\mathcal{O}(1)$*. To successfully address input, we have the following restriction:* $W \geq \log n$*, where* n *stands for the size of the input. As arithmetic with* $\mathcal{O}(1)$*-times accuracy could be simulated with a constant slowdown, we can assume that* $W = \Omega(n)$*, so we could work with numbers polynomially big in respect to* n*. These operations will be mentioned in the previous section.*

- AC^0*-RAM: This model supports any function which is evaluable by a network of gates of polynomial size and constant depth. The gates could have any number of inputs. This a cleaner definition, yet it lacks some important functions such as parity, multiplication, division or modulo. It can be also proven [3] that parity is not in* AC^0*, which also implies that multiplication is not in* AC^0 *too.*

We will mainly focus on the model with limited size of integers.

1.5 RAM Operations

Definition 3. *In RAM model with limited size of integers we use word of size* W*. We consider of basic operations in Table 1.1 to only take* $\mathcal{O}(1)$ *time to be executed:*

$\&$	Binary AND
$\mid$	Binary OR
$\oplus$	Binary XOR
$+$	Addition
$-$	Subtraction
$*$	Multiplication
$/$	Division
$\%$	Modulo
$\sim$	Bitwise negation
$\gg$	Right bit-shift
$\ll$	Left bit-shift
$>$	Comparison (greater)
$<$	Comparison (lesser)
$=$	Comparison (equal)

Table 1.1: Basic RAM operations

There might be possibly more operations than this chosen set yet we either will not use them or they could be derived from these operations. An example of easily derived operations might be "$\geq$" or "$\leq$", which are simple combination of comparisons.

Also, note that 2^k could be also obtained in $\mathcal{O}(1)$ as "$1 \ll k$".

Also note that this definition of $\mathcal{O}(1)$ basic functions isn't as clean as of AC^0. The most troublesome are some arithmetic operations such as $*$, / or %. It is convention to consider them (see Mareš [2]) to have constant slowdown, even though it is not possible to achieve a circuit with constant depth (at least if polynomial size is demanded). Anyway note that all these instructions are pretty common in programming languages. Also note that the slowdown of these instructions is currently not that big—see[4].

1.6 Bit Tricks

Definition 4. *Let us introduce the concept of* ***buckets****. We say we split a word of size W bits into κ buckets of size t. i-th bucket has assigned bits of on segment $[i \cdot t, (i+1) \cdot t - 1]$.*

1.6.1 Simple functions

In Table 1.2 there are several functions, which cost constant time. These functions are designed as composition of basic operations. You can read about most of these functions here [2].

Name	Description	Realization
ERASE	This function will set all bits of an integer α between i and j to 0	$\alpha = \alpha \,\&\, ((2^i - 1) \mid ((2^{W-j} - 1) \ll (j + 1)))$
REPLICATE	Replicate m bit integer α, κ times so it will be concatenated after each other.	$\alpha \cdot (0^{m-1}1)^\kappa$
SUM	Sums all components of vector α, with κ buckets of size b (we assume it will fit into b bits). The sum of all buckets will appear in the position of last bucket.	$\alpha \cdot (0^{b-1}1)^\kappa$
UNPACK	This function will interlace bits of integer x by zeroes.	We will REPLICATE x, AND it with proper constant c so that only i-th bit will remain in i-th bucket. Afterwards, we subtract it from constant η in form of $(0^{b-1}1)^\kappa$ (where κ stans for number of buckets), AND it again with the constant and properly shift.
POPCOUNT	Finds the number of bits of b-bit integer with usage of b^2 bit integer. Note that sometime instruction POPCOUNT is available.	We firstly execute UNPACK and then SUM.
LSB	Finds the least significant one-bit (which is the rightmost one) of integer α.	We call POPCOUNT on $\alpha \oplus (\alpha - 1)$.
PACK	Scratches out all zeroes between some bits—making it more compact. For instance we could imagine a b bit buckets while we are interested only in last bit (the rest are 0).	We imagine the buckets to be one bit shorter and we call SUM (and properly extract the bits).
ZR	Recognizes buckets (of some variable a) which are filled with 0. The time complexity of this function is $\mathcal{O}(1)$. Note that for this operation we have to have an additional bit for in every bucket (which will be 0 in a).	This can be done by following formula: $(c - a) \gg \kappa$ (where c is an constant filled by 0-bits with exception of the first bit of every bucket which is set to 1 and κ is the size of bucket).

Table 1.2: List of simple bitwise functions

1.6.2 Finding the most significant bit

In this subsection, we are going to discuss the algorithm for finding the position of **the most significant bit**, see Demaine [5], of an integer χ in constant time. Also let us assume for simplicity that W is a square (so for some r,

$W = r^2$).

Some processors might have the operation CLZ which would do this procedure pretty fast, yet it is not a general possession.

Definition 5. *Let us define a constant λ in form of $(10^{\sqrt{W}-1})^{\sqrt{W}}$ (the power means concatenation).*

For example for $W = 16$, it would look like: 1000100010001000.
Now the MSB could be found by following these steps:

- Firstly imagine the integer to be split into $\sqrt{W}$ buckets of size $\sqrt{W}$. We would like to find out the empty buckets.
- The first step is the identification of buckets which have first bit set to 1: $t_1 = \chi \,\&\, \lambda$.
- As next step we will erase those bits from χ:

$$t_2 = \chi \oplus t_1$$

- The third step is finding, whether there is a 1 bit in the rest of the bucket. This can be done by:

$$t_3 = ((\lambda - t_2) \,\&\, \lambda) \oplus \lambda$$

  As we can see, the first step is subtraction. We know that every bucket of λ is strictly greater than any bucket of t_2, since it has set the most significant bit to 1 (which means the buckets are independent). As we can see, if the rest of the bucket of t_2 is lesser/equal, the 1 will remain there, otherwise, it will get borrowed. Note that it is lesser/equal only if it is empty. As we will be interested only to the first bit, we have to AND it with λ (which will get rid of the rest—garbage). Also as we can see, now it is inverse to what we wanted—there is 1 if the bucket is empty. It is not desired so we have to convert zeroes to ones (at the first positions) and vice versa. This could be done by XOR with λ.

- The fourth step is combining the first bits and the rest together, which could be simply done as:

$$t_4 = t_2 \,|\, t_3$$

- The fifth step is to compress the important bits to a single $\sqrt{W}$ bucket. This operation is slightly problematic in general, yet here the positions are in a regular pattern. As we can see, the i-th (interesting) bit of our number t_4 is on position $(i+1)\sqrt{W} - 1$. We will now define a special number m (with respect to W) with j-th bit set on one: $m_j = W - \sqrt{W} + 1 - j\sqrt{W} + j$. We will multiply t_4 by m and see what happens.

The first key fact is that multiplication of such integers can be seen as an addition of powers of important bits (bits which are set on). That means, we are looking for $t_{4_i} + m_j$ which is $W + (i-j)\sqrt{W} + j$. As we can observe, these sums will be distinct for all $0 \leq i, j < \sqrt{W}$, which means there will be no collisions and nothing could go wrong. Also, we shall observe that $t_{4_i} + m_i$ is equal to $W + i$, which means the important bits got sequentially one after each other in the right order. Even though they are in some weird position, it is no problem to repair this by doing a bit shift:

$$t_5 = ((t_4 \cdot m) \gg W) \,\&\, (2^{\sqrt{W}} - 1).$$

- The sixth step will simply REPLICATE the first and only bucket with appended 1-bit to each of $\sqrt{W}$ buckets. This can be simply done by multiplication with appropriate constant

$$\zeta = \sum_{i=0}^{\sqrt{W}-1} 2^{i \cdot (\sqrt{W}+1)}$$

(which is not dependent on the integer so it could be created before this process)

$$t_6 = (t_5 + 2^{\sqrt{W}}) \cdot \zeta.$$

Note that size of buckets is now $\sqrt{W} + 1$.

- The seventh step is the parallel comparison with powers of 2. Again, we have to create a constant

$$\rho = \sum_{i=0}^{\sqrt{W}-1} 2^i \ll (i \cdot (\sqrt{W} + 1)).$$

As we can see, we also append 0 to each such power (the slot for parallel comparison). Now if we do:

$$t_7 = ((t_6 - \rho) \,\&\, (\zeta \ll (\sqrt{W} - 1))) \oplus (\zeta \ll (\sqrt{W} - 1))$$

We obtain integer, which has the first bit of each $(\sqrt{W} + 1)$ bucket set on, as long as it is lesser than the power of 2 in it.

- The eight step is to find the most significant bit of such integer. This would yield us the first power of 2 which is lesser than t_5, which could identify the first non-empty bucket of χ. Obviously, we cannot use our method since it would lead to infinite recursion. Luckily for us, the integer t_7 fulfills two important properties: Firstly, the important bits are spread in a predictable and regular pattern. Secondly, the number of the bucket with the most significant bit on is equal to the number of 1-bits in t_7 (as we can observe, there will be only zeroes since the powers of

two will be greater and then there will be just ones). This could be done by multiplying the number by ζ. We can look at the multiplication as on addition of integers t_7 always shifted by $(\sqrt{W}+1)\cdot i$. As we can see, it will always shift the important bits into the next important bits which means that starting by first important bit's position, there will be the sum of the previous $\sqrt{W}$ important bits. We might also observe, that there will not be any collisions of such places since $\mathcal{O}(\sqrt{W}) > \mathcal{O}(\log\sqrt{W})$. Now we simply take the number in the position of $\sqrt{W}$-th bucket (by proper SHIFT and AND) which can identify the first non-empty bucket's position.

- The last few steps are finding the most significant bit of the bucket we have identified. The size of the bucket is $\sqrt{W}$ so we can shift it to the first bucket and continue with step **six**.

The process above can be summarized by Algorithm 1:

Algorithm 1 The Most Significant Bit

1: **function** MSB(χ)
2: $t_1 = \chi \,\&\, \lambda$
3: $t_2 = \chi \oplus t_1$
4: $t_3 = ((\lambda - t_2) \,\&\, \lambda) \oplus \lambda$
5: $t_4 = t_3 \,|\, t_1$
6: $t_5 = ((t_4 \cdot m) \gg W) \,\&\, (2^{\sqrt{W}} - 1)$
7: $t_6 = (t_5 + 2^{\sqrt{W}}) \cdot \zeta$
8: $t_7 = ((t_6 - \rho) \,\&\, (\zeta \ll (\sqrt{W} - 1))) \oplus (\zeta \ll (\sqrt{W} - 1))$
9: $t'_5 = t_7 \gg ((\sqrt{W} - 1 - \text{SUM}(t_7)\sqrt{W}) \,\&\, (2^{\sqrt{W}} - 1))$
10: $t'_7 = ((t'_6 - \rho) \,\&\, (\zeta \ll (\sqrt{W} - 1))) \oplus (\zeta \ll (\sqrt{W} - 1))$ **return** $(\sqrt{W} - 1 - \text{SUM}(t_7)\sqrt{W} + (\sqrt{W} - 1 - \text{SUM}(t'_7))$
11: **end function**

Lemma 1. *The complexity of* MSB *is* $\mathcal{O}(1)$.

Proof. Despite the big number of steps, every single of them takes $\mathcal{O}(1)$ so the total complexity is $\mathcal{O}(1)$ too. □

As we might observe, the similar method might be used for LSB too, the only thing we would need to change is the behavior of step 8.

Example 1. *We provide an example 1 of finding the most significant bit of integer*

$$\chi = 0101000010001101$$

by the described algorithm.

Step	**Form of number**
Let us have an integer χ	0101000010001101
We split the integer to 4 buckets of size 4	0101 0000 1000 1101
Then we find $t_1 = \chi \,\&\, \lambda$	0000 0000 1000 1000
Afterwards we extract the first bits of every bucket of χ as $t_2 = \chi \oplus t_1$	0101 0000 0000 0101
Thereafter we find whether the bucket is nonempty: $t_3 = ((\lambda - t_2) \,\&\, \lambda) \oplus \lambda$	1000 0000 0000 1000
In the following step we combine the results of step 1 and step 3: $t_4 = t_3 \,\vert\, t_1$	1000 0000 1000 1000
The next procedure is to compress the first bits into a single bucket: $t_5 = ((t_4 \cdot m) \gg W) \,\&\, (2^{\sqrt{W}} - 1)$	0000 0000 0000 1011
The further step is to enlarge and REPLICATE the last bucket: $t_6 = (t_5 + 2^{\sqrt{W}}) \cdot \zeta$	11011 11011 11011 11011
Now let us have a constant with powers of 2 in each bucket:	01000 00100 00010 00001
In the subsequent step we do parallel comparison with a constant: $t_7 = ((t_6 - \rho) \,\&\, (\zeta \ll (\sqrt{W} - 1))) \oplus (\zeta \ll (\sqrt{W} - 1))$	00000 00000 00000 00000
As the next step we do SUM to find there is no 1, so we identified the first bucket, so let us continue from step 6 with following integer:	0000 0000 0000 0101
So let us follow the step 6: $t_6 = (t_5 + 2^{\sqrt{W}}) \cdot \zeta$	10101 10101 10101 10101
And so with step 7: $t_7 = ((t_6 - \rho) \,\&\, (\zeta \ll (\sqrt{W} - 1))) \oplus (\zeta \ll (\sqrt{W} - 1))$	10000 00000 00000 00000
Now we will find 1 by SUM so we know we the position is 1-st bit from left (indexed from 0) for 0-th bucket.	0101000010001101

Table 1.3: Example of finding the most significant bit of an integer

1.6.3 Stall

Let us define the STALL function. This takes two integers χ and λ. The aim of this function is to temporarily remove some buckets. The fact, whether the bucket will be removed depends on whether last bit of the appropriate bucket of λ is 1 or 0. Also note that we will denote the size of buckets as κ.

