



Explore | Expand | Enrich



Explore | Expand | Enrich

Helping Students
to Realize Their Dreams
for the Last 14 Years



VIT AP Winter Session

Explore | Expand | Enrich

Competitive Programming



Topic: Linked List

Introduction

A loop in a linked list is a condition that occurs when the linked list does not have any end.

When the loop exists in the linked list, the last pointer does not point to the Null as observed in the singly linked list or doubly linked list and to the head of the linked list observed in the circular linked list.



When the loop exists, it points to some other node, also known as the linked list cycle.

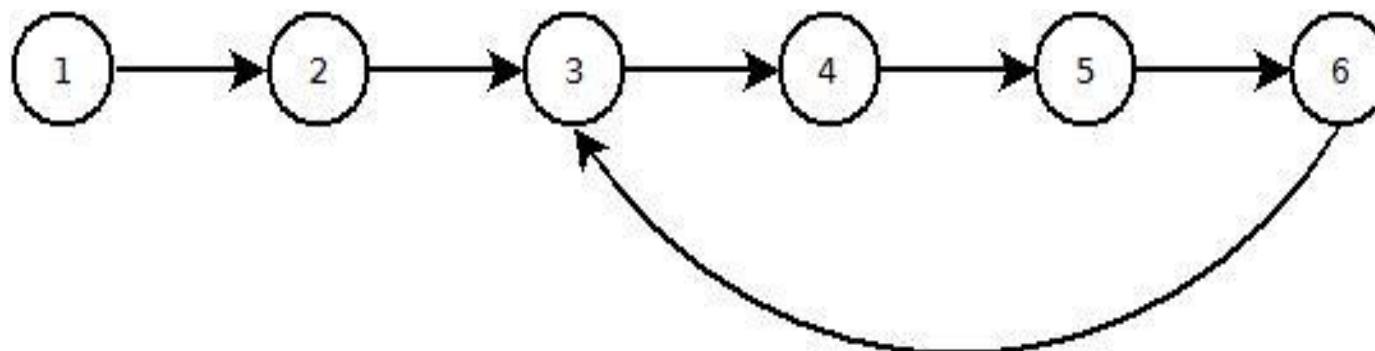


Loop Detection



Explore | Expand | Enrich

Example



Solution: Using hashing

The simplest approach to solve this problem is to check whether a node in the linked list has been visited before. To perform this operation, a hashmap can be used.

Algorithm

- Initialise a hashmap.
- Traverse the linked list till the head pointer isn't NULL:
 - If the current node is already present in the hashset, it ensures that the linked list contains a loop. Hence, terminate and return True.
 - Else, continue traversing and continue inserting the node into the hashset.
- Return False if it doesn't satisfy the above conditions.



Loop Detection



Explore | Expand | Enrich

Program: Using hashing Approach

linkedlistloop1.java

Time Complexity: $O(n)$

Auxiliary Space Complexity: $O(n)$



Solution: Two Pointer approach

Also known as Floyd's Cycle Detection Algorithm

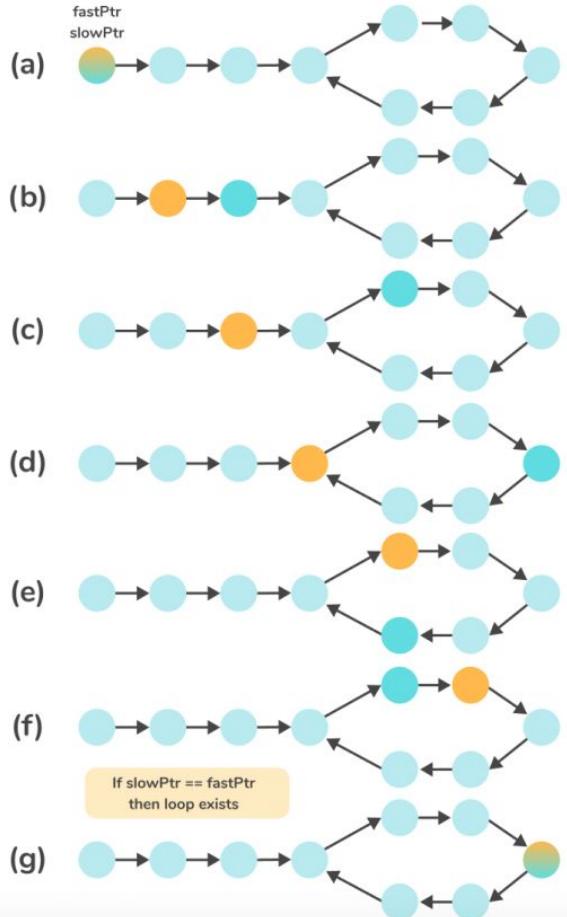
This approach uses a two-pointer – a fast pointer and a slow pointer to determine if there exists a cycle in the loop.

The slow pointer moves one node ahead at a time, while the fast pointer moves two nodes ahead at a time.

If a loop exists in the linked list, the fast and slow pointers are bound to meet at some point.

Loop Detection

Example



`slowPtr`



`fastPtr`

Explore | Expand | Enrich



Algorithm: Using Two Pointers

- Initialise two pointers, fast and slow to the head of the linked list.
- Traverse through the linked list until the fast pointer doesn't reach the end of the linked list.
- If the fast pointer reaches the end, it means that the linked list doesn't contain any cycle. Hence, return False.
- Else, move the slow pointer by one node i.e. $\text{slow} = \text{slow} \rightarrow \text{next}$ and fast pointer by two nodes i.e. $\text{fast} = \text{fast} \rightarrow \text{next} \rightarrow \text{next}$.
- At any point, if the fast and the slow pointers point to the same node, return True as a loop has been detected.



Loop Detection



Explore | Expand | Enrich

Program: Using Two Pointers

linkedlistloop2.java

Time Complexity: $O(n)$

Auxiliary Space Complexity: $O(1)$





Sort the bitonic DLL



Explore | Expand | Enrich

Sort the bitonic doubly linked list

Sort the given bitonic doubly linked list.

A bitonic doubly linked list is a doubly linked list which is first increasing and then decreasing.

A strictly increasing or a strictly decreasing list is also a bitonic doubly linked list.





Sort the bitonic DLL



Sort the bitonic doubly linked list

Input : DLL: 2<->5<->7<->12<->10<->6<->4<->1

Output : 1<->2<->4<->5<->6<->7<->10<->12

Input : DLL: 20<->17<->14<->8<->3

Output : 3<->8<->14<->17<->20



Sort the bitonic DLL



Algorithm

- Find the first node in the list which is smaller than its previous node. Let it be current.
- If no such node is present then list is already sorted.
- Else split the list into two lists, first starting from head node till the current's previous node and second starting from current node till the end of the list.
- Reverse the second doubly linked list.
- Merge the first and second sorted doubly linked list.
- The final merged list is the required sorted doubly linked list.



Sort the bitonic DLL



Program: Approach #1

bitonicDLL.java

Time Complexity: $O(n)$

Sample IO

• Input

2 5 7 12 10 6 4 1

Output

1 2 4 5 6 7 10 12





Segregate even & odd nodes in a LL



Explore | Expand | Enrich

Introduction

Given a linked list, rearrange it by separating odd nodes from even ones.

