Programming Challenges (GB21802)

Week 2 - Data Structures

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

2020/5/12
(last updated: May 10, 2020)

Version 2020.1

Lecture 02 – Data Structures
Part I – Introduction

Outline

When writing any program (and not just programming challenges!) the right data structure makes a great difference in how easy the program is to write and how time and memory efficient the algorithm is.

- In this lecture, we will review some data structures that commonly appear in programming challenges;
- This lecture covers Chapter 2 of the "Competitive Programming" book;
- In this lecture, we focus the **description and implementation** of data structures more than on their theoretical analysis.

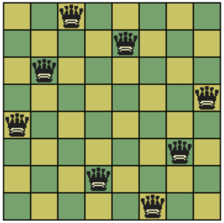
Comments on the last week:

- Numbers of Problems solved;
- Orders of Problems;
- Lecture Calendar for this week;

Motivating Problems

To introduce the topics of this class, let's show three problems where the choice of data structure can make a big difference;

Example 1: 8 Queen Problem (UVA 750)



For a board of size $n \times n$, you have to find how many safe configurations of n queens exist.

Because you need to count how many configurations exist, it is necessary to test **all** valid configurations.

```
for (int i = 0; i < #configurations; i++)
  if configurationIsSafe(i) sum++
return(sum)
```

Example 1: 8 Queen Problem (UVA 750)

Last lecture we talked about how **pruning** can be used to reduce the problem size. This time we review this concept more concretely.

Consider how we store information about all the configurations. Imagine that we have an array, *conf*, which contains all configurations that we want to test.

Approach 1: For each queen, we store the pair (col, row).

```
conf[0]   = {{a,1}, {a,1}, {a,1}, ... {a,1}, {a,1}}
conf[1]   = {{a,1}, {a,1}, {a,1}, ... {a,1}, {a,2}}
conf[2]   = {{a,1}, {a,1}, {a,1}, ... {a,1}, {a,3}}
  ...
conf[k]   = {{a,1}, {b,2}, {b,2}, ... {c,8}, {d,8}}
conf[k+1] = {{a,1}, {b,2}, {b,2}, ... {c,8}, {e,1}}
  ...
```

Looping through all options: n^{n^2} steps

Example 1: 8 Queen Problem (UVA 750)

Approach 2: We fix each queen on a column (a,b,c,d...). Our data structure only needs to represent the row of each queen.

We store an array of arrays, containing 8 integers representing the row:

```
conf[0]   = {0,0,0,0,0,0,0,0}
conf[1]   = {0,0,0,0,0,0,0,1}
conf[2]   = {0,0,0,0,0,0,0,2}
  ...
conf[k]   = {0,0,0,3,3,6,7,7}
conf[k+1] = {0,0,0,3,3,7,0,0}
  ...
```

Looping through all options: n^n steps

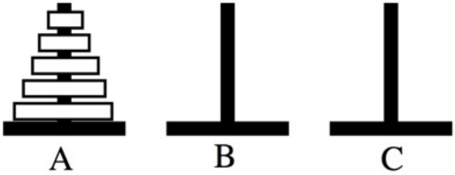
Example 1: 8 Queen Problem (UVA 750)

Approach 3: We fix each queen on a column (a,b,c,d...), and each configuration is a permutation of rows where we place the queens.

We store a string of rows, and each configuration is a permutation accessed using "next_permutation" function from C++ stl's "algorithm" header.

```
conf[0] = "01234567"
conf[1] = "01234576"
conf[2] = "01234657"
  ...
```

Example 2: The Towers of Hanoi

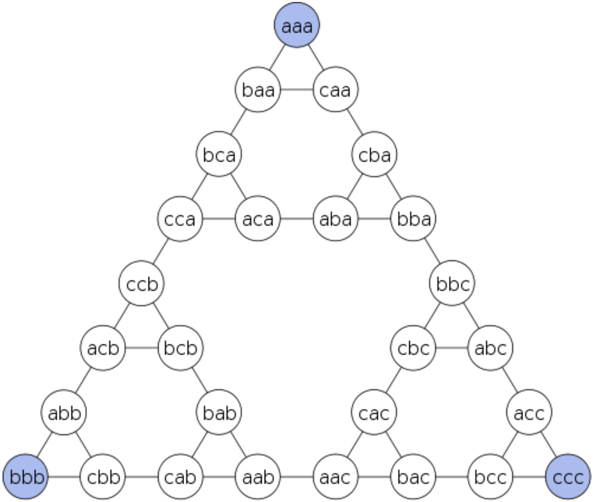


- You have N disks and K poles. Each disk has unique size s_i.
- A disk i can be moved from one pole to another.
- A move of disk i to pole k is only valid if k has no disks smaller than i
- Find the list of moves to move all disks from pole 1 to pole K.