We will call first phase STALLPHASEIN. We would like to get rid of all uninteresting bits by the following operation: $\lambda = \lambda \,\&\, c$ (where c is constant which has only the last bit of each bucket set on 1). Firstly we will make the appropriate buckets filled by 1's which could be done as $\lambda = \lambda \cdot (2^{\kappa} - 1)$. Now

we can easily back up the appropriate nodes to variable t as $t = \lambda \,\&\, \chi$.

In between these steps we could apply anything, yet it will not affect the buckets which are phased out.

As last step which we denote as stallPhaseOut we have use our backup: $\chi = (\chi \,\&\, \sim \lambda) \,|\, t$. The complexity of this operation is $\mathcal{O}(1)$.

1.6.4 Cswap

We would like to define cswap (conditioned swap). It has two parameters x and y and after this operation for every bucket if it is greater in x then it will be swapped with the appropriate bucket in y (in fact we do not really care whether equal elements will be swapped too as some overhead).

Definition 6. *Let us define a constant c, as an integer with all bits set to* 0 *except for the last bit of every bucket.*

The pseudocode of cswap looks as follows 2:

Algorithm 2 cswap

```
1: function CSWAP((x, y))
2:     x | = c ≪ κ
3:     STALLPHASEIN(x, (x − y) ≫ κ)
4:     STALLPHASEIN(y, (x − y) ≫ κ)
5:     SWAP(x, y)
6:     STALLPHASEOUT(x, (x − y) ≫ κ)
7:     STALLPHASEOUT(y, (x − y) ≫ κ)
8: end function
```

Firstly we have to find whether the i-th bucket in x is greater or equal. The first step is to fill the first bit of x by 1: $x = x \,|\, (c \ll \kappa)$ (where c is a constant with last bit of each bucket set as 1 and κ is the size of buckets). Now if we subtract y from x, the first bit will remain if and only if x is greater/equal (otherwise we will have to borrow). Since all first-bits of y are empty it is guaranteed that all buckets are independent of each other. Now we have to deliver the bit to the position of last bit, which can be simply done by shift: $a = (x - y) \gg \kappa$. Now we would like to stall by $a \oplus c$ (for both: x and y). This will phase out all buckets which are already in the right position. Now we simply swap x and y (which now works as swapping of bits which were are not phased out) and then phase back.

Lemma 2. *The complexity of* cswap *is* $\mathcal{O}(1)$

Proof. As it could be seen from pseudocode, there are only 6 steps. In each step, all operations are constant—they are either basic operations or stall, whose complexity was shown to be $\mathcal{O}(1)$ in the previous subsection. □

Chapter 2

Segment Data Structures

2.0.1 Word-Encoded B-Trees

Before we will immerse in segmental data structures, we would like to introduce Word-Encoded B-Trees [6]. Word-Encoded B-Tree is vector variant of *B-Tree*. The idea is simple and its structure will be used in multiple following data structures. This structure is a classical *B-Tree* with data in leaves (let us say each leave will have k bits storage). The inner nodes will contain only auxiliary keys and will have at most β children. The depth of tree will be denoted as h (note that it could be derived from the number of leaves n and the branching factor β as $h = \mathcal{O}(\log_\beta n)$).

We use such tree as any *B-Tree*, yet operations on nodes will be done vectorally. This simple structure can implement operations such as *insert*, *delete*, *find*, *predecessor*, *successor* or similar.

Example 2. *For predecessor/successor we will navigate through the tree while using* LSB/MSB *functions to find next nonempty subtree. For these operations the only information we have to store is* OR *of all values in subtree, while storing only* 0 *or* 1 *in leaves, depending on whether the element is inserted or not.*

2.1 Sum-elements

2.1.1 Introduction

Definition 7. *The problem which we will denote as* ***sum-elements*** *will be as follows: This data structure will work over the universe of discrete integers on the interval* $[0, n-1]$*. Then we will have to handle the following queries:*

- SUM*: Answer the number of elements* x_i*, such that* $begin \leq i \leq end$*, which are in the set. This query has begin and end as parameters.*

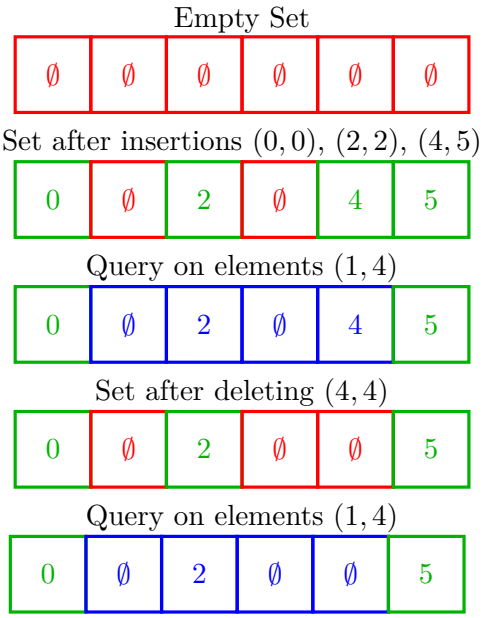

Figure 2.1: Example of sum-elements problem

- DEL*: Delete all elements x_i, such that begin $\leq i \leq$ end, from the set. This query has begin and end as parameters.*

- INSERT*: Insert all elements x_i, such that begin $\leq i \leq$ end, into the set. This query has begin and end as parameters.*

Every element of the set might be present at most once. This means that inserting an element which is already in the set will not do anything.

Example 3. *Let us have such problem for n equal to 6 (empty at fist). We can imagine the set as bit-string of zeroes and ones, where* 1 *means, that element is present (and* 0 *means it is not). Our empty set would look like* 000000*. If we execute insertion on numbers* $\{0, 2, 4, 5\}$*, our set will be* 101011*. If we would ask for the number of elements within an interval with bounds* $(1, 4)$*, the answer shall be* 2 *(*2 *and* 4 *are present on the interval). As we can see, if we delete number* 4 *and ask for the same query as we did before, the answer will be just* 1 *(since we deleted* 4 *and only* 2 *remains on the desired interval). This is illustrated in Figure 2.1.*

2.1.2 Known solutions

2.1.2.1 Fenwick Tree

Fenwick Tree structure invented by Peter Fenwick [7] allows us to add a value to an index and then count prefix sum (both in $\mathcal{O}(\log n)$. This method uses $\mathcal{O}(n)$ memory. As we can see, it easy to bend this structure to solve our problem. Firstly, query 1 and 2 can be solved by adding 1 and -1 to given index (it is also easy to find out whether the index is empty or not). To find out the number of elements on interval (l, r), we simply answer PREFIXSUM(r) − PREFIXSUM$(l-1)$.

This method is highlighted since it is very elegant and easy to code. Unlike the next two methods, the hidden constant here is pretty small.

In fact, there are many more functions than just prefix sum which could be grafted onto Fenwick tree, yet it is not important for our problem.

2.1.2.2 Segment Tree

This data structure, firstly shown by Jon Luis Bentley [8] is slightly similar to Fenwick Tree (for our purposes)—at least in sense of asymptotic complexities (both—execution time of queries and memory).

This structure will be described a little bit more since our new data structure has multiple similar ideas as Segment Tree. Segment Tree has even more functionalities then Fenwick Tree, yet we will focus just on the application, which could solve our problem.

Firstly, imagine a boolean array of size n, with 0 if an element is not present and 1 otherwise. The size of the array will be without loss of generality (further as *w.l.o.g.*) $2^{\lceil \log_2 n \rceil}$ (as we can see, this size is at most two times bigger than original size n). As we have this, we pair every even element with the next element and create a new node, which will be **parent** of both of the elements. As we have all the parent we recursively do the same thing with them, until only one node remains—which we will denote as *root*.

The first and most important observation is, that this structure forms a full binary tree. This implies that the depth of this structure is going to be $\mathcal{O}(\log n)$. We can also easily observe the number of nodes:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2 + 1$$

which sums to $2n - 1$ (so $\mathcal{O}(n)$). In each node, we store the sum of elements in the subtree.

The query of summation on an interval is a little bit more complicated. We have to travel through nodes (beginning in root) and follow these steps:

- If the whole subtree of the actual node lies inside the desired interval, return the value summed in the node.

- If the whole subtree of actual node lies outside the interval, return 0.

- Otherwise (a part of subtree lies in the interval), return the answer of the left son + the answer of the right son.

The complexity of this method in each node is $\mathcal{O}(1)$ and it will travel at most $\mathcal{O}(\log n)$ nodes. To fully understand the number of nodes visited, we have to observe, that third step can be followed by two third-steps at most one time during the whole query—in all other cases it will be followed by at least one step 1 or step 2, either in left son or in right son.

Other two queries are very similar—at least the traversal is.

Another advantage is the strength of this structure—it can not only support many other functions, yet it can convert insertion/deletion operations to be segment too.

We would recommend the reader to fully understand this part before moving to the newly designed structure. As little help for understanding, we could take look at pseudocode of INSERT 3 and GET 4.

Algorithm 3 Insertion of one element to Sum-value Tree

```
1: function INSERT(i, b = 0, e = n − 1)
2:     if i < b || i > e then
3:         return 0
4:     end if
5:     if b == e then
6:         if sum == 1 then
7:             return 0
8:         end if
9:         sum = 1
10:        return 1
11:    end if
12:    tmp = right.INSERT(i, b, (b + e)/2) + left.INSERT(i, (b + e)/2 + 1, e)
13:    sum = tmp + sum
14:    return tmp
15: end function
```

Algorithm 4 Get the number of elements on range from i to j.

```
1: function GET(i, j, b = 0, e = n − 1)
2:     if j < b || i > e then
3:         return 0
4:     end if
5:     if i ≥ b && j ≤ e then
6:         return sum
7:     end if
8:     return right.GET(i, j, b, (b + e)/2) + left.GET(i, j, (b + e)/2 + 1, e)
9: end function
```

2.1.2.3 Balanced Binary Tree

Most of Balanced Binary Trees could be modified so it would work in very similar manner as segment trees. The advantage here is that the complexity of this solution depends on the number of inserted elements and not on the size of the universe. This might come to use if the number of present elements would be asymptotically lesser than the universe of elements. An example of balanced binary search tree might be Red-Black Tree, which was firstly derived from the symmetric B-Trees by Guibas and Sedgewick [9].

2.1.3 Sum-element Tree

Operation	**Complexity**
GET	$\mathcal{O}(\log_{W/\log n} n)$
DEL	$\mathcal{O}(\log_{W/\log n} n)$
INSERT	$\mathcal{O}(\log_{W/\log n} n)$

Table 2.1: Complexities of operations of *Sum-element Tree*

The complexities in Table 2.1 coulde be obtained from Lemmas 7, 8 and 9.

The structure is similar to β-ary Tree. It has one root and every node has β children. The root covers all elements. Then the i-th child of root (indexed from 0) covers all elements between $i\lceil n/\beta \rceil$ and $(i+1)\lceil n/\beta \rceil$ (here n stands for size of subtree). This applies recursively.

This would lead to β^k leaves (for some k). Anyway the last $\beta^k - n$ leaves are not necessary so we would rather allow the last child of each node to possibly cover fewer elements than each previous child. An example of such structure could be seen in Figure 2.2.

Lemma 3. *The depth of Sum-element Tree is* $\mathcal{O}(\log_\beta n)$ *nodes.*

Proof. The size of subtree of every child of node with size n is $\mathcal{O}(\frac{n}{\beta})$. This is an obvious development of logarithm with a base of β. □

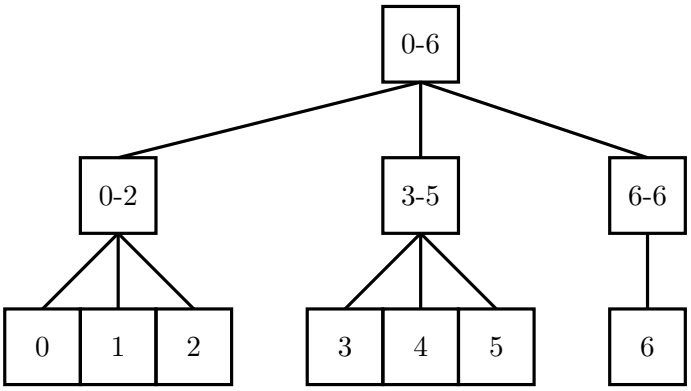

Figure 2.2: Example of tree with β=3 and 7 nodes

With the advantage of depth, a few complications appear. First one is the navigation in the tree, which is straightforward, yet a little bit complicated here. The second problem is the management of the content of nodes: Note that it is also easy in a binary tree, where simple integer with the number of elements in the whole subtree is enough there.

Definition 8. *Each inner node has an integer size which indicates the size of its subtree. Every child, except the last one, must have equal size.*

Imagine we are in a node and we want to reach i-th leaf. To decide which child we have to ask, we simply divide i by *size* of first child: $\lfloor i/SZ \rfloor$. Also, we can recursively ask the children, yet note that we have to ask him for ($i \mod SZ$)-th node (as we just skipped $\lfloor i/SZ \rfloor \cdot SZ$ nodes, where SZ stands for the size of the subtree of first child).

Example of traversal to 5-th node could be found in Figure 2.3.