All even nodes should come before all odd nodes in the output list

The relative order of even and odd nodes should be maintained.

Example

Given a linked list of the following elements in order {1, 2, 3, 4, 5, 6, 7}, the final result should be {2, 4, 6, 1, 3, 5, 7}



Program: Approach #1: Using Iteration

seggLL1.java

Time Complexity: O(n)

Sample IO

- Input

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → null

Output

2 → 4 → 6 → 8 → 10 → 1 → 3 → 5 → 7 → 9 → null

Program: Approach #2: Using Recursion

seggLL2.java

Time Complexity: O(n)

Sample IO

- Input

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → null

Output

2 → 4 → 6 → 8 → 10 → 1 → 3 → 5 → 7 → 9 → null





Merge sort for DLL



Introduction to merge sort

Merge Sort is a Divide and Conquer algorithm.

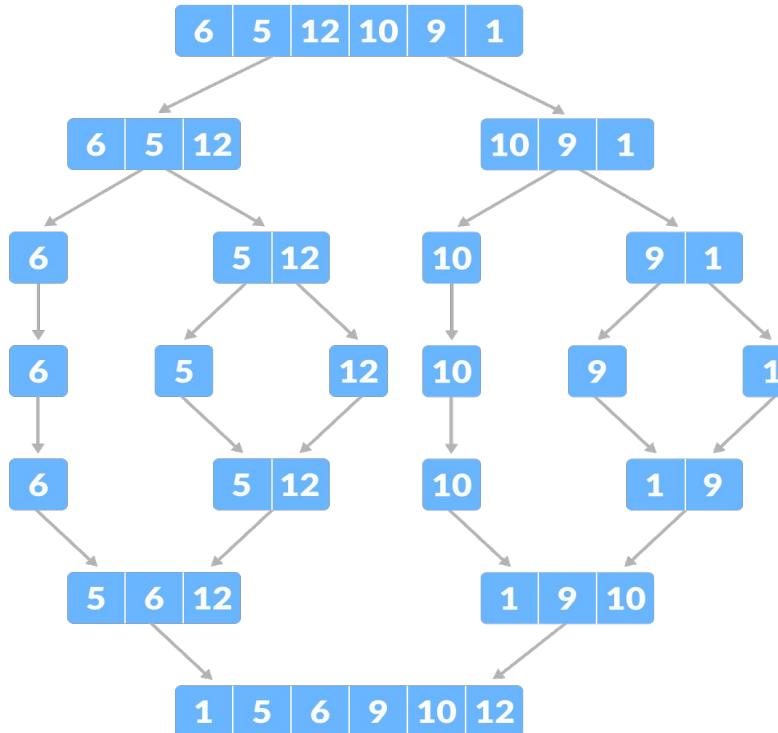
It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves

The merge() function is used for merging two halves.

The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Merge sort for DLL

Introduction to merge sort





Merge sort for DLL



Explore | Expand | Enrich

Introduction

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort

The merge sort algorithm on the doubly linked list works by splitting the list into two halves, sorting each sublist recursively

It finally merges both the sorted lists together to get a single sorted list.





Merge sort for DLL



Program: Merge sort for DLL

mergesortDLL.java

Time Complexity: $O(n \log n)$

Sample IO

• Input

6 —> 4 —> 8 —> 7 —> 9 —> 2 —> 1 —> null

Output

1 —> 2 —> 4 —> 6 —> 7 —> 8 —> 9 —> null



Introduction

The problem statement is to design a stack to support an additional operation that returns the minimum element from the stack in constant time.

The stack should support the following regular stack operations

- push(x) – Push element x onto stack.
- pop() – Removes the element on top of the stack.
- top() – Get the top element.
- getMin() – Retrieve the minimum element in the stack



Minimum Stack



Explore | Expand | Enrich

Example

					 Value popped is equal to Min. Pop a second time and update Min		
MinStack() Min=Integer.MAX_Value	push(-2) Min= -2	push(0) Min= -2	push(-3) Min= -3	getMin() Min= -3	pop() Min= -3 Min= -2 (value popped is equal to Min. Pop a second time and update Min)	top() Min= -2 (return 0)	getMin() Min= -2 (return -2)



Problem Statement

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.





Minimum Stack



Explore | Expand | Enrich

Program: Uses 2 stacks

minStack1.java

We maintain 2 stacks. One for normal operations on stack and one for maintaining the minimum value after each pop.

The idea here is that for each element we push in the stack we need to check that if this element is less than the one we are pushing to the min stack, if yes we push the new value or

- we repeat the value from peek().

This way we ensure we keep track of the minimum value at any state in the stack



Minimum Stack



Example

Input

```
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[ ],[-2],[0],[-3],[ ],[ ],[ ],[ ]]
```