How do you represent the data in this problem?

Example 2: The Towers of Hanoi

A string with “n” disks, from smaller to larger.



Example 3: Army Buddies (UVA 12356)
Problem Description

- There is a line of S soldiers: $0, 1, 2, 3, 4, ..., S$
- There are Q queries that remove soldiers from i to j:

```
Q1: 2,4          (removes soldiers 2, 3, 4)
Q2: 6,7          (removes soldiers 6, 7)
Q3: 1,1          (removes soldier 1)
```

- For each query, list the soldier to the left and to the right

```
A1: 1,5      1 x x x 5 6 7
A2: 5,*      1 - - - 5 x x
A3: *,5      x - - - 5 - -
```

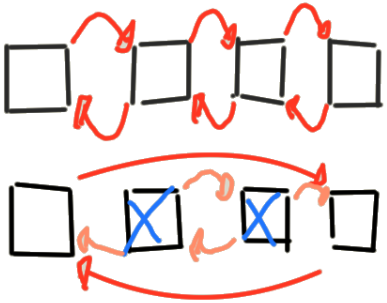
How do we solve this problem?

Example 2: Army Buddies (UVA 12356)

Idea 1: Linked Lists

For each query, we find the first soldier, and we remove each soldier until we find the second soldier.

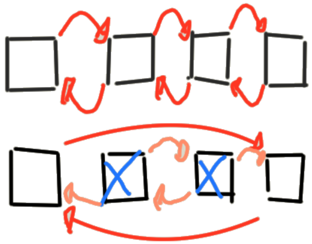
We use the linked list to reduce the size of the list after each query.



- Represent the line as a linked list.
- Find the 1st soldier and 2nd soldiers ($O(n)$ steps)
- Repeat the operation above for each query. ($O(nm)$ steps)

Example 2: Army Buddies (UVA 12356)

A solution using linked lists



Problem! The input is too big, and $O(nm)$ takes too much time.

- $1 \leq S \leq B \leq 10^5$; $(O(10^5 \times 10^5)) = 10^{10}$
- Also multiple cases; $(O(n^2k)) = 10^{10}k$

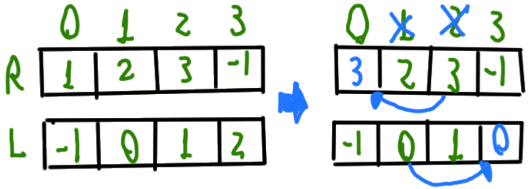
Let's think of a different solution! (Before looking at the next slide)

Example 2: Army Buddies (UVA 12356)

A solution using arrays

The problem with last solution is that it costs n to search the soldiers. We need to access the sodier position in $O(1)$ using an **index**. We also need to keep track of neighbors when removing soldiers.

- Idea: To use **two** Neighbor Arrays
  - Let **R** be: **Int** Array of Right neighbors
  - Let **L** be: **Int** Array of Left neighbors



- Question: how do we update R and L after query (r, l)?

Motivating Data Structure

As you can see, the choice of data structure and problem representation is very important.

- Choosing the right data structure:
  - Changes the time or memory complexity of the implementation;
  - Makes the programming task simpler or more complex;
- Hints for programming contests;
  - Avoid using pointers (source of bugs, programming overhead);
  - Prefer multiple variables, instead of complex structs;
  - In larger programs (not challenges) you want more complex structures;
- Learn the library tools of your language (STL, java.utils, etc);

End of part I

Lecture 02 – Data Structures
Part II – The Array Data Structure

Introducing the simple array!

Arrays are the simplest data structure, but also the ones most often used for programming challenges.

Merits

- They are easy to implement and manipulate (no pointers);
- Random access is usually very fast;
- Pointers can be *simulated* using index operations;
- Many library functions for array manipulation;

Concerns

- Inserting many items in the middle of an array can be expensive;

Implementing arrays/vectors (C++)

```
#include <vector>

int arr[5] = {7,7,7};       // arr = {7,7,7,0,0}
vector<int> v(5, 5);        // v = {5,5,5,5,5}

int x = arr[2] + v[2];      // x = 12

arr[5] = 5;                 // Runtime error
cout << v[7];               // 0 !! Be careful.

v.push_back(6);             // v = {5,5,5,5,5,6}
```

Trying to access indexes outside of an array is a common source of Runtime Errors (RTE)

How do you reset an array?

