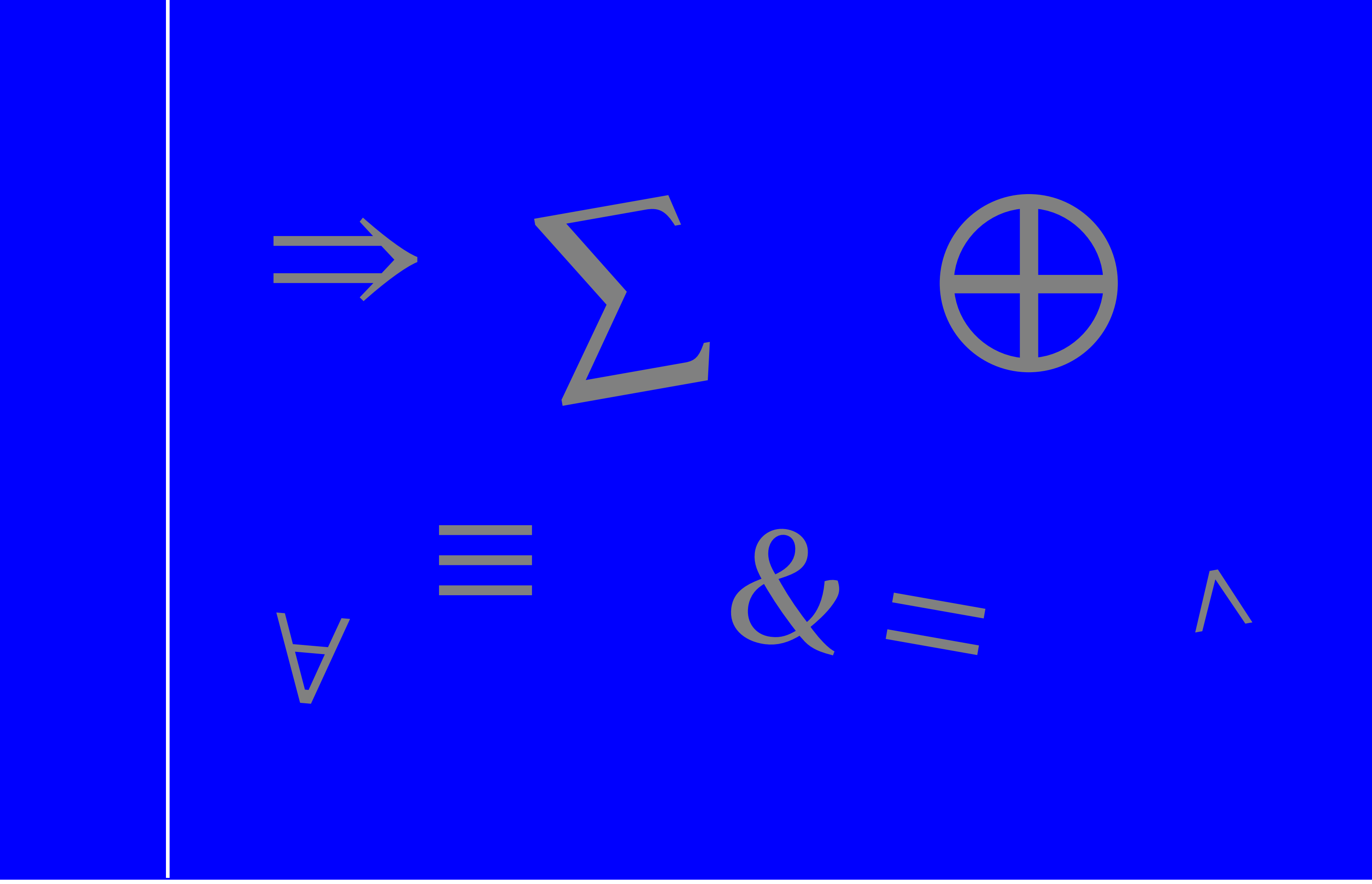


Tricks And Algorithms

C++

2022

Edward Ju

Preface

The main goal of this book is to give thoughts about approaches and strategies you can use when doing competitive programming. This book covers various algorithms and techniques used in different programming problems. The book also explains how and why the algorithm works and provides detailed implementation of how this thought can be written as a code.

This book is intended for all interested in learning algorithms in depth, and no previous knowledge of algorithms is required. I believe it will also help those studying competitive programming.

Each chapter of the book discusses the different algorithms, and for the algorithms that do include more than two different algorithms, I have tried putting them in a later chapter though I will remind you in those cases.

I hope you have a fun time while reading this book. Enjoy the book!

Table of Contents

I Basic Techniques **1**

1 Data Structures 1

1.1 Vectors 1

1.2 Iterators 2

1.3 Stacks and Queues 2

1.4 Maps 4

1.5 Sets 6

2 Greedy Algorithms 9

2.1 Minimizing Number of Coins 9

2.2 Maximum Number of Task 10

2.3 Minimizing Number of Operations With Profits 12

2.4 Limitation on Greedy Algorithms 13

3 Searching Algorithms 15

3.1 Linear Search 15

3.2 Binary Search 16

3.3 Jump Search 17

3.4 Ternary Search 19

3.5 Two Sum Problem 20

3.6 Three Sum Problem 22

4 Bit Algorithms 24

4.1 Bit Operations 24

4.2 Bit Manipulations 25

4.3 Processing Subsets 28

5 Dynamic Programming 30

5.1 Introduction 30

5.2 Revisiting the Greedy Algorithm Problems 32

5.3 Longest Increasing Subsequence 35

5.4 Maximum Path Sum 37

5.5 Counting Ways 38

6 Sorting Algorithms 41

6.1 Introduction 41

6.2 Bubble Sort 41

6.3 Merge Sort 43
6.4 Selection Sort 46
6.5 Comparator Functions 47
7 Range Queries 50
7.1 Introduction 50
7.2 Sum Queries with Updates 51
7.3 Minimum Queries 52
7.4 Minimum Queries with Updates 55
7.5 Other Queries 56
7.6 Sparse Table 58

Data Structures 1

The data structure allows us to store data in different ways. There are a variety of data structures that support different functions. We usually need to determine which data structure to use depending on its function, like whether it is sorted or how we can access the elements. This chapter gives an idea of data structures that are often used.

1.1 Vectors

A `vector` is a dynamic array, that is array that can change its size. The main advantage of the vector is that the size is not fixed, unlike arrays. In addition, it is easy to delete the element in the vector.

```
vector<int> v;
v.push_back(5); // [5]
v.push_back(2); // [5,2]
v.push_back(3); // [5,2,3]
v.push_back(7); // [5,2,3,7]
v[3] = 4; // [5,2,3,4]
cout << v[1]; // 2
v.erase(v.begin(), v.begin() +3); //[7]
v.push_back(3); // [7,3]
v.erase(v.begin() +1) // [7]
```

We can initialize the vector by a particular number with a specified number of elements.

```
vector<int> v(5); //initialize vector of size 5 which all the value is 0
vector<int> v1(4,5) //initialize vector of size 4 which all value is 5.
```

We can also sort, reverse, and print all the element in vector as follows.

```
vector<int> v = {2,4,6,4,3}
sort(v.begin(), v.end()); // [2,3,4,4,6]
reverse(v.begin(), v.end()); // [6,4,4,3,2]
```

```
for(auto it:v){
cout << it << " "; // 6 4 4 3 2
}
```

1.2 Iterators

Iterator acts as a pointer of the elements of the data structure. Using iterator, one can traverse the elements in the data structure (like the one shown for vectors). As it is a pointer to get the actual value, we need to put the * in front to access the real value.

```
// vector iterator
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it){
   cout <<  *it << " ";
}
```

Given a sorted data structure, we can also use `lower_bound` and `upper_bound`. The lower_bound returns a pointer to the least value greater than or equal to the value, while the upper_bound returns a pointer that points to the least value that is strictly greater than the value.

```
vector<int> v = {3,2,5,3,8,5,2};
sort(v.begin(),v.end()); // [2,2,3,3,5,5,8]
auto it1 = lower_bound(v.begin(),v.end(),5); // 5
auto it2 = upper_bound(v.begin(),v.end(),5); // 6
cout << it2 - v.begin(); // Prints the index of it2
```

Important: The vector needs to be sorted to use the lower_bound and upper_bound.

1.3 Stacks and other Queues

Stacks

A `stack` is a Last In First Out (LIFO) data structure. This data structure can access the most recently added element but can not check the other elements. The `push` inserts a new element at the top of the stack, and the `pop` removes the element at the top of the stack. We can also check the size of the stack and access the top of the element using the top and check whether it is empty. All the operation in the stack has a time complexity of $O(1)$.

```
stack<int>s;
s.push(5); // [5]
s.push(2); // [2,5]
s.push(7); // [7,2,5]

cout << s.size() << endl; // 3

while(!s.empty()){  //This will print all the element in s.
   cout << s.top() <<" "; // 7 2 5
   s.pop();
}
```

Deques

A deque is a data structure that allows us to manipulate the elements in the front and the back of the structure. It supports `push_back` and`push_font` as well as `pop_back` and `push_font`. Adding and deleting these have average time complexity of $O(1)$.

```
deque<int> d;
d.push_front(3); // [3]
d.push_front(5); // [5,3]
d.push_back(7); // [5,3,7]
d.pop_back(); // [5,3]
d.pop_front(); // [3]
```

Using the iterator, we can also print the element in deque as follows.

```
//deque iterator
for(deque<int>::iterator it = d.begin(); it != d.end(); ++it){
   cout << *it <<" ";
}
```

Queues

A `queue` is a First In First Out (FIFO) data structure. Unlike deque, the element can be added only in one direction. We can add an element with `push`, delete the first element with `pop` and access the first element using the `front`. All these operation run in $O(1)$ on average.

```
queue<int>q;
q.push(3); // [3]
q.push(7); // [3,7]
q.push(5); // [3,7,5]

cout << q.front() << endl; // 3
```

```
q.pop(); // [7,5]
q.push(2); // [7,5,2]
cout << q.front(); // 7
```

Priority queues

A `priority queue` is an ordered set of elements. By default, the priority queue is sorted in decreasing order. The priority queue has three main functions, where inserting and removing an element takes $O(\log n)$ time, while the accessing the elements takes $O(1)$ time.

```
priority_queue<int> pq;
pq.push(3); // [3]
pq.push(9); // [9,3]
pq.push(5); // [9,5,3]
pq.push(4); // [9,5,4,3]
pq.push(2); // [9,5,4,3,2]

cout << pq.top() << endl; // 9
pq.pop(); // [5,4,3,2]
pq.size(); // 4
while(!pq.empty()){
   cout << pq.top()<<" ";
   pq.pop();
}
```

Ascending Prioirty Queues

We can also make a priority queue that sorts the elements in ascending order that we can use as follows.

```
priority_queue<int,vector<int>, greater<int>> pq;
```

1.4 Maps

A map is an array that is stored as key-value pairs. This means that for each key, there is a unique value that it corresponds to. There are two different maps, an unordered map, and an ordered map. An unordered map, `unordered_map`, is a map where hashing is used. And thusunordered map supports $O(1)$ operations while the `map` operation takes $O(\log n)$ time complexity.

Unordered Maps

An unordered map is a map that has been hashed. It supports count and erase operations that run in $O(1)$. As it is a map, we can access the value using the key; each key contains a single value. And since it is unordered, the key-value pair is stored in random order.

```
unordered_map<int,int> mp;

mp[1] = 3; // [(1,3)]
mp[2] = 6; // [(1,3), (2,6)] in random order
mp[8] = 13; // [(1,3), (2,6), (8,13)] in random order
mp[5] = 6; // [(1,3), (2,6), (8,13), (5,6)] in random order

cout << mp[1] << endl; // 3
mp[1] = 1; // [(1,1), (2,6), (8,13), (5,6)] in random order
cout << mp[1] << endl; // 1

mp.erase(2); // [(1,1), (8,13), (5,6)] in random order

if(mp.count(1)){
   cout << "mp[1] exists" << endl;
}else{
   cout << "mp[1] does not exist"<< endl;
}
```

Ordered Maps

The `ordered_map` supports operations that unordered map supports in addition to the `lower_bound` and `upper_bound`. The time complexity for this is $O(\log n)$.

```
map<int,int> mp;

mp[1] = 3; // [(1,3)]
mp[2] = 6; // [(1,3), (2,6)]
mp[8] = 13; // [(1,3), (2,6), (8,13)]
mp[5] = 9; // [(1,3), (2,6), (5,9), (8,13)]

cout << mp.lower_bound(2) -> first << " " << mp.lower_bound(2) -> second
   << '\n'; // 2 6

mp.erase(2); // [(1,3), (5,9), (8,13)]

cout << mp.lower_bound(2) -> first << " " << mp.lower_bound(2) -> second
   << '\n'; // 5 9

if(mp.upper_bound(8) == mp.end()){
   cout << "Last element";
}
```

Descending map

We can also sort the map in descending order by doing as follow.

```
map<int,int,greater<int>>mp;
```

1.5 Sets

A set is a data structure that contains a collection of elements. Similar to a map, there is unordered_set and set. There also is a multiset that allows storage same value multiple times.

