Lebanese University
Faculty of Science
Computer Science BS Degree

Advanced Algorithms I3341

Dr Siba Haidar & Dr Antoun Yaacoub

Course Chapters

as per the textbook

1. Introduction
2. Data Structures and Libraries
3. Problem Solving Paradigms
4. Graph
5. Mathematics
6. String Processing
7. Computational Geometry

Problem Solving Paradigms

Chapter 3

Chapter Outline

1. Complete Search = Brute Force
   a. Iterative Complete Search
   b. Recursive Complete Search
   c. Tips
2. Divide and Conquer
   a. Interesting Usages of Binary Search
3. Greedy
   a. Examples
4. Dynamic Programming
   a. DP Illustration
   b. Classical Examples
   c. Non-Classical Examples

Overview and Motivation

Let us focus on Divide & Conquer

- 1. Find largest & smallest element
  - → $O(n)$ Complete Search
- 2. Find kth smallest element
  - find smallest and replace with big number
  - repeat k times
  - if k = n/2 → $O(n \times n/2) \sim O(n^2)$
  - better sort then choose kth → $O(nlogn)$ → Divide and Conquer
  - better $O(n)$ → also Divide and Conquer
- 3. Find largest gap g / x, y ∈ A & g = |x – y|
  - consider every pair → $O(n^2)$
  - largest – smallest → $O(n)$ → Greedy
- 4. Find longest increasing subsequence
  - try all $O(2^n)$ possible subsequences
  - not feasible for all n ≤ 10K
  - → $O(n^2)$ DP
  - → $O(nlogn)$ Greedy

Divide & Conquer

Introduction

- Based on recursion.
- How it works ?
  - Recursively break down a problem into two or more sub problems of the same type, until they become simple enough to be solved directly.
  - The solutions to the sub problems are then combined to give a solution to the original problem.

Strategy?

- The D & C strategy solves a problem by:

1. ***Divide****:* Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2. ***Recursion****:* Recursively solving these sub problems.
3. ***Conquer****:* Appropriately combining their answers.

Does Divide and Conquer Always Work?

- NO !!
- For all problems it is not possible to find the subproblems which are the same size and *D* & C is not a choice for all problems.

Visualization

- Assume that *n* is the size of the original problem.
- Divide the problem into *b* sub problems with each of size *n/b* (for some constant *b*).
- Solve the sub problems recursively and combine their solutions to get the solution for the original problem.

Visualization

```
DivideandConquer(P){
        if(small(P))
                // P is very small so that a solution is obvious
                return solution(P);

        Divide P into b subproblems: P1, P2, … Pb

        return (
                combine(
                        DivideandConquer(P1),
                        DivideandConquer(P2),
                        …,
                        DivideandConquer(Pb)
                        )
                )
}
```


Dr Siba Haidar & Dr Antoun Yaacoub

Examples

- We have already solved many problems based on *D* & *C* strategy: like Binary Search, Merge Sort, Quick Sort, etc....

- Looking for a name in a phone book: We have a phone book with names in alphabetical order. Given a name, how do we find whether that name is there in the phone book or not?
- Finding our car in a parking lot.
- …

Advantages of Divide and Conquer

- **Solving difficult problems:** *D* & *C* is a powerful method for solving difficult problems: Tower of Hanoi problem.

- **Parallelism:** Since *D* & *C* allows us to solve the subproblems independently.

Disadvantages of Divide and Conquer

- Recursion is slow.
  Overhead of the repeated subproblem calls.
  Stack for storing the calls.

- Sometimes more complicated than an iterative approach.
  Example: add *n* numbers, a simple loop to add them up in sequence is much easier than a *D* & *C* approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

Divide and Conquer Applications

- Binary Search
- Merge Sort and Quick Sort
- Median Finding
- Min and Max Finding
- Matrix Multiplication
- Closest Pair problem

Binary Search: The Ordinary Usage

- As the size of search space is halved (in a binary fashion) after each check, the complexity of this algorithm is *O(log n).*
- There are built-in library routines for this algorithm, e.g.
  - the C++ STL `algorithm::lower_bound` / `algorithm::upper_bound`
  - the Java `Collections.binarySearch`

Binary Search: The Ordinary Usage

```cpp
#include <iostream>     // std::cout
#include <algorithm>    // std::lower_bound,
std::upper_bound, std::sort
#include <vector>       // std::vector
int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  std::vector<int> v(myints,myints+8);
  std::sort (v.begin(), v.end());
  std::vector<int>::iterator low,up;
  low=std::lower_bound (v.begin(), v.end(), 20);
  up= std::upper_bound (v.begin(), v.end(), 20);

  std::cout << "lower_bound at position "
            << (low- v.begin()) << '\n';
  std::cout << "upper_bound at position "
            << (up - v.begin()) << '\n';
  return 0;
}
```

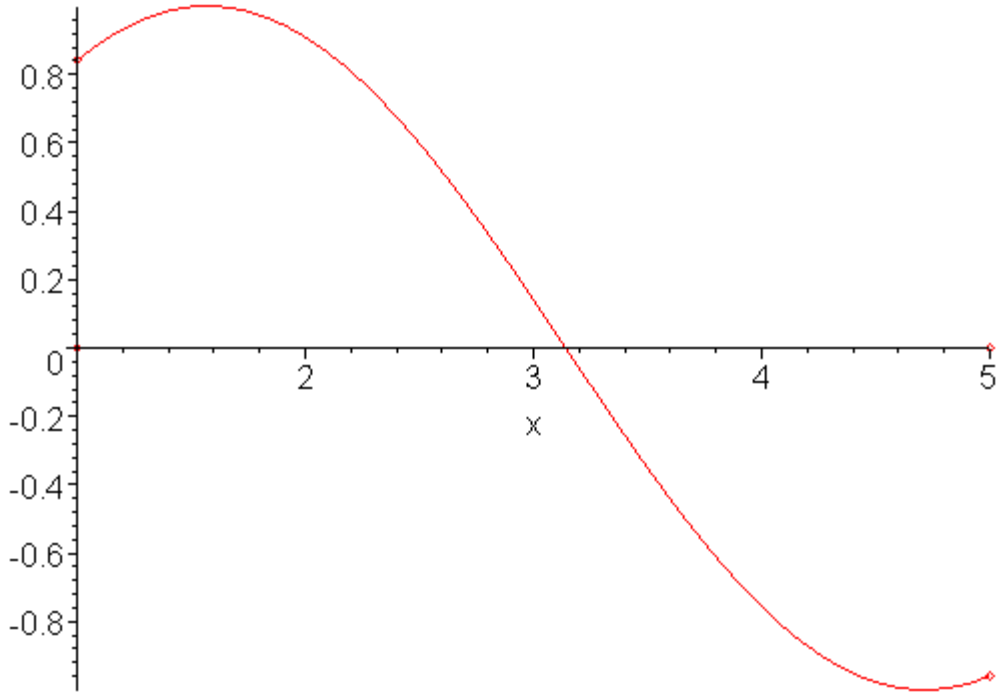
Output:
lower_bound at position 3
upper_bound at position 6

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
public class GFG
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(1);
        al.add(2);
        al.add(3);
        Collections.sort(al);

        int index = Collections.binarySearch(al, 10);
        System.out.println(index);