Implementation matters

```
#include <vector>
#include <string.h>
vector<int> v(10000,7)

memset(v, 0, 10000*__SIZEOF_INT__);          // Method 1
fill(v.begin(), v.end(), 0);                 // Method 2
for (int i = 0; i < 10000; i++) v[i] = 0;    // Method 3
v.assign(v.size(), 0);                       // Method 4
```

```
Method       |  executable size    |  Time Taken (in sec) |
             |  -O0     |  -O3     |  -O0       |  -O3    |
-------------|----------|----------|------------|---------|
1. memset    | 17 kB    | 8.6 kB   | 0.125      | 0.124   |
2. fill      | 19 kB    | 8.6 kB   | 13.4       | 0.124   |
3. manual    | 19 kB    | 8.6 kB   | 14.5       | 0.124   |
4. assign    | 24 kB    | 9.0 kB   | 1.9        | 0.591   |
```

Operations in Arrays

Problem Example

Example – Vito's Family (UVA 10041)

Vito wants to move to an address that is closest to his entire family.

Input: A list of integers (street addresses):
10, 20, 10, 10, 40, 80, 30, 90, 20, 55, 20

Output: The address (integer) with minimal distance to all others.

- **10**: $0 + 10 + 0 + 0 + 30 + 70 + 20 + 80 + 10 + 45 + 10 = 275$
- **40**: $30 + 20 + 30 + 30 + 0 + 40 + 10 + 50 + 20 + 15 + 20 = 265$
- **20**: $10 + 0 + 10 + 10 + 20 + 60 + 10 + 70 + 0 + 35 + 0 = 225$

Result: 20!
How would you solve this problem?

Operations in Arrays

Problem Example

- The solution to this problem is to find de **median** address.
- 1- sort the address array, 2- select the middle value.

```
#include<iostream>
#include<algorithm>
using namespace std;

int main() {
    int n; int add[100];
    cin >> n;
    for (int i=0; i<n; i++) { cin >> add[i]; }

    sort(add, add+n);
    cout << add[n/2] << endl;
}
```

Operations in Arrays
Sorting

In the last problem example, we used sorting to calculate the median. In fact, you can **solve many, many problems using sorting**.

Some examples:

- Finding the Highest n values, Finding duplicate values;
- Binary Search ($O(\log n)$)
- Pre-processing data for other algorithms.

Operations in Arrays

The "algorithm" header: sorting and binary search

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int n, t, search; vector<int> v;
  cin >> n >> search;
  for (int i=0; i<n; i++) { cin >> t; v.push_back(t); }

  sort (v.begin(), v.end());
  vector<int>::iterator low,up;
  low = lower_bound (v.begin(), v.end(), search);
  up  = upper_bound (v.begin(), v.end(), search);
  cout << (low-v.begin()) << " and " << (up-v.begin());
}
```

Operations in Arrays

Sorting with specific funtions

In some cases, you need to do a complex sort on several variables.

```
#include <algorithm>
#include <vector>
#include <string>
struct team{ string name; int point; int penal;
             team(string _n, int _po, int _pe) :
               name(_n), point(_p), penal(_g){} };

bool cmp(team a, team b) {      % Sorting Function
  if (a.point != b.point) return a.point > b.point;
  if (a.penal != b.penal) return a.penal < b.penal;
  return strcmp(a.name,b.name); }

vector<team> v;
sort(v.begin(), v.end(), cmp); // sort using cmp
reverse(v.begin(), v.end()); // and reverse
```

Lecture 02 – Data Structures
Part III – Data Structures from Libraries

Learn your data structure libraries

- The standard C++ library (STL) implements many useful data structures;
- By using data structures from the STL, you make your program simpler and avoids bugs;

- In this section, we will review some of the data structures used most often.
- We will focus on usage, rather than theory.

- For more information about how these data structures work, we recomment the the website `https://visualgo.net/`;

Vector variations: Deque, Queue, Stack

While the *vector* is a general and useful data structure, for some applications you will want special access to the **start** and **end** of a vector.

- stack: *pop* and *push* from the front of the vector;
- queue: *pop* from the back, *push* from the front;
- deque: *pop_front, push_front, pop_back, push_back*;

Behind C++

Actually, *Queue* and *Stack* are high level constructs, List or Deque are used to implement them.

Queue and Stacks

Queues and Stacks are useful to simplify common cases of vectors

Stack Example: Test if a set of parenthesis is balanced.

