ALGO.API

PREPARED BY

KUNAL BHATIA

INTRODUCTION

As of today there are 18.2 million software developers worldwide, these developers build products and algorithms for usage in systems.

However in various programming and related disciplines, they encounter a very significant problem.As they have to write thousands of lines of code, time complexity becomes an important factor and so does writing the same functions time and again.

We present a novel and integrated solution to this problem in the form of an easily available and easy to use API that contains the shortest possible and optimised codes to reduce time complexity.All these features are integrated into a single header file to be included by the programmer.

INSPIRATION

Our Motivation

As school students greatly interested in computers and programming.We used to attend various competitive programming competitions.This is where we encountered the problem Upon looking further we realised that this problem was not just limited to such competitions but also posed an issue in the landscape of systems built on similar technologies.

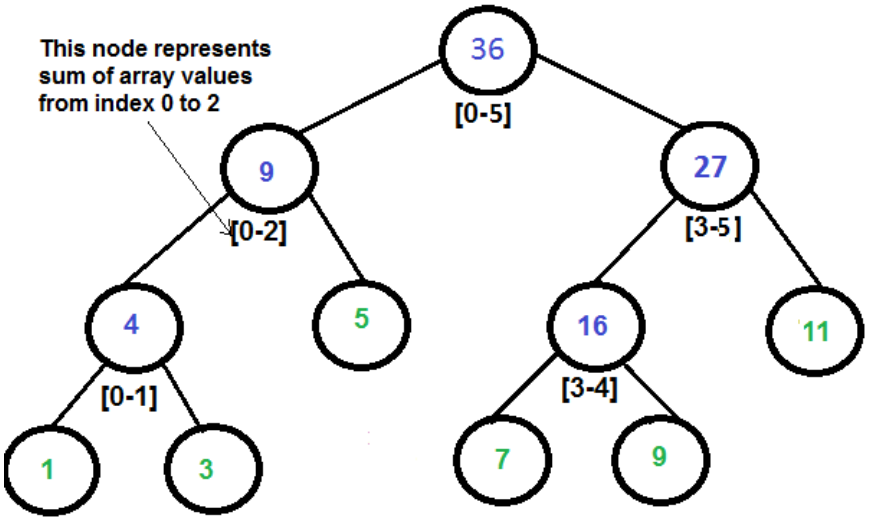
We set out to solve this problem, after arduous researching and development we finally came with our product Algo.api.

BUILDING

In order to build our project further, we decided to first make prototypes of a few functions and include them in our library.

the functions we first chose were graphing functions.Functions such as segment trees, this was because .these functions can be optimised in a variety of fashions.

There were some options that we could explore such as the A* and IDA* algorithms but for the sake of simplicity we decided to start with segment trees.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Current Functions

FUNCTIONS

the library currently includes four segment tree functions and one Dijkstra's shortest path function.These have the least time complexity, the code for these is included in the prototyping section.

Segment Tree
- Build
- Upgrade
- Query
- Merge

Dijkstra's Shortest Path

FUTURE AND APPLICATION

Currently we are adding more and more algorithms to this library and corroborating it with the latest methods of reducing time complexity and exploring more optimised methods as they come up.We are working on over 50 algorithms to be added to our library.

The applications for this software are many, but primarily include reducing time complexity in systems that deal with large computations and need to have the most optimised approaches available from accessible and trusted sources,

PROTOTYPING

Our algorithms library can be found attatched on the platform, it is named **algo.hpp.** The code of the primary graphing functions are also provided below

Segment Tree Build

The build function intializes the segment tree. Each node in the tree contains three things:-
1. A value
2. A left pointer
3. A right pointer
The value in the node is represents the merger of all the nodes in it's subtree, which are pointed by its left and right children. The base case or the leaf nodes of our segment tree contain the original values stored in our array.

```
void build_r(T a[], int l, int r, node *cur) {
  if ( l == r ) {
     cur->val = a[l];
  }
  else {
     cur->left = new node;
     int mid = (l+r)/2;
     build_r(a,l,mid,cur->left);
     cur->right = new node;
     build_r(a,mid+1,r,cur->right);
     t->val = merge( t->left->val , t->right->val );
  }
}
```

Segment Tree Update

The update function is used to change values in the segment tree. For the position where the value has to be changed, all the values on the nodes will change from it's path to the root node which contains the merger of the whole array

```
void update_r(int pos, T x, int l, int r, node *cur){
  if ( l == r && l == pos ) {
    cur->val = x;
  } else {
    int mid = (l+r)/2;
    if ( pos <= mid ) {
      update_r(pos,x,l,mid,cur->left);
    } else {
      update_r(pos,x,mid+1,r,cur->right);
    }
    cur->val = merge( cur->left->val , cur->right->val );
  }
}
```

Segment Tree Query

The query function retrieves a value from the segment tree. Whenever we want the merger of l to r, there are only 3 cases possible.
1. The range lies in the left half, so we query for that.
2. The range lies in the right half, so we query for that.
3. The range lies in between, so we divide it into two parts and return the merger of the query of those two parts.

```
T query_r( int lx, int lr, int l, int r,node *cur){
  if ( lx == l && lr == r) {
    return cur->val;
  } else {
    int mid = (l+r)/2;
    if( lr <= mid ) {
      return query_r(lx,lr,l,mid,cur->left);
    } else if ( lx > mid ) {
      return query_r(lx,lr,mid+1,r,cur->right);
    } else {
      return merge( query_r(lx,mid,l,mid,cur->left) ,
query_r(mid+1,lr,mid+1,r,cur->right) );
    }
  }
}
```

Segment Tree Merge

By default, we have made the sum segment tree, but this can easily be adjusted according to use by overwriting the merge function in the code.

```
T merge(T a, T b){
return a + b;
}
```

Dijkstra's shortest path

Dijkstra's shortest algorithm gives the shortest path from a single node in the graph to every other node. The function takes the adjacency list in the form of vector, the number of vertices, source from which the shortest paths are to be calculated, and the array dist where the path lengths would be stored.

We maintain a min. priority queue in the format (distance,node) and initially push the source.
For every element in the priority queue, we visit it's neighbours, check if the distance from the source to the element + the distance between element and it's neighbour is shorter than the distance from the source to the neighbour.
We update the distance and push the neighbour in the priority queue

```
void dijkstra(std::vector< std::pair >graph[], int v, int source, int dist[]){
  std::priority_queue< std::pair , std::vector< std::pair > , std::greater<
std::pair > > pq;
  for(int i=1; i<=v; i++) dist[i] = INT_MAX;
  int vis[v+1] = { };
  dist[source] = 0;
  pq.push(std::make_pair(dist[source],source));
  while(!pq.empty()){
    int node = pq.top().first , cur = pq.top().second;
    pq.pop();
    if(dist[node] == cur && !vis[node]){
      for(int i=0; i < graph[node].size(); i++){
        int v = graph[node][i].first, d = graph[node][i].second;
        if(cur + d < dist[v]){
          dist[v] = cur + d;
          pq.push_back(std::make_pair(dist[v],v));
        }
      }
      vis[node] = 1;
    }
  }
```