        // 13 is not present. 13 would have been inserted
        // at position 4. So the function returns (-4-1)
        // which is -5.
        index = Collections.binarySearch(al, 13);
        System.out.println(index);
    }
}
```

Output:
3
-5

Bisection Method

- Find the root of a function that may be difficult to compute directly.



Bisection Method - Example

- You buy a car with loan and now want to pay the loan in monthly installments of *d* dollars for m months.
- Suppose the value of the car is originally *v* dollars and the bank charges an interest rate of *i%* for any unpaid loan at the end of each month. What is the amount of money d that you must pay per month (to 2 digits after the decimal point)?

Bisection Method - Example

- Suppose $d = 576.19$, $m = 2$, $v = 1000$, and $i = 10\%$.
  - After one month, your debt becomes $1000 \times (1.1) - 576.19 = 523.81$.
  - After two months, your debt becomes $523.81 \times (1.1) - 576.19 \approx 0$.

Now let's reverse the process:

- If we are only given $m = 2$, $v = 1000$, and $i = 10\%$, how would we determine that $d = 576.19$?
  In other words, find the root d such that the debt payment function $f(d,m, v, i) \approx 0$.

Bisection Method - Example

- Pick a reasonable range [*a..b*] as starting points.
  For the bisection method to work, we must ensure that the function values of the two extreme points in the initial Real range [*a..b*], i.e. *f(a)* and *f(b)* have opposite signs

- Fix d within the range [*a..b*]
  where a = 0.01 as we have to pay at least one cent and b = (1+i%) × v as the earliest we can complete the payment is m = 1 if we pay exactly (1 + i%) × v dollars after one month. In this example, b = (1+0.1) × 1000 = 1100.00 dollars.

Bisection Method

a	b	$d = \frac{a+b}{2}$	Status: $f(d,m.v.i)$	action
0.01	1100.00	550.005	Undershoot by 54.9895	↑ d
550.005	1100.00	825.0025	Overshoot by 522.50525	↓ d
550.005	825.0025	687.50375	Overshoot by 233.757875	↓ d
550.005	687.50375	618.754375	Overshoot by 89.384187	↓ d
550.005	618.754375	584.379688	Overshoot by 17.197344	↓ d
550.005	584.379688	567.192344	Undershoot by 18.896078	↑ d
567.192344	584.379688	575.786016	Undershoot by 0.849366	↑ d
…	…	…. After few iterations		…
…	…	576.190476 stop; error is now less than ε answer = 576.19		

(1.1)*1000 – 550.005 = 549.995
(1.1)*549.995 – 550.005 = 54.9895

Bisection Method

- It requires $O\left(log_2\left(\frac{b-a}{\varepsilon}\right)\right)$iterations to get an answer
- In this example, bisection method only takes $log_2\left(\frac{1099.99}{\varepsilon}\right)$ tries.
  - Using a small ε = 1*e*-9, this yields only ≈ 40 iterations.
  - Even if we use a smaller ε = 1*e*-15, we will still only need ≈ 60 tries.
  - Notice that the number of tries is *small*.
  - The bisection method is much more efficient compared to exhaustively evaluating each possible value of *d* =[0.01..1100.00]/ε.
- Note: The bisection method can be written with a loop that tries the values of *d* ≈ 40 to 60 times using 'binary search' technique.

Dr Siba Haidar & Dr Antoun Yaacoub

Bisection Method

```
double lo = 0.01, hi = 1100.00, mid ;

for (int i = 0; i < 50; i++) { // log_2 ((1099.99 - 0.0) / 1e-9) ~= 40
        mid = (lo + hi) / 2.0; // looping 50 times should be precise enough
        if (f(mid)>0) low= mid;
        else hi = mid;
}
return mid;
```


Dr Siba Haidar & Dr Antoun Yaacoub

Segment Tree
Range Minimum Query (RMQ) problem

Dr Siba Haidar & Dr Antoun Yaacoub

Segment Tree
Range Minimum Query (RMQ) problem

- Data structure which can efficiently answer *dynamic* range queries.
- One such range query is the problem of finding the index of the minimum element in an array within range [*i..j*].
- This is more commonly known as the Range Minimum Query (RMQ) problem.

Example

Array	Values	18	17	13	19	15	11	20
A	Indices	0	1	2	3	4	5	6

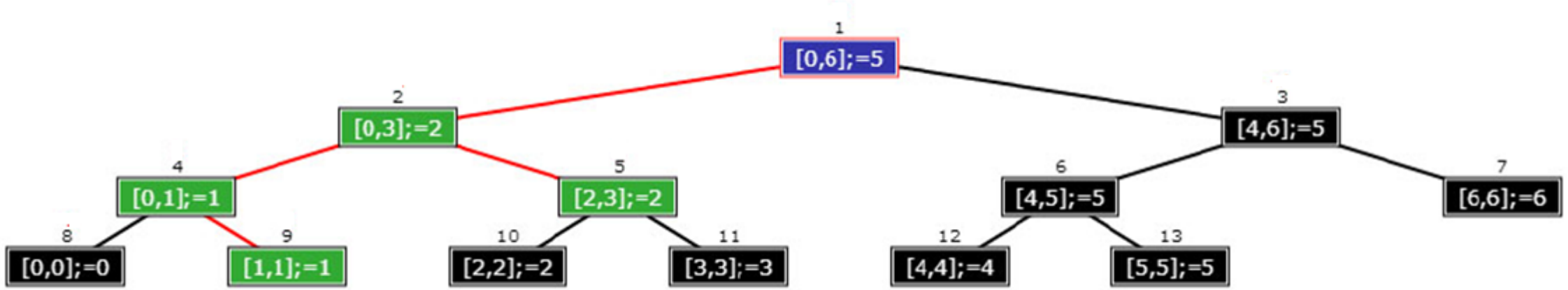
- Given an array *A* of size *n = 7,*
- RMQ(1, 3) = 2, as the index 2 contains the minimum element among A[1], A[2], and A[3].
- RMQ(3, 4) = 4, RMQ(0, 0) = 0, RMQ(0, 1) = 1, and RMQ(0, 6) = 5.

- There are several ways to implement the RMQ. One trivial algorithm is to simply iterate the array from index *i* to *j* and report the index with the minimum value, but this will run in *O(n)* time per query. When *n* is large and there are many queries, such an algorithm may be infeasible.

Implementation

- Segment tree like binary heap using static implementation.
  - We call the array `st`.
  - Index 1 (skipping index 0) is the root and the left and right children of index p are index 2 × p and (2 × p) + 1 respectively.
  - The value of `st[p]` is the RMQ value of the segment associated with index *p*.

Implementation



- The root of segment tree represents segment [0, n-1].
- For each segment [L, R] stored in index p where L != R, the segment will be split into [L, (L+R)/2] and [(L+R)/2+1, R] in a left and right vertices.
- The left sub-segment and right sub-segment will be stored in index 2×p and (2×p)+1 respectively.
- When L = R, it is clear that st[p] = L (or R). Otherwise, we will recursively build the segment tree, comparing the minimum value of the left and the right sub-segments and updating the st[p] of the segment.

Implementation

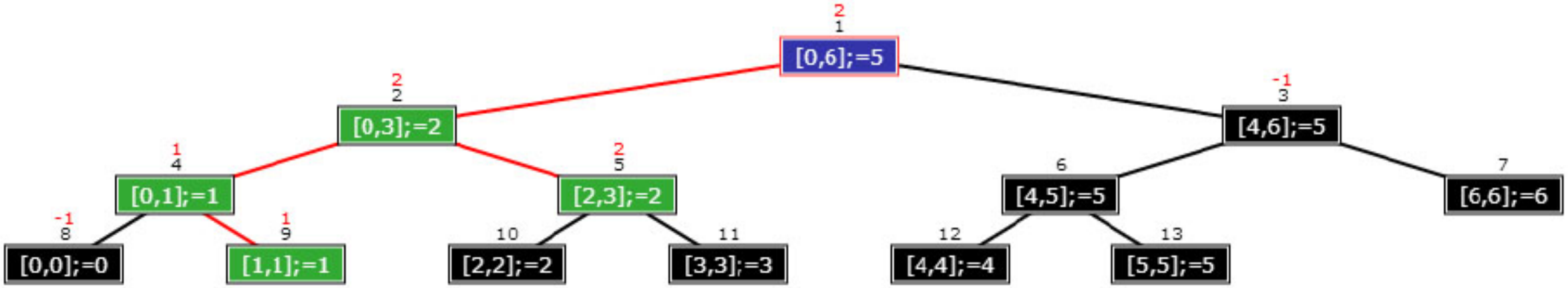
```
typedef vector<int> vi;
class SegmentTree {
  private: vi st, A;
  vector<int> vi;
  int n;
  int left (int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }
  void build(int p, int L, int R) {
      if (L == R) st[p] = L;
      else {
            build(left(p) , L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            int p1 = st[left(p)], p2 = st[right(p)];
            st[p] = (A[p1] <= A[p2]) ? p1 : p2;
      }
     }
```