```
#include <stack>
stack<char> s;
char c;

while(cin >> c) {
  if (c == '(') s.push(c);
  else {
    if (s.size() == 0) { s.push('*'); break; }
    s.pop();
  }
}
cout << (s.size() == 0 ? "balanced" : "unbalanced");
```

Maps and Sets

Problem Example: CD – 11849

Jack and Jill are comparing their CD collections. The problem wants to know: How many CDs are in the both collections?

Input:

- Jack CD collection: List of 10^6 CD IDs
- Jill CD collection: List of 10^6 CD IDs
- Each CD ID is an integer from 1 to 10^9

Output:

- Number of CD IDs that appear in both collections.

Easy problem, right?

Maps and Sets

Problem Example: CD – 11849

Naive Solution:

1. Store all IDs in collection 1 in a Vector (n steps)
2. Sort the Vector (nlogn steps)
3. For each ID in collection 2, use Binary Search to find it in Vector 1 (nlogn steps)

Total Cost: $n + n\log n + n\log n$

Unfortunately, this is not fast enough for the time limit of this problem. Using the right data structure, you can reduce the time cost to $n + \log n$.

Solving CD with a map data structure

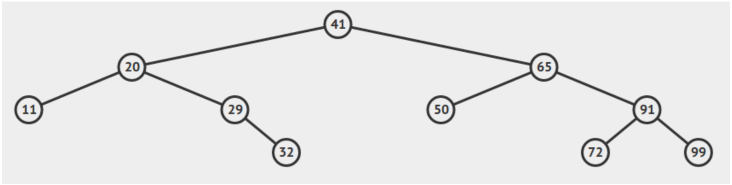
Approximate Solution

The STL provides **set**, which is a data structure that uses balanced search trees to allow finding items in $\log n$

```
#include <iostream>
#include <set>
using namespace std;

int main() {
  int N, M, num; cin >> N >> M;
  set<int> first;
  while (N--) { cin >> num; first.insert(num); }
  int count = 0;
  for (int i = 0; i < M; i++) {
    cin >> num;
    if (first.find(num) != first.end()) ++count; }
  cout << count << '\n';
}
```

Balanced Search Trees



- *Search Trees* Keep items in an ordered relationship.
- For example: Left children always have smaller values, Right children always have larger values;
- Insertion/Search/Deletion in a tree costs $O(h)$, where h is the height of the tree;
- For a tree with n elements, the minimum height is $\log n$
- For a balanced tree, the maximum height is also $\log n$
- How to keep the tree balanced?

Balanced Search Trees

How to keep the tree balanced?

There are many Tree implementations/algorithms for keeping an BST balanced, and minimizing the tree height efficiently:

- AVL Tree (Adelson-Velskii-Landis);
- Red-Black Tree;
- B-Tree;
- Splay Tree;

However, in for short programs (such as programming challenges) implementing these trees from scratch is a good way to create **bugs.**

Luckly, most standard libraries include some implementation of BST.

ABLs in C++: Map and Set

- In C++, the *Map* and *Set* classes are implemented using BSTs
- *Map* Accept Key-value pairs;
- *Set* Accepts only Keys;

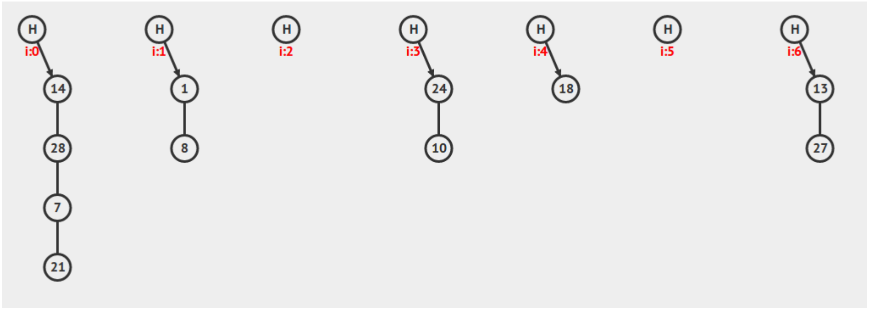
Using Map in C++