Output

```
[null,null,null,null,-3,null,0,-2]
```

- ### Explanation

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();   // return 0
minStack.getMin(); // return -2
```



Minimum Stack



Explore | Expand | Enrich

Program: Without Aux Stack

minStack2.java

Sample IO

Input

In the program

Output

6

6

5

3

5

6





The Celebrity Problem



Introduction

In a party of N people, only one person is known to everyone.

Such a person may be present in the party, if yes, (s)he doesn't know anyone in the party.

We can only ask questions like “does A know B? ”. Find the stranger (celebrity) in minimum number of questions.

We can describe the problem input as an array of numbers/characters representing persons in the party.





The Celebrity Problem



Problem Statement

A square NxN matrix $M[][]$ is used to represent people at the party.

In the matrix, if an element of row i and column j is set to 1 it means i th person knows j th person.

Here $M[i][i]$ will always be 0.





The Celebrity Problem



Explore | Expand | Enrich

Problem Statement

A square NxN matrix $M[][]$ is used to represent people at the party.

In the matrix, if an element of row i and column j is set to 1 it means i th person knows j th person.

Here $M[i][i]$ will always be 0.





Celebrity Problem



Explore | Expand | Enrich

Program: Brute Force

celebprob1.java

Time Complexity: $O(n^2)$

Space Complexity: $O(n)$

Sample IO

Input

- N = 8

M=

0, 0, 1, 0

0, 0, 1, 0

0, 0, 0, 0

0, 0, 1, 0

Output

Celebrity ID: 2



Celebrity Problem



Program: Better Approach

celebprob2.java

Sample IO #1

Input

N = 3

M[][] = {{0 1 0},
 {0 0 0},
 {0 1 0}}

Output

Celebrity ID: 1

Explanation

0th and 2nd person both
know 1. Therefore, 1 is the celebrity

Time and Space Complexities

O(N) and O(1) respectively

Sample IO #2

Input

N=2

M[][] = {{0 1},
 {1 0}}

Output

No Celebrity

Explanation

The two people at the party both
know each other. None of them is a celebrity.



Iterative Tower of Hanoi



Explore | Expand | Enrich

Introduction

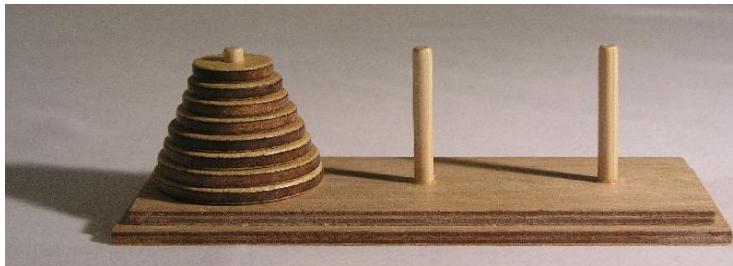
The Tower of Hanoi, is a mathematical problem which consists of three rods and multiple disks.

Initially, all the disks are placed on one rod, one over the other in ascending order of size similar to a cone-shaped tower.

The objective of this problem is to move the stack of disks from the initial rod to another rod,

- following these rules:

- A disk cannot be placed on top of a smaller disk
- No disk can be placed on top of the smaller disk.





Iterative Tower of Hanoi



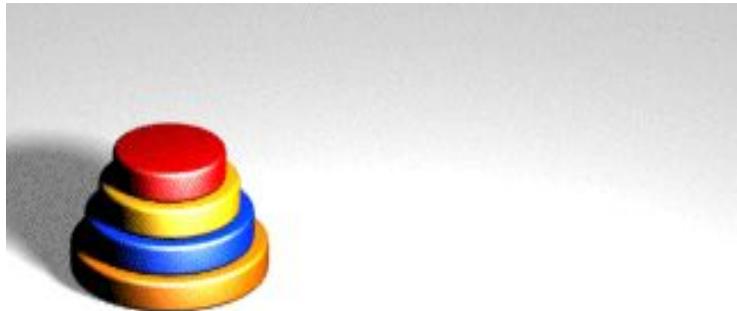
Explore | Expand | Enrich

Introduction

The goal is to move all the disks from the leftmost rod to the rightmost rod. To move N disks from one rod to another, $2^N - 1$ steps are required.

So, to move 3 disks from starting the rod to the ending rod, a total of 7 steps are required.

Below is an example of solving the Tower of Hanoi





Iterative Tower of Hanoi



Explore | Expand | Enrich

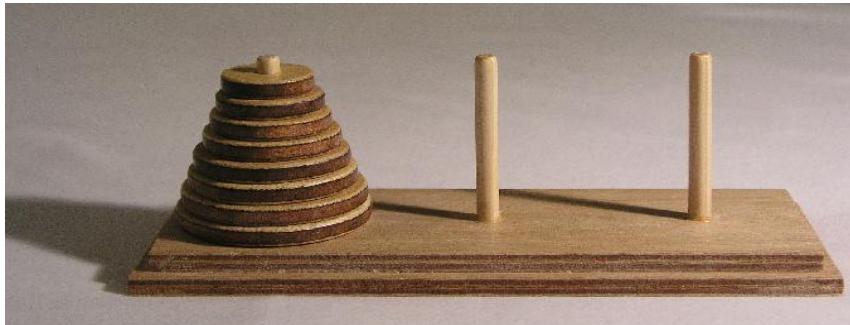
Problem Statement

Move all the disks stacked on the first tower over to the last tower using a helper tower in the middle. While moving the disks, certain rules must be followed. These are :

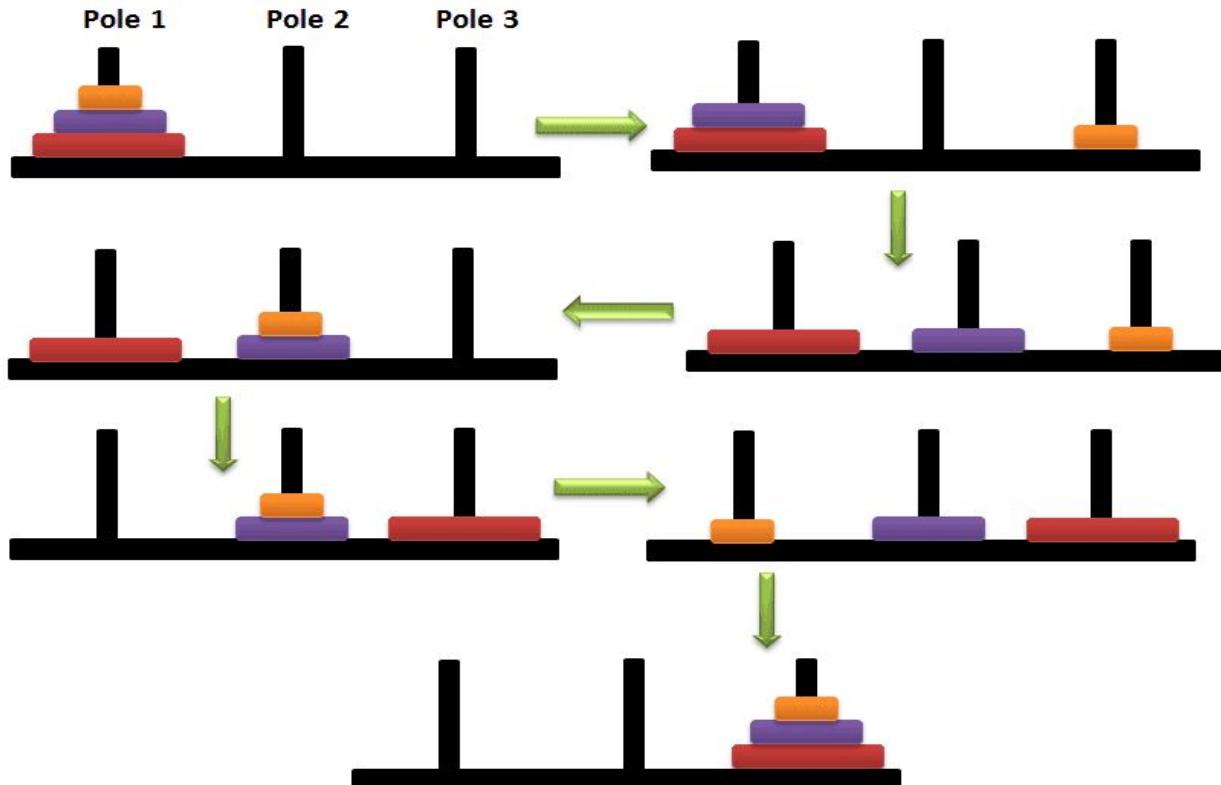
1. Only one disk can be moved.
2. A larger disk can not be placed on a smaller disk.

You can only move one disk at a time and never place a smaller disk over a larger disk.

- To do this you have an extra tower, it is known as helper/auxiliary tower.



Iterative Tower of Hanoi





Iterative Tower of Hanoi



Explore | Expand | Enrich

Approach

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