```
class SegmentTree {
  private int[] st, A;
  private int n;
  private int left (int p) { return p << 1; }
  private int right(int p) { return (p << 1) + 1; }

  private void build(int p, int L, int R) {
    if (L == R) st[p] = L;
    else {
          build(left(p) , L, (L + R) / 2);
          build(right(p), (L + R) / 2 + 1, R);
          int p1 = st[left(p)], p2 = st[right(p)];
          st[p] = (A[p1] <= A[p2]) ? p1 : p2;
    }
   }
```

Implementation

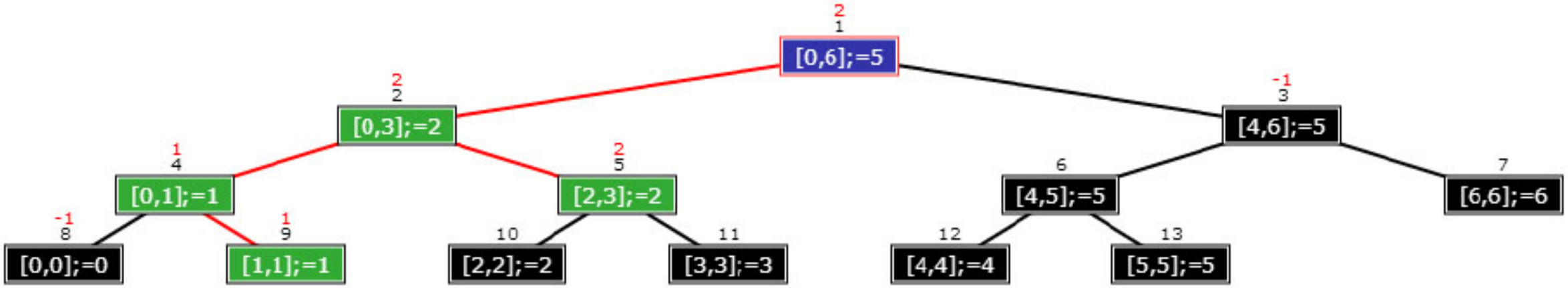
- With the segment tree ready, answering an RMQ can be done in *O(log n)*.
- The answer for RMQ(i, i) is trivial—simply return i itself.
- However, for the general case RMQ(i, j), further checks are needed.
  Let `p1=RMQ(i,(i+j)/2)` and `p2=RMQ((i+j)/2+1, j)`.
  Then RMQ(i, j) is p1 if A[p1] ≤ A[p2] or p2 otherwise.

Implementation



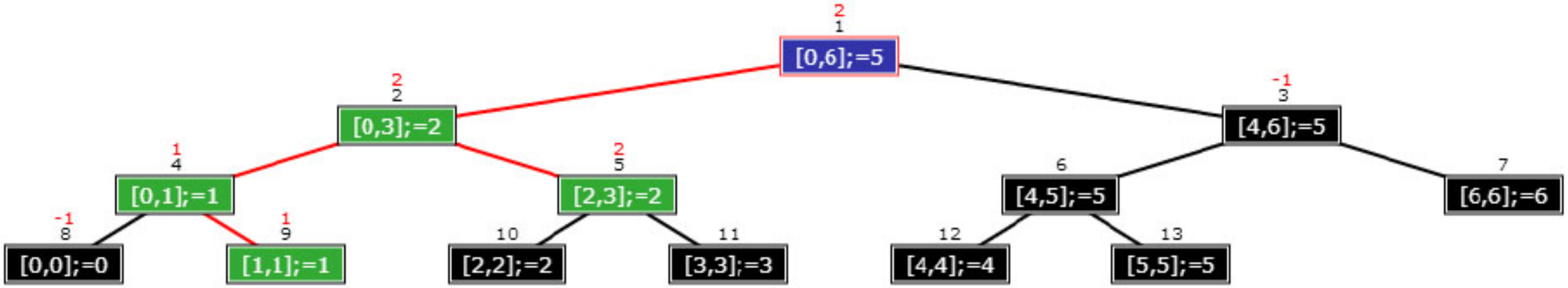
- RMQ(1, 3):
  - Start from the root (index 1) which represents segment [0, 6]. We cannot use the stored minimum value of segment [0, 6].
  - From the root, we only have to go to the left subtree as the root of the right subtree represents segment [4, 6] which is outside the desired range in RMQ(1, 3).

Implementation



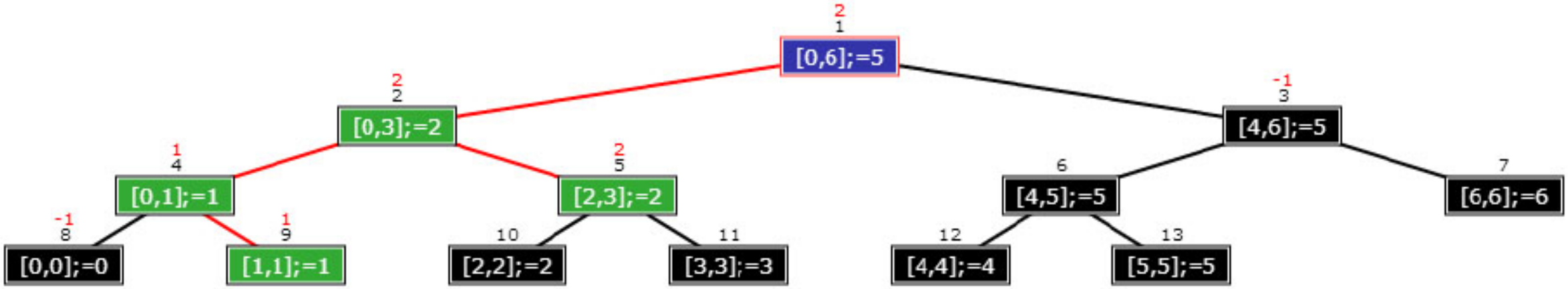
- RMQ(1, 3):
  - We are now at the root of the left subtree (index 2) that represents segment [0, 3]. This segment [0, 3] is still larger than the desired RMQ(1, 3).
  - RMQ(1, 3) intersects both the left sub-segment [0, 1] (index 4) and the right sub-segment [2, 3] (index 5) of segment [0, 3], so we have to explore both subtrees (sub-segments).

Implementation



- RMQ(1, 3):
  - The left segment [0, 1] (index 4) of [0, 3] (index 2) is not yet inside the RMQ(1, 3). From segment [0, 1] (index 4), we move right to segment [1, 1] (index 9), which is now inside [1, 3].
  - At this point, we know that RMQ(1, 1) = st[9] = 1 and we can return this value to the caller. The right segment [2, 3] (index 5) of [0, 3] (index 2) is inside the required [1, 3]. From the stored value inside this vertex, we know that RMQ(2, 3) = st[5] = 2. We do not need to traverse further down.

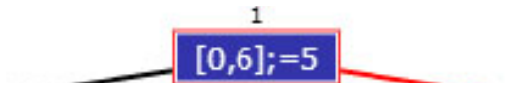
Implementation



- RMQ(1, 3):
  - Now, back in the call to segment [0, 3] (index 2), we now have p1 = RMQ(1, 1) = 1 and p2 = RMQ(2, 3) = 2. Because A[p1] > A[p2] since A[1] = 17 and A[2] = 13, we now have RMQ(1, 3) = p2 = 2. This is the final answer.

Implementation

- Build the segment tree of Array A = {18, 17, 13, 19, 15, 11, 20} and RMQ(4, 6)



Implementation