Unordered Sets

Hashing is used in the unordered set, which allows the inserting, removing, and searching operation to be $O(1)$ time complexity. But since it is unordered, when we iterate through the elements in unordered_set, it will print in arbitrary order.

```
unordered_set<int> s;

s.insert(3); // [3]
s.insert(5); // [3, 5] in random order
s.insert(7); // [3,5,7] in random order
s.insert(9); // [3, 5, 7, 9] in random order

for(int i=0; i<10; i++){
   if(s.count(i)){
      cout << i << " "; // 3 5 7 9
   }
}

s.erase(3); // [5,7,9] in random order
```

Ordered Sets

The set is ordered by default, and thus the insertion, deletion, and searching take $O(\log n)$ time. Ordered set supports `lower_bound` and `upper_bound` as well as `begin()` and `end()` unlike the unordered set. We can also find the size of the set using `size` and the index using distance().

```
set<int> s;
s.insert(1); // [1]
s.insert(4); // [1, 4]
s.insert(3); // [1, 3, 4]
s.insert(10); // [1, 3, 4, 10]

cout << *s.upper_bound(7) << '\n'; // 10
cout << "Index for lower_bound(4) is " << distance(s.begin(),s.
   lower_bound(4)) << '\n'; // 2

auto it = s.end();
--it;
cout << *(--it); // 4
s.erase(s.lower_bound(2)); // [1, 4, 10]
```

Unordered Multisets

An unordered multiset is similar to an unordered set, with the difference that elements may be repeated. Similar to an unordered set, an unordered multiset uses hashing.

```
unordered_multiset<int>ums({1,2,3,4,5,5,5,5,4,3,7,8,23}); // stores in
   random order

if(ums.empty()){
   cout << "multiset is empty \n";
}else{
   for(unordered_multiset<int>::iterator it =ums.begin(); it != ums.end()
   ; ++it){
      cout << *it <<" ";
   }
}
```

Ordered Multisets

A multiset is an ordered set that allows to store same values multiple times. Multiset is sorted by default and it supports `count()`, `erase()`, `find()`, and more. If we use erase, it will erase all the occurrences of that number. If we want to erase one of the elements, we can use erase(find(x)) to erase one of the x from the multiset.

```
multiset<int> ms;
ms.insert(2); // [2]
ms.insert(3); // [2, 3]
ms.insert(7); // [2, 3, 7]
ms.insert(3); // [2, 3, 3, 7]
ms.insert(3); // [2, 3, 3, 3, 7]

cout << ms.count(3); // 3

ms.erase(3); // [2, 7]

cout << ms.count(3); // 0

ms.insert(2); // [2, 2, 7]
ms.insert(2); // [2, 2, 2, 7]

ms.erase(ms.find(2)); // [2, 2, 7]

cout << ms.count(2); // 2
```

Greedy Algorithms 2

A **greedy algorithm** allows us to find the optimal way for finding the task at the moment. Greedy does not look forward or consider the solution as a whole, and it also does not consider what has happened before, so it will not take back the choices that have been made. Instead, in greedy algorithms, it only considers the instant we are performing the task.

In greedy algorithm problems, we try to maximize or minimize some value. The greedy algorithm can mislead, and finding the correct strategy for applying the algorithm is not easy. However, when we do use greed, it becomes very efficient. Unfortunately, not all greedy approaches work, and some will end up with the wrong solution.

2.1 Minimizing Number of Coins

In this type of problem, we are asked to find the minimum number of coins that we can use to create a certain amount of money. Greedy can solve this problem by taking the largest amount of value we can still put in to get the value.

```
vector<int> coins = {1,5,10,100};

int desired = 133;

sort(coins.begin(),coins.end());
reverse(coins.begin(), coins.end());

int pos = 0,NumberOfCoins =0;

while(desired >0){
   while(desired >= coins[pos]){
      desired -=coins[pos];
      NumberOfCoins++;
   }
   pos++;
}
cout << NumberOfCoins; // 7
```

Unfortunately, there are some cases where the greedy algorithm fails for the coins problem. The example might be when the values of coins are [1, 3, 6, 7] and the desired value is 9. In this case, the optimal strategy will be to take 3 and 6, which will require 2 coins. However, following the greedy approach, it will select 7. 1, and 1, which requires 3 coins.

Important: Greedy Approach does not always work in coin problems.

2.2 Maximum Number of Task

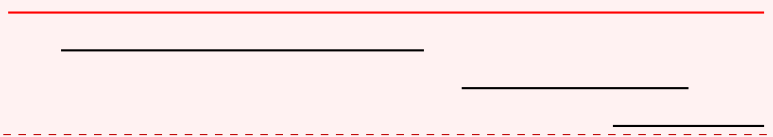
Let us say we are performing a task and want to maximize the number of tasks we do. When we are given when each of the tasks starting time and ending time, we can greedily choose the task to maximize the number of task we do.

Wrong approaches

First approach

Select the task whenever you are free, then as soon as that task ends, choose the next possible task that begins earliest after the current task end.

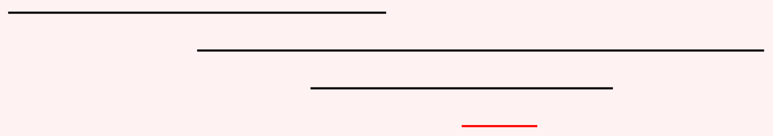
However, this approach fails is the following case:



Second approach

Always choosing the task that takes the shortest amount of time. Then continue choosing the following shortest possible tasks.

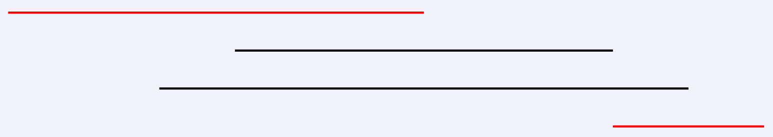
However, this approach fails is the following case:



We can see that both of the promising approaches do not work. The correct approach for solving this problem turns out to be to continuously select the fastest ending task regardless of the time it takes.

Correct approach

The correct approach is to choose the task that ends the earliest and keep selecting the ones that complete the earliest after that, which we can take and continue.



How can we implement this? We can put all the starting and ending times of the task in one array and then sort the array by their time, along with whether it is the starting time or the ending time of the task. In addition, we could connect the ending time of the task with the starting time of the task to check whether it is better to take the task or not.

First approach

We can store the pairs and sort them by the ending time of the task. Then we can choose the soonest ending task such that the starting time is after the ending time of the current task

```
//First approach
int main(){
   sort(v.begin(),v.end(),Sort_by_Ending_time); // sorting by ending time
   int Current_Task_Ending_time = 0; // To select the first ending task
   for(auto next:v){
      if(next.startingTime >= Current_Task_Ending_time){ // checking if we can do this task
         Current_Task_Ending_time = next.Ending_Time;
         Number_of_task++;
      }
   }
return 0;
}
```

Second approach

We can also use dynamic programming to solve this problem (If you are unfamiliar with dynamic programming, try coming back after reading chapter 5). We again sort by the ending time of the tasks and see whether choosing this task is optimal or not.

We will have three states to consider: the time which is the time that this happens, the index which stores which task it is so that when we reach the end of the task, we can move to the start of the task to see whether or not we should do this task or not. In addition, we store type, which will store whether this is the start time or end time of the task.

```
struct Task{
   int time,index,type
   bool operator(Task other){return t == other.t? index < other.
   index: t < other.t;} //sorting by time
}task[2*Max]
```

After sorting, we can go from 1 to n, moving to the next if nothing is happening at this time, and others check whether taking this task gives the answer we want or not.

```
startingTime[n];
map<int,int>pos;
for(int i=0; i<2*n; i++) pos[task[i].time] = i;

for(int i=0; i<2*n; i++){
   if(!task[i].type)dp[i] = dp[i-1] // If nothing happened
   else dp[i] = max(dp[i-1], dp[pos[startingTime[task[i].index]]]+1)
   ;
}
```

2.3 Minimizing Number of Operations With Profits

Suppose that we want to maximize the profit while minimizing the number of operations. Let us say we initially have an array of numbers, and we get a profit when we get a certain specified value. If we have the array distance, which gives us the minimum number of operations that we need to make to get a certain value, then we can find the maximum profit within K operation.

```
vector<int> initial_numbers(n);

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

for(auto numbers:initial_numbers){
   for(int j = K - Distance[numbers]; j>=0; j--){
      TotalProfit[j+Distance[numbers]] = max(TotalProfit[j+Distance[
   numbers]],TotalProfit[j] + profit[numbers]);
   }
}
```

2.4 Limitation on Greedy Algorithms

Like the coin problem shown at the beginning of the chapter, a greedy algorithm does not always necessarily produce the most optimal solution. The reason is because of the fact that it only looks at the best movement at that moment and does not think about later moves.

Path with maximum sum from a complete binary tree

The greedy approach will not work in this case, as there could be a node with greater value later on the other side of the tree.

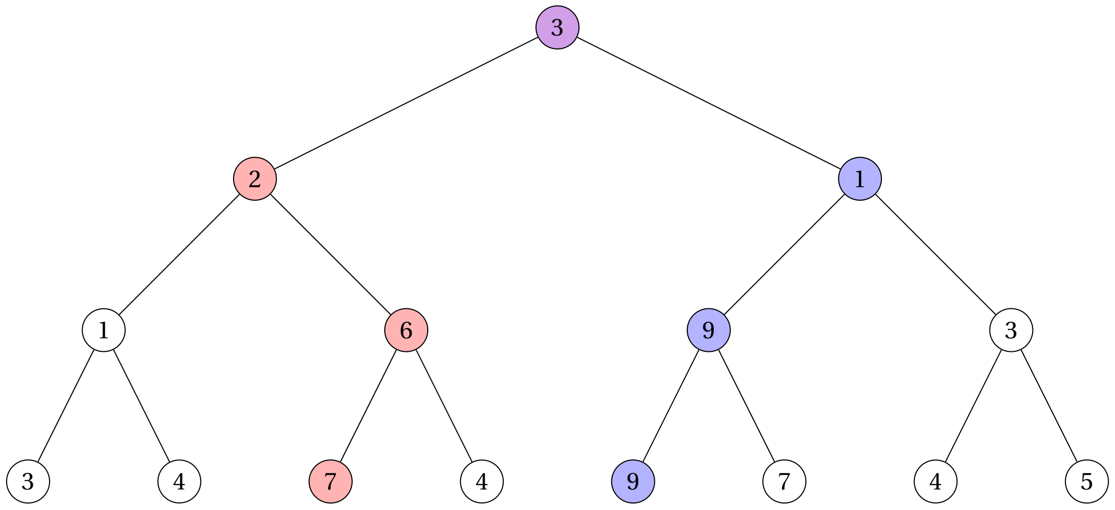


FIGURE 2.1: Maximum path sum

For example, from the following binary tree, from the purple node as 2 > 1, the maximum sum using the greedy algorithm will follow the red path, while the correct solution would have followed the blue path.

Knapsack

Knapsack is a problem that involves maximizing the profit given its capacity and the weight of each item and its profit. The obvious greedy approach would be to choose the most efficient one, where efficiency will be determined by the one that has a maximum value when you divide the profit by its weight. However, this does not always work. For example, consider the following example where we want to maximize our satisfaction level. Say we have 20 dollars and consider the following chart.