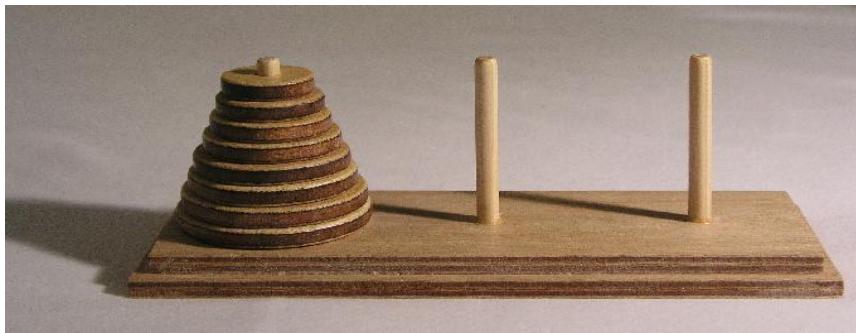
Step 3 : Shift first disk from 'B' to 'C'.

- The pattern here is :

Shift ' $n-1$ ' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift ' $n-1$ ' disks from 'B' to 'C'.





Iterative Tower of Hanoi



Examples

For two disks i.e. $n = 2$

Disk 1 moved from A to B
Disk 2 moved from A to C
Disk 1 moved from B to C

For three disks i.e. $n = 3$

Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C



Program: Recursive Solution

tohanoi1.java

Sample IO #1

Input

N = 4

Output

Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C



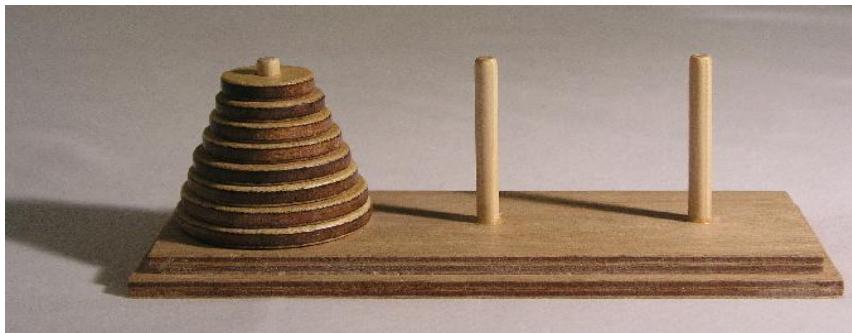
Iterative Tower of Hanoi



Explore | Expand | Enrich

Iterative Approach

We have seen that for n disks, a total of $2^n - 1$ moves are required.





Iterative Tower of Hanoi

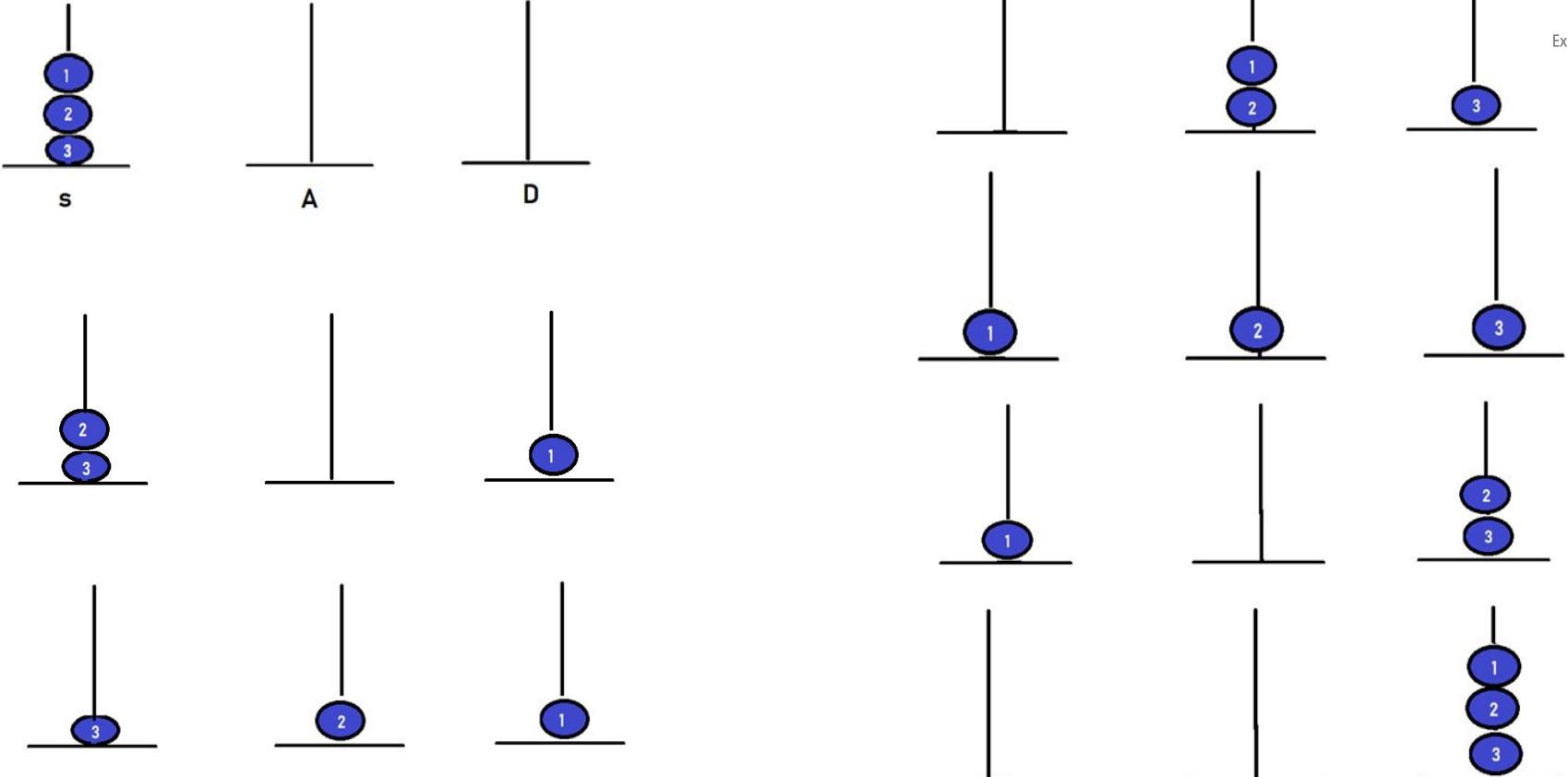
Iterative Algorithm

1. Calculate the total number of moves required i.e. $\text{pow}(2, n) - 1$. Here n is number of disks.
2. If number of disks (i.e. n) is even then interchange destinationpole and auxiliary pole
3. for i = 1 to total number of moves:
 - if $i \% 3 == 1$:
 Legal movement of top disk between source pole and destination pole
 - if $i \% 3 == 2$:
 Legal movement top disk between source pole and auxiliary pole
 - if $i \% 3 == 0$:
 Legal movement top disk between auxiliary pole and destination pole

Iterative Tower of Hanoi



Explore | Expand | Enrich



Program: Iterative Solution

tohanoi2.java

For the iterative approach, we will use Stack Data Structure. The stack will store and remove the disks in LIFO(Last In First Order) principle

Sample IO #1

Input

N = 3

Output

Move the disk 1 from 'S' to 'D'
Move the disk 2 from 'S' to 'A'
Move the disk 1 from 'D' to 'A'
Move the disk 3 from 'S' to 'D'
Move the disk 1 from 'A' to 'S'
Move the disk 2 from 'A' to 'D'
Move the disk 1 from 'S' to 'D'



Stock Span Problem



Explore | Expand | Enrich

Introduction

The stock span problem is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate the span of stocks price for all n days.

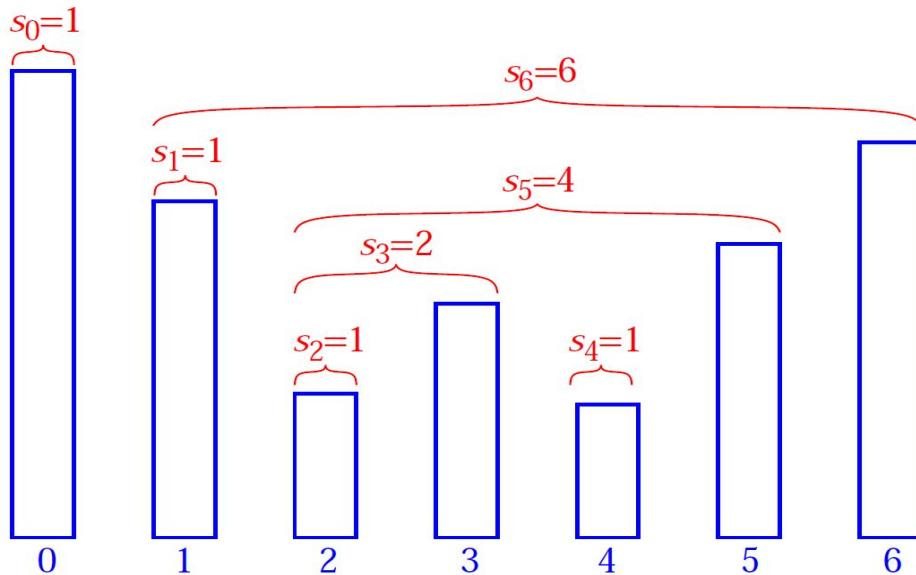
The span S_i of the stocks price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.

For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}.

Stock Span Problem

Introduction

prices = {100, 80, 60, 70, 60, 75, 85}
span = {1, 1, 1, 2, 1, 4, 6}.





Stock Span Problem



Explore | Expand | Enrich

Program: Basic Solution

stockspan1.java

Time Complexity: $O(n^2)$

Sample IO #1

Input

- `price[] = { 10, 4, 5, 90, 120, 80 };`

Output

1 1 2 4 5 1





Stock Span Problem



Program: Linear Solution

stockspan2.java

Time Complexity: $O(n)$

Auxilliary Space Complexity: $O(n)$

Sample IO #1

Input

- `price[] = { 10, 4, 5, 90, 120, 80 };`

Output

1 1 2 4 5 1





Stock Span Problem



Explore | Expand | Enrich

Program: Another Linear Solution
(Without using Stack)

stockspan3.java

Time Complexity: O(n)

Sample IO #1

• Input

price[] = { 10, 4, 5, 90, 120, 80 };

Output

1 1 2 4 5 1



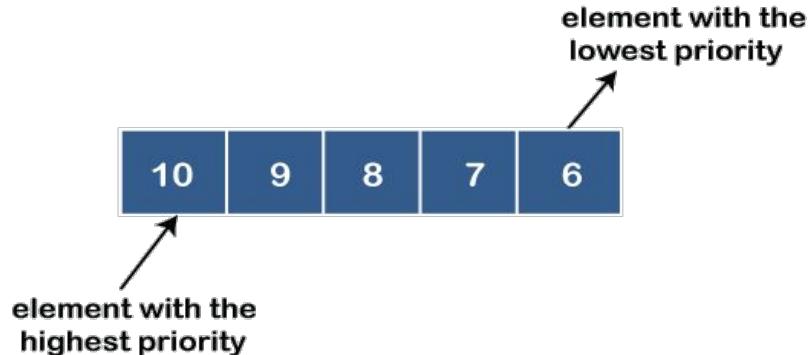
Priority Queue using DLL

Introduction

A priority queue is a special type of queue in which each element is associated with a priority value.

Elements are served on the basis of their priority. That is, higher priority elements are served first.

However, if elements with the same priority occur, they are served according to their order in the queue.



Priority Queue using DLL

Introduction

Generally, the value of the element itself is considered for assigning the priority.

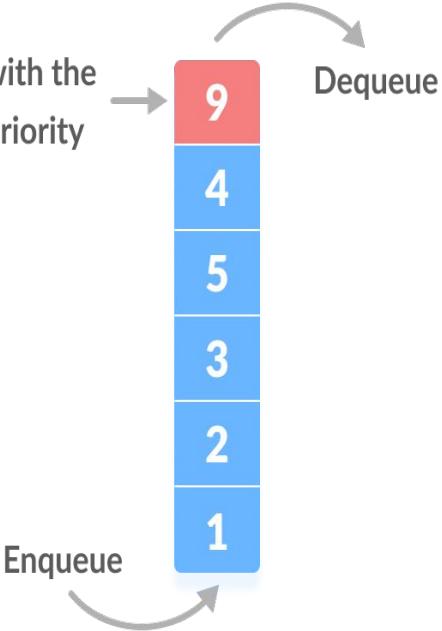
However, in other cases, we can assume the element with the lowest value as the highest priority element.