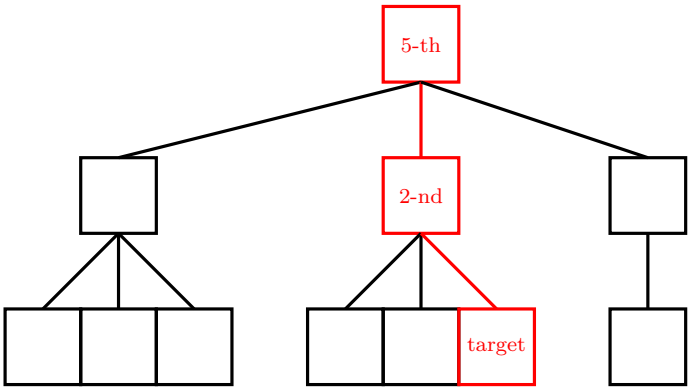

Figure 2.3: Example of traversal to 5-th node in tree from previous sample (indexed from 0)

Traversal to an interval of nodes $[\ell,r]$ looks as follows:

- If the leftmost node of the actual subtree is greater/equal to ℓ and the rightmost node of the actual subtree is lesser/equal to r: End in this node.
- If ℓ and r are in the same subtree: Traverse to the subtree.
- Otherwise: Visit the child with ℓ unless it is the first node in the subtree. Also, visit the child with r unless it is the last node in the subtree.

Observe that with such definition we cannot even visit leaves since they contain only one node which is always the first and the last node of the subtree.

Lemma 4. *With the method above we will visit at most* $\mathcal{O}(\log_\beta n)$ *of Sum-elements Tree.*

Proof. To analyze this, it is key to observe, that 3^{rd} item—which is the only one allowing traversal to multiple children—could split at most *once* during the whole traversal. In the worst case this leads into two paths from **root** to nodes above leaves, which is of the same length as the depth of the tree. □

Example to traversal for the whole segment could be found in Figure 2.4.

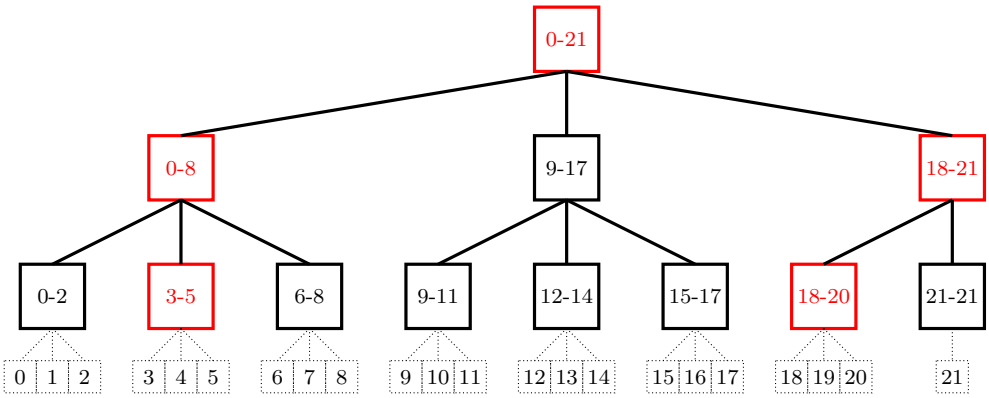

Figure 2.4: Example of traversal from 4 to 18 in tree with $\beta = 3$ and 22 nodes

2.1.4 Node structure

Definition 9. *Every node will have a word sum, which will be split into buckets, such that i-th bucket will contain the number of elements in the subtrees of first i children.*

Lemma 5. *The size of buckets of root of Sum-elements Tree is at least* $\lceil \log_2 n \rceil$.

Proof. We might observe that the rightmost bucket of root might contain number n in itself if all elements are filled-in. This means we need $\lceil \log_2 n \rceil$ bits to represent it. □

Lemma 6. *The depth of Sum-element Tree is* $\mathcal{O}(\log_{\frac{W}{\log n}} n)$.

Proof. The number in buckets $\lceil \log_2 n \rceil$ leads to bound of $\lfloor \frac{W}{\log_2 n} \rfloor$ children for each node. This means it will be a $\lfloor \frac{W}{\log_2 n} \rfloor$-ary Tree, which means the depth is $\mathcal{O}(\log_{\frac{W}{\log n}} n)$. □

An example of a partially filled tree could be seen in Figure 2.5.

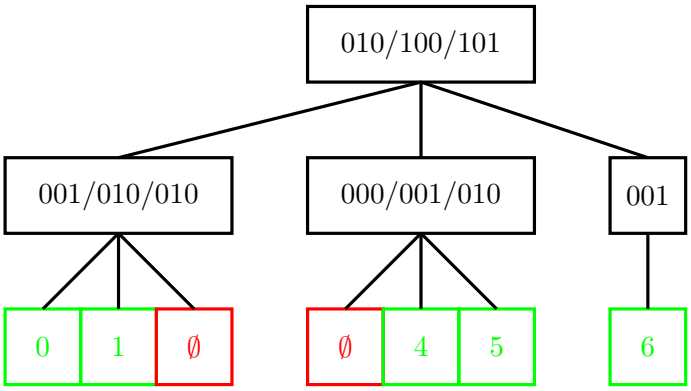

Figure 2.5: Example of tree with $\beta = 3$, 7 nodes, where $0, 1, 4, 5$ and 6 are inserted

Note that the depth could be slightly improved as we observe that size $\lceil \log_2 n \rceil$ bits for buckets is necessary only for *root*. Obviously, the size of buckets of a *root* of a subtree is just $\lceil \log_2 SZ \rceil$, where SZ stands for the size of the subtree. As the branching factor of a node with m elements is $W/\lceil \log m \rceil$, then the size of the child subtree is approximate $\mathcal{O}\left(m \cdot \log m/W\right)$. If we would follow this observation, we could describe the depth by the following recursion:

$$T(m) = T\left(\left\lceil \frac{m \log m}{W} \right\rceil\right) + \mathcal{O}(1), \qquad T(1) = \mathcal{O}(1).$$

Every node will also possess two integers *del* and *add*. Both of the integers use only β bits (β stands for branching factor), while i-th bit of *add* means, that every element of i-th child is in the tree. Likewise, if the i-th bit of *del* is 1, then the i-th subtree is empty. The purpose of these two integers is to enable lazy updates.

2.1.5 Queries

Let us define a simple function BUCK which would return i-th bucket of some integer α (for simplicity the size of buckets will be denoted as κ).

In Algorithm 5 it is described how to obtain the content of i-th bucket.

Algorithm 5 Get the content of i-th bucket

```
1: function BUCK(α, i)
2:     if i < 0 then
3:         return 0
4:     end if
5:     return (α ≫ (κ · i)) & (2^κ − 1)
6: end function
```

If we would travel to any child (in the following subsection), check *del* and *add* first. If any of these bits is set on, do a proper update (which is $\mathcal{O}(1)$ so it will not spoil the complexity) and erase this bit.

The pseudocodes in the following subsection will for simplicity assume the size of each subtree is equal. As it might not be, it is left as an implementation detail.

2.1.5.1 Get

A special case of **get** query is the sum of elements of the whole subtree. This can be easily done by returning the content of the last bucket. Thus the time complexity is $\mathcal{O}(1)$.

This kind of query asks for *sum* of elements on an interval. Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost is in the j-th child:

- If the beginning of the interval is not the leftmost element of the i-th child, then add to the answer the answer of i-th child and increase i by 1.
- If the end of the interval is not the rightmost element of j-th child, then add to the answer the answer of j-th child and decrease j by 1. Do this query only if we did not ask j-th child before (this could have happened if begin and end belong to the same subtree).
- Find the number of elements of all buckets between i and j and add it to answer. This can be found if we subtract content of $(i-1)$-th bucket from the content of j-th bucket (in case of i being the 0-th bucket, we subtract 0 instead). Do this only if $i \leq j$.

This procedure could be seen in Algorithm 6.
A content of i-th bucket of *sum* can be extracted by function BUCK.

Lemma 7. *The complexity of* GET *of Sum-element Tree is* $\mathcal{O}(\log_\beta n)$*, where* β *stands for branching factor and is equal to* $\mathcal{O}\left(\frac{W}{\log n}\right)$.

Proof. As we already shown in section "*Tree structure*", this traversal visits $\mathcal{O}(\log_\beta n)$ nodes. All of the items above could be done in $\mathcal{O}(1)$ time, so the total complexity of this kind of query remains $\mathcal{O}(\log_\beta n)$. □

2.1.5.2 Delete

A special case of *deletion* query is the deletion of the whole subtree. This could be done by setting *sum* to 0, setting *add* to 0 and *del* to $2^\beta - 1$ (where β means the branching factor). As we can observe, the complexity of this operation is $\mathcal{O}(1)$.

Algorithm 6 Get the number of elements on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size$

```
1: function GET(beg, end)
2:     SZ = child[0].size
3:     ans = 0
4:     if beg/SZ == end/SZ then
5:         if (del ≫ (beg/SZ)) & 1 then
6:             return 0
7:         end if
8:         if (add ≫ (beg/SZ)) & 1 then
9:             return end − beg + 1
10:        end if
11:        return child[beg/SZ].get(beg%SZ, end%SZ)
12:        //End of recursion is implementation detail
13:    end if
14:    if beg%SZ > 0 then
15:        if (del ≫ (beg/SZ)) & 1 then
16:
17:        else if (add ≫ (beg/SZ)) & 1 then
18:            ans = ans + SZ − beg%SZ
19:        else
20:            ans = ans + child[beg/SZ].GET(beg%SZ, SZ − 1)
21:        end if
22:        beg = beg + SZ − beg%SZ
23:    end if
24:    if (end + 1)%SZ > 0 then
25:        if (del ≫ (end/SZ)) & 1 then
26:
27:        else if (add ≫ (end/SZ)) & 1 then
28:            ans = ans + end%SZ
29:        else
30:            ans = ans + child[end/SZ].GET(0, end%SZ)
31:        end if
32:        end = end − end%SZ − 1
33:    end if
34:    return ans + BUCK(sum, end/SZ) − BUCK(sum, beg/SZ − 1)
35: end function
```

This kind of query is *deletion*. Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost node is in the j-th child.

- If the beginning of the interval is not the leftmost element of i-th child, update i-th child and increase i by 1.
- If the end of the interval is not the rightmost element of j-th child, then update the j-th child and decrease j by 1. Update j-th child only if we have not done this already (this could have happened if $i = j$).
- Let τ be the sum of all elements in nodes in between i-th and j-th subtrees (inclusive). This can be found by subtracting content of $(i-1)$-th bucket from j-th bucket. As we will delete these elements, we surely have to erase τ from all buckets after j—which could be done by copying τ (multiplying it by proper constant, which has 1 in every bucket), shifting it by $(j+1) \cdot SZ$ (where SZ stands for the size of first subtree) and subtract it from *sum*. The erase all buckets between i and j (for example by calling ERASE). Then copy the contents of $(i-1)$-th bucket into all buckets between i and j (0 in case of i being the 0-th bucket). Also call ERASE(add, i, j) and set all bits between i and j of *del* to 1: $del = del \mid ((2^{j-i+1} - 1) \ll i)$.

This procedure could be seen in Algorithm 7.

The first step in updating i-th subtree is to subtract the number of elements in i-th subtree from every bucket beyond i (inclusive). Then call update on the proper subtree. The last step is to add the number of elements of i-th subtree to every bucket beyond i (inclusive).

Lemma 8. *The complexity of deletion query of Sum-element Tree is* $\mathcal{O}(\log_\beta n)$*, where* β *stands for branching factor and is equal to* $\mathcal{O}\left(\frac{W}{\log n}\right)$.

Proof. All operations in the node are $\mathcal{O}(1)$ and the number of visited nodes is at most $\mathcal{O}(\log_\beta n)$, so the total complexity of this operation is $\mathcal{O}(\log_\beta n)$. □

2.1.5.3 Insert

Special case of **insert** query is insertion of whole subtree. This could be done by setting *del* to 0, setting *add* to $2^\beta - 1$ and setting *sum* to $SZ \cdot c$ (here SZ stands for size of first subtree). Here c stands for constant, which has $i+1$ in i-th bucket. Note that the last bucket might be done possibly separately. As we can observe, the complexity of this operation is $\mathcal{O}(1)$.

This kind of query is *insertion*. Before we will run the query itself, run *deletion* on the same segment (we will avoid reinserting an element for the second time). Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost node is in the j-th child.