```
int rmq(int p, int L, int R, int i, int j) {
      if (i >  R || j <  L) return -1;
              // current segment outside query range
       if (L >= i && R <= j) return st[p];
              // inside query range
     // compute the min position in the left and right
       int p1 = rmq(left(p) , L, (L+R) / 2, i, j);
       int p2 = rmq(right(p), (L+R) / 2 + 1, R, i, j);
       if (p1 == -1) return p2;
          // if we try to access segment outside query
       if (p2 == -1) return p1;
          // same as above
       return (A[p1] <= A[p2]) ? p1 : p2;
}
```

```
private int rmq(int p, int L, int R, int i, int j) {
    if (i >  R || j <  L) return -1;
              // current segment outside query range
    if (L >= i && R <= j) return st[p];
             // inside query range
  // compute the min position in the left and right
    int p1 = rmq(left(p) , L, (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R, i, j);
    if (p1 == -1) return p2;
        // if we try to access segment outside query
     if (p2 == -1) return p1;
        // same as above
     return (A[p1] <= A[p2]) ? p1 : p2; }
```

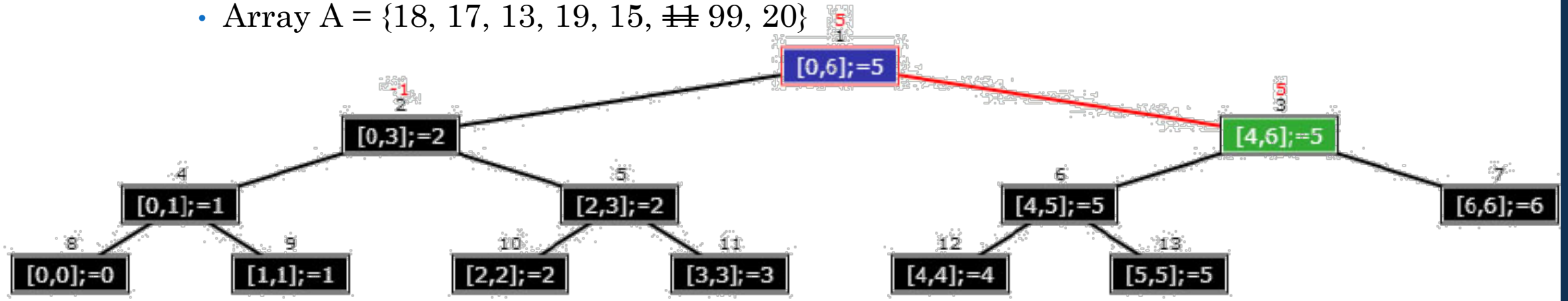
Usage

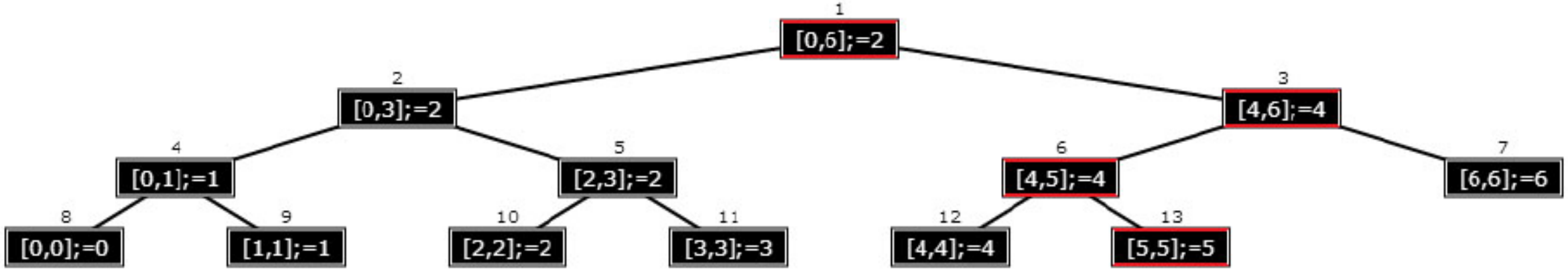
- If the array A is static (i.e. unchanged after it is instantiated), then using a Segment Tree to solve the RMQ problem is **OVERKILL** as there exists a Dynamic Programming (DP) solution that requires O(n log n) one-time pre-processing and allows for O(1) per RMQ.