Item	Price	Satisfaction level
Water	1.5	3
Apple Juice	2	6.5
Pizza	5	18
Hot dog	3	10

FIGURE 2.2: Knapsack example

According to the greedy approach, pizza is the most efficient, giving the greatest satisfaction level per price. We would buy 2 pizzas, and since we have 4 dollars left, we would choose the next efficient one, which would be Hot dog. As we would only have 1 dollar left, we could buy no more, and the satisfaction level would have been 46. However, this is not the optimal solution. If we buy 2 pizzas with 2 apple juice, the total price would have been 20 dollars within the capacity, while the satisfaction level would have been 48. The correct approach for solving this problem is using dynamic programming, which we will discuss more in chapter 5.

Searching Algorithms 3

Searching Algorithms allows us to retrieve values among the data that has been stored. The two main types of search methods are the `sequential search` which traverses the elements sequentially, while `interval search` repeatedly divides the range into intervals and searches.

An example of sequential search is the `linear search` which basically goes through all the elements and checks whether it satisfies the condition. An example of interval search would be `binary search`, which repeatedly divides the interval in half and checks which side the desired element is. Generally, if we are given a sorted data list, the interval search is more efficient than the sequential one.

3.1 Linear Search

In `linear search` given a value, we can iterate through the entire data one by one, compare and check if it is or not. If the value is what we are looking for, we can return the position, and if it is not, we continue on. The time complexity is $O(n)$.

```
// n is the size of the array
void linear_search(int a[], int n, int value){
   for(int i=0; i<n; i++){
      if(a[i] == value){
         cout << "Value is in the array" << endl;
         return;
      }
   }
   cout << "Value is not in the array" <<endl;
}

int main(){
   //exmple from Fig 3.1
   int a[] = {13,7,5,6,23,14,12,10,7,3};
   int n =size(a);
   linear_search(a,n,23);
```

```
    return 0;
}
```

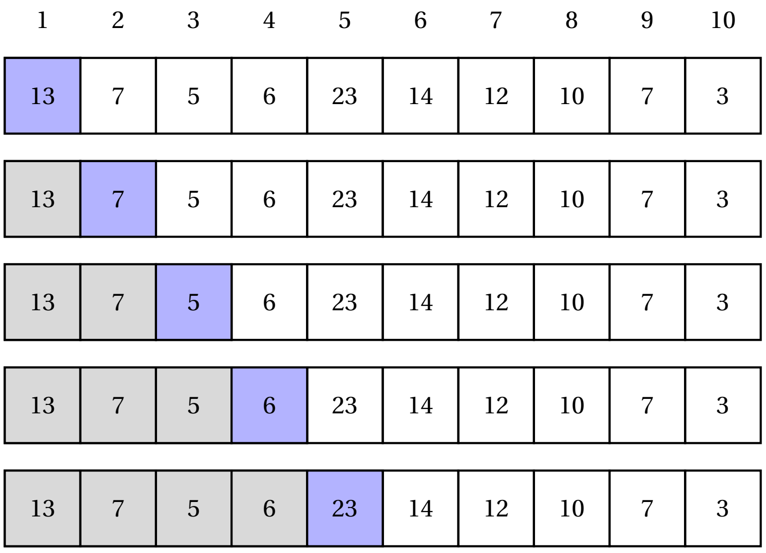


FIGURE 3.1: Linear Search

3.2 Binary Search

`Binary search` is a more efficient way to search for elements. Given the sorted array, we divide the array in half and check whether the value we are looking for fits on the left or right side of the array. Then we continually divide the array in half until we finally find the element. The time complexity for binary search is $O(log\ n)$

```
void binary_search(int a[],int l, int r, int val){
     while(l <=r){
        int mid = (l+r)/2;

        if(a[mid] == val){
            cout << mid <<endl;
            return;
        }

        if(a[mid] >val){
            r = mid-1;
        }else{
            l = mid+1;
        }
    }
}

int main(){
   //exmple from Fig 3.2
    int a[] = {3,7,9,13,17,24,29,33,37,41};
    int n =size(a);
    sort(a, a+n); // In case the array is not sorted
    binary_search(a,0,n,7);
   return 0;
}
```

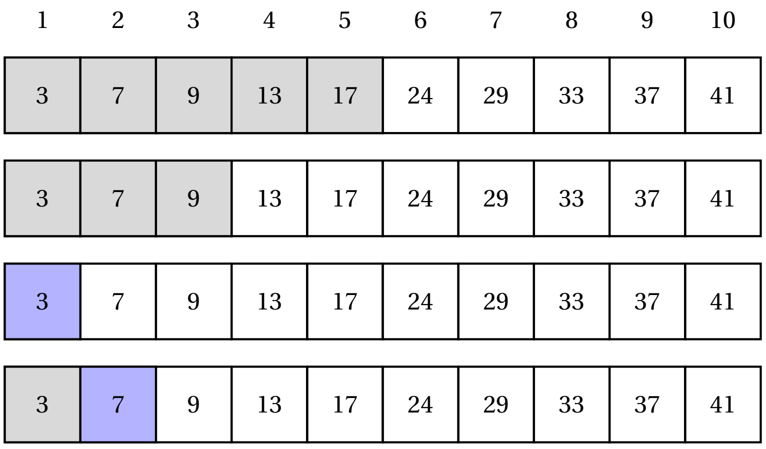


FIGURE 3.2: Binary Search

Important: In order to use binary and Jump search the array needs to be sorted

3.3 Jump Search

`Jump search` is a searching method where we jump from one position to the next until we find the value or report that the value does not exist in the array. We could choose the number of the index we will jump each time.

To calculate the time complexity when we make a jump by k index, since we will make at most n/k jumps then do at most $k-1$ comparisons through linear search, the time complexity will be $O(n/k+k-1)$. By the AM-GM inequality we can see that $n/k+k-1 \geq 2\sqrt{n}-1$ where the equality will occur when $n/k = k$ or $k = \sqrt{n}$. So we can say that the optimal time complexity happens when k, the number of jumps, is $\sqrt{n}$.

```
void jump_search(int a[], int n, int val){
   int l =0, r = sqrt(n);

   while(r < n && a[r] <=val){
      l = r;
      r += sqrt(n);
   }
   if(r >= n){
      cout << "Value does not exist";
      return;
   }

   for(int pos = l; pos < r; pos++){
      if(a[pos] == val){
         cout << "Value exist at index " << pos;
         return;
      }
   }

   cout << "Value does not exist";

}

int main(){
   int a[] = {1,3,4,8,13,44,59,79,84,94,135,153,167,172,184,213};
   int n = size(a);
   int val = 94;
   sort(a, a+n); // In case the array is not sorted
   jump_search(a,n,val);

return 0;
}
```

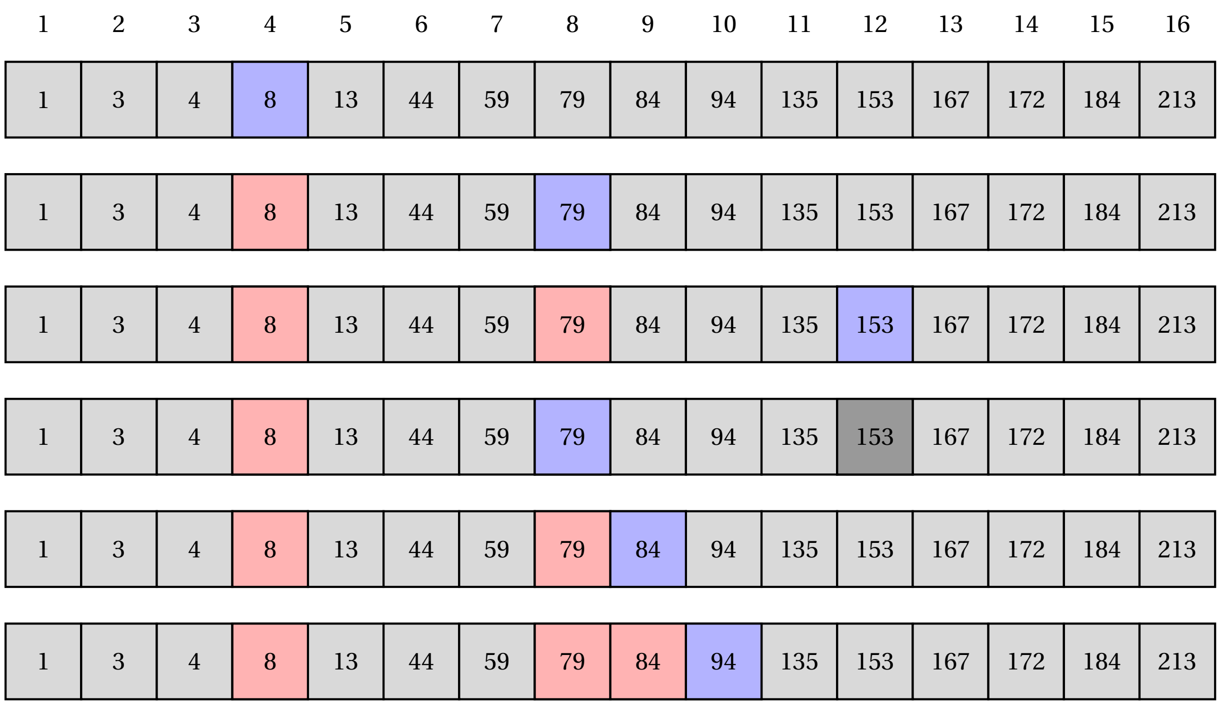


FIGURE 3.3: Jump Search

3.4 Ternary Search

`Ternary search` is similar to binary search in that it also considers the blocks after dividing the array. However, unlike binary search, which divides the array into two each time, ternary divides the array into three pieces. Similar to binary search, the array needs to be sorted to use ternary search.

After dividing it into three pieces, it checks b1, b2, end of the 1st block, and beginning of the 3 rd block among the three blocks. If the value is equal to either the end of the first block or the beginning of the third block, we can return that index. Otherwise, we now consider the divided intervals. So if the value is less than b1, we could change the interval to the first block and perform the ternary search. Otherwise, if the value is greater than the beginning of the third interval, we can do a ternary search on the third interval. If the value is between, we can check the 2 nd block and apply a ternary search.

Because we will be dividing the array into three different blocks, we will have

$$b_1 = l + (r - l)/3$$

$$b_2 = r - (r - l)/3$$

and the time complexity will be $O(\log_3 n)$.

```
void ternary_search(int a[], int l,int r, int val){
   while(l <=r){
      int b1 = l + (r-l)/3;
      int b2 = r - (r-l)/3;

      if(a[b1] == val){
         cout << b1 << " ";
         return;
      }

      if(a[b2] == val){
         cout << b2 <<" ";
         return;
      }

      if(val < a[b1]){
         r = b1-1;
      }else if(val > a[b2]){
         l = b2+1;
      }else{
         l = b1 +1;
```

```
            r = b2 -1;
        }
    }

    cout << "Value does not exist";

}

int main(){

    int a[] = {1,3,5,7,12,15,17,19,20,25};
    int n = size(a);
    int val = 15;

    sort(a, a+n); // In case the array is not sorted

    ternary_search(a,0,n-1,val);

return 0;
}
```

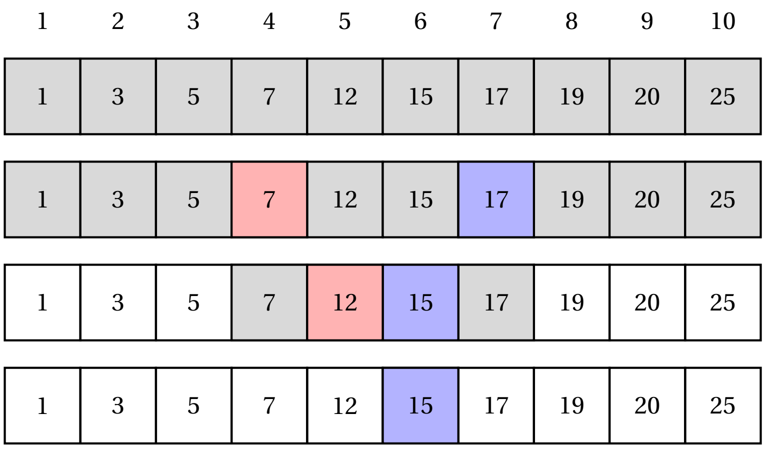


FIGURE 3.4: Ternary Search

3.5 Two Sum Problem

In two sum problems, we would like to find two numbers from the array that sum up the value we are looking for. The naive way to solve this would be to check all the possible pairs as follows, which would be $O(n^2)$.

```
void two_sum(int a[], int n, int val){

    for(int i=0; i<n; i++){
      for(int j=i+1; j<n;j++){
         if(a[i] + a[j] == val){
            cout << i <<" " << j << '\n';
            return;
         }
      }
   }

   cout << "Pair does not exist";

}
```