- We can also set priorities according to our needs.

In a normal queue, the first-in-first-out rule is implemented

In a priority queue, the values are removed on the basis of priority.

Element with the highest priority



Priority Queue using DLL

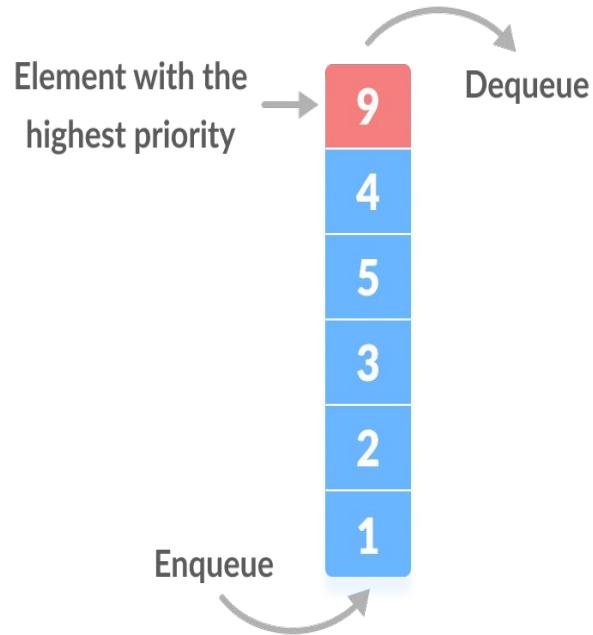
Implementation

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.

Among these data structures, heap data structure provides an efficient implementation of priority queues.

- However we would be using a DLL to implement priority queue in this section.

The operations would be: push(), pop(), peek()





Priority Queue using DLL



Introduction

To implement Priority Queue using Doubly Linked Lists, we need to have the operations

- `push()`: This function is used to insert a new data into the queue.
- `pop()`: This function removes the element with the highest priority from the queue.
- `peek() / top()`: This function is used to get the highest priority element in the queue without removing it from the queue.

Approach

1. Create a doubly linked list having fields

- info: hold the information of the Node
- priority: hold the priority of the Node
- prev: point to previous Node
- next: point to next Node

2. Insert the element and priority in the Node.

3. Arrange the Nodes in the increasing order of priority.

Program: PQ using DLL

pqueuedll.java

Here, in pop() operation, we remove node with the least priority, and in peek(), we peek at the node with the least priority. This can be reversed

Sample IO #1

- Input

DLL of

2 3 4 5 6 1

with respective priorities of

3 4 5 6 7 2

Time Complexity: O(n) for push, linear for peek() and pop()

Output

1

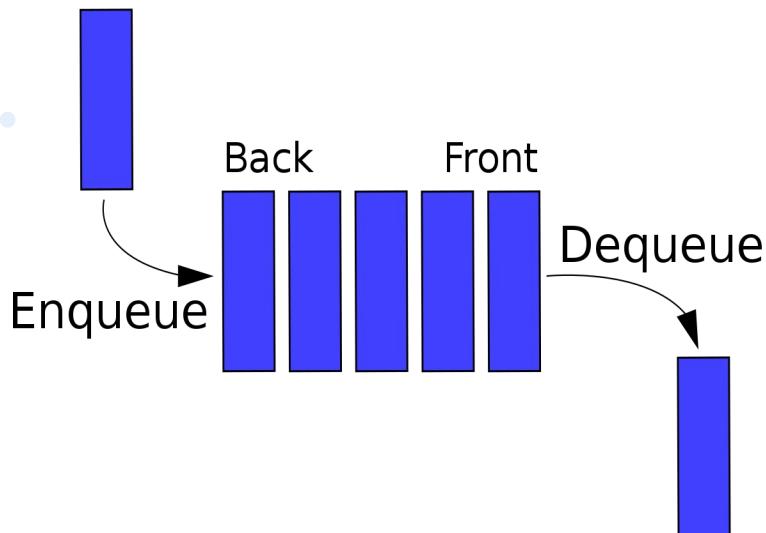
2

Sort without using extra space

Introduction

We have been given a queue with random integer elements. The problem is that we need to sort the elements of the queue without using extra space.

Remember, the operations that are possible with a queue are enqueue, dequeue.



Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue



Sort without using extra space



Introduction

If we are allowed extra space, then we can simply move all items of queue to an array, then sort the array and finally move array elements back to queue.

Examples

Input

10 -> 7 -> 2 -> 8 -> 6

Output

2 -> 6 -> 7 -> 8 -> 10

Input

66 -> 1 -> 18 -> 23 -> 39

Output

1 -> 18 -> 23 -> 39 -> 66



Algorithm

Consider the queue made up of two parts, one is sorted and the other is not sorted.

Initially, all the elements are present in not sorted part.

At every step, find the index of the minimum element from the unsorted queue

- and move it to the end of the queue, that is, to the sorted part.

Repeat this step till all the elements are not present in the sorted queue.





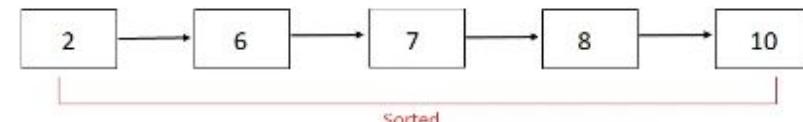
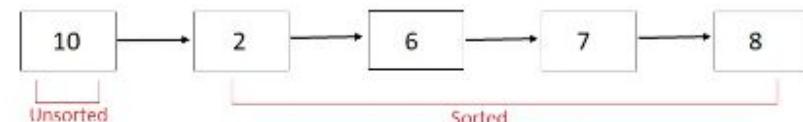
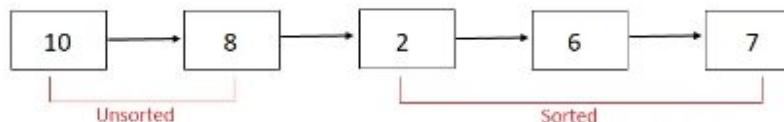
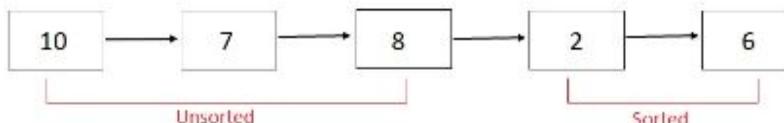
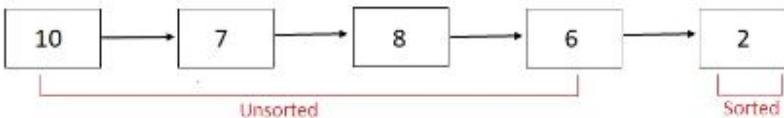
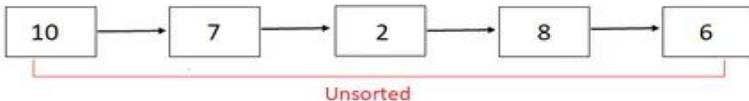
Sort without using extra space

Algorithm

1. Run a loop for $i = 0$ to n (not included), where n is the size of the queue.
 2. At every iteration initialize minIndex as -1 and minValue as -infinity.
 3. Run another loop with variable j that finds the minimum element's index from the unsorted queue.
 - 1. The unsorted queue is present from index 0 to index $(n - i)$ for a particular value of i .
 - 2. At every iteration, if the current element is less than minValue , update minValue as current element and minIndex as j .
 4. Traverse in the queue and remove the element at position minIndex and push it at the end of the queue.
 5. The queue is sorted, print its elements.