- Segment Tree is useful if the underlying array is frequently updated (**dynamic**).

Implementation

- Array A = {18, 17, 13, 19, 15, ~~11~~ 99, 20}

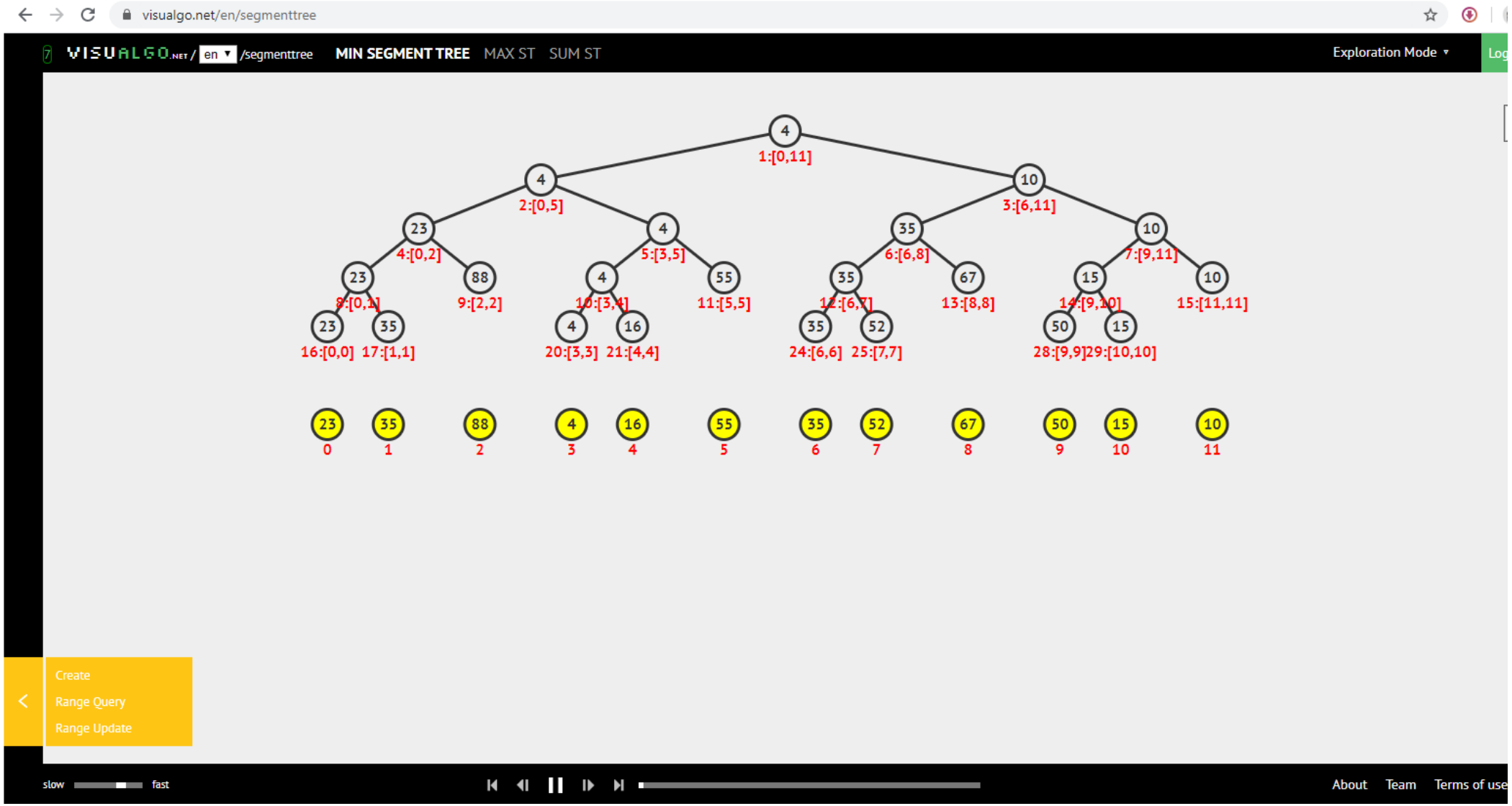




Implementation

- We just need to update the vertices along the leaf to root path in *O(log n).*
- For comparison, the DP solution requires another O(n log n) pre-processing to update the structure and is ineffective for such dynamic updates.

https://visualgo.net/en/segmenttree



Dr Siba Haidar & Dr Antoun Yaacoub

Binary Indexed (Fenwick) Tree

Binary Indexed (Fenwick) Tree

- The Fenwick Tree is a useful data structure for implementing *dynamic cumulative frequency tables.*

- Example
  - we have test scores of m = 11 students
    f = *{2,4,5,5,6,6,6,7,7,8,9}* integer values ranging from [1..10].

  In the following:

  - The frequency of each individual test score ∈ [1..10]
  - The cumulative frequency of test scores ranging from [1..i] denoted by cf[i]—that is, the sum of the frequencies of test scores 1, 2, ..., i.

Binary Indexed (Fenwick) Tree

- *{2,4,5,5,6,6,6,7,7,8,9}*

Index/ Score	Frequency f	Cumulative Frequency cf	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	cf[1] = f[1] = 0.
2	1	1	cf[2] = f[1] + f[2] = 0 + 1 = 1.
3	0	1	cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1.
4	1	2	cf[4] = cf[3] + f[4] = 1 + 1 = 2.
5	2	4	cf[5] = cf[4] + f[5] = 2 + 2 = 4.
6	3	7	cf[6] = cf[5] + f[6] = 4 + 3 = 7.
7	2	9	cf[7] = cf[6] + f[7] = 7 + 2 = 9.
8	1	10	cf[8] = cf[7] + f[8] = 9 + 1 = 10.