The better way to solve this problem would be to use two pointers to iterate to find the sum. First, given a sorted array, if not sort the array, we could choose the first and last element and find their sum. So here is what we can do. If the sum is greater than the value we are looking for, we can move the right pointer one unit to the left, which will reduce the sum, or else if the sum is less than the value, then we can move the left pointer one unit to the right which will increase the sum. The time complexity will be $O(n)$.

```
void two_sum(int a[], int n, int val){

   int l = 0, r = n-1;

   while(l<r){
      if(a[l] + a[r] == val){
         cout << l <<" " << r <<'\n';
         return;
      }else if(a[l] + a[r] < val){
         l++;
      }else{
         r--;
      }

   }

   cout << "Pair does not exist";

}

int main(){
   int a[] = {1,4,7,40,15,23,12,-14,26,37};
   int n = size(a);
   int val = 24;

   sort(a, a+n); // In case the array is not sorted

   two_sum(a,n,val);
return 0;
}
```

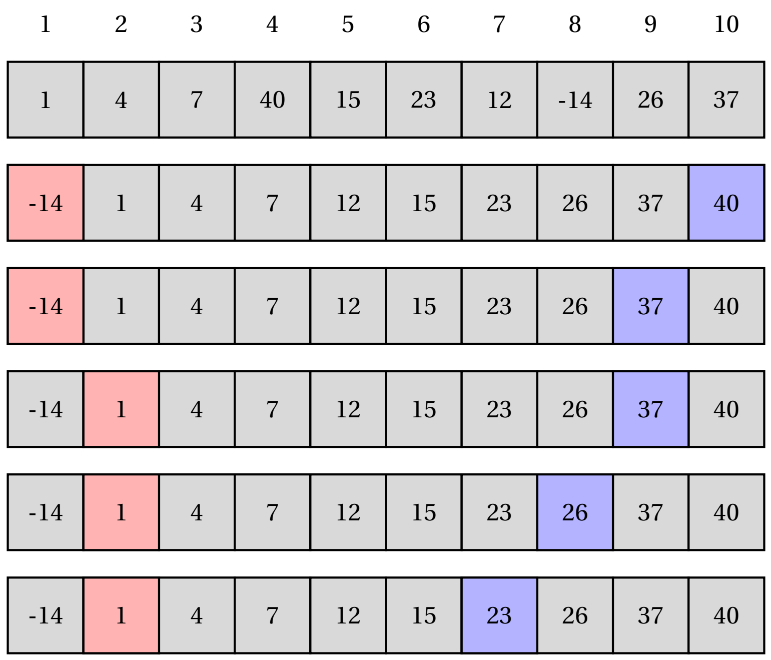


FIGURE 3.5: Two Sum

3.6 Three Sum Problem

Three sum problem is to search for the three numbers in the array that sums to a certain value. Similar to two sum problems, the naive algorithm would be to check all three possible triplets and check if their sum is equal to the value we are looking for. The time complexity for this would be $O(n^3)$.

However, there is a more straightforward way to solve this problem. Here is what we could do. First, we can go through the array from the first value to the 3rd to the last value. And here again, we assume that the array is sorted (If not, we should sort before we apply the algorithm). After selecting the leftmost number, say X, if val is the value we are looking for, then this essentially reduces to finding the sum of two values which would be greater than X that sums to Val – X. In this case, the time complexity would be $O(n^2)$.

```
void three_sum(int a[], int n, int val){

   for(int i=0; i<n-2; i++){
      int l = i+1;
      int r = n-1;
      int target = val - a[i];

      while(l<r){
            if(a[l] + a[r] == target){
                cout << i <<" " << l << " " << r <<'\n';
                return;
            }else if(a[l] + a[r} < sum){
                l++;
            }else{
                r--;
            }
        }
    }
    cout << "Sum does not exist";
}
```

Bit Algorithms 4

Bit algorithms are the way to manipulate the bits in different ways to solve the problem. Sometimes using bits can lead to a shorter and more elegant solution. There is a lot of application of bit representation and many operations relating to bits.

4.1 Bit Operations

The main bit operations are and, or, and xor. Each bit is compared to its corresponding bit when given two-bit and performing the bit operations. If we ignore the leading zeros, assume we want to operate on 3 and 6 in which the bit representation are 0011 and 0110, respectively. Then the operation would be done as follows

`And`, &, returns 1 if both of the bit has 1 and otherwise returns zero. So 0011 & 0110 will be 0010.

`Or`, |, returns 0 if both of the bit at that position is 0 and returns 1 otherwise. So 0011 | 0110 will be 0111.

`Xor`, ^, returns 1 if the two-bit is different and returns 0 otherwise. So 0011 $\wedge$ 0110 will be 1010.

`Not`, ~, for a number x is written as $\sim x$. The interesting thing is that $x + \sim x$ if always equal to -1. Another way to say this is $\sim x = -x - 1$.

`bit shift`, << and >>, shifts the bit leftward by certain unit and adds 0 at the end. If $x = 10$, then its bit representation, ignoring the leading zeros, will be 1010 then $x << 3$ will be 1010000. You can think of this as multiplying the number by 2^3. In general $x << k$ will by multiplying x by 2^k. The right shift works similarly but the bits shift rightward. So $x >> k$ can by the thought of as dividing the number by 2^k.

The other useful built-in functions are __builtin_popcount(x) which returns the number of one's, __builtin_clz(x) which returns the number of leading zeros, __builtin_ctz(x) which counts the trailing zeros, and __builtin_parity(x) which returns 1, or true, if the number of one's in the bit representation is odd and 0, or false, if the number of one's in bit representation is even.

```
int main(){

   int x = 10; // 00000000000000000000000000001010
   int y = 12; // 00000000000000000000000000001100

   cout << (x&y) <<'\n'; // 8
   cout << (x|y) <<'\n'; // 14
   cout << (x^y) <<'\n'; // 6
   cout << (~x) <<'\n'; // -11
   cout << (x<<3) <<'\n'; // 80
   cout << (x>>3) <<'\n'; // 1
   cout << __builtin_popcount(x)<<'\n'; // 2

    if(__builtin_parity(x)){
        cout << "number of one's in x is odd: " << __builtin_popcount(x)
   << '\n';
    }else{
        cout << "number of one's in x is even: " << __builtin_popcount(x)
    << '\n';
    }

    cout << __builtin_clz(x)<< '\n'; // 28

   cout << __builtin_ctz(y)<<'\n';  // 2

return 0;
}
```

4.2 Bit Manipultations

Bit has a lot of usages and depending on how you use them bit could be highly efficient.

Checking parity of number

Since when x is odd then the last bit in binary form is 1 we can check whether x is odd or not by checking $(x \& 1)$.

```
int main(){

   int x = 29;

   if(x&1){
      cout << "x is odd";
   }else{
      cout << "x is even";
   }
return 0;
}
```

Checking whether k th bit is set

We can check by operating with x and $(1 << k)$. If the k th bit is set, it will return $(1 << k)$ and 0 otherwise. (Here, we assume the 0th bit is the most rightward bit)

```
int main(){
   int x = 7541; // 00000000000000000001110101110101
   int k = 7;

   if(x&(1<<k)){
      cout << "7 th bit is set";
   }else{
      cout << "7 th bit is not set";
   }
return 0;
}
```

Setting the k th bit in the number

This is the application of `or` operator. To set the k th bit we can do $x = (x|(1 << k))$.

```
int main(){
   int x = 5; // 00000000000000000000000000000101

   int k = 3;

      x = (x|(1<<k));

   cout << x; // 00000000000000000000000000001101

return 0;
}
```

Unsetting the k th bit in the number

This is the application of `and` and `not` operator. To unset the k th bit we can do $x = (x\&(\sim(1 << k)))$.

```
int main(){
   int x = 5; // 00000000000000000000000000000101

   int k = 2;

   x = (x& (~(1<<k)));

   cout << x; // 00000000000000000000000000000001

return 0;
}
```

Unsetting first i bits

Similarly, we can unset to first i bits from the right as follow.

```
int k = ~((1<<(i+1)) -1);
x = (x&k);
```

Finding the binary form of the number

We can go through from 31 th bit to 0 th bit, checking if the bit at that position is set. If the bit is set we add 1 at the back of the string and 0 otherwise.

```
int main(){
   int x = 7541; // 00000000000000000001110101110101

   string s;

   for(int i=31; i>=0; i--){
      if(x&(1<<i)){
         s+='1';
      }else{
         s+='0';
      }
   }

   cout << "Binary representation of 7541 is " << s << '\n';


return 0;
}
```

Determining whether the number is power of 2

Notice that when x is the power of two, the binary representation would be one, followed by zeros. That means that $x-1$ will have the binary form of zero at the front, followed by ones. Therefore $x\&(x-1)$ will be zero if x is power of two.

```
int main(){
   int x = 1024; // 00000000000000000000010000000000

   if(x && !(x&(x-1))){
      cout << x <<" is power of two";
   }else{
      cout << x <<" is not a power of two";
   }

return 0;
}
```

x&(-x)

We can also determine whether a number is a power of 2 by using $x\&(-x)$. If $x = x\&(-x)$ then x is a power of 2. In general, $x\&(-x)$ returns the position of the rightmost set bit, assuming that the rightmost bit is position 1.

4.3 Processing Subsets

Let's say we have some numbers and want to generate a subset using those numbers. Using bit, we can elegantly do this. Let's say we have n numbers. Then we can have an n-digit bit with either 0 or 1 where the k the bit is 0 if we do not select the k the element and 1 otherwise. Then we can iterate through and check whether the k the bit is 1 or not.

```
// n is the number of element
for(int i=0; i<(1<<n); i++){
   for(int j=0; j<n; j++){
      if(i&(1<<j){
         //Incude in a subset
      }
   }
}
```

K sum problem

Using the concept above, we can now go on further and determine the number of pairs among the array such that two of the number sums to a given value.

```
int main(){
    int a[] = {1,2,5,6,9,12,34};

    int val = 11;
    int n = size(a);

    int ans = 0;


    for(int i=0; i<(1<<n); i++){
        int sum =0;

        if(__builtin_popcount(i) ==2){
            for(int j=0; j<n; j++){
                if(i&(1<<j)){
                    sum += a[j];
                }
            }

            if(sum == val){
                ans++;
            }
        }
    }
    cout << ans;

return 0;
}
```

Note that the number of elements being chosen is 2 by the if statement that we are checking whether the number of 1 in i is 2. So by simply switching the number 2 to another number $k \le n$, we can find the number of ways to select k of the n elements that sum to a value.

Dynamic Programming 5

Dynamic programming is a way to find the most optimal solutions using a recursion. Dynamic programming also allows us to count the number of solutions holding some property. Dynamic programming can sometimes be efficient, allowing the solution to better time complexity.

5.1 Introduction

Dynamic programming problems ask to find the minimum or maximum solution or count the number of possible ways. In dynamic programming problems, we can consider each case as a state. And what we are trying to do is to connect those states to make all possible combinations that are possible and carry on the best cases. When counting the number of ways, we can also store the number and relate it to the possible combination to find the final answer. And in the problem where we are trying to find the minimum or maximum solutions, the initial state will usually be smaller than the minimum if it is the maximum solution problem and is greater than the maximum if it is the minimum solution problem.

Stair Problem

Let us say there are n stairs that you have to go up. How many different ways are there to go up n stairs if you can go up the stairs by 1 or 2?