Algorithm 7 Delete all the elements on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size$

1: **function** DEL(beg, end)
2: $SZ = child[0].size$
3: **if** $beg/SZ == end/SZ$ **then**
4: **if** $(del \gg (beg/SZ)) \,\&\, 1$ **then**
5: **return**
6: **end if**
7: **if** $(add \gg (beg/SZ)) \,\&\, 1$ **then**
8: $add = add \oplus 2^{beg/SZ}$
9: $child[beg/SZ]$.ADD$(0, child[beg/SZ].size - 1)$,update sum
10: **end if**
11: $child[beg/SZ]$.DEL$(beg\%SZ, end\%SZ)$,update sum
12: **return**
13: //End of recursion is implementation detail (perhaps stop when covering whole subtree)
14: **end if**
15: **if** $beg\%SZ > 0$ **then**
16: **if** $(del \gg (beg/SZ)) \,\&\, 1$ **then**
17: break from all ifs
18: **end if**
19: **if** $(add \gg (beg/SZ)) \,\&\, 1$ **then**
20: $child[beg/SZ]$.INSERT$(0, child[beg/SZ].size - 1)$,update sum
21: $add = add \oplus 2^{beg/SZ}$
22: **end if**
23: $child[beg/SZ]$.DEL$(beg\%SZ, SZ - 1)$,update sum
24: $beg = beg + SZ - beg\%SZ$
25: **end if**
26: **if** $(end + 1)\%SZ > 0$ **then**
27: **if** $(del \gg (end/SZ)) \,\&\, 1$ **then**
28: break from all ifs
29: **else if** $(add \gg (end/SZ)) \,\&\, 1$ **then**
30: $child[end/SZ]$.INSERT$(0, child[end/SZ].size - 1)$,update sum
31: $add = add \oplus 2^{end/SZ}$
32: **end if**
33: $child[end/SZ]$.DEL$(0, end\%ST)$,update sum
34: $end = end - end\%SZ - 1$
35: **end if**
36: ERASE$(add, beg/SZ, end/SZ)$
37: $del = del \mid (2^{end/SZ - beg/SZ + 1} - 1)$
38: REPLICATE BUCK$(sum, end/SZ)$ − BUCK$(sum, beg/SZ - 1)$ and subtract it (shifted by $end + 1$) from all suffix buckets.
39: ERASE(sum, beg, end)
40: REPLICATE BUCK$(sum, beg/sz - 1)$, SHIFT it by beg and OR it to sum
41: **end function**

- If the beginning of the interval is not the leftmost element of i-th child, update i-th child and increase i by 1.
- If the end of the interval is not the rightmost element of j-th child, then update the j-th child and decrease j by 1. Update j-th child only if we have not done this already (this could have happened if $i = j$).
- We need to update the *del* by deleting elements between i and j (for example by calling ERASE). Then we set all the bits of *add* to 1—for example by following formula: $add = add \mid ((2^{j-i+1} - 1) \ll i)$. We also have to update buckets of *sum*: Firstly we have to add $(k + 1) \cdot SZ$ to all $(i + k)$-th buckets, for all $i + k$ between i and j (inclusive). This could be done by multiplying SZ with a constant which has $i+1$ in i-th bucket, *erasing* all bits beyond first $j-i+1$ and properly shifting them. We also have to add $(j - i + 1) \cdot SZ$ to all buckets beyond j (exclusive), which could be done by proper copy and shift.

The described procedure could be seen in Algorithm 8.

As we have already deleted all proper elements, the update of i-th subtree can be done by calling update for the proper subtree and then adding the number of added elements to all buckets after i (i inclusive). This can be as usually done by *copying* and adding.

Lemma 9. *The complexity of insertion of Sum-element Tree is* $\mathcal{O}(\log_\beta n)$*, where* β *stands for branching factor and is equal to* $\mathcal{O}\left(\frac{W}{\log n}\right)$.

Proof. The number of visited nodes of **insertion** is $\mathcal{O}(\log_\beta n)$. As we can see, the complexity in each node is $\mathcal{O}(1)$. We also have to count the complexity of **deletion** which is also $\mathcal{O}(\log_\beta n)$ (so it will not spoil the complexity). □

2.2 Sum-values

2.2.1 Introduction

The most general wording of this problem is that every element has assigned a value, which can be either increased or decreased. We can easily observe that this problem is very similar to the previous problem and can be converted with little effort. Yet not that this would have one hole in it—the size of the bucket cannot be properly detected with such definition (unless done dynamically).

One problem derived from above is providing the answer modulo some 2^κ. This immediately gives us a lower bound onto the size of buckets, which is κ—but for our purposes, we will use 2κ.

It is a convention to initialize the structure so that every element's values is 0 at the beginning.

Algorithm 8 Insert all the elements on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size$

1: **function** INSERT(beg, end)
2: $SZ = child[0].size$
3: **if** $beg/SZ == end/SZ$ **then**
4: **if** $(add \gg (beg/SZ)) \,\&\, 1$ **then**
5: **return**
6: **end if**
7: **if** $(del \gg (beg/SZ)) \,\&\, 1$ **then**
8: $del = del \oplus 2^{beg/SZ}$
9: $child[beg/SZ]$.DEL$(0, child[beg/SZ].size - 1)$,update sum
10: **end if**
11: $child[beg/SZ]$.INSERT$(beg\%SZ, end\%SZ)$,update sum
12: **return**
13: //End of recursion is implementation detail (perhaps when covering whole subtree)
14: **end if**
15: **if** $beg\%SZ > 0$ **then**
16: **if** $(add \gg (beg/SZ)) \,\&\, 1$ **then**
17: break from all ifs
18: **end if**
19: **if** $(del \gg (beg/SZ)) \,\&\, 1$ **then**
20: $child[beg/SZ]$.DEL$(0, child[beg/SZ].size - 1)$,update sum
21: $del = del \oplus 2^{beg/SZ}$
22: **end if**
23: $child[beg/SZ]$.INSERT$(beg\%SZ, SZ - 1)$,update sum
24: $beg = beg + SZ - beg\%SZ$
25: **end if**
26: **if** $(end + 1)\%SZ > 0$ **then**
27: **if** $(add \gg (end/SZ)) \,\&\, 1$ **then**
28: break from all ifs
29: **else if** $(del \gg (end/SZ)) \,\&\, 1$ **then**
30: $child[end/SZ]$.DEL$(0, child[end/SZ].size - 1)$,update sum
31: $del = del \oplus 2^{end/SZ}$
32: **end if**
33: $child[end/SZ]$.INSERT$(0, end\%ST)$,update sum
34: $end = end - end\%SZ - 1$
35: **end if**
36: ERASE$(del, beg/SZ, end/SZ)$
37: multiply SZ by proper constant, SHIFT it by beg and OR it to sum
38: REPLICATE BUCK$(sum, beg/SZ - 1)$ and add it to sum (shifted by beg)
39: REPLICATE BUCK$(sum, end/SZ)$ − BUCK$(sum, beg/SZ - 1)$ and add it (shifted by $end + 1$) to all suffix buckets.
40: **end function**

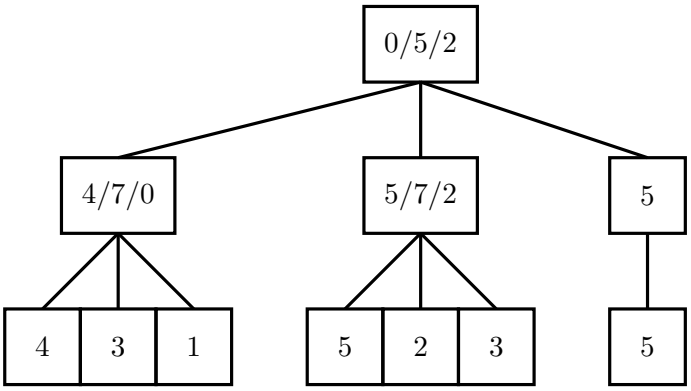

Figure 2.6: Example of *sum* of tree with β=3, 7 nodes, and κ=3 (shown in decimal)

2.2.2 Sum-values Tree

Operation	**Complexity**
SUM	$\mathcal{O}(\log_{W/\kappa} n)$
ADD	$\mathcal{O}(\log_{W/\kappa} n)$

Table 2.2: Complexities of operations of *Sum-values Tree*

The complexities in Table 2.2 coulde be obtained from Lemmas 10 and 11.

For our purposes, we would need branching factor to be at most $\mathcal{O}(\sqrt{W})$.

Note that the structure of this tree is very similar to the previous data structure. It is also a *β-ary Tree* (here β is equal to $\lfloor \frac{W}{2\kappa} \rfloor$). This means the depth of such structure will be $\mathcal{O}(\log_\beta n)$, where n stands for the size of the universe.

In every node, there is alike in the previous data structure, an integer *sum*. The i-th bucket in *sum* means the sum of values of elements from all subtrees between 0 and i (inclusive) modulo 2^κ. Another integer we will need is *add*. This integer is similarly divided into buckets of size 2κ. The content of i-th bucket means the sum of values between 0-th and i-th buckets (inclusive), which were not added to the proper subtrees. Another integer will be *addOne*, which will be also chopped into buckets of size 2κ and content of i-th bucket means that this value was added to every element in the subtree, but it was not propagated.

The example of a tree could be found in Figure 2.6.

2.2.3 Queries

2.2.3.1 Sum

Definition: The first kind of query asks for the sum of values of elements on an interval. Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost is in the j-th child:

- If the beginning of the interval is not the leftmost element of i-th child, then add to the answer the answer of i-th child modulo 2^κ. Also, add the number of elements of the segment in i-th child multiplied by the content of i-th bucket of *addOne* modulo 2^κ. Then increase i by 1.
- If the end of the interval is not the rightmost element of j-th child, then add to the answer the answer of j-th child modulo 2^κ. Also, add the number of elements of the segment in j-th child multiplied by the content of j-th bucket of *addOne* modulo 2^κ. Then decrease j by 1. Do this procedure only if we did not ask j-th child before (this could have happened if $i = j$).
- To find the sum of elements on the rest of the segment, subtract $(i-1)$-th bucket of *sum* from j-th bucket modulo 2^κ. Also add the subtraction of $(i-1)$-th bucket of *add* from j-th bucket modulo 2^κ. Note that if we subtracting two numbers in modulo, we must ensure the answer to be positive. This can be for example achieved by adding 2^κ before using a modulo operation.

This procedure is described by Algorithm 9.

Lemma 10. *The complexity of* SUM *of Sum-values Tree is* $\mathcal{O}(\log_\beta n)$*, where* β *stands for branching factor and is equal to* $\mathcal{O}\left(\frac{W}{\kappa}\right)$.

Proof. The operations in the node have $\mathcal{O}(1)$ complexity and the number of visited nodes is $\mathcal{O}(\log_\beta n)$, so the total complexity is $\mathcal{O}(\log_\beta n)$ too. □

2.2.3.2 Add

The second kind of query adds (modulo 2^κ) some value χ to value of every node on an interval. Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost is in the j-th child:

- If the beginning of the interval is not the leftmost element of i-th child, update the i-th child.
- If the end of the interval is not the rightmost element of j-th child, update the j-th child. Do this procedure only if we did not update j-th child before (this could have happened if $i = j$).

Algorithm 9 Sums the values of elements on interval modulo 2^κ

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size$

```
1: function GET(beg, end)
2:     SZ = child[0].size
3:     ans = 0
4:     if beg/SZ == end/SZ then
5:         return (child[beg/SZ].GET(beg%SZ, end%SZ) + buck(addOne, beg/SZ) · (end − beg + 1))%2^κ
6:         //End of recursion is implementation detail
7:     end if
8:     if beg%SZ > 0 then
9:         ans = ans + child[beg/SZ].GET(beg%SZ, SZ − 1)
10:        ans = ans + BUCK(addOne, beg/SZ) · (SZ − beg%SZ)
11:        beg = beg + SZ − beg%SZ
12:    end if
13:    if (end + 1)%SZ > 0 then
14:        an = ans + child[end/SZ].GET(0, end%ST)
15:        ans = ans + BUCK(addOne, end/SZ) · (end%SZ)
16:        end = end − end%SZ − 1
17:    end if
18:    ans = ans + BUCK(sum, end/SZ) − BUCK(sum, beg/SZ − 1) + 2^κ
19:    ans = ans + BUCK(add, end/SZ) − BUCK(add, beg/SZ − 1) + 2^κ
20:    return ans%2^κ
21: end function
```

- Now we need to update *add*. Firstly, we have to find out the total sum of values added to each subtree in modulo, which could be done as follows: $tot = (SZ \cdot \chi) \mod 2^\kappa$ (where SZ stands for the size of the subtree of first child). Then copy this value $i + 1$ times to i-th bucket—this could be done by multiplication with a constant c which has $i + 1$ in each bucket (obviously cut it to first $j - i + 1$ bucket only, which could be done for example by ANDing it with $2^{2 \cdot \kappa \cdot (j-i+1)} - 1$). Shift this value by $2 \cdot \kappa \cdot i$ and add it to *sum*. Now modulo every bucket. This could be done by copying $2^\kappa - 1$ into each bucket and ANDing it with *sum*.

- We also have to update *addOne*. This could be simply done by copying value χ $(j - i + 1)$-times, shifting it, adding it to *addOne* and proceed modulo by ANDing it with the same value as in the previous procedure.

This procedure is also described by Algorithm 10.

If we update i-th bucket, we also have to take care of *sum*. Firstly we have to subtract the sum of values of i-th subtree, then update and afterward add it. The process is similar to previous processes (copying, modulo,...).

Anyway, note that there are some obstacles. Firstly we have to obtain the sum of the whole subtree. This could be luckily done very easily by summing the last bucket of *sum* and the last bucket of *add* of the i-th child modulo 2^κ. Another problem is subtraction, since overflowing a bucket could ruin our structure (keep on mind that buckets have to be independent). This could be solved by adding 2^κ to every bucket before subtracting.

Lemma 11. *The complexity of* ADD *query of Sum-values Tree is* $\mathcal{O}(\log_\beta n)$*, where* β *stands for branching factor and is equal to* $\mathcal{O}\left(\frac{W}{\kappa}\right)$.