9	1	11	cf[9] = cf[8] + f[9] = 10 + 1 = 11.
10	0	11	cf[10] = cf[9] + f[10] = 11 + 0 = 11.

Binary Indexed (Fenwick) Tree

- The cumulative frequency table can also be used as a solution to the Range Sum Query (RSQ) problem. It stores RSQ(1, i) $\forall$ i $\in$ [1..n] where n is the largest integer index/score.
  - In the example, we have n = 10, RSQ(1, 1)= 0, RSQ(1, 2) = 1, . . . , RSQ(1, 6) = 7, . . . , RSQ(1, 8) = 10, . . . , and RSQ(1, 10) =11.
  - when i $\neq$ 1 , RSQ(i, j) = RSQ(1, j) - RSQ(1, i - 1). Example: RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6.

Index/ Score	Frequency f	Cumulative Frequency cf
0	-	-
1	0	0
2	1	1
3	0	1
4	1	2
5	2	4
6	3	7
7	2	9
8	1	10
9	1	11
10	0	11

Binary Indexed (Fenwick) Tree

- Fenwick Tree operations are also extremely efficient as they use fast bit manipulation techniques.

- We will use the function `LSOne(i)` (which is actually (i & (-i))) .
- The operation (i & (-i)) produces the first Least Significant One-bit in i.

LSOne(i)

- To get the value of the least significant bit that is on (first from the right), use `T = (i & (-i))`.
- Example
  - i = 40 (base 10) = `000...000101000` (32 bits, base 2)
  - -i = -40 (base 10) = `111...111011000` (two's complement)
  - ---------- AND
  - T = 8 (base 10) = `000...000001000` (3rd bit from right is on)

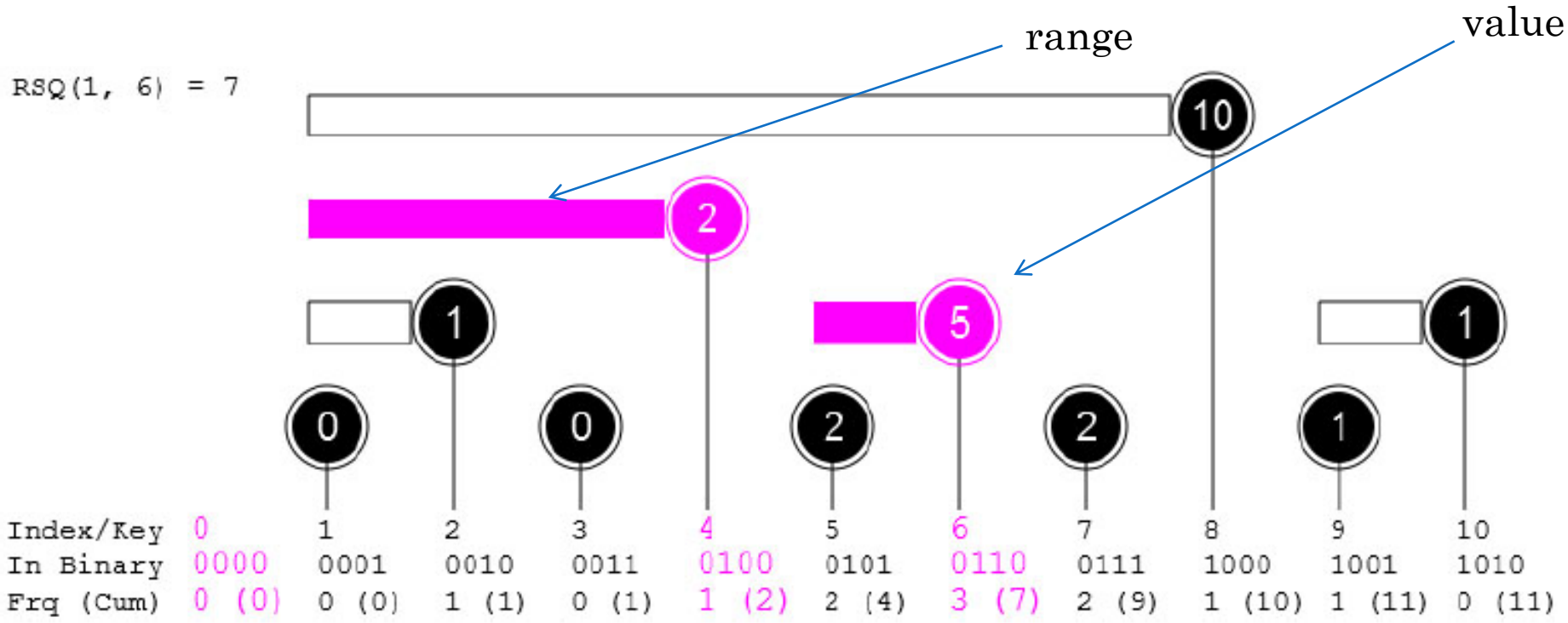
Implementation

- The Fenwick Tree is typically implemented as an array (vector).
- The Fenwick Tree is a tree that is indexed by the *bits of its integer keys.*