```
#include <map>
map<string, int> ages; ages.clear();

ages["john"] = 40;
ages["billy"] = 39;
ages["andy"] = 29;
ages["steven"] = 42;
ages["felix"] = 33;

  // What is the age of andy?
map<string, int>::iterator it = ages.find("andy");
cout << it->second << endl;
  // Which names are between "f" and "m" ??
for (map<string, int>::iterator it =
     age.lower_bound("f");                 // finds felix
     it != age.upper_bound("m"); it++)   // finds johm
        cout << " " << ((string)it->first).c_str();
```

Hash Tables



- Insertion and Search: O(1) – Slow iteration;
- C++ library: *std::unordered_map*;
- Hash parameter – Defines Collision results.
- Learn more about hash tables here: `https://visualgo.net/ja/hashtable`

Lecture 02 – Data Structures
Part IV – Hand-made Data Structures

Hand-making Data Structures

For certain problems, it is necessary to extend existing data structures;

- Extensions of Arrays and Vectors for complex data;
- New features for indexing and/or searching;
- Express special relationship between data items (ex: graphs)

We will examine two data structures now: UFDS and Segment Tree

Union-Find Disjoint Set (UFDS)

Motivating Problem

Network Connections – UVA793

Imagine a network with *n* computers, some are connected to others.

Input: A series of "commands"

- c i j – New Info: Computer *i* is connected to computer *j*
- q i j – Query: Is computer *i* connected to computer *j*?

Output: The number of "q" with answer yes, and the number of "q" with answer no.

Union-Find Disjoint Set (UFDS)

Motivating Problem – Naive answer

Neighborhood Graph

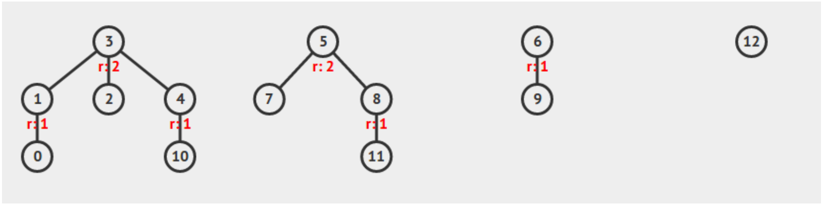
- Initialize an $n \times n$ matrix with zeros.
- For every "c i j" input, $N_{i,j}$ and $N_{j,i}$ becomes 1.
- For every "q i j", we perform a breadth first search on the graph.

How good is this solution?

- Cost to insert a new connection: $O(1)$
- Cost to check if "q i j": $O(V + E)$

We can do better!

Union-Find Disjoint Set



- The UFDS keeps sets of items, each is represented by a parent;
- When you join two sets You join their parents;
- When you test the parent of an item You flatten the tree;
- Test_item and Join_item are both O(1); *(amortized)*
- Visualization: `https://visualgo.net/ja/ufds`;

UFDS Implementation using Arrays

```
int p[MAX], r[MAX];
                              # which groups x belong to?
int find(int x) { return x == p[x] ? x : p[x]=find(p[x]); }

int join(int x, int y) {      # x and y are the same group
    x = find(x), y = find(y);
    if(x != y) {
        if(r[x] < r[y])      { p[x] = y; r[y] += r[x]; }
        else                 { p[y] = x; r[x] += r[y]; }
        return 1;
    }
    return 0;
}

void init() { # Initialize each element as separate group
    for(int i = 0; i < MAX; i++)  { p[i] = i; r[i] = 1; }
}
```

Union Find Disjoint Set

Problem II – War

From a set of 10k people, some are friends, other are enemies.

- If A,B are friends, and B,C are friends, then A,C are friends
- If A,B are friends, and B,C are enemies, then A,C are enemies
- If A,B are enemies, and B,C are enemies, then A,C are friends

Input: A series of commands from the set below:

- SetFriends(i,j) SetEnemies(i,j)
- TestFriends(i,j) TestEnemies(i,j)

Output:

- If a “SetFriends” or “SetEnemies” is impossible, output “-1”
- For a “TestFriends”, “TestEnemies”, output 0 - false, 1 - true

Union Find Disjoint Set

Problem II – War

This problem is similar to “Networking”, but now you need to keep track of **TWO** relations: Friends and Enemies.

There are different ways to implement this:

- Create one UFDS for friends, and one UFDS for enemies?
- Add a “friend/enemy” flag for each person?

What other ideas can you think? Which one is easy/hard to implement?

Range Maximum Query – RMQ

Suppose you have an array of values:

```
Value: 18 17 13 19 15 11 20
Index:  0  1  2  3  4  5  6
```

The Range Maximum Query problem asks you to find the index with the maximum value between two indexes:

- RMQ(0,0) = 0
- RMQ(0,6) = 6
- RMQ(1,4) = 3

Naive Method: loop from i to j, find maximum value. $O(nk)$ steps

But what is the number of **Values (n)** or **Queries (k)** is too big?

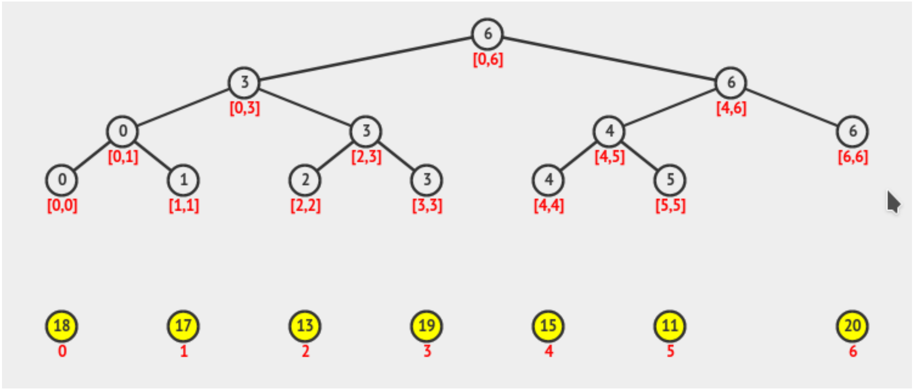
Segment Tree

- **Basic idea**: Binary tree with the max index in of each subtree.
- **Operation Costs:**
  - Creation of the tree: $O(n)$
  - Query of a segment: $O(\log n)$
  - Update of the tree: $O(\log n)$

Important Part

- There are many implementations. We will show a vector based heap.

Segment Tree



Segment Tree animation at VISUALGO:
`https://visualgo.net/en/segmenttree`

Coding the Segment Tree

Tree Creation

```
typedef vector<int> vi; // Summarizing types that we use often.

class SegmentTree { // OOP implementation,
private: vi st, A;  // vi: typedef vector<int> vi;
  int n;
  int left (int p) { return  p<<1; }  // heap-like index;
  int right(int p) { return (p<<1) + 1; }

  void build(int p, int L, int R) {   // O(n log n)
    if (L == R)
      st[p] = L;                      // store the index
    else {                            // recursive build
      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;
  } }
```

Coding the Segment Tree

Query the Tree

rmq(1, 0, n-1, i, j) – Query from i to j.

```
int rmq(int p, int L, int R, int i, int j) // O(log n)
{
  if (i >  R || j <  L)
    return -1;     // outside query range
  if (L >= i && R <= j)
    return st[p]; // inside query range

  // compute the min position in the left and right part
  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;    // segment outside query
  if (p2 == -1) return p1;    // segment outside query
  return (A[p1] <= A[p2]) ? p1 : p2;
}
```

Coding the Segment Tree

Update the Tree

update(1, 0, n-1, i, v) – update index i to value v

```
int update(int p, int L, int R, int idx, int new_value) {
  int i = idx, j = idx;    //for point update i = j = idx
  // if the curr interval does not intersect the update,
  if (i > R || j < L) return st[p];  //return node value!
  // if the current interval is in the update range,
  if (L == i && R == j) {
    A[i] = new_value;     // update the underlying array
    return st[p] = L;     // this index
  }
  // compute the min pos in L/R part of the interval
  int p1, p2;
  p1=update(left(p) , L        , (L+R)/2, idx, new_value);
  p2=update(right(p), (L+R)/2+1, R      , idx, new_value);
  // return the position where the overall minimum is
  return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
}
```

Conclusion

- The choice of data structure and its implementation has great influence in the algorithm used to solve a programming challenge.
- It is important to be familiar with the main data structures of your programming language (arrays, matrices), and their utility functions.
- For those structures not available in the library, I recommend that you create a "library" of code you have created. It will come in handy time and time again in your career!

About these Slides

Image Credits I

[Page 5] Chessboard by Lee Daniel Crocker. CC-BY-SA 3.0
[Page 10] Tower of Hanoi's graph image by nonenmac
[Page 48] Segment Tree Code from
`https://github.com/stevenhalim/cpbook-code`