Proof. As we can see, every operation in a node is $\mathcal{O}(1)$ and the number of visited nodes is $\mathcal{O}(\log_\beta n)$, so the total complexity of this process is $\mathcal{O}(\log_\beta n)$. □

Algorithm 10 Adds χ to all elements on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size \wedge \chi \geq 0 \wedge \chi < 2^\kappa$

1: **function** ADD(beg, end, χ)
2: $SZ = child[0].size$
3: **if** $beg/SZ == end/SZ$ **then**
4: $child[beg/SZ]$.ADD($beg\%SZ, end\%SZ, \chi$)
5: $sum = sum +$ REPLICATE$(\chi \cdot (end - beg + 1)\%2^\kappa) \ll beg/SZ \cdot 2\kappa$
6: $sum = sum\&$REPLICATE$(2^\kappa - 1)$
7: **return**
8: //End of recursion is implementation detail
9: **end if**
10: **if** $beg\%SZ > 0$ **then**
11: $child[beg/SZ]$.ADD$(beg\%SZ, SZ - 1, \chi)$
12: $sum = sum +$ REPLICATE$(\chi \cdot (SZ - beg\%SZ)\%2^\kappa) \ll beg/SZ \cdot 2\kappa$
13: $sum = sum\&$REPLICATE$(2^\kappa - 1)$
14: $beg = beg + SZ - beg\%SZ$
15: **end if**
16: **if** $(end + 1)\%SZ > 0$ **then**
17: $child[end/SZ]$.ADD$(0, end\%SZ, \chi)$
18: $sum = sum +$ REPLICATE$(\chi \cdot (end\%SZ)\%2^\kappa) \ll end/SZ \cdot 2\kappa$
19: $sum = sum\&$REPLICATE$(2^\kappa - 1)$
20: $end = end - end\%SZ - 1$
21: **end if**
22: $addOne = addOne +$ (REPLICATE(χ) & $(2^{2\kappa \cdot (end/SZ - beg/SZ + 1)})) \ll (beg/SZ \cdot 2\kappa)$
23: $addOne\,\&\, =$ REPLICATE$(2^\kappa - 1)$
24: $add = add +$ $((\chi \cdot SZ\%2^\kappa \cdot \rho)\,\&\,(2^{2\kappa(end/SZ - beg/SZ + 1)} - 1)) \ll (beg/SZ \cdot 2\kappa)$
25: $add = add +$ REPLICATE$((end - beg + 1) \cdot \chi\%2^\kappa) \ll ((end/SZ + 1) \cdot 2\kappa)$
26: $add\,\&\, =$ REPLICATE$(2^\kappa - 1)$
27: **end function**

2.3 Max-value

2.3.1 Introduction

Definition 10. *In this problem we have n sequentially ordered elements—initially set to 0. We have two kinds of queries:*

- *We ask for **maximum** element on some interval.*
- *Set value of i-th element to χ.*

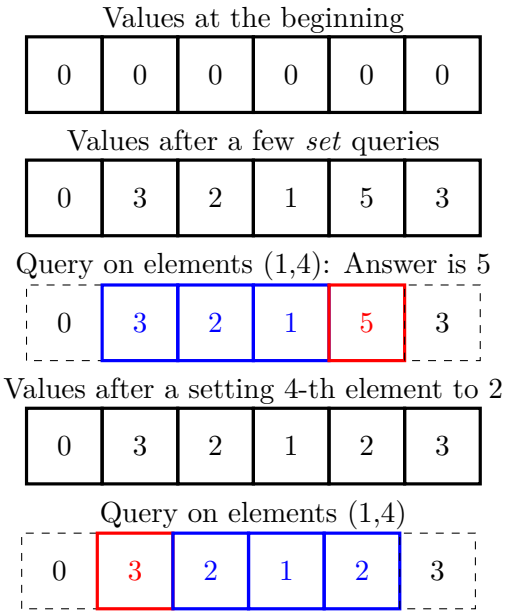

Figure 2.7: Example of max-value problem

Definition 11. *We will restrict the values of χ on integers from range 0 to $2^{W^{1/3}-1}$. We can obviously increase the upper bound by some constant if we will chain a few words together, so $\mathcal{O}(2^{W^{1/3}})$ would probably fit better.*

Similar problem **min-value** could be trivially derived from this problem. To bend the structure to solve this problem instead, we simply have to insert $\mu - \chi$ (here μ stands for maximal possible value) instead of χ, and also let query number **1** return $\mu - ANS$ instead.

This structure can support any range of numbers which is properly long—one just has to properly shift the numbers before insertion or after harvesting the answer.

In this problem, we return for simplicity the value of the maximum element. A very similar problem might ask for index instead but that would be a little more complicated.

A small example of this problem is described in Figure 2.7.

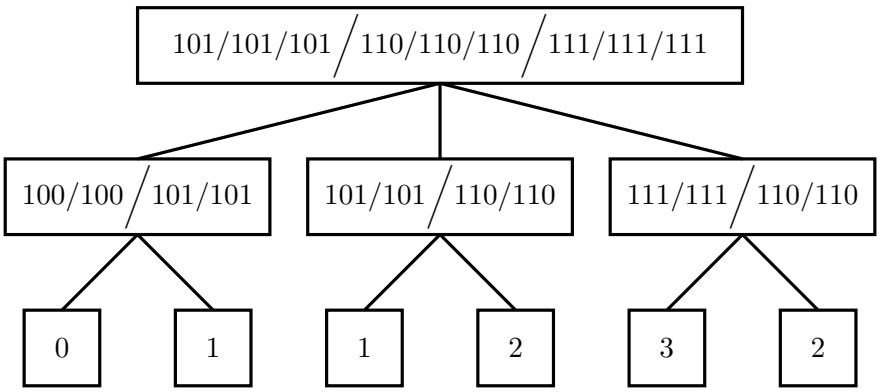

Figure 2.8: Example of tree with R=3, 6 nodes: The variable q

2.3.2 Max-value Tree

Operation	Complexity
MAX	$\mathcal{O}(\log_W n)$
SET	$\mathcal{O}(\log_W n)$

Table 2.3: Complexities of operations of *Max-value Tree*

The complexities in Table 2.3 coulde be obtained from Lemmas 14 and 15.

This data structure will also have the form of *β-ary Tree*. Here our branching factor will be r, which will be set as some power of 2 which is approximately $\mathcal{O}(W^{\frac{1}{3}})$. The traversal style will be similar to previous data structures—the only difference will be in the processing of nodes.

Lemma 12. *The depth of Max-value Tree is $\mathcal{O}(\log_W n)$.*

Proof. As the branching factor is $\mathcal{O}(W^{\frac{1}{3}})$, the depth is going to be $\mathcal{O}(\log_{W^{\frac{1}{3}}} n)$. Note that as it is in asymptote, so the complexity is equal to $\mathcal{O}(\log_W n)$. □

Definition 12. *Every node will posses an integer Q. This integer will be divided into R buckets (of size r^2, which is $\mathcal{O}(W^{\frac{2}{3}})$). Each of those buckets will be also divided into r **mini-buckets**, each of size r. The first bit of every mini-bucket will always be 1. The remaining $r-1$ bits of each mini-bucket will consist of **maximum** of i-th subtree, where i is the number of the bucket in which the mini-bucket is.*

In the example, the buckets in the middle layer have only 2 mini-buckets. Even though it is an implementation detail, in fact, it would be best to have 3^{rd} mini-buckets, filled with 0 (except for the first bit, which is 1)

Definition 13. *Every node will also have an integer h. This integer will consist of one bucket with r mini-buckets of size r (this means we only need $\mathcal{O}(W^{\frac{2}{3}})$ bits for it). The first bit of every mini-bucket will be* 0*. All other bits of i-th mini-bucket will be filled with the maximum of i-th subtree.*

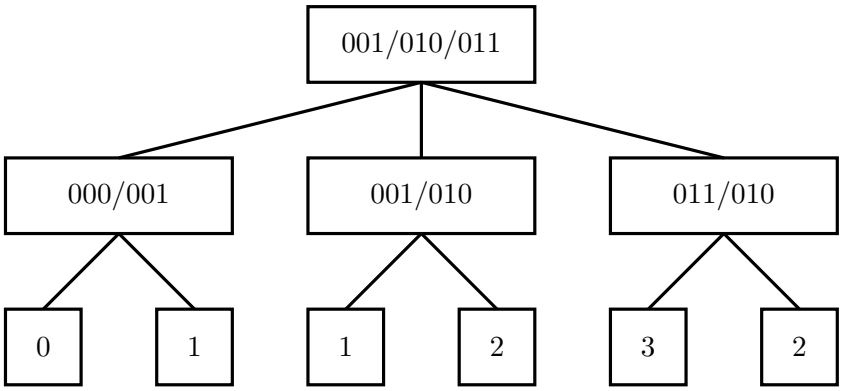

Figure 2.9: Example of tree with r=3, 6 nodes: The variable h

The examples of trees and their variables q and h could be seen in following Figures 2.8 and 2.9.

Definition 14. *Let us have constant c, which has* 1 *as the last bit of every bucket. This means it is in form of $(0^{r^2-1}1)^r$, where power means concatenation.*

Definition 15. *Let us define a constant k, which has* 1 *at the end of each mini-bucket and consists of r^2 bits (the rest might be for example filled with zeroes). This means it is in form of $(0^{r-1}1)^r$, where power means concatenation.*

2.3.3 Queries

2.3.3.1 Maximum of the whole subtree

This query will find the maximum of the whole subtree without traveling to any of the children.

- As the first step, we do $q-h\cdot c$. This is a very simple formula yet it is crucial to realize the impact of such a formula. Firstly note that $h\cdot c$ copies h to each bucket. Afterward, there will be r^2 parallel comparisons—one in every mini-bucket. Since every number in mini-buckets of q is greater than those in $h\cdot c$ (notice the first bit), the comparisons are independent.
- Observe, that the first bit of every mini-bucket will be 1 as long as the remaining bits of the mini-bucket of q will be greater or equal to the

content of mini-bucket of $h \cdot c$. We will care only for the first bit of every mini bucket. To get rid of the *garbage*, we could shift the result by $r - 1$ (so that the 1's move to the last bit of the mini-buckets) and AND it with $k \cdot c$. Another simple note is that the bucket(s) with the most ones in it has the maximum of this subtree in it.

- In the next step, we multiply the result by k. Again, this step is just a simple formula yet it is important to understand what happens here. For each mini-bucket, we shift the integer α by r and add it:

$$\sum_{n=0}^{r} \alpha \ll r$$

  As we might observe, in every mini-bucket, there appears the number of 1 which are in the previous r mini-buckets (actual mini-bucket inclusive). This implies that the leftmost mini-bucket of each bucket will contain the number of ones in the bucket. Note that each mini-bucket has size r and we only need $\lceil \log_2 r \rceil$ bits to represent the sum, so it certainly cannot overflow.

- It might look like we are in the same situation as at the beginning since we have to detect the bucket with the biggest number in it (which is in the last mini-bucket). Anyway as we might observe, we know that the biggest number will be equal to r, which is a power of 2 (as stated in the previous subsection). This means that as long as r is let us say in form of 2^b, we know that the b-th bit of the last mini-bucket is 1. We can AND the result with c shifted by $r^2 + b$.

- Now any bucket with a 1 in it has also the maximum in it. The last step is to detect **any** position of one-bit, divide the number by r^2, and this will identify the desired bucket. As we have the desired bucket, we can obtain the maximum from q or h.

- Note that position of the random bit could be done for example by MSB [Most Significant Bit] algorithm or LSB [Least Significant Bit] algorithm—both of them can be executed in constant time. Note that LSB presented by us in the *Bit Tricks* section would need $\mathcal{O}(W^2)$ bits, yet we could overcome this problem by calling PACK first, by which it will compress to $\mathcal{O}(W^{\frac{1}{3}})$ bits only.

This could be also found in Algorithm 11.

Lemma 13. *The complexity of* MAX *query, asking for whole tree of Max-value Tree is* $\mathcal{O}(1)$.

Proof. Each step of the procedure takes constant time and we do not have to visit any other nodes, so the total complexity is $\mathcal{O}(1)$. □

Algorithm 11 Finds the maximum from values of the whole subtree

```
1: function MAX
2:     tmp = (q − h · c) · k
3:     tmp &= c ≪ (r^2 + b)
4:     return buck_mini(h, LSB(tmp)/r)
5: end function
```

2.3.3.2 Maximum on segment

This kind of query will find the biggest value of elements from a segment. Imagine we are in a node and we detect the leftmost node of our query is in the i-th child and the rightmost is in the j-th child:

- If the beginning of the interval is not the leftmost element of i-th child, then let one of the potential maximum candidates be the answer of i-th subtree. Then increase i by 1.

- If the end of the interval is not the rightmost element of j-th child, then let one of the potential maximum candidates be the answer of j-th subtree. Then increase j by 1. Do this procedure only if we did not ask j-th child before (this could have happened if $i = j$).

- For the rest of the segment we might proceed in the same way in subsection "**Maximum of the whole subtree**", with the exception we have to properly adjust h and q. To do this, we have to get rid of all buckets/mini-buckets which are not included between i and j. As they are consecutive, we can simply use ERASE to do so. In case of q we also have to OR it with shifted $k \cdot c$ so we will not break our rule with 1 at the beginning.

The described algorithm could be also found in Algorithm 12.

Lemma 14. *The complexity of maximum on segment of Max-value Tree is* $\mathcal{O}(\log_W n)$.

Proof. The complexity in each node is $\mathcal{O}(1)$ and we will visit at most $\mathcal{O}(\log_W n)$ nodes so the complexity is equal to $\mathcal{O}(\log_W n)$. □

Algorithm 12 Finds the maximal value of elements on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size$

```
 1: function MAX(beg, end)
 2:     SZ = child[0].size
 3:     ans = −∞
 4:     if beg/SZ == end/SZ then
 5:         return child[beg/SZ].MAX(beg%SZ, end%SZ)
 6:         //End of recursion is implementation detail
 7:     end if
 8:     if beg%SZ > 0 then
 9:         ans = child[beg/SZ].MAX(beg%SZ, SZ − 1)
10:         beg = beg + SZ − beg%SZ
11:     end if
12:     if (end + 1)%SZ > 0 then
13:         ans = max(ans, child[end/SZ].MAX(0, end%SZ))
14:         end = end − end%SZ − 1
15:     end if
16:     backup variables
17:     ERASE(h, 0, beg/SZ · r − 1)
18:     ERASE(h, (end/SZ + 1) · r, W − 1)
19:     ERASE(q, 0, beg/SZ · r^2 − 1)
20:     ERASE(q, (end/SZ + 1) · r^2, W − 1)
21:     ans = max(ans, MAX())
22:     use backup
23:     return ans
24: end function
```

2.3.3.3 Update

This kind of query will set the value of any element to some value χ. It must be in the proper range between 0 and 2^{r-1}.