- These integer keys fall within the fixed range [1..n]—skipping index 0.

- In the previous table, the scores [1..10] are the integer keys in the corresponding array with size n = 10 and m = 11 data points.

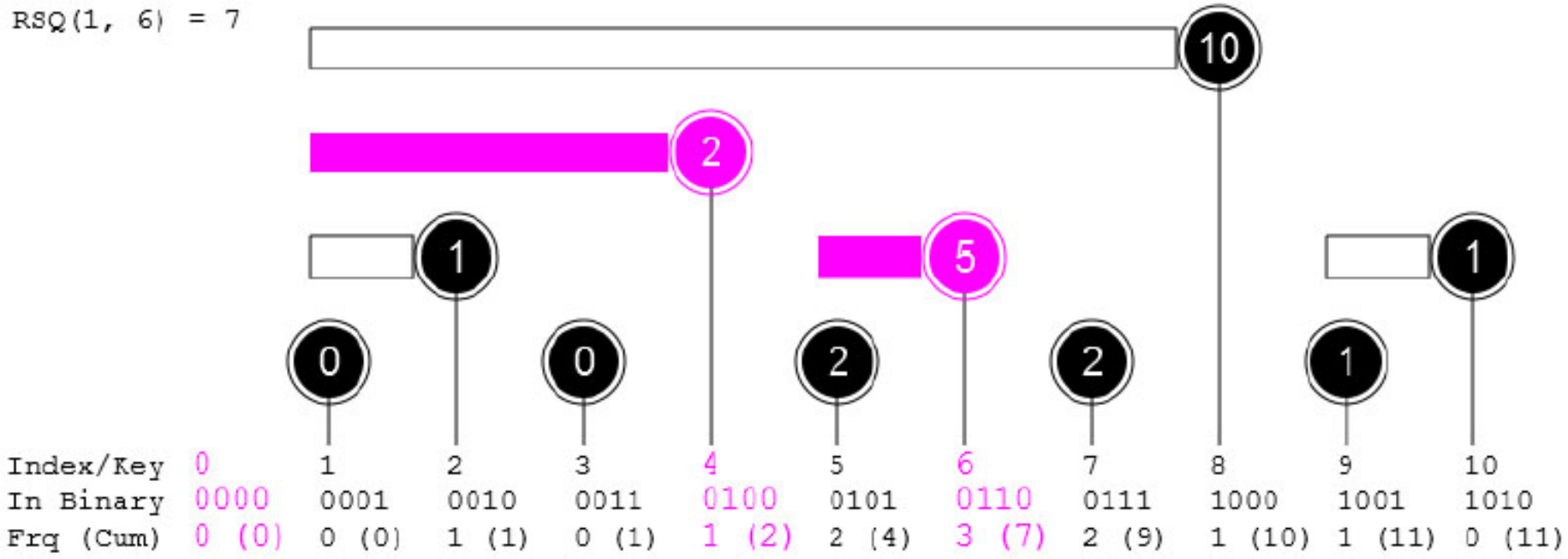
Implementation

- Let the name of the Fenwick Tree array be ft:
  - element at index *i* is responsible for elements in the range [i-LSOne(i)+1..i]
  - ft[i] stores the cf of {i-LSOne(i)+1,i-LSOne(i)+2,i-LSOne(i)+3, .., i}



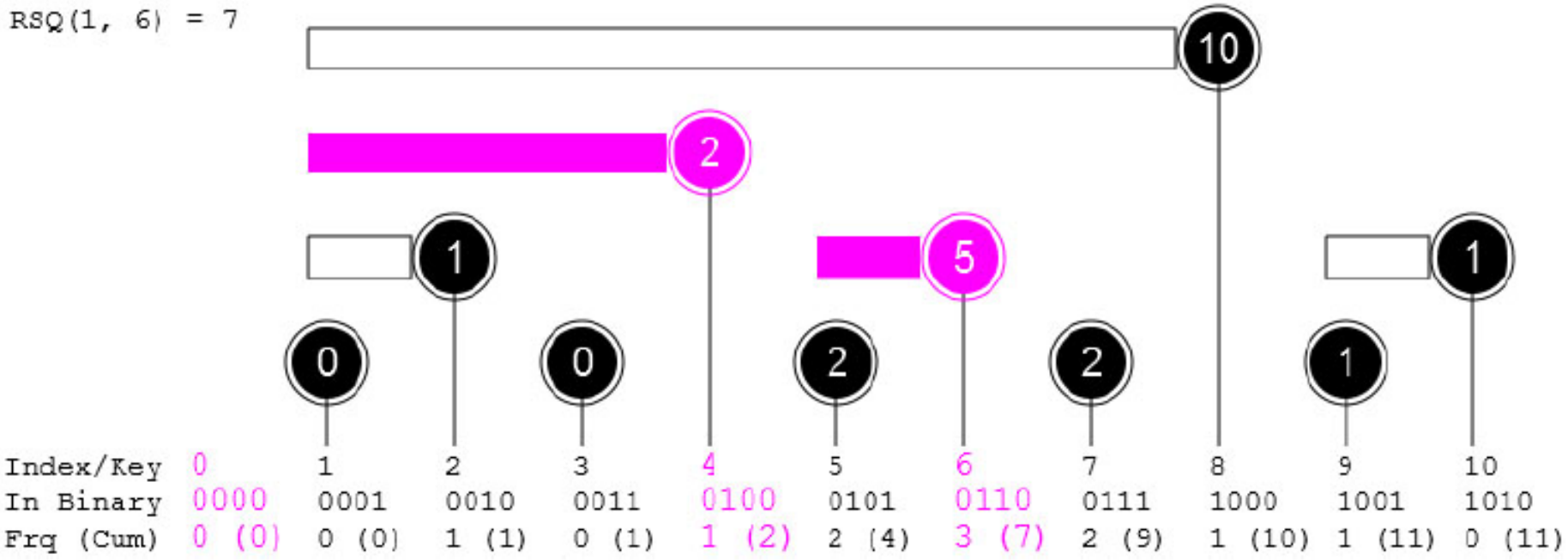
Implementation

- ft[4] = 2 is responsible for range [4-4+1..4] = [1..4]
  ft[6] = 5 is responsible for range [6-2+1..6] = [5..6]
  ft[7] = 2 is responsible for range [7-1+1..7] = [7..7]
  ft[8] = 10 is responsible for range [8-8+1..8] = [1..8]



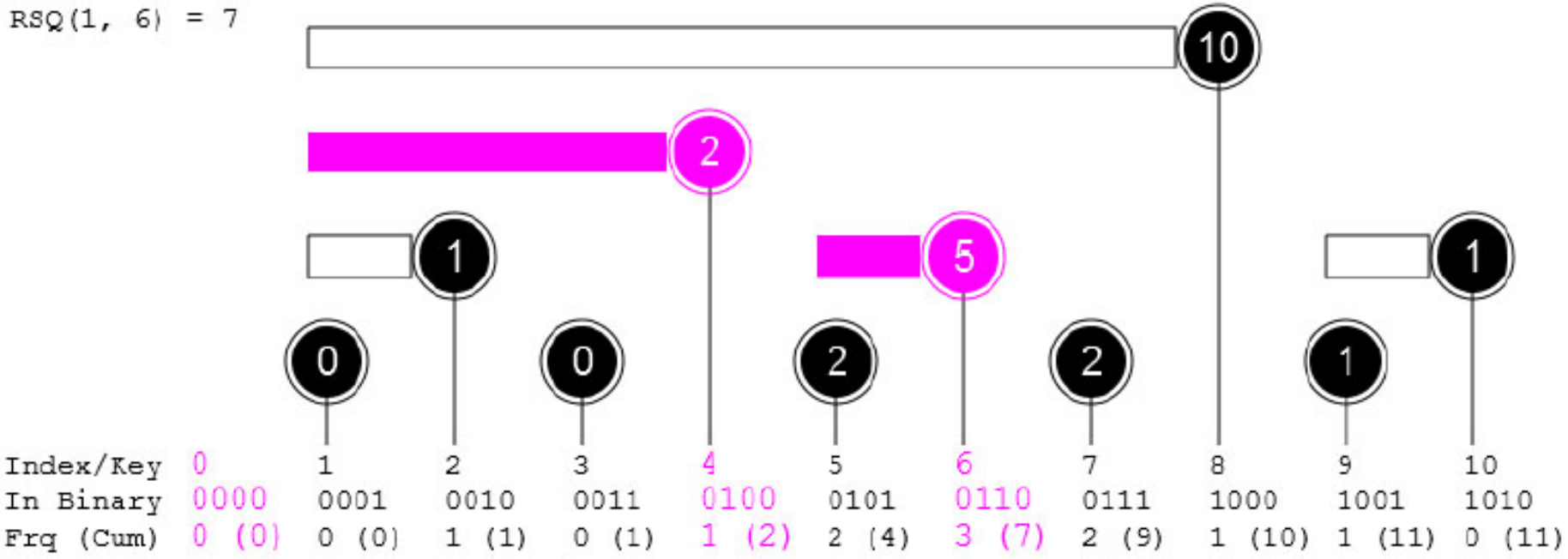
Implementation

- To obtain the cf between [1..b], i.e. rsq(b), we simply add ft[b], ft[b'], ft[b''], . . . until index b^i is 0.
  - This sequence of indices is obtained via subtracting the Least Significant One-bit via the bit manipulation expression: b' = b - LSOne(b).



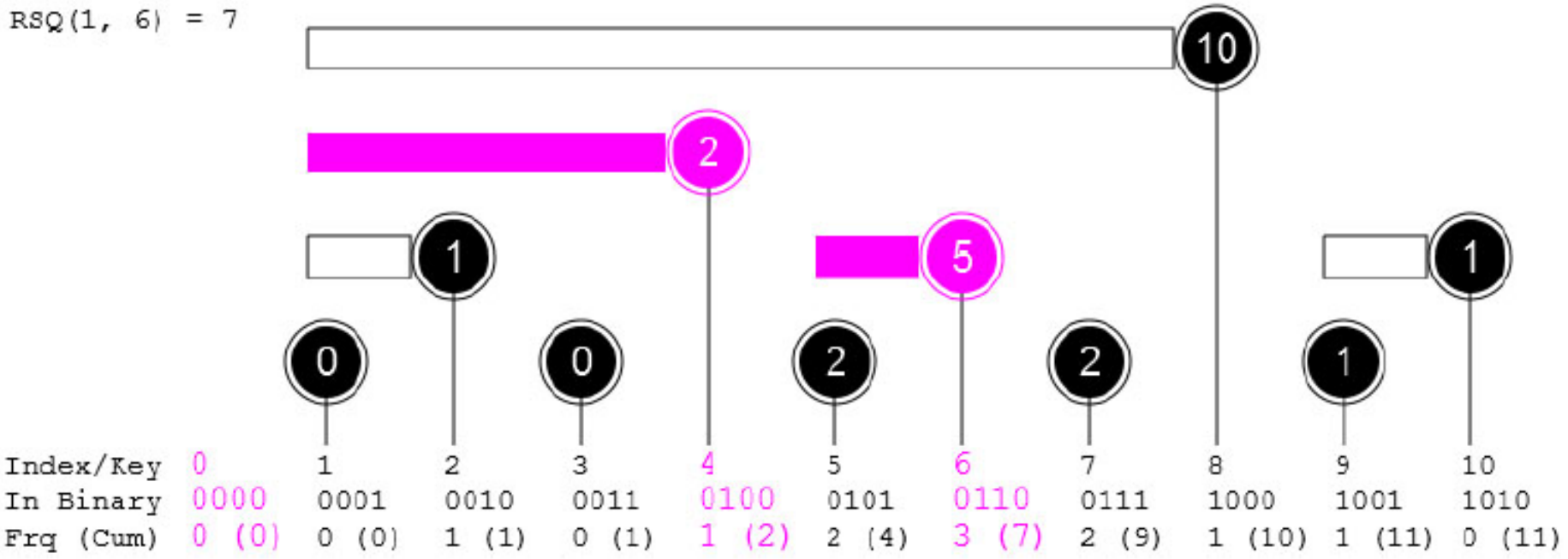
Implementation

- rsq(6) = ft[6] + ft[4] = 5 + 2 = 7. Notice that indices 4 and 6 are responsible for range [1..4] and [5..6], respectively.
  - The indices 6, 4, and 0 are related in their binary form: b = 6_{10} = $(110)_2$ can be transformed to b' = 4_{10} = $(100)_2$ and subsequently to b'' = 0_{10} = $(000)_2$.



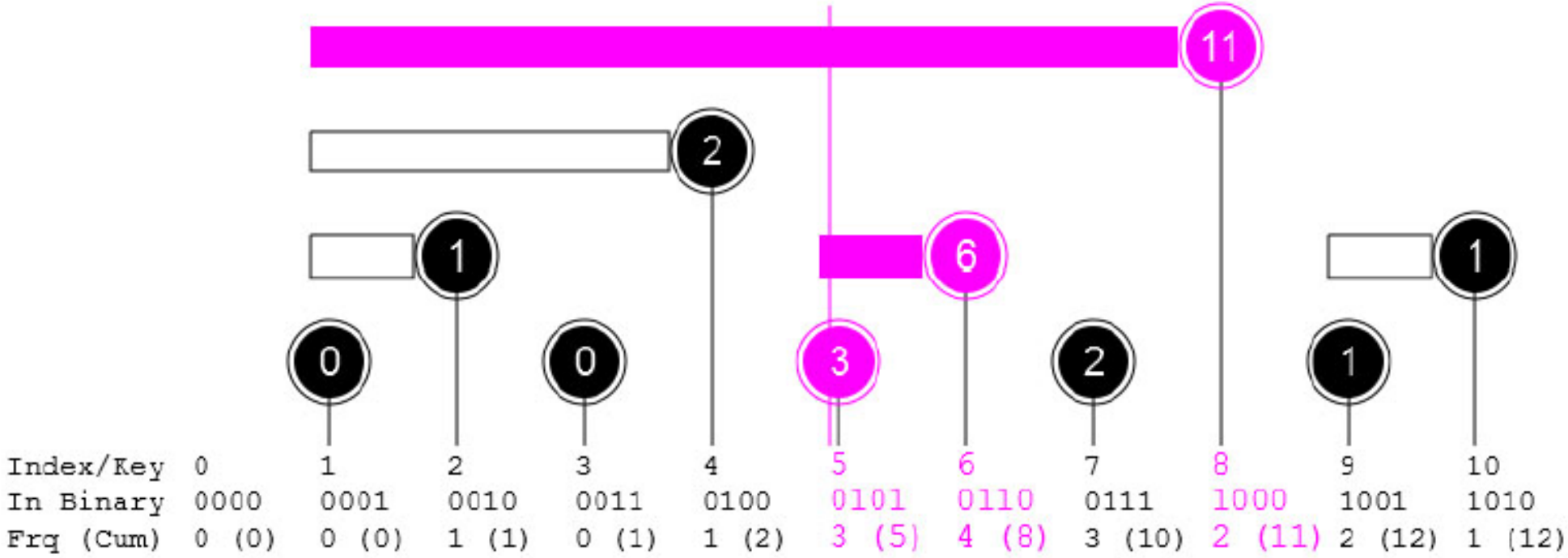