- 

Sort without using extra space

Illustration



- At every step, find the index of minimum element in the unsorted part and move it to the end of the queue, that is, sorted part.



Sort without using extra space



Explore | Expand | Enrich

Program: Sort queue without using extra space

sorttwoextraspaces.java

Sample IO

Input

- 2 6 7 8 10

Output

1 18 23 39 56 66

Time Complexity: $O(n*n)$

Space Complexity: $O(1)$



Max Sliding Window



Explore | Expand | Enrich

Algorithm

Given an array A and an integer K, the problem is to find the maximum for each and every contiguous subarray of size k aka the sliding window

There is a sliding window of size K which is moving from the very left of the array to the very right.

- You can only see the w numbers in the window. Each time the sliding window moves rightwards by one position.

You have to find the maximum for each window.





Max Sliding Window

Problem Statement

Return an array C, where C[i] is the maximum value from A[i] to A[i+K-1].

If k > length of the array, return 1 element with the max of the array.

Example

- Input: A[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}, K = 4
Output: {10, 10, 10, 15, 15, 90, 90}

Explanation

Maximum of first 4 elements is 10, similarly for next 4 elements (i.e from index 1 to 4) is 10, So the sequence generated is 10 10 10 15 15 90 90



Max Sliding Window



Illustration

Input: $A[] = \{1, 3, -1, -3, 5, 3, 6, 7\}$, $K = 3$

Output: $\{3, 3, 5, 5, 6, 7\}$

Explanation

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7



Brute Force Algorithm

The simplest approach to solve this problem is to iterate over all possible sliding windows and find the maximum for each window.

There can be a total of $N - K + 1$ sliding window and there are K elements in each window.

- 1. Run a loop i from 0 to $N - K + 1$, denoting the current window.
- 2. Initialise a variable max to INT_MIN to find the maximum value of each window.
- 3. Run a nested loop from j from i to $i + K$ to iterate over the elements of the current window and maximize max .
- 4. Store the value of max for each window and return.



Max Sliding Window



Program: Brute force approach for Max Sliding Window

maxslidingwindow1.java

Sample IO

Input

- arr = {1, 3, -1, -3, 5, 3, 6, 7}
k = 3

Time Complexity: O(N*k)

Space Complexity: O(N – k + 1)

Output

3 3 5 5 6 7



Using Dequeue

The previous approach takes $O(N * K)$ time, which would give a Time Limit Exceeded error for large values of K.

One of the ideas to reduce the time complexity could be to use a max heap, since the maximum element is always present at the top of the heap, but an insertion in a max heap of size K takes $O(\log K)$ time.

Therefore, the time complexity becomes $O(N \log K)$, which is efficient. Can we reduce it further?

The idea is to use a deque i.e. double ended queue which performs the push() and pop() operations in $O(1)$ constant time.

Max Sliding Window

Initially:

arr	20	5	3	8	6	15
-----	----	---	---	---	---	----

K=3, N=6

Take the first window and keep the necessary index in deque

0	2	3	4	5	6
---	---	---	---	---	---

Step 1:

arr	20	5	3	8	6	15
-----	----	---	---	---	---	----

Deque : {0, 1, 2}

Maximum element is arr[0] = 20

0	2	3	4	5	6
---	---	---	---	---	---

Step 2:

arr	20	5	3	8	6	15
-----	----	---	---	---	---	----

Deque : {3}

Maximum element is arr[3] = 8

0	2	3	4	5	6
---	---	---	---	---	---

Step 3:

arr	20	5	3	8	6	15
-----	----	---	---	---	---	----

Deque : {3, 4}

Maximum element is arr[3] = 8

0	2	3	4	5	6
---	---	---	---	---	---

Step 4:

arr	20	5	3	8	6	15
-----	----	---	---	---	---	----

Deque : {5}

Maximum element is arr[5] = 15