Definition 16. *Let us define m as the maximum of the subtree in which we updated the value.*

We would like to properly adjust variable Q. Firstly, we would like to erase all bits of i-th bucket (for example with the usage of function ERASE). Then take m and prepend a 1 bit before it (could be done by ORing with 2^{r-1}). Next step is to copy this r times (to $r-1$ following mini-buckets), which could be done by multiplying it with k. The last step is to stick this value to q, which might be done by shifting this by $r^2 \cdot i$ and ORing it to q (note that addition would work too since the i-th bucket shall be empty at this moment).

To update h, we first have to erase the i-th mini-bucket (which could be done for example by ERASE). Then we simply add m shifted by $r \cdot i$ to it.

These described operations of update could be also found in Algorithm 13.

Lemma 15. *The complexity of update of Max-value Tree is* $\mathcal{O}(\log_W n)$.

Proof. If the navigation is done correctly, we only need to visit the leaf-node itself plus all of its ancestors. This means we will visit $\mathcal{O}(\log_W n)$ nodes. □

Note that the update has to be done only if the m really changed its value. Even though it does not improve the asymptotical complexity, it might lighten the number of conducted operations in an average case.

Algorithm 13 Sets i-th element to χ

Require: $i \geq 0 \wedge i < size \wedge \chi \geq 0 \wedge \chi < 2^{r-1}$

```
1: function SET(i, χ)
2:     if size == 1 then
3:         return
4:     end if
5:     SZ = child[0].size
6:     child[i/SZ].SET(i%SZ, χ)
7:     m = child[i/SZ].MAX()
8:     //here we could end if maximum have not changed
9:     ERASE(q, i/SZ · r², (i/SZ + 1) · r²)
10:    q | = (REPLICATE(m | 2^(r−1)) & (2^(r²) − 1)) ≪ (i/SZ · r²)
11:    ERASE(h, i/SZ · r, (i/SZ + 1) · r)
12:    h = h|(m ≪ (i/SZ · r))
13: end function
```

2.4 Persistence

2.4.1 Intro

Definition 17. *The structures describe above are all* ***ephemeral****. That means that every modification "destroys" the last version of the data structure and creates a new one so it is always possible to access the last version only.*

Definition 18. *Unlike ephemeral data structures,* ***persistent*** *[10] data structure could access all previously modified versions and possibly modify them. There are multiple levels of persistence. The aim of such a structure is to minimize its space and time complexities. A version of such persistent data structure is uniquely identifying the stat of the data structure at the given moment.*

Definition 19. *In the* ***partial persistence*** *[11] we may query any previous version of the data structure, but we may only update the latest version. This implies a linear ordering among the versions.*

Definition 20. *In the* ***fully persistent*** *model, both queries and updates are available to previous versions of the data structure.*

Definition 21. *In **confluently persistent** model, we combine the input of multiple previous versions to output a new version. These combinations makes the model a **DAG** [Directed Acyclic Graph] by its structure.*

2.4.2 Partial Persistence

It was shown by Brodal [12] that it is possible to make a data structure persistent with $\mathcal{O}(\iota + \sigma)$ overhead per updated node, where ι stands for in-degree and σ stands for out-degree of the node. Even though ι is constant for our data structures σ is not.

A suitable strategy for achieving partial persistence is **path copying** [13]. The strategy is simple:

- As we would modify a node, we allocate a new node instead which will be the copy of our node and modify this node instead. Then we return the *pointer* of the modified node to its parent and then we will modify the parent (its values and pointers to modified children).
- As we return the pointer of the root (note that root will be always modified) we store it in some time-array (or some structure with random access—yet note that array is perfectly suitable). Now we can observe that we can easily access the data structure at any time (by looking into our time-array). We can also observe that as long as we would need to access some nodes which were not modified in desired time, it will simply jump into *older* version of data structure and count the result from it.

Lemma 16. *The complexity of one update of persistent tree will be $\mathcal{O}(prev + depth \cdot \sigma)$, where prev stands for previous update complexity.*

Proof. The number of modified nodes will be the same as the number of nodes accessed and the time complexity of the update will be $\mathcal{O}(prev + depth \cdot \sigma)$. The complexity of getting segment answers will remain the same. As we can see, we can slightly balance the complexity by making a lesser/greater number of children. □

Note that even though the update of a node is $\mathcal{O}(1)$ the copying of a node is $\mathcal{O}(\sigma)$ since it has to hold $\mathcal{O}(\sigma)$ pointers to its children. Even though some of the structures previously described could do its job without pointers (by some arithmetic operation—even though it is not necessary). Here we need some kind of pointers to children because we have to change them during updates.

Obviously, we can make some improvement if we will be able to make some analysis. If we could estimate some u, such that the number of updates will be lesser then u we know we will need only $\mathcal{O}(n + u \cdot depth)$ nodes. This also means that the pointers will range between 0 and $\mathcal{O}(n+u \cdot depth)$ which means

we could represent a pointer by $\mathcal{O}(\log(n + u \cdot depth))$ bits. So in the end, we could end with $\delta = \mathcal{O}\left(\frac{\sigma \cdot \log_2(n+u \cdot depth)}{W}\right)$ integers in each node. This could be achieved by having an array of δ integers, splitting them into σ buckets. Each bucket will be a pointer to a child. We can simply access it in $\mathcal{O}(1)$ by making some ANDs and SHIFTs.

Example 4. *In Figure 2.10 is an example of tree structure after updating node number 7 and node number 9:*

2.4.3 Another Approach

There is also a similar approach which has very very small overhead on both: query and update. It has pretty much in common with the previous approach. It is also path copying and the only different behavior is operating with an array of links to children.

Unlike previous structure, here we will only have a pointer to the array of children. Obviously, copying of such pointer costs $\mathcal{O}(1)$ which is great. The problem is in updating the structure. We can use *persistent arrays* proposed by Milan Straka [14] which could access the array in arbitrary time. Here we simply update the proper children which would be the only change.

Straka's persistent arrays achieve the complexity of $\mathcal{O}(\log\log(min(\sigma, q)))$, where σ stands for the size of the array, which is the number of children and q stands for the number of updates. Updates for this approach seems to be better (at least if W is not enormously large) yet the $\mathcal{O}(\log\log(min(\sigma, q)))$ factor will show up in access queries too (note that this factor is very small—despite that it might be still undesired).

2.5 k-th element of segment

2.5.1 Introduction

Definition 22. *In this problem we will get static array with some values. Then we will have only one kind of queries, asking for k-th element on some range [b,e] (**b**egin → **e**nd), where* $0 \leq k < e - b + 1$ *(so* 0*-th element is the lowest one).*

Operation	**Complexity**
INITIALIZATION	$\mathcal{O}\left(n \log n + n \log_{W/\log n}(n) \frac{\log(n \log_{W/\log n} n)}{\log n}\right)$
KTH	$\mathcal{O}(\log n \log_{W/\log n} n)$

Table 2.4: Complexities of operations of *Max-value Tree*

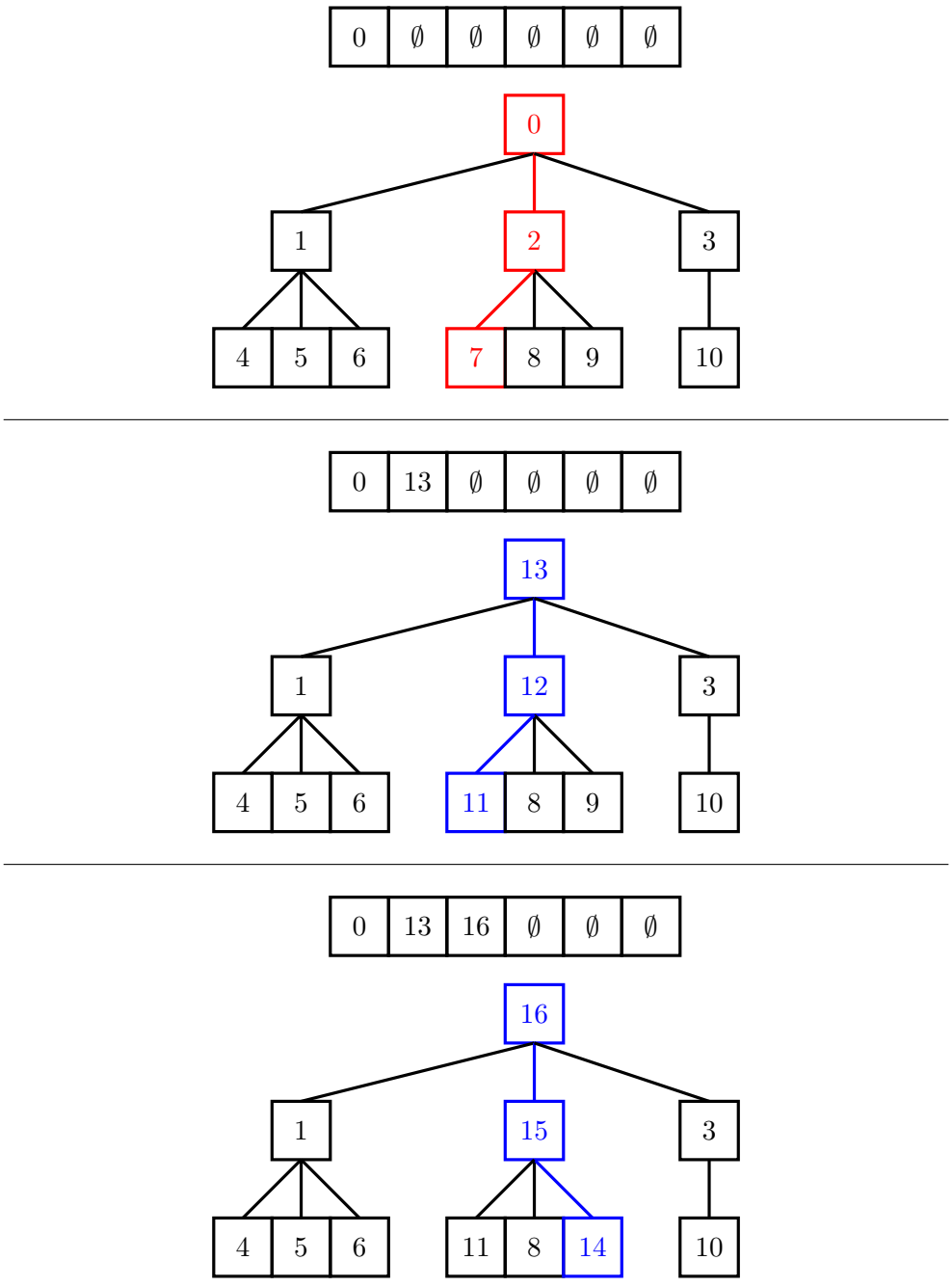

Figure 2.10: Example of path copying, while updating nodes number 7 and 9

The complexities in Table 2.4 coulde be obtained from Lemmas 17 and 18.

This problem might also have a dynamic version which we will not discuss here.

For further processing, we will use the persistent version of our **sum-elements** structure only as a blackbox.

Example 5. *In Figure 2.11 is an example of searching k-th element on segment from 1 to 4 (indexed from 0), in array of $\{1, 2, 5, 4, 3, 6\}$, while $k = 2$.*

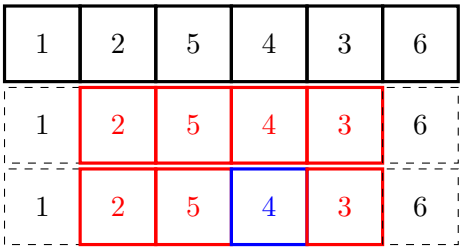

Figure 2.11: Example of k-th element problem

2.5.2 Initialization

The input for initialization is an array α (of length n) full of some values. The initialization will work in following steps:

- We transform such array into an array of pairs where the first element is the value and the second one is the index into the array.
- The next step is to sort such array. To make it a little easier, we simply omit the index (it still has to be there yet we do not care what is the order of elements with the same value).
- The third step is to insert the elements into our data structure. We do this in order from left to right according to our sorted array. We insert element on the position of the second value of the pair (we do not care about the original value of the element).

Lemma 17. *The complexity of initialization for k-th element problem will be:*

$$\mathcal{O}\left(n \log n + n \log_\beta(n) \frac{\log(n \log_\beta n)}{\log n}\right)$$

.

Proof. To make the analysis simple, we will do so with the structure before branching factor improvement. As we can see, there will be exactly n queries and each of them will affect $\mathcal{O}(\log_\beta n)$ (where β stands for branching factor and is equal to $\mathcal{O}(W/\log n)$) nodes (note that the number of affected nodes is almost accurate since it will always be just path from the root to leaf—nothing else). For each such affected node the biggest overhead will be in making a new copy and so the complexity will reach

$$\mathcal{O}\left(\frac{\beta \cdot \log(n \log_\beta n)}{W}\right),$$

which shall be

$$\mathcal{O}\left(\frac{\log(n \log_\beta n}{\log n}\right).$$

So in the end the total complexity of initialization will reach:

$$\mathcal{O}\left(n \log n + n \log_\beta(n) \frac{\log(n \log_\beta n)}{\log n}\right)$$

. □

In the second step, we sorted the array. If we know nothing about the values there is nothing to deal with: The sort will take $\mathcal{O}(n \log n)$ time. As long as the values will be integers (or actually some of many other data types with information—such as strings), we can get a little improvement for this part. Specifically for integers, we could use *Radix Sort*.

The example of this procedure could be found in Figure 2.12.