Implementation

- cf between [a..b] where a != 1 is simple, evaluate rsq(a, b) = rsq(b) - rsq(a - 1).
- rsq(4, 6), we can simply return rsq(6) - rsq(3) = (5+2) - (0+1) = 7 - 1 = 6.



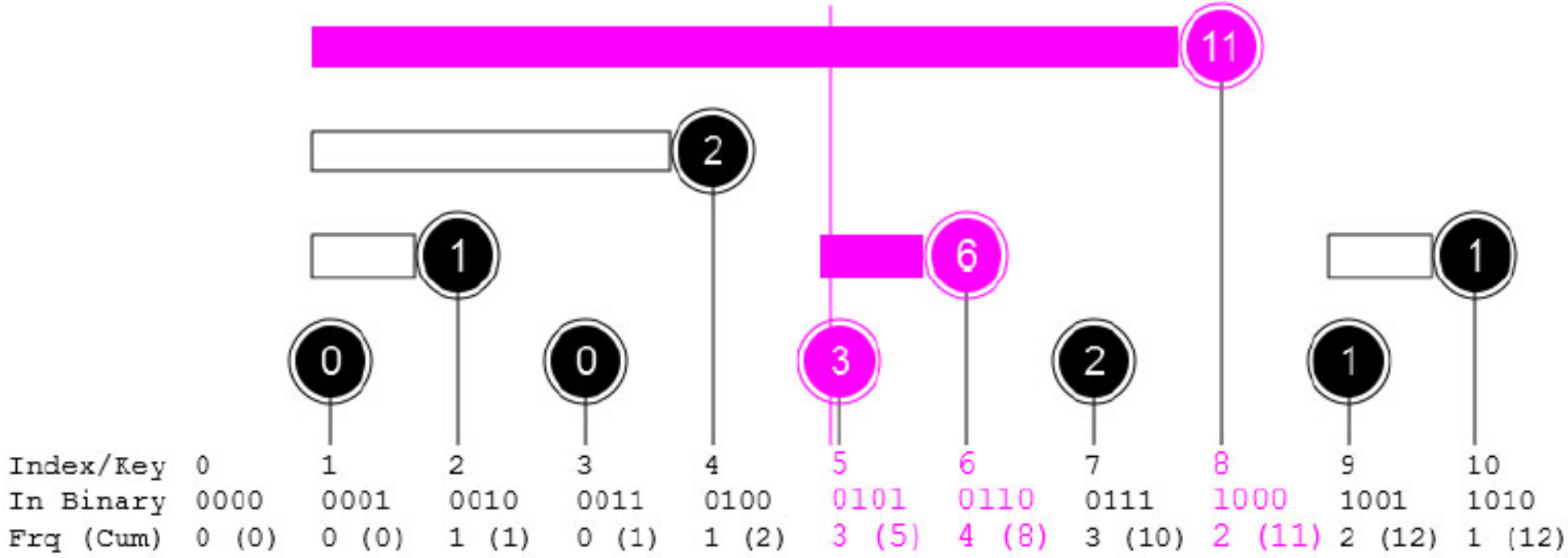
Implementation

- When updating the value of the element at index k by adjusting its value by v (note that v can be either positive or negative), i.e. calling adjust(k, v), we have to update ft[k], ft[k'], ft[k''], . . . until index k^i *exceeds n.* k' = k + LSOne(k).



Implementation

- adjust(5, 1) will affect (add +1 to) ft[k] at indices k = $5_{10} = (101)_2$, k' = $(101)_2 + (001)_2 = (110)_2 = 6_{10}$, and k'' = $(110)_2 + (010)_2 = (1000)_2 = 8_{10}$ via the expression given above.



Implementation

```
class FenwickTree {
  private Vector<Integer> ft;

  private int LSOne(int S) { return (S & (-S)); }

  public FenwickTree() {}

  // initialization: n + 1 zeroes, ignore index 0
  public FenwickTree(int n) {
    ft = new Vector<Integer>();
    for (int i = 0; i <= n; i++) ft.add(0);
  }

  public int rsq(int b) {                              // returns RSQ(1, b)
    int sum = 0; for (; b > 0; b -= LSOne(b)) sum += ft.get(b);
    return sum; }

  public int rsq(int a, int b) {                       // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }

  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                        // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft.set(k, ft.get(k) + v); }
};
```

https://visualgo.net/en/fenwicktree