Algorithm: Using Dequeue

1. Initialise the first K elements of the array into the deque.
2. Iterate over the input array and for each step:
 1. Consider only the indices of the elements in the current window.
 2. Pop-out all the indices of elements smaller than the current element, since their value will be less than the current element.
 3. Push the current element into the deque.
 4. Push the first element of the deque i.e. deque[0] into the output array.

Program: Using Dequeue

maxslidingwindow2.java

Sample IO

Input

- arr = {1, 3, -1, -3, 5, 3, 6, 7}
k = 3

Time Complexity: O(N)
Space Complexity: O(N)

Output

3 3 5 5 6 7

Introduction

A stack permutation is a permutation of objects in the given input queue which is done by transferring elements from input queue to the output queue with the help of a stack and the built-in push and pop functions.

The conditions are

- 1. Only dequeue from the input queue.
- 2. Use inbuilt push, pop functions in the single stack.
- 3. Stack and input queue must be empty at the end.
- 4. Only enqueue to the output queue.



Problem Statement

Given two arrays $a[]$ and $b[]$ of size n .

All the elements of the array are unique.

Create a function to check if the given array $b[]$ is the stack permutation of given array $a[]$ or not.

The idea to do this is we will try to convert the input queue to output queue using a stack, if we are able to do so then the queue is permutable otherwise not.

Stack permutations



Let array $a[] : 1, 2, 3$ and array $b[] : 2, 1, 3$

Steps :

- push 1 from array $a[]$ to stack
- push 2 from array $a[]$ to stack
- pop 2 from stack to array $b[]$
- pop 1 from stack to array $b[]$
- push 3 from array $a[]$ to stack
- pop 3 from stack to array $b[]$

Therefore, Output : Yes

Algorithm

Initialize two arrays $a[]$ and $b[]$ of size n .

Create a function to check stack permutation which accepts the two integer arrays and the size of the array as its parameters.

After that, create a queue data structure of integer type.

Traverse through the array $a[]$ and push/insert all the elements of the array $a[]$ in the queue.

After that, create a second queue data structure of integer type.

Traverse through the array $b[]$ and push/insert all the elements of array $b[]$ in the queue.



Stack permutations



Algorithm -- continued

Similarly, create a stack data structure of integer type.

Traverse while the size of the first queue is not 0.

Create an integer variable and store the element at the front of the first queue in it and pop/remove it from the first queue.

- Check if the value in the integer variable is not equal to the element at the front of the second queue, push/insert the integer variable in the stack.

Else pop/remove the element at the front of the second queue.

Traverse again while the size of the stack is not 0. Check if the element at the top of the stack is equal to the element at the front of the second queue, pop / remove the element at the front of the second queue, and from the top of the stack. Else break.

Check if both the stack and the first queue are empty, print “Yes” else print “No”.



Program: Check if an array is stack permutation of other

stackperm.java

Sample IO

Input

- int a[] = { 1, 2, 3 };
- int b[] = { 2, 1, 3 };

Output

Yes

Time Complexity: O(n)

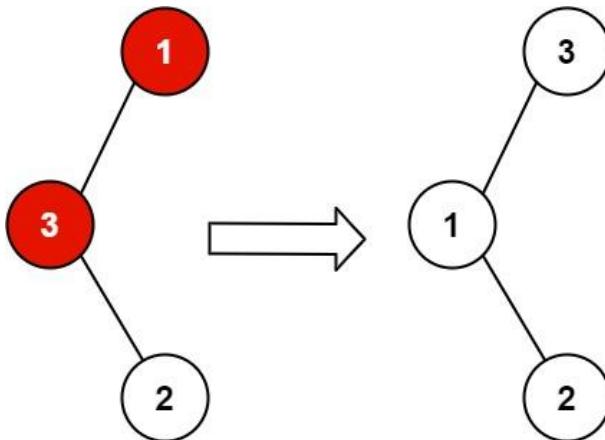
Space Complexity: O(n)

Note that 'n' is the number of elements in the arrays a and b

Introduction

You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake.

The task is to recover the tree without changing its structure

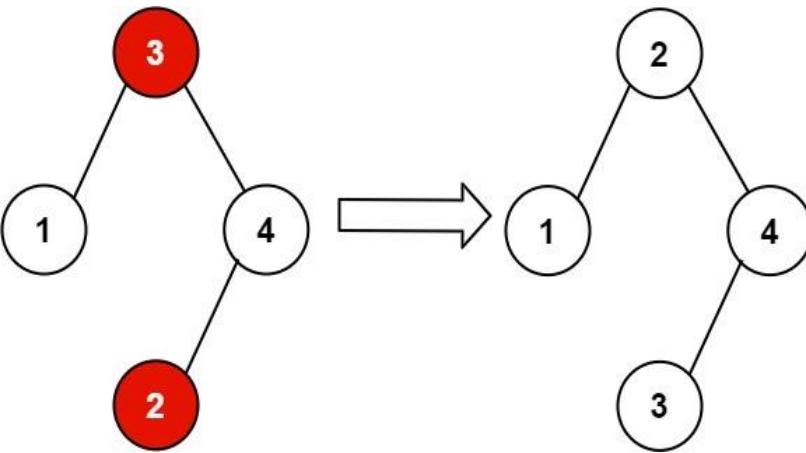
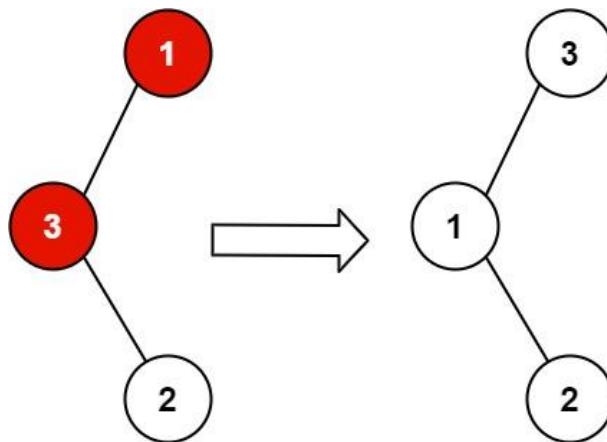




Recover the BST