This is a well-known math problem. Let's say a_n is the number of ways to go up to the n th stair by going up by 1 or 2 stairs. Then notice that $a_n = a_{n-1} + a_{n-2}$. We can solve this equation and get the general formula for a_n. Dynamic programming provides an alternative. Since $a_1 = 1$ and $a_2 = 2$, we can continuously add up to find a_k for $3, 4, ..., n$.

```
int main(){

   int a[1] = 1,a[2] = 2; //initial conditions

   for(int i=3; i<=n; i++){
      a[i] = a[i-1] + a[i-2];
   }

   cout << a[n] << '\n';

return 0;
}
```

Though here we said we can go up the stairs by 1 or 2 we could have said that it could go up by 1,2,..., k steps by modifying the code as follows.

```
int main(){

   a[0] = 1; // initially at stair 0

   for(int i=0; i<=n; i++){
      for(int jump = 1; jump <=k; jump++){
         if(jump + i <=n) a[jump+i]  += a[i];
      }
   }

   cout << a[n] << '\n';

return 0;
}
```

Now let's say the tiredness when you go to the k th stair increases by tired[k]. And since you do not want to get as tired as possible, you want to know the minimum tiredness you can achieve by optimally selecting the stairs. Say we assume that $0 \le \text{tired}[k] \le 100 \ \ \forall 1 \le k \le n$. Then since this is the minimization problem, we should set all initial dp array to the number greater than the maximum and iterate through. And make another auxiliary array dp, where dp[k] is the minimum tiredness you can have when you go up stairs k. We can continuously store the minimum tiredness up to stair n.

```
const int N = 100; // N will be greater than the maximum number of n

int main(){

    int tired[] = {0,4,5,3,36,54,23,54,23}; // tired[k] is tiredness when
    moving to stair k

    int dp[N];

    int n = 8;

    int k =3; // maximum number of jumps of stair in one jump

    for(int i=1; i<=n; i++){
        dp[i] = 10000;
    }

    dp[0] = 0; // since tiredness at the beginning is 0


    for(int i=0; i<=n; i++){
        for(int jumps =1; jumps <=k; jumps++){
            if(jumps+i <=n) dp[jumps+i] = min(dp[jumps+i], dp[i] + tired[
   i+jumps]);
        }
    }

    cout << dp[n];


return 0;
}
```

5.2 Revisiting the Greedy Algorithm Problems

We have seen from the greedy algorithm that for coin problem and greedy knapsack approach do not necessarily work. The correct solution can be solved using dynamic programming.

Coin Problem

Since we are trying to find the minimum number of coins needed to get a certain value, dynamic programming is a good candidate. Set the states to be the current amount of coins we have, and in the dp we store the minimum number of coins required to make that value. Because it takes 0 number of coins to make 0 value, dp[0] is 0, and the time complexity will be $O(mn)$ if we say n to be the number of coins and m is the value we are hoping to make.

```
const int N = 100;
int main(){
    vector<int> coins = {1,3,4,7,8,15};
    int dp[N];
    int val = 63;

    for(int i=1; i<=val; i++){
        dp[i] = 1000;
    }

    dp[0] = 0;
    for(int i=1; i<=val; i++){
        for(auto coin:coins){
            if(i-coin >=0) dp[i] = min(dp[i], 1+dp[i-coin]);
        }
    }

    cout << dp[val];

return 0;
}
```

Knapsack

Now we revisit the knapsack problem. Say we have n items of which the item k has their price p[k] dollars and satisfaction level s[k]. Say we have m dollars that we can spend, and we don't mind how much money we are left with but just want to increase the satisfaction level. What is the maximum satisfaction you can have?

From here, we can let dp[i][j] to be the maximum satisfaction level we can have spent j dollars and currently are iterating the item i. Then similar to the stair problem, we can iterator through and check whether that item is in the budget and see if it is better to have or not have by comparing it with the current value as follows. And so by doing this, we check both cases when we select and do not select this item.

```
if(j + p[i] <= m){
   dp[i+1][j+p[i]] = max(dp[i+1][j+p[i]], dp[i][j] + s[i]);
}

dp[i+1][j] = max(dp[i+1][j], dp[i][j];
```

And notice how since we are adding the satisfaction level in the iteration initially, all the values in the dp array should be 0.

```
int main(){

    int p[] = {2,3,6,8,14};
    int s[] = {3,4,9,12,34};

    int n = size(p);

    int m = 20;

    int dp[100][100];

    memset(dp,0,sizeof(dp)); //setting all values in dp to 0

    for(int i=0; i<n; i++){
        for(int j=0; j<=m; j++){
            if(j+ p[i] <=m){ // if in the budget
                dp[i+1][j+p[i]] = max(dp[i+1][j+p[i]], dp[i][j] + s[i]);
            }

            dp[i+1][j] = max(dp[i+1][j], dp[i][j]); // if we do not
   select this item
        }
    }

    cout << dp[n][m];

return 0;
}
```

Path with maximum sum from the complete binary tree

The maximum sum path can be solved by assigning the nodes by (l,r). From here, we can decrease l or r to see which one returns the greater value.

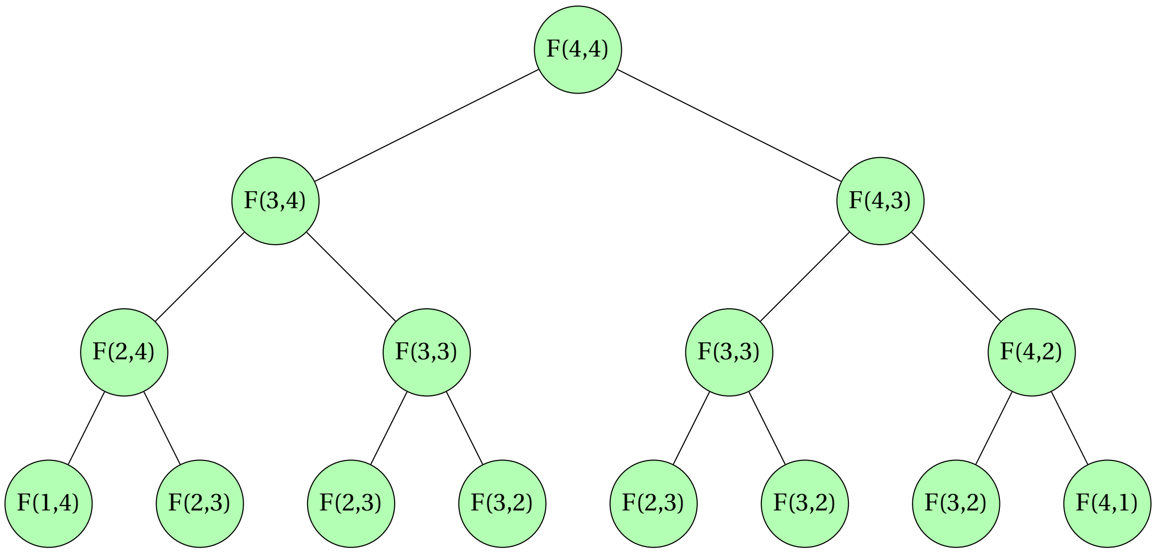


FIGURE 5.1: Maximum Path Sum (Binary Tree)

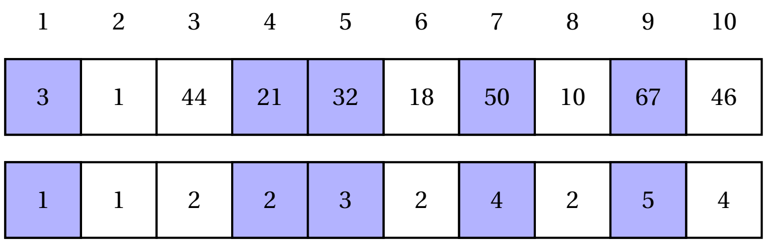
If we say that the value at (3,4) to be F(3,4) and dp[3][4] to be the maximum path sum until that subtree, we can solve the problem by dp[n][n] where n is the height of the tree. And we can relate the dp by dp[a][b] = F[a][b] + max(dp[a-1][b],dp[a][b-1]).

```
int solve(int a,int b){

   if(a<1 || b<1)return 0;

   dp[a][b] = F[a][b] + max(solve(a-1,b), solve(a,b-1));

   return dp[a][b];
}
```

5.3 Longest Increasing Subsequence

Given the array of numbers, we want to find the longest possible subsequence of the array such that it is strictly increasing. The simplest $O(n^2)$ method would be to iterate through and keep the maximum possible length in the temporary array.

```
int LIS(int a[], int n){
   vector<int> dp(n,1);

   for(int i=1; i<n; i++){
      for(int j=0; j<i; j++){
         if(a[i] > a[j] && dp[i] < dp[j] +1){
            dp[i] = dp[j]+1;
         }
      }
   }
   return *max_element(dp.begin(), dp.end());
}

int main(){

   int a[] = { 3,1,44,21,32,18,50,10,67,46};

   int n = size(a);

   cout << "Length of the longest increasing subsequence is " << LIS(a,n)
   ;

return 0;
}
```



1	2	3	4	5	6	7	8	9	10
3	1	44	21	32	18	50	10	67	46
1	1	2	2	3	2	4	2	5	4

FIGURE 5.2: Longest Increasing Subsequence

Using `lower_bound` the longest increasing subsequence problem can be solved in $O(\log n)$ time complexity. We will iterate through the vector, and if it is smaller than the smallest value, that will be our smallest value. Otherwise, we add to the back if possible to lengthen the overall length; however, if that is not possible, we can use the lower_bound function to find the index where we can replace that number since then it will decrease the number in that place which will be more likely to provide the chance to increase the length of the subsequence afterward.

```
const int INF = 1e9;

int LIS(vector<int> v, int n){

    // we don't need to check if the vector is empty
    if(n ==0){
        return 0;
    }

    vector<int> dp(n,INF);

    int len = 1;

    for(auto u: v){

      // This wil be able to replace the smallest value
        if(u < dp[0]){
            dp[0] = u;

         // This will be the candidate for so far largest value and is
   put into the subsequences
        }else if(u > dp[len -1]){
            dp[len++] = u;

         // Replace the lowest place that we can by using lower_bound

        }else{
            dp[lower_bound(dp.begin(), dp.end(), u) - dp.begin()] = u;
        }
    }

return len;
}


int main(){

    vector<int> v = { 3,1,44,21,32,18,50,10,67,46};

    int n = v.size();

    cout << "Length of the longest increasing subsequence is " << LIS(v,n
    );

return 0;
}
```

5.4 Maximum Path Sum

When given the n by m grid with numbers, we want to find the maximum path sum from (0,0) to any point in the grid if we can only move to the right, down, or right down diagonally from the grid. To solve this problem, we can consider a dp array where dp$[x][y]$ stores the maximum path sum from (0,0) to (x,y). Then we can relate dp$[x][y]$ with its previous term as follows.

$$\text{dp}[x][y] = \text{grid}[x][y] + \max(\{\text{dp}[x-1][y], \text{dp}[x][y-1], \text{dp}[x-1][y-1]\})$$

This is true since the the path (x,y) can be reach from either $(x-1,y)$, $(x,y-1)$, or $(x-1,y-1)$.

```
const int n =5,m =6;
int dp[n][m];
int grid[n][m] = {  {3,5,7,6,4,-7},
                    {3,4,6,-3,3,7},
                    {2,3,-6,8,3,2},
                    {8,4,-2,6,-4,1},
                    {2,-9,4,0,1,5} };


void MaxPathSum(){

    dp[0][0] = grid[0][0];

    for(int i=1; i<n; i++){
        dp[i][0] = grid[i][0] +dp[i-1][0];
    }

    for(int j=1; j <m; j++){
        dp[0][j] = grid[0][j] + dp[0][j-1];
    }

    for(int i=1; i<n; i++){
        for(int j=1; j<m; j++){
            dp[i][j] = grid[i][j] + max({dp[i][j-1], dp[i-1][j], dp[i-1][
   j-1]});
        }
    }
}


int main(){

    MaxPathSum();

    vector<pair<int,int>> test_values = {{5,3}, {5,6}};

    for(auto v: test_values){
        cout << dp[v.first -1][v.second -1] <<'\n';
    }


return 0;
}
```

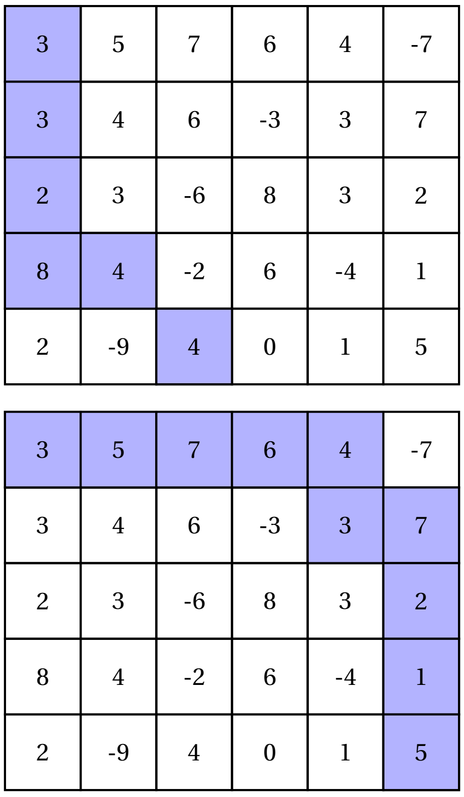


FIGURE 5.3: Maximum Path Sum (Grid)