1	2	5	4	3	6
{1,0}	{2,1}	{5,2}	{4,3}	{3,4}	{6,5}
{1,0}	{2,1}	{3,4}	{4,3}	{5,2}	{6,5}
0	1	4	3	2	5

Figure 2.12: Example of the first two steps of initialization

The pseudocode of this procedure could be found in Algorithm 14.

Algorithm 14 Initialize structure for searching k-th element

```
1: function INI(α, n)
2:     b ← new array of pairs(n)
3:     τ ← new persistent sum − elements(n)
4:     for i : 0 → n − 1 do
5:         b[i] = pair(α[i], i)
6:     end for
7:     sort(b)
8:     for i : 0 → n − 1 do
9:         τ.INSERT(b[i].second)
10:    end for
11: end function
```

2.5.3 Query

Before we will discuss the query on k-th element we will take look on another kind of query: *What is the number of element lesser/equal than i-th lowest element on input?*. In fact, this is pretty easy with what we have: We simply ask for the number of elements on the segment in i-th time (meaning we ask root whose pointer is on i-th position in our time-array). As we have discussed before, this query takes $\mathcal{O}(\log_{\frac{W}{\lceil \log n \rceil}} n)$.

Now if we would have some guess, we could verify the number of elements on the interval which is lesser than the number. So basically we have to find the index of an element for which the number of elements is greater than k and the $index - 1$ (unless the index is 0) is equal to k. This could be achieved by binary search by the answer. The whole process could be found in Algorithm 15.

Lemma 18. *Complexity of query in k-th element problem is $\mathcal{O}(\log n \log_\beta n)$, where β stands for branching factor and is equal to $\mathcal{O}(W/\log n)$.*

Proof. As we used binary search on the method mentioned in the first paragraph, the complexity will be $\mathcal{O}(\log_n)$ times higher than the complexity of the method which was mentioned in the paragraph. This leads to $\mathcal{O}(\log n \log_\beta n)$ complexity. □

This might be even better if the number of distinct elements will be asymptotically lesser than $\mathcal{O}(n)$.

Algorithm 15 Finds the k-th element on interval

Require: $beg \geq 0 \wedge beg \leq end \wedge end < size \wedge k \geq 0 \wedge k < end - beg + 1$

```
1:  function KTH(beg, end, k)
2:      b ← 0
3:      e → end − beg
4:      while b < e do
5:          if τ.GET(beg, end, time = (b + e)/2) < k then
6:              b = (b + e)/2 + 1
7:          else
8:              e = (b + e)/2
9:          end if
10:     end while
11:     return b
12: end function
```

2.5.4 Second approach of persistence

We have discussed multiple kinds of persistence. We will show the impact of the second approach of persistence onto the complexity.

Lemma 19. *The complexity of initialization of k-th element problem with the second approach is:*

$$\mathcal{O}(n \log_\beta n \log\log\log_\beta n + n \log n),$$

where β stands for branching factor and is equal to $\mathcal{O}\left(\frac{W}{\log n}\right)$.

Proof. As we can see, the only additional factor apart from the confluent data structure is factor resulting from Straka's *persistent arrays* is triple logarithm (which has very slow development). Thus the complexity will reach $\mathcal{O}(n \log_\beta n \log\log\log_\beta n + n \log n)$. □

Lemma 20. *The complexity of query of k-th element problem with second approach is:*

$$\mathcal{O}(\log n \log_\beta(n) \log\log\log_\beta n)$$

Proof. This complexity is a multiplication of three factors: Binary Search, Confluent Sum-Elements-Tree and Straka's *Persistent Arrays*. Thus the complexity will reach:

$$\mathcal{O}(\log n \log_\beta(n) \log\log\log_\beta n)$$

. □

As we might observe, the query of the second approach will be strictly slower than a query with the previous version (at least asymptotically).

Chapter 3

Algorithms

The first goal of this chapter is to parallelize an algorithm for greatest common divisor in such a way that it will calculate several pairs of integers at once. This result is achieved by bitwise parallelization. In the next section, a simple application is shown.

Next, we discuss the usage of operation MULTIPLICATION.

We also design a circuit for the maximum which could be applied in *Max-value Tree* if it would be done under AC^0.

Last part of this chapter focuses on bitwise parallelization of Number Theoretical Transform. We design a parallel MODULO operation to achieve so.

3.1 GCD

3.1.1 Introduction

In this section, we are going to take peek onto GCD [Greatest Common Divisor]. There are multiple ways how to proceed with such operation on a pair of two integers (of size κ bits) such as comparison of prime factors or Euclidean algorithm (which can do so in $5 \cdot \log_{10}(\kappa)$ steps as stated in Lamé's theorem [15]).

The algorithm we will be interested in is **Stein's Algorithm**, published by Josef Stein [16], also known as **Binary GCD Algorithm**. The number of steps is $\mathcal{O}(\log \kappa)$ which is asymptotically same as Euclidean Algorithm.

Even though $\text{GCD}(0, 0)$ is not defined, we traditionally return 0 as result, which we will keep up too.

Our intention is to do some bit parallelism, which means doing multiple GCD's at once (depends on how many integers of length κ fits into the word of length W) with as similar asymptotic complexity (to classical GCD) as possible.

3.1.2 Stein's algorithm

As we will discuss this algorithm to detail, we have to introduce it a little bit more. There are several ways how to make such algorithm—for our purposes we have chosen a simple iterative method which is described in Algorithm 16.

Algorithm 16 Pseudocode of Stein's Algorithm

```
1: function GCD(u, v)
2:     if u == 0 then
3:         return v
4:     end if
5:     if v == 0 then
6:         return u
7:     end if
8:     shift = 0
9:     while (u | v) & 1 == 0 do
10:        u >>= 1
11:        v >>= 1
12:        ++shift
13:    end while
14:    while u & 1 == 0 do
15:        u >>= 1
16:    end while
17:    while v ≠ 0 do
18:        while v & 1 == 0 do
19:            v >>= 1
20:        end while
21:        if u > v then
22:            SWAP(u, v)
23:        end if
24:        v = v − u
25:    end while
26:    return u << shift
27: end function
```

3.1.3 Parallelism

As can be seen, there are several operations which have to be discussed. Similarly to previous algorithms, we have to split the word to $\lfloor W/(\kappa+1) \rfloor$ buckets. Each bucket will consist of $\kappa + 1$ bits: The last κ bits of each bucket will be used to represent numbers while the first bit will be used in our algorithm.

Definition 23. *We will define a constant c which is filled by* 0*-bits with exception of the last bit of every bucket which is set to* 1*.*

The algorithm modification takes the following steps:

- We would like to store (in the end) our answer in u so we start with CSWAP(v, u). With this property, we know we are done if the number of 1-bits of ZR(v) is equal to the number of buckets. Also, note that we shall STALL on all buckets for which ZR(v) is true during each operation since we already know the answer for such buckets. It is always just $\mathcal{O}(1)$ overhead yet for sake of simplicity we will not mention this in the rest of the subsection. Also, note that we solve the first two ifs by this since we are not able to end the function here (unless ZR(v) would be true for *all* buckets).

- Now let us peek on $shift$. Obviously, we have to have such variable in our algorithm yet it is not that simple since it has to behave independently for each bucket. The definition of our shift will be following: If the shift of i-th integer would be r, our shift will have $2^r - 1$ in i-th bucket (observe that r cannot be greater than κ so no problem can appear with such definition). Now until we will not STALL every bucket (we would STALL an integer with all 1 bits of c on), we STALL $(u \,|\, v) \,\&\, c$ for words u, v and $shift$. We apply right shift on u and v while we apply $shift = (shift \ll 1) \,|\, c$ (note that there are some bits added wrongly, yet the STALL function will get rid of them). This phase will have at most $\mathcal{O}(\kappa)$ steps which are all $\mathcal{O}(1)$.

- The next loop will be very similar: we STALL by $u \,\&\, c$ while shifting to the right. The asymptotic complexity is the same as of the previous operation.

- The next loop which is presented in pseudocode has three steps:

  - The first step is similar to the loop we discussed above.
  - The second step is a simple call of CSWAP.
  - The third step can be left as it is.
  - All steps have $\mathcal{O}(1)$ complexity and the number of such cycles is same as in original algorithm (in fact it is equal to maximum among the numbers of steps for all buckets). The loop ends when ZR(v) is equal to c.

- Last operation is $u \ll shift$. It could be done by a loop which goes while $shift$ is not equal to 0. In each step it STALLs all buckets which have last bit empty: $shift \oplus c$. It shifts u (the buckets which are not STALLed) and also updates $shift$ as $shift = (shift \;\&\; \sim c) \gg 1$ which erases the last bits of $shift$ and shifts $shift$ to right. This is the only operation which does not have the same number of steps as in the original algorithm:

It was $\mathcal{O}(1)$ but now it is $\mathcal{O}(\kappa)$. Luckily it will not endanger the total complexity of the function.

To sum it up, we made a GCD function which works in $\mathcal{O}(\kappa)$ as the original version (obviously only if $\kappa \leq W$) yet now it can find the answer for $\mathcal{O}(W/\kappa)$ integers at once.

3.2 Sparse Table

3.2.1 Introduction

Definition 24. ***Sparse Table*** *is very simple static data structure which initializes from an array in $\mathcal{O}(\Psi n \log n)$ time, and answers the result of an operation on segment in $\mathcal{O}(\Psi)$ time where Ψ stands for time of operation (an example of operation might be min or max). This structure occupies $\mathcal{O}(n \log n)$ space.*

Sparse Table consists of 2D table where first dimension has $\mathcal{O}(\log_2 n)$ levels while the second has n levels. The element on position ij-th position is the result of operation on segment between j and $j + 2^i - 1$ (inclusive). Any field of table can be filled simply in $\mathcal{O}(\Psi)$ by checking field on $[i-1][j]$ and $[i-1][j+2^{i-1}]$. The method for initialization could be seen in Algorithm 17.

Algorithm 17 Pseudocode for initialization of data structure

```
1: function INIT(input, n)
2:     INIT(Table[log2 n][n])
3:     for i : 0 → n − 1 do
4:         Table[0][i] = input[i]
5:     end for
6:     for k = 0 : 2^k ≤ n do
7:         for i : 0 → n − 2^k − 1 do
8:             Table[k][i] = OPERATION(Table[k − 1][i], Table[k − 1][i + 2^(k−1)])
9:         end for
10:    end for
11: end function
```

The result for segment from *begin* to *end* can be obtained in $\mathcal{O}(\Psi)$ (while ℓ is the greatest power of 2 lesser than $e - b + 1$) by taking operation on $[\ell][begin]$ and $[\ell][end - 2^\ell + 1]$. As we can observe, the intervals intersect which might be problematic for operation for which $x(operation)x$ is not x (for any x) such as addition, xor or similar (anyway luckily those operations could be usually solved by prefix sum which is even easier). Anyway for many operations this structure is fine: such as MIN, MAX, GCD, AND, OR, etc... It also relies on getting the greatest power of 2 which might not be simple (to

obtain it in $\mathcal{O}(1)$) but we can (in worst case) precompute an array in $\mathcal{O}(n)$ during initialization and the simply look up the answer as could be seen in Algorithm 18.

Algorithm 18 Pseudocode for operation on interval

```
1: function RMQ(begin, end)
2:     k = log2(end − begin + 1)
3:     return OPERATION(Table[k][begin], Table[k][end − 2^k + 1])
4: end function
```

Even though this structure is not the fastest one, the results are fine and it is pretty easy to code.

3.2.2 Bitwise Parallelisation

Even though the asymptotic complexity of query part is hard to improve (without n^2 table: which could do so from $\mathcal{O}(\Psi)$ to $\mathcal{O}(1)$), there is still space for improvement during initialization part.

As long as κ (number of bits to represent a number) would be significantly lesser than W (size of the word), we can have set of n buckets in each level (instead of n words we would need only $\mathcal{O}(\frac{n \cdot \kappa}{W})$ words).

Now instead of finding out the result of every cell separately, we could always take the previous level, shift it by $\kappa 2^{i-1}$ and apply our operation onto the previous level and our shifted previous level.

Here not only the time complexity improved to $\mathcal{O}(\frac{n \cdot \kappa}{W})$ yet same happened with space complexity. Obviously note that the complexity can get better than $\mathcal{O}(\log_2 n)$, yet such a case would not probably matter anyway.

We also have to know how to proceed the bitwise parallel operation. The operation even has to be trivial (as bitwise operations like & or |) or we have to come up with one (as in the case of GCD).

3.3 Operations

3.3.1 Multiplication

Some instruction set systems do not have operations as *multiplication* (such as AC^0). In our structures, this operation has key importance. One example of usage is to copy bucket b to κ following buckets in $\mathcal{O}(1)$.

A very naive way might obviously be copying of the bucket to each of the κ buckets one by one with help of a few shifts in $\mathcal{O}(\kappa)$. This is indeed very slow, yet it is not a problem to improve this approach. First copying will remain the same. The second one will copy both of the last two buckets and

shift it by two times the size of the bucket. Now as we have filled four buckets already, we can fill eight by a similar operation.

As we can see, we can grow the integer to $2^{\lfloor \log_2 \kappa \rfloor}$ in $\mathcal{O}(\log_2 \kappa)$ steps. The last step (if necessary) is a simple shift by the size of unfilled buckets and ORing those two integers to itself. As we can see, OR operation will not corrupt the bits which overlap. The total complexity of this detour is $\mathcal{O}(\log_2 \kappa)$

If this would be designed as AC^0 (circuit with constant depth and polynomial size) operation, it would be simply wiring of the first bucket to each other bucket.