Examples



Approach

The inorder traversal of a BST produces a sorted array.

So a simple method is to store inorder traversal of the input tree in an auxiliary array.

- After traversal, this auxilliary array must be sorted, and an a BST is supposed to be built out of this auxilliary array

Approach

However, sorting requires $O(n \log n)$ time and on introspecting, we can reduce the time complexity

Since only two nodes will be misplaced, we can traverse the auxiliary array and swap the elements which are not in order.

By doing this our problem gets reduced to $O(n)$

This is not straightforward and involves two cases to be handled.



Recover the BST



Approach

Consider a BST with inorder traversal 3 5 7 8 10 15 20 25,

The two cases that need to be handled are:-

1. The swapped nodes are not adjacent in the inorder traversal of the BST

- If nodes 5 and 25 are swapped, then the inorder traversal would be
3 25 7 8 10 15 20 5

2. The swapped nodes are adjacent in the inorder traversal of BST.

If nodes 7 and 8 are swapped, the inorder traversal would be

3 5 8 7 10 15 20 25



Approach

1. We will maintain three-pointers, first, middle, and last.
2. When we find the first point where the current node value is smaller than the previous node value, we update the first with the previous node & the middle with the current node.
3. When we find the second point where the current node value is smaller than the previous node value, we update the last with the current node.
4. In the second case, we swap the consecutive elements which are out of order.
5. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.

Program: Correcting the BST

recoverbst.java

Sample IO

Input

- Original Tree(Inorder traversal)

1 10 3 6 7 2 12

Time Complexity: O(n)

Output

Inorder Traversal of the fixed tree

1 2 3 6 7 10 12

Approach

1. We will maintain three-pointers, first, middle, and last.
2. When we find the first point where the current node value is smaller than the previous node value, we update the first with the previous node & the middle with the current node.
3. When we find the second point where the current node value is smaller than the previous node value, we update the last with the current node.
4. In the second case, we swap the consecutive elements which are out of order.
5. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.

Introduction

Different Types of Views in Binary Tree are:

1. Left View
 2. Right View
 3. Top View
 4. Bottom View
- 

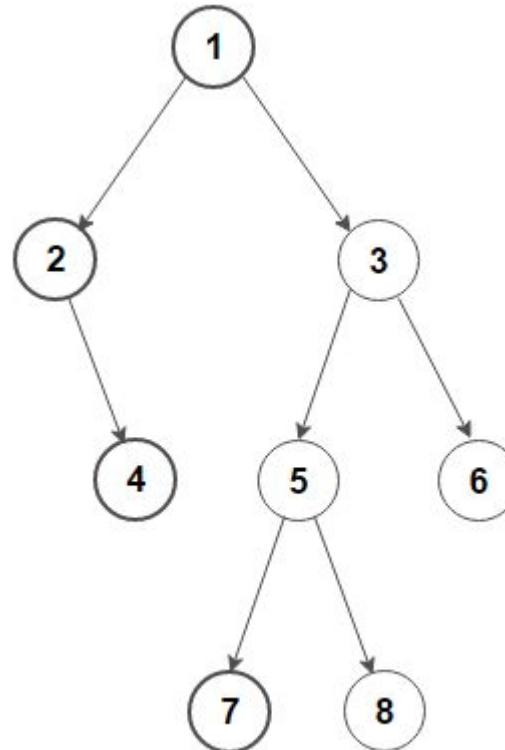
Introduction

Different Types of Views in Binary Tree are:

1. Left View
2. Right View
3. Top View
4. Bottom View

The given example shows the left view of a binary tree.

Given this tree as the input, the output would be 1 2 4 7

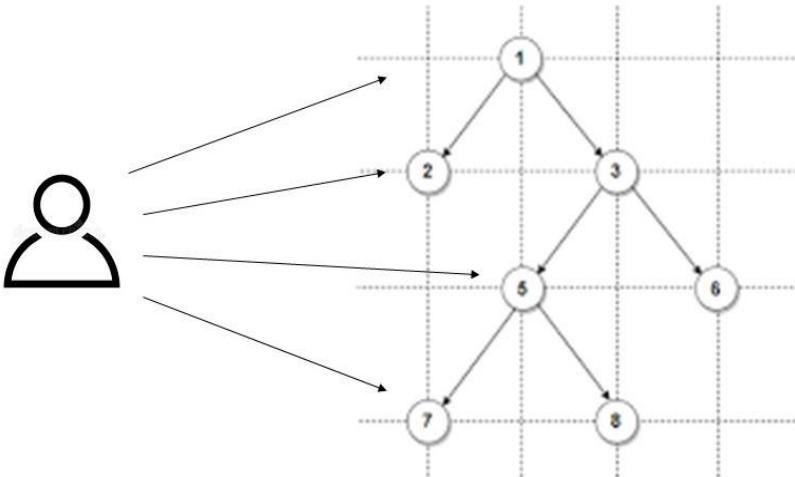




Left view of a Binary Tree



Explore | Expand | Enrich



Left view : 1 2 5 7



Left view of a Binary Tree

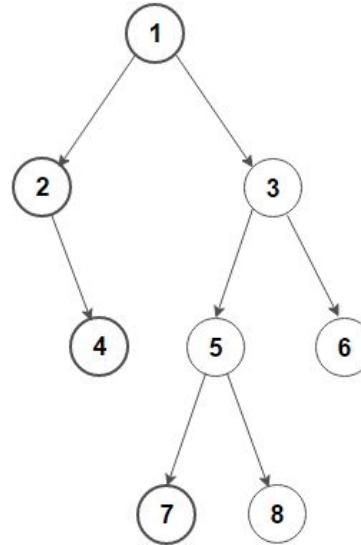
Approach

We can also solve this problem by using hashing.

The idea is to traverse the tree in a preorder fashion and pass level information in function arguments.

- If the level is visited for the first time, insert the current node and level information into the map.

Finally, when all nodes are processed, traverse the map and print the left view.



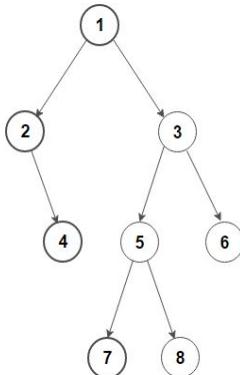
Left view of a Binary Tree



Program: Left view of a BT (Using Hashing)

leftview1.java

Sample IO
Input



Output: 1 2 4 7

Time Complexity: O(n)
Space Complexity: O(n)

n: size of the BT

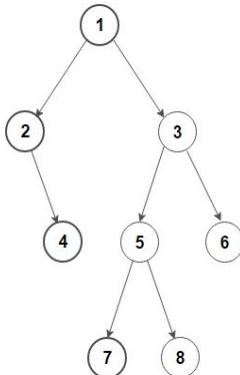
Left view of a Binary Tree



Program: Left view of a BT (Using Hashing)

leftview2.java

Sample IO
Input