5.5 Counting Ways

Suppose we have the cards with numbers 1 through 10 and an infinite supply for each. Then what is the number of different ways of choosing the cards with the sum n if order matters? We can make a dp array and here say dp$[x]$ to be the number of ways to choose the card with sum x where order matters. We can relate the previous term by adding the previous 10 terms to the current.

$$\text{dp}[x] = \text{dp}[x-1] + \text{dp}[x-2] + ... + \text{dp}[x-10]$$

```
int main(){

   int dp[N];

   dp[0] = 1; // 1 way to make sum of 0

   for(int i=1; i<=n; i++){
      for(int j=1; j<=10; j++){
         dp[i] += dp[i-j];
      }
   }

return 0;
}
```

Bitmask dp

Using bitmask dp, each of the bits can represent a state. Suppose we want to find the number of ways to fill in the N by M grid using 1 by 2 blocks. Here the state can be represented by the bits where if the bit is set, then that place is not occupied, while if the bit is 0, then the place has been occupied. We will make an array from it, iterate through, and continuously add the result to the next line. We can make an auxiliary array dp here where dp$[X]$[mask][checked] where X is the position we currently are at and mask is the state of that current row. Checked is needed to relate the terms from the previous position to the current position. We relate and store the following position info when checked is 0 and transfer them to when checked is 1 so that we again can relate that to the next position of that. If mask = 0, then we can conclude that line i has been filled. So if we have N rows and M columns, then the solution to the problem will be dp[N*M][0][1]. The time complexity will be $O(\text{NM}2^M)$, and the initial state dp[0][0][1] will be 1 since that would be the initial board.

```
int main(){

    int N,M;
    cin >> N >> M;

    int dp[N*M+1][(1<<N)][2];

    memset(dp,0,sizeof(dp));

    dp[0][0][1]=1;

    for(int j=0; j<M; j++){
        for(int i=0; i<N; i++){
            for(int mask =0; mask < (1<<N); mask++){
                if(mask &(1<<i)){
                    dp[j*N+i+1][mask][0] += dp[j*N+i][mask^(1<<i)][1];
                }else{
                    dp[j*N+i+1][mask][0] += dp[j*N+i][mask^(1<<i)][1];
                    if(i>=1 &&!(mask &(1<< i-1))){
            // since we know !(mask&(1<<i)) is true
                        dp[j*N+i+1][mask][0] += dp[j*N+i][mask^(1<<i-1)][
    1];
                    }
                }
            }
           for(int mask =0; mask < (1<<N); mask++){
                dp[j*N+i+1][mask][1] = dp[j*N+i+1][mask][0];
           }
        }
    }


    cout << dp[M*N][0][1];

return 0;
}
```

Sorting Algorithms 6

A **sorting algorithm** is a way to rearrange the elements. Sorting can often be helpful and many times allows us to perform more efficiently task, which otherwise might have been more difficult to solve.

6.1 Introduction

Say given n numbers of more than half are guaranteed to be the same, and we want to find that number. If the numbers are not sorted, we would have to check the numbers one by one to see which appears the most which are $O(n)$. However, if the array was sorted, we could have simply chosen the median value from the array as that would be the number we can do in $O(1)$ time complexity.

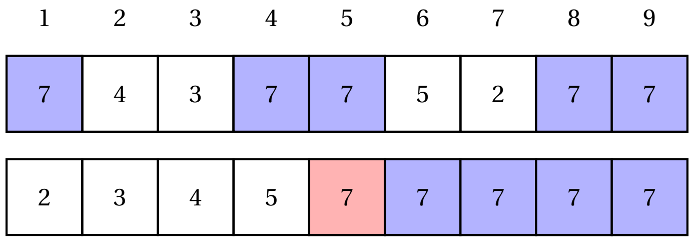


FIGURE 6.1: Sorting Algorithm Introduction

6.2 Bubble Sort

`Bubble sort` is one of the most straightforward sorting methods. It iterates through while continuously swapping the element if the order is the wrong way. Each time we iterate through each element, we would have at least one more step closer to where it is supposed to be when we perform this, the number of element times, then we will have a sorted element all in the correct position.

```
void BubbleSort(int a[], int n){
   for(int i=0; i<n-1; i++){
      for(int j =0; j< n-1; j++){
         if(a[j] > a[j+1]){
            swap(a[j],a[j+1]);
         }
      }
   }

}


int main(){

   int a[] = {5,3,2,2,3,8,13,9,18,31};

   int n = size(a);

   BubbleSort(a,n);

   for(auto v:a){
      cout << v <<" ";
   }

return 0;
}
```

We could also modify the `BubbleSort` function a little more. Say we have n elements. An important observation we can make is that the largest element will end up at the n th position from the first iteration. Next, the next greatest element would have been at $n-1$, the position at the second iteration. Like this, in the k th iteration, the $n-k+1$ th greatest element would end up at the correct position. Using this logic, we can reimplement the bubble sort function as follows.

```
void BubbleSort(int a[], int n){

   for(int i=0; i<n-1; i++){
      for(int j =0; j< n - i-1; j++){
         if(a[j] > a[j+1]){
            swap(a[j],a[j+1]);
         }
      }
   }

}
```

In both cases, the algorithm's time complexity would be $O(n^2)$ though the second im-

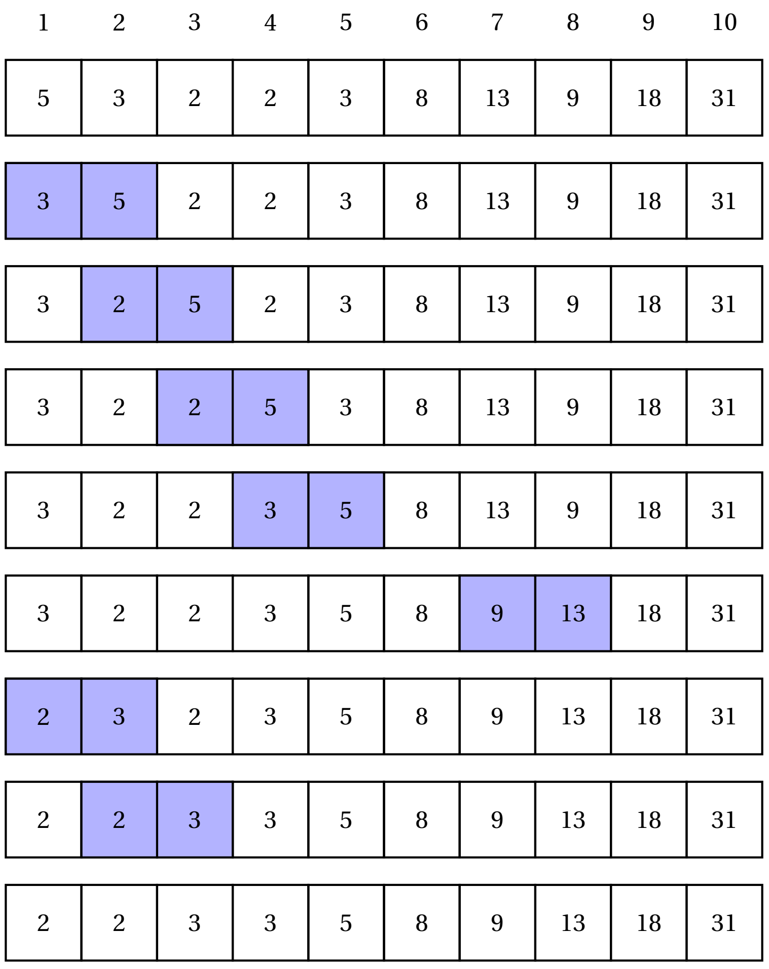


FIGURE 6.2: Bubble Sort

plementation will perform fewer operations.

6.3 Merge Sort

`Merge sort` is an efficient sorting method with the time complexity of $O(\log n)$ which uses the concept of `divide and conquer`. How we sort, the array is that given an array, we can continuously divide the array into half, sort each of the divided arrays, and then merge the arrays back to get a sorted array.

```
void Merge(int a[], int l, int mid, int r){
   int LeftSize = mid - l +1;
   int RightSize = r - mid;

   int LeftArray[LeftSize];
   int RightArray[RightSize];

   for(int i=0; i < LeftSize; i++){
      LeftArray[i] = a[l+i];
   }

   for(int i=0; i<RightSize; i++){
      RightArray[i] = a[mid + i + 1];
   }

   int L = 0, R = 0, p = l;

   while(L < LeftSize && R < RightSize){ //merging the array
      if(LeftArray[L] <= RightArray[R]){
         a[p] = LeftArray[L];
         L++;
      }else{
         a[p] = RightArray[R];
         R++;
      }
      p++;
   }

   while(L<LeftSize){ //In case where there are leftovers from left array
      a[p] = LeftArray[L];
      L++, p++;
   }

   while(R<RightSize){ //In case where there are leftovers from right
   array
      a[p] = RightArray[R];
      R++, p++;
   }
}

void MergeSort(int a[], int l, int r){

   if(l >= r) return;

   int mid = (l+r)/2; //finding the midpoint to divide the array
   MergeSort(a,l,mid); //left side of the divided array
   MergeSort(a,mid+1,r); //right side of the divided array
   Merge(a, l, mid, r); //putting the array back together
}

int main(){

   int a[] = {10,35,34,2,6,53,4,2,35,43};
   int n = size(a);

   MergeSort(a,0,n-1);

   for(int i=0; i<n; i++){
      cout << a[i] <<" ";
```

```
    }

return 0;
}
```