Another usage is to copy bucket b to i-th bucket exactly $(i+1)$ times. This was done by multiplying b to a constant which has $(i+1)$ in i-th bucket. Here we could use a very similar approach as before. Firstly we will do the previous step, so let us have an integer A which has b copied in every bucket. Then, for each possible bit, in j-th step, we will add b shifted by j. This could be done by shifting integer α by j and adding it to result.

We already discussed the first step has $\mathcal{O}(\log \kappa)$ complexity. Then the number of steps, which is each $\mathcal{O}(1)$ is at most $\lceil \log_2 \kappa \rceil$ so the total complexity is at most $\mathcal{O}(\log \kappa)$.

Anyway, *note* that for some data structures the size of the bucket is at most $\mathcal{O}(\log n)$, which means at most $\mathcal{O}(n)$ possible values. We can do a table of size $\mathcal{O}(n)$, while under i-th index it will be stored the bucket properly multiplied with given constant. This will take time of an operation for every index which means $\mathcal{O}(n \log \kappa)$ during initialization, yet then there will be no overhead afterward.

3.3.2 Maximum

Definition 25. *An interesting task is to propose operation maximum from buckets as AC^0 operation. Let us denote an integer of size W filled with κ buckets, each of size h bits. Let us denote integer in i-th bucket as m_i.*

Firstly let us make a circuit which yields 1 if i-th bucket is one of the buckets which has the maximum in it (0 otherwise). The first step is a set of circuits which yields carry of subtraction of two elements (which is not a basic operation of AC^0 but it is known). For each j from 0 to $\kappa - 1$ let us lead i-th and j-th bucket into such circuit so $m_i - m_j$ would yield 0 if m_i is greater/equal to m_j. We lead the results to an OR bucket. If we would lead negation from this bucket, we would already have what we came for.

If we would need only maximum itself, we could simply wire these circuits to each position of the bucket (so they would either make 0 or $2^h - 1$). The we will AND this with original value m_i. If we would OR the results from each bucket, it would yield us maximum.

If we would need the index of maximum (instead of maximum itself) we could simply use the first step (1 if maximum), chain them one after another

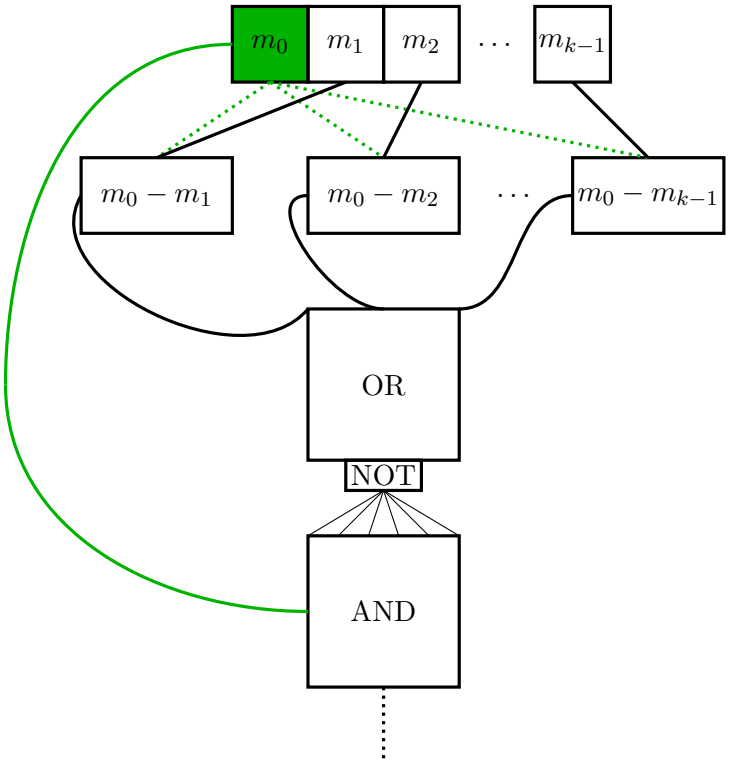

Figure 3.1: Circuit which returns m_0 if it is maximal

and find the most significant bit. This might be for example done by the following process: For each bit, we can OR all previous bits, negate this and AND this with the bit on the actual position. This will result in only one bit. Now if we AND each bit with copied $\lceil \log_2 \kappa \rceil$ times with i. As we OR all the results together, we will get the index of the most significant bit.

This operation could be used in **max-value** trees if they would be done in AC^0. This could lead to significant complexity improvement and also milder demands on the size of values.

The part of the circuit which returns m_0 if it is one of the maximums could be found in Figure 3.1.

3.4 Number Theoretic Transform

3.4.1 Introduction

Definition 26. *Number Theoretic Transform (further as **NTT**) is a kind of Discrete Fourier Transformation. Modular arithmetic (in some modulo q) is used to calculate such transformation, so all integers of input/output vector*

*are calculated in modular arithmetic. The advantage of **NTT** is avoidance of rounding errors.*

To transform vector of size $n = 2^k$, we have to find the vector

$$[g^0, g^1, g^2, \ldots, g^{n-1}],$$

where g is $2n$-th root of unity in modulo q. According to Michael Scott [17], such g could be found easily if we choose q such that $q = 1 \mod n$. As we have g, our vector could be generated in $\mathcal{O}(n)$ by simply making $g^i = g^{i-1} \cdot g$. To proceed with our algorithm, we also have to sort the elements of vector g by its powers as if they would be bitwise reversed. This could be done for example by simple recursion generating all integers (bit by bit) while also generating such integers from backward (in i-th step try to set i-th bit to 0 and then 1. Do the same for $(k-1-i)$-th bit for the second integer). This recursion has $2n$ steps.

3.4.2 Parallel modulo

Probably the most important operation of modular arithmetic is modulo itself. It is indeed bad news that this operation is kinda problematic to be done by bitwise parallelism. We would like to present a method, which could do such operation bitwise parallel yet sadly with some non-constant slowdown. Imagine we would like to do parallel modulo, where the numbers might have up to b bits (note that b is probably bigger than $\log_2 m$ since otherwise, we could do such operation quickly by subtracting the m), where m stands for modulus (which is operand of modulo operation). Firstly, we shall possess buckets which are slightly bigger than the maximal number of bits we need to represent numbers (by a constant: 3 additional bits shall be enough).

Before we will introduce the approach, we shall adopt a few constants. Firstly it is c, which is m copied to all buckets. Secondly, it is constant ℓ which has 1 at the end of every bucket. Then it is the difference between the representation size of the number and of modulo k: where $k = b - \lceil \log_2 m \rceil$.

The approach itself will happen for each bit between 0 and k (for each bucket at once). Starting by $i = k$ (descending down to 0) we create a temporary variable t as sum of result and c shifted by i to left: $t = result + (c \ll i)$. Now we detect all the buckets in which t is bigger than the input. The detection could be done by an above-mentioned trick with ORing ℓ shifted by bucket size -1 to t and subtracting input. Now we can simply STALL *result* and adding $(c \ll i)$ to buckets which were not greater. By this approach, we will create the biggest multiple of m which is lesser/equal to input (in each bucket separately). Now simple $input - result$ would do the same job as moduling every bucket.

Lemma 21. *The complexity of parallel modulo is* $\mathcal{O}(b - \log m)$.

Proof. There is a constant step for each bit greater than the highest bit of m. □

3.4.3 Pseudocodes

The **NTT** algorithm has to proceed not only the transformation itself but also the inverse transformation [17]. There is Algorithm 19 for NTT and Algorithm 20 for INTT.

Algorithm 19 Pseudocode of NTT Algorithm

```
1:  function NTT(x, n)
2:      t = n/2
3:      m = 1
4:      while m < n do
5:          k = 0
6:          for i : 0 → m − 1 do
7:              s = g[m + i]
8:              for j : k → k + t − 1 do
9:                  u = x[j]
10:                 v = x[j + t] · s%q
11:                 x[j] = (u + v)%q
12:                 x[j + t] = (u − v)%q
13:             end for
14:             k = k + 2t
15:         end for
16:         t = t/2
17:         m = t ∗ 2
18:     end while
19: end function
```

3.4.4 Parallel NTT

Good news for us is that the only thing we are concerned about is the most inner loop since operations in outer loops will be executed at most $\mathcal{O}((\log n)^2)$ times.

As we can see, obtaining variables u and v is a simple question of shifts.

The addition is also easy as long as the buckets size is at least $\log_2 q + 1$.

The subtraction is somehow a catch since we cannot afford it to go negative (which would overflow a bucket). Luckily, this has an easy solution, which is adding c to u before subtraction.

We also need multiplication to be executed. This seems to be a pretty tough problem since multiplication is not bucket-independent operation. Anyway, note that we already have to do $\mathcal{O}(\log m)$ operations during modulo so

Algorithm 20 Pseudocode of inverse NTT Algorithm

```
1: function INTT(x, n)
2:     t = 1
3:     m = n/2
4:     while m > 0 do
5:         k = 0
6:         for i : 0 → m − 1 do
7:             s = g^{-1}[m + i]
8:             for j : k → k + t − 1 do
9:                 u = x[j]
10:                v = x[j + t]
11:                x[j] = (u + v)%q
12:                w = (u − v)%q
13:                x[j + t] = w · s%q
14:            end for
15:            k = k + 2t
16:        end for
17:        t = t ∗ 2
18:        m = m/2
19:    end while
20: end function
```

another $\mathcal{O}(\log m)$ operations will not spoil asymptotic complexity. Instead of multiplication, we add the number shifted by all i, such that i-th bit is on in s. Another good news is, that s is same for all buckets so we do not need STALL operation. Obviously, this is somehow problematic since we could add number shifted by $\lfloor \log_2 q \rfloor$, so we need to make the buckets bigger. Anyway it is enough to make them $2 \cdot \lfloor \log_2 q \rfloor + 3$, which is still $\mathcal{O}(\log q)$.

Lemma 22. *The total complexity of* NTT *will reach:*

$$\mathcal{O}\left(\frac{n \log(n) \log(m)}{W} + n + (\log_2 m)^2\right)$$

Proof. The complexity of operations, unless multiplication and modulo will remain same so the complexity will be the multiplication of NTT's original complexity and the maximum of complexities of modulo and multiplication (which are same—$\mathcal{O}(\log q)$). □

Chapter 4

Future work

There are several more questions related to this work which were not answered in this paper:

- What is the exact complexity of Sum-elements Tree's queries with described optimization?
- It it possible to construct Sum-values Tree with general modulo and no additional overhead?
- Is it possible to design Max-value Tree with the same complexities yet a better range of numbers on RAM model with limited size of integers?
- Could the NTT algorithm be designed without the overhead of modulo and multiplication?

Bibliography

[1] Reinders, J. Intel® AVX-512 Instructions. Available from: https://software.intel.com/en-us/blogs/2013/avx-512-instructions

[2] Mares, M. Vypocentni Modely. Charles University in Prague, January 2014. Available from: https://mj.ucw.cz/vyuka/ga/7-ram.pdf

[3] Arora, S.; Barak, B. *Computational complexity: a modern approach*. 2016.

[4] Forg, A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. Available from: https://www.agner.org/optimize/instruction_tables.pdf

[5] Demain, E. Integer: fusion trees, sketching, parallel comparison, most significant set bit. Massachusetts Institute of Technology, April 2012.

[6] Mares, M. Q-Heaps. Charles University in Prague, December 2009. Available from: https://mj.ucw.cz/vyuka/ga/8-qheap.pdf

[7] Fenwick, P. M. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, volume 24, no. 3, mar 1994: pp. 327–336, doi:10.1002/spe.4380240306. Available from: https://doi.org/10.1002/spe.4380240306

[8] Bentley, J. L.; Friedman, J. H. Data Structures for Range Searching. *ACM Computing Surveys*, volume 11, no. 4, dec 1979: pp. 397–409, doi:10.1145/356789.356797. Available from: https://doi.org/10.1145/356789.356797

[9] Guibas, L. J.; Sedgewick, R. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, oct 1978, doi:10.1109/sfcs.1978.3. Available from: https://doi.org/10.1109/sfcs.1978.3

[10] Malík, P. Persistent data structures and their application. 2014.

[11] Kim, S.-S. Partial Persistence. Available from: http://sungsoo.github.io/2014/01/18/partial-persistence.html

[12] Brodal, G. S. Partially Persistent Data Structures of Bounded Degree with Constant Update Time. *BRICS Report Series*, volume 1, no. 35, nov 1994, doi:10.7146/brics.v1i35.21608. Available from: https://doi.org/10.7146/brics.v1i35.21608

[13] Karger, D. Persistent Data Structures. Massachusetts Institute of Technology, September 2005.

[14] Straka, M. Fully persistent arrays with optimal worst-case complexity Milan Straka. 2013.

[15] Weisstein, E. W. Lamé's Theorem. Available from: http://mathworld.wolfram.com/LamesTheorem.html

[16] Stein, J. Computational problems associated with Racah algebra. *Journal of Computational Physics*, volume 1, no. 3, feb 1967: pp. 397–405, doi:10.1016/0021-9991(67)90047-2. Available from: https://doi.org/10.1016/0021-9991(67)90047-2

[17] Scott, M. A Note on the Implementation of the Number Theoretic Transform. 2017: pp. 247–258, doi:10.1007/978-3-319-71045-7_13. Available from: https://doi.org/10.1007/978-3-319-71045-7_13

Appendix A

Acronyms

GCD Greatest Common Divisor

NTT Number Theoretic Transform

DAG Directed Acyclic Graph

MSB Most Significant Bit

LSB Least Significant Bit

w.l.o.g. Without Loss On Generality

AVX Advanced Vector Extensions

Appendix B

Contents of enclosed CD

root
- segment.pdf
- segment.tex
- mybibliographyfile.bib