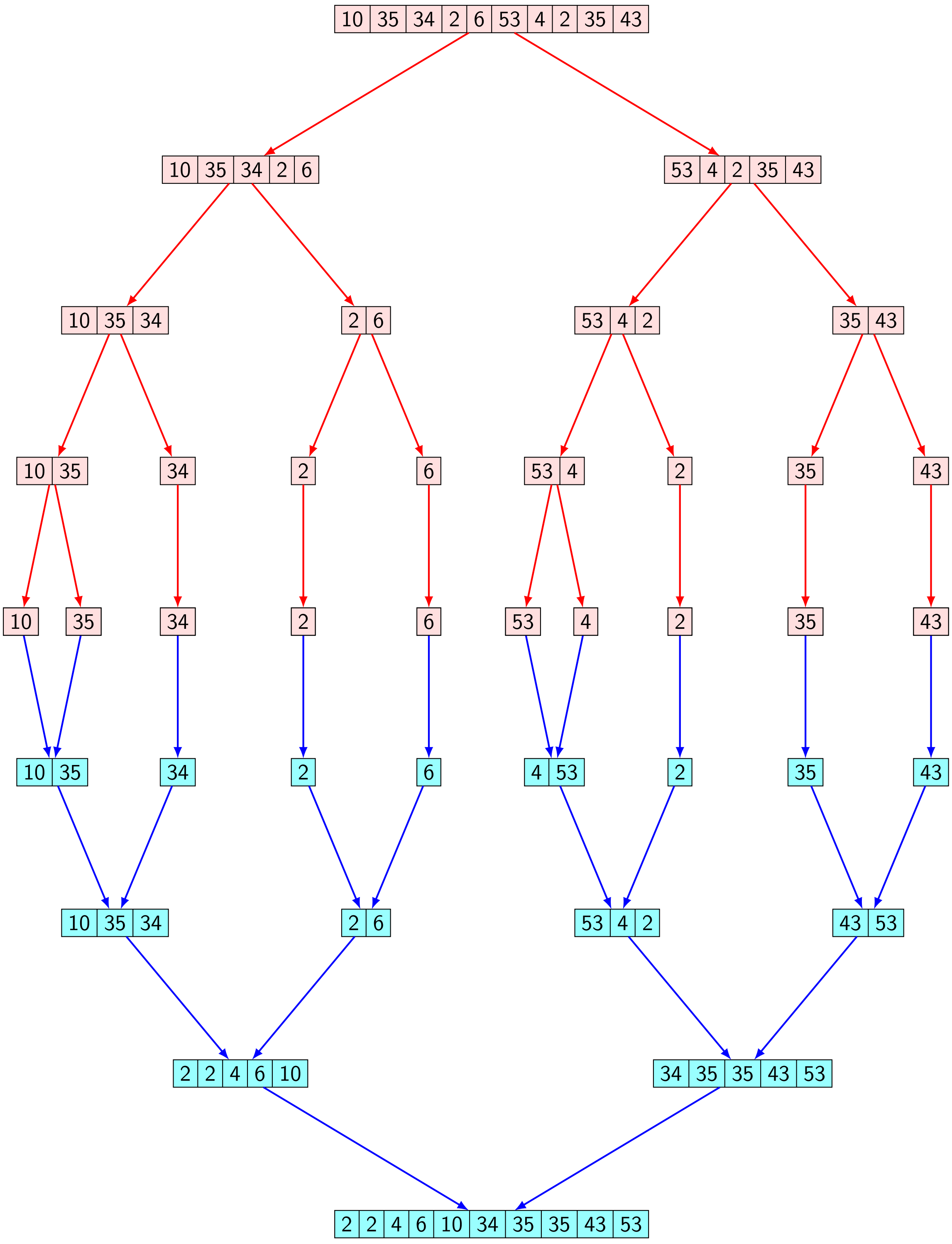


FIGURE 6.3: Merge Sort

The implementation works because, as shown in figure 6.3, it first continuously calls the function itself, MergeSort, which would then divide the array by half until there is one element for each. Then afterward, we can take the elements and start building them by merging them, shown in green in figure 6.3. How the merging works are that we make two temporary arrays, one storing the values on the left side, and another storing value on the right side. One thing to note here is that the values of these two arrays would have already been sorted as the Merge function would already have been performed to those arrays. Then using those, we could start to put them and merge them in order again and repeat this process until we merge all of the elements into one array. Then this array would have been sorted.

Merge Function

There already is an implementation for the merge function in C++ STL.

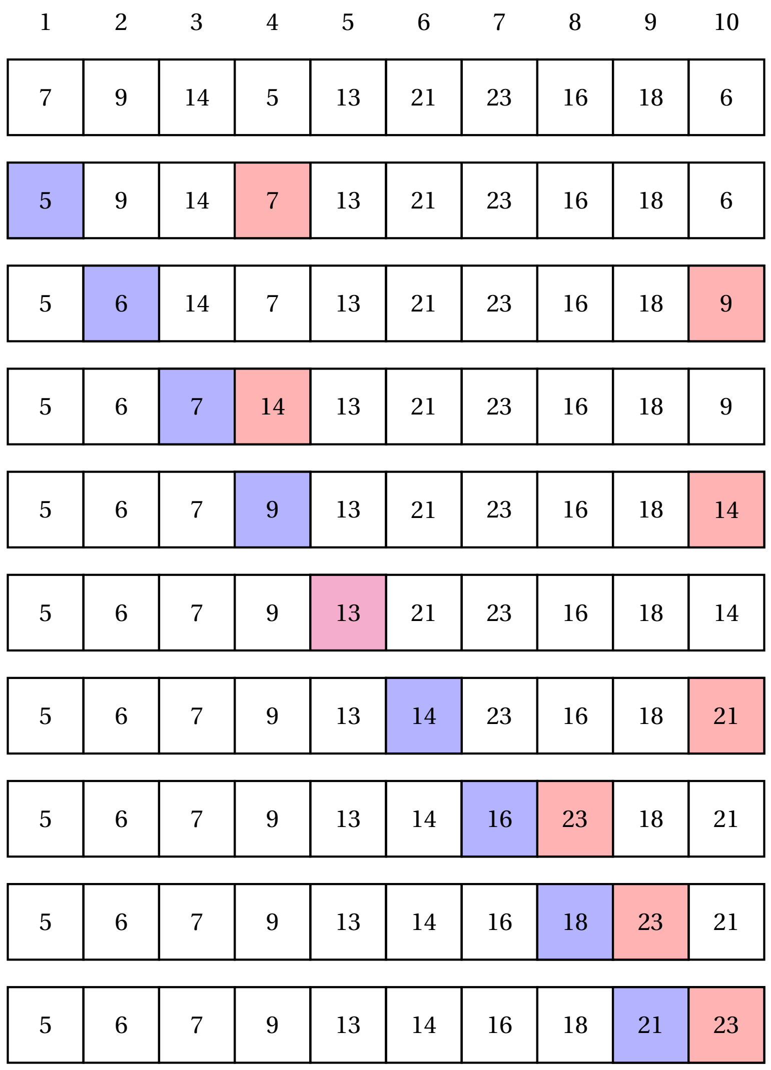
6.4 Selection Sort

The `selection sort` is another sorting method that performs at a time complexity of $O(n^2)$. How this sorting method work is that it continuously iterate through the array for which, in the i th iteration, it finds the i th smallest element in the array, which should be positioned in the i th place in the array. Using this idea, we can keep finding the minimum, excluding the ones found in the previous iteration, and reposition the next smallest element of the array.

The following is the implementation of the selection sort. We go through each of the positions in the array and check what element should be in there by checking the possible elements that could be in that position, taking the minimum element, and swapping with that element. If the element itself is already in the correct place, it will just swap with itself and move on to the next position until it reaches the end of the array, for which all elements would have been sorted in order.

```
void SelectionSort(int a[], int n){

   int i, j, MinIndex;

   for(int i=0; i<n; i++){
      MinIndex = i;
      for(int j=i+1; j<n; j++){
         if(a[MinIndex] > a[j]){
            MinIndex = j;
         }
      }
      swap(a[MinIndex], a[i]);
   }


}
```

```
int main(){

   int a[] = {7, 9, 14, 5, 13, 21, 23, 16, 18, 6};
   int n = size(a);

   SelectionSort(a, n);

   for(auto x: a){
      cout << x << " ";
   }
}
```



1	2	3	4	5	6	7	8	9	10
7	9	14	5	13	21	23	16	18	6
5	9	14	7	13	21	23	16	18	6
5	6	14	7	13	21	23	16	18	9
5	6	7	14	13	21	23	16	18	9
5	6	7	9	13	21	23	16	18	14
5	6	7	9	13	21	23	16	18	14
5	6	7	9	13	14	23	16	18	21
5	6	7	9	13	14	16	23	18	21
5	6	7	9	13	14	16	18	23	21
5	6	7	9	13	14	16	18	21	23

FIGURE 6.4: Selection Sort

6.5 Comparator Functions

The above methods show how we can sort them in increasing order of the elements. However, we can sort in various ways in the way we want by writing a `comparator functions`. In addition, we can sort various elements like strings, pairs, and more. For example, the following function sorts the pair by its second element.

```
bool Comp(pair<int,int> a, pair<int,int> b){
   return a.second < b.second;
}


int main(){

   vector<pair<int,int>> v = {{1,3}, {5, 10}, {2, 9}, {4,12}, {5,7}};

   sort(v.begin(), v.end(), Comp);

   for(auto i:v){
      cout << i.first <<"   " << i.second << '\n';
   }

return 0;
}
```

The above function will allow sorting based on one factor. However, we could have made it sorted based on multiple factors. So instead of just sorting based on the second element, we could also have considered the first element in sorting. The following code is the example that first sorts based on the first element; however, if the first element is equal, then sort it in the increasing order of the second element.

```
bool Comp(pair<int,int> a, pair<int,int> b){
   if(a.first == b.first){
      return a.second < b.second;
   }
   return a.first < b.first;
}

int main(){

   vector<pair<int,int>> v = {{1,3}, {5, 10}, {2, 9}, {4,12}, {5,7}, {1,7
   }, {1,5}};

   sort(v.begin(), v.end(), Comp);

   for(auto i:v){
      cout << i.first <<"   " << i.second << '\n';
   }
return 0;
}
```

Comparator Function for Struct

We can similarly sort the struct with the sorting functions. However, another method to sort the struct is using the comparison operator, `operator<`. The following sorts by the first element; if they are equal, it sorts by the second element.

```
struct Pair{
    int s1, s2;

    bool operator<(Pair &s){
        if(s1 == s.s1){
            return s2 < s.s2;
        }
        return s1 < s.s1;
    }
};

int main(){

    vector<Pair> v = {{1,3}, {5, 10}, {2, 9}, {4,12}, {5,7}, {1,7},
   {1,5}};

    sort(v.begin(), v.end());

    for(auto i:v){
        cout << i.s1 <<"     " << i.s2 << '\n';
    }

return 0;
}
```

Range Queries 7

A **range queries** is how to efficiently handle the queries performed in some subarray from the array. For the range queries, we can always answer certain queries by simply going through all of the elements in that subarray. However, we will be able to answer the queries more efficiently using range queries.

7.1 Introduction

The most simple range queries Say given n numbers and given to integers a and b, $1 \le a \le b \le n$, and we would like to find the sum of the values in the interval $[a, b]$. The simple way to solve this problem will be to simply go through each element from a to b and add them all. However, we can solve this problem more efficiently by making the array Sum and storing the sum of the first k values as Sum$[k]$. By this construction we can now answer the sum range query of $[a, b]$ by Sum$[b]$ - Sum$[a-1]$.

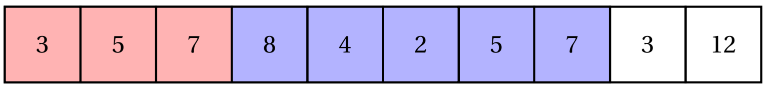


FIGURE 7.1: Sum for the range [4, 8]

```
int main(){

    int q; //number of queries
    long long a[10] = {3,5,7,8,4,2,5,7,3,12};
    long long Sum[10];
    int n = size(a);

    memset(Sum, 0, sizeof(Sum));

    Sum[0] = a[0];//first element

    for(int i=1; i<n; i++){
        Sum[i] = Sum[i-1] + a[i]; //building the prefix sums
    }

    for(int i=0; i<q; i++){ //queries
```

```
        int a,b;
        cin >> a >> b; //say the query was the sum of range [a,b]
        a--;b--; //array index starts from 0
        cout << Sum[b] - Sum[a-1] << '\n';
    }


return 0;
}
```

7.2 Sum Queries with Updates

In the previous example, the values of the array remained the same. However, what if there can be a modification in the array? Though again, the simplest way to solve this problem is to go through all arrays to do better simply. We will need the new data structure. The one that we will be presenting here is the Fenwick tree, also known as the binary-indexed tree. Fenwick tree has various applications for range queries. As discussed in chapter 4, $x = x\&(x-1)$ returns the rightmost bit from the binary representation. Using this idea, we can go through the array updating and calculate the range's sum more efficiently.

First, we can make the update function by getting the position and the value as input where we are trying to update the element's position by the value. We can review the position and continue adding the last bit of the number. this can be done by adding $x\&(-x)$ to x. Then we can continuously update the values.

```
void Update(int a, int b){
   for(int i = a; i<=X; i+= i&(-i)){ //X will be some value greater then
   the size of the array
      Sum[i] += b;
   }
}
```

To answer the queries we would need to make another function called queries. We will go through such that we get rid of the least significant bit and sum the range from $[1, x]$. And as we go through, we get the sum of the values by adding the values from the array Sum.

```
long long Query(int x){
   long long sum =0;

   for(int i = x; i > 0; i -= i&(-i)){
      sum += Sum[i];
   }
return sum;
}
```

The idea for getting the answer is similar to the previous range sum queries discussed in the introduction. When trying to calculate the sum of the range $[a, b]$, we will first calculate the sum of the range $[1, b]$ and then subtract $[1, a-1]$ to get the range $[a, b]$.

```
int main(){

    long long a[10] = {3,5,7,8,4,2,5,7,3,12};
    int n = size(a);

   for(int i =1; i<=n; i++){ //First updating the values of the array a
   to the array Sum
      Update(i, a[i-1]);
   }

   int U; // number of updates

   for(int i=0; i<U; i++){
      int x,y;
      cin >> x >> y; // say you want to update position x to y
      Update(x, -a[x-1]); // First erase the previous value
      a[x-1] = y; //assign the new value to the array
      Update(x, a[x-1]); // Update using the new value
   }

   int q; //number of queries

   for(int i=0; i<q; i++){
      int x,y; // Say we want to find the sum of the range [x, y]

      cout << Query(y) - Query(x-1);
   }


return 0;
}
```

7.3 Minimum Queries

Another well-known range query is the `range minimum query` which is the query that identifies the minimum element in some arrange $[a, b]$. Though again, for each query we will be able to answer the query simply by going through each element and finding the minimum we can do more efficiently by using the segment tree. The concept of the segment tree is straightforward. We give each node an ID which will be used to find the minimum value of itself and the subset of its child. And here, using the tree is convenient as the height of the tree is at most $\log n$, so instead of having to go through each of the elements with $O(n)$ time complexity which we will have to if it was in the form of the array by representing them as a tree we could reduce the number of times we have to perform operations.

So for each array, we first have to give it a tree representation of the array. This can be done by putting the first element as a root, multiplying by 2 if we move to the left, multiplying by 2, and adding 1 if we move to the right. And by this, it is not hard to see that each node has at most two children directly connected.

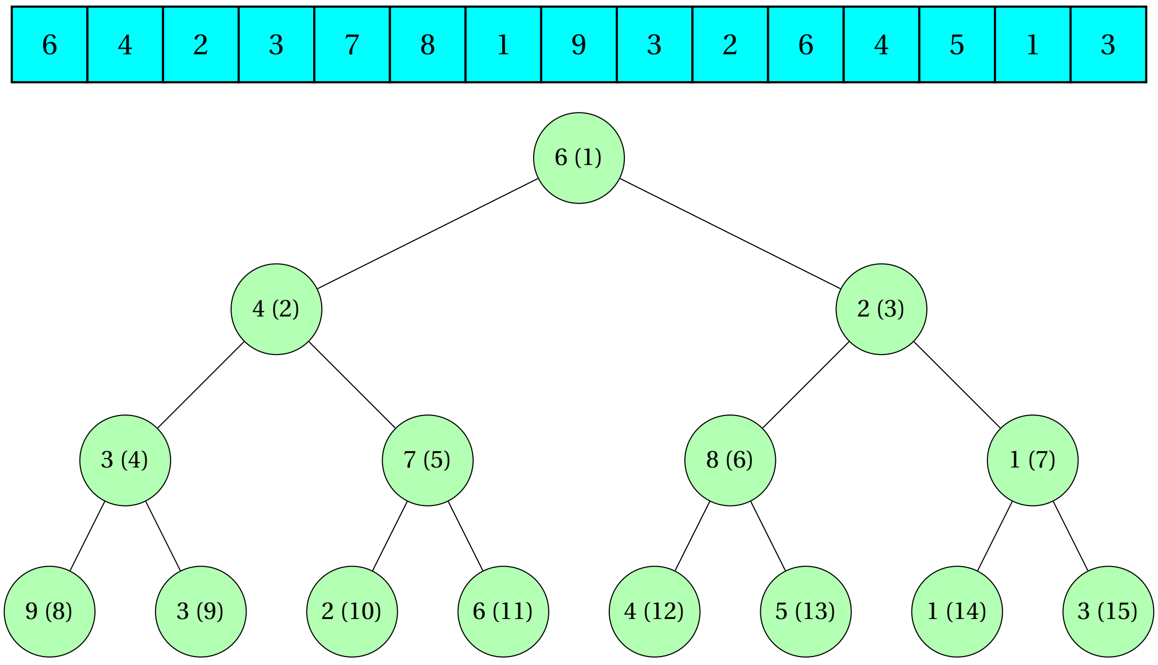


FIGURE 7.2: Tree representation of the array

Now to answer the question of what the minimum is in a given range, what we can do is first build the segment tree. First, we have to make a temporary array rmq which will store the minimum value of its subtree. This can be done by passing three arguments the ID of the node and the left and right bound of the array. If the left and right bound are the same, then we can store that value in the rmq array, while if not, we can first build the left and right subtree and then store the minimum value among them.

```
int BuildRmq(int id, int l, int r){
    if(l == r){
        return rmq[id] = x[l];
    }else{
        int mid = (l+r)/2;
        return rmq[id] = min(BuildRmq(id*2,l,mid),BuildRmq(id*2+1, mid+1,
    r));
    }
}
```

And so if we were to answer the minimum in the range $[a, b]$, we can go through the tree giving its ID. This can be done by passing an additional left and right bound, which we will use to go through the tree. If the additional left and right bound, called $landr$ respectively, satisfies $a \le l$ and $r \le b$, then we can return the minimum of its subtree as part of the range. Another thing is if $r < a$ or $b < l$, meaning none of the elements is part of the array, then we can return any value greater than the maximum possible element, which means that this element will not get considered. Suppose both situations are not the case when there is partial overlap, representing that some of the element is in the array while some are not. In that case, we can divide it into its left and right subtree again and go through the query again, returning the minimum value among them.

```
int Query(int id, int l,int r, int a, int b){

    if(a<=l && r<= b){
        return rmq[id];
    }

    if(r < a || b < l){
        return MAX_ELEMENT;
    }

    int mid = (l+r)/2;

    return min(Query(id*2, l, mid, a, b), Query(id*2 +1, mid +1, r, a, b)
   );
}
```

Now, answering the queries can be done, and building the tree can be done as follows. First building the tree can be done by BuildRmq(1, 1, n) where the first element will be 1 as we are starting from the root node which will be the tree with all of the elements so l = 1, r = n. For answering each of the queries calculating the minimum value in the range $[a, b]$, we should call the functions Query(1, 1, n, a, b) as first we will be considering all element. We added two more values, a and b, as this will be constant throughout and will be used to compare the value while the second and third arguments l and r change.

```
int main(){

    int q; //number of queries
    int x[10] = {1,5,3,2,16,12,4,8,6,9};
    int n = size(x);
    BuildRmq(1, 1, n);

    for(int i=0; i<q; i++){
        int a,b;
        cin >> a >> b;
        cout << Query(1, 1, n, a, b) << '\n';
    }

return 0;
}
```

7.4 Minimum Queries with Updates

Now, what if we want to do minimum queries while updating the elements? We can use the same as the minimum query with no updates for the build and the query function. The only additional function that we have to make now is the update function. How the update function works is that we get input for the position we are trying to change and the value we are trying to update to. Once we know the position first, we can compare it with our boundaries which would be the additional arguments of the function. If the position is out of the boundary, we ignore it, while if not, we check whether we are in the position for which we will have to update the element. In this situation $l = \text{pos} = r$ so the condition is $l = r$. If this is also not the case, then we can continue on finding it by moving on to its children node by dividing the tree into its left child node and its right child node then, after we go through both sides update the final value as the minimum of the two.

```
void Update(int id, int pos, int val, int l, int r){

    if(pos < l || pos > r){
        return;
    }

    if(l == r){
        rmq[id] = val;
        return;
    }

    int mid = (l+r)/2;

    Update(id*2, pos, val, l, mid);
    Update(id*2+1, pos, val, mid + 1, r);
    rmq[id] = min(rmq[id*2], rmq[id*2 + 1]);
}
```

Maximum Queries

The `maximum query` problem can be solved the same way as the minimum query with little modification.

7.5 Other Queries

In the previous sections, we discussed common range queries and how to construct the build, query, and/or update functions. Some other types of queries can also be solved using those ideas.

Counting Queries

In the introduction, we calculated the sum in the range using the prefix array Sum. When the query is something related to counting or something that accumulates, we can use a similar idea to answer the query though if there is an update, this might not be the best choice. For example, let us say we answer the prime range query where we try to count the prime in the given range. As with the sum query we first have to build the prefix array. And for this, as we are trying to count the number of primes, we have to traverse the array and increase by one if that element is a prime and just continue if not. And first, we need to check whether an element is a prime for each element. However, if we do that for each element, it will take a while. To make the program more efficient, we can use `Sieve of Eratosthenes`.

Sieve of Eratosthenes

Here is how the algorithm works. First, starting from 2, as 1 is not a prime, we go through the numbers until we reach the number that has not yet been erased, then select that number to be prime. Then we delete all its proper multiples (all numbers greater than that number and divisible by that number) in the range. Then we select the undeleted number, mark that as a prime, erase all its proper multiples, and repeat this throughout the range. (Note how this algorithm always works as an undeleted number means no smaller number divides this number)

	2	3	~~4~~	5	~~6~~	7
~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~
~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~
~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~
29	~~30~~	31	~~32~~	~~33~~	~~34~~	~~35~~
~~36~~	37	~~38~~	~~39~~	~~40~~	41	~~42~~
43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~

Now combining these ideas, we can develop a build function as follows.

```
void BuildPrefix(){
    bool prime[N + 5]; // N should at least be greater than or equal to
   the maximum element in the array
```

```
    int ans[N + 5];

    memset(prime, true, sizeof(prime));

    for(int x = 2; x*x <= N; x++){
        if(prime[x]){
            for(int y = x*2; y <= N; y += x){
                prime[y] = false;
            }
        }
    }

    if(prime[array[0]]){ //if first element is prime increase by 1 else
   just let be 0
        ans[0] = 1;
    }else{
        ans[0] = 0;
    }

    for(int i = 0; i< n; i++){ // n is the number of elements in the
   array
        ans[i] = ans[i-1]; //store previous value
        if(prime[array[i]]){ //if this element in the array is a prime
   then we increase the answer by 1
            ans[i]++;
        }
    }
}
```

GCD Queries

The following range query that we will discuss is the GCD Queries, for which we try to calculate the GCD of the given range from the array. This range query can be solved similarly to the minimum range query. We can first construct the tree by splitting it into half, which will be its children, and take the GCD of them after going through that array, for which we input the value of the array if we reach the tree leaves.

```
int BuildGCD(int id, int l, int r){

    if(l == r){
        return Gcd[id] = x[l];
    }else{
        int mid = (l+r)/2;

        return Gcd[id] = __gcd(BuildGCD(id*2,l, mid), BuildGCD(id*2 +1,
   mid + 1, r));
    }
}
```

The query function can also be solved with a segment tree in the same way where we consider the additional pointers of the array l and r for, which will be used as with the id to go through the tree until the number range fits within the range for which we will return the subtree from it and if it is out entirely out of the range we can just return 0 as $GCD(n,0) = n$ it will not have any effect in calculating the final GCD.

```
int Query(int id, int l, int r, int a, int b){

    if(a <=l && r <= b){
        return Gcd[id];
    }

    if(r < a || b < l){
        return 0;
    }

    int mid = (l + r)/2;

    return __gcd(Query(id*2, l, mid, a, b), Query(id*2 + 1, mid + 1, r, a
   , b));
}
```

Good practice

As with the previous queries, different-looking queries can be solved with similar methods. When solving problems, it might be often helpful to think about how to answer the query and how to modify the array to be used for the query function.

7.6 Sparse Table

Another technique for solving range query is `sparse table`. It can sometimes be used to solve a range query that has no updates. The idea is that for any array for the range $[l, r]$, we can divide the array into two power of 2 (which may include some overlaps). And we have to keep the power of 2 to be the minimum such that if we have two of them, it will be at least as great as the length of the array. The trick to doing this is to use the log2 function. So if the range was $[l, r]$, we could use two $2^{log2(r-l+1)}$ size arrays to include all of the elements in the array. For example if we wanted to calculate the range $[5, 17]$, the length of the array is $17 - 5 + 1 = 13$ so we can use the $2^{log2(13)} = 2^3 = 8$.

Size of the block

It is important to note that it has to be the minimum satisfying number as if it was something greater, it may overflow the array and include some elements that are not in the range.

So now how we can answer the range query is that we make an additional array which will include the element we are starting from and will also store some number k for which it will consist of the function applied to the next 2^k elements. This can be done as follows.

```
for(int i =1; i< M2 ; i++){//2^(M2) should be at least N
   for(int j =0; j<N - (1<<i); j++){
      x[j][i] = Function(x[j][i-1], x[j + (1<<(i-1))][i-1]); //Function
    will be the question that we are trying to answer
   }
}
```

From here, we originally would have to store the array in this additional array. So for example if we were trying to perform the operation on array a then for each of the element $a[i]$ should be stored in the additional array as $x[i][0]$.

Once we construct this additional array, we can now perform queries in the array by the logic discussed above.

```
//Query of the range [a,b]
void Query(int a, int b){
   int k = log2(b - a + 1); //Finding the minimal such that whole range
    will be represented when we approach from both side of the subarray
   cout << Function(x[a][k], x[b - (1<<k) + 1][k]);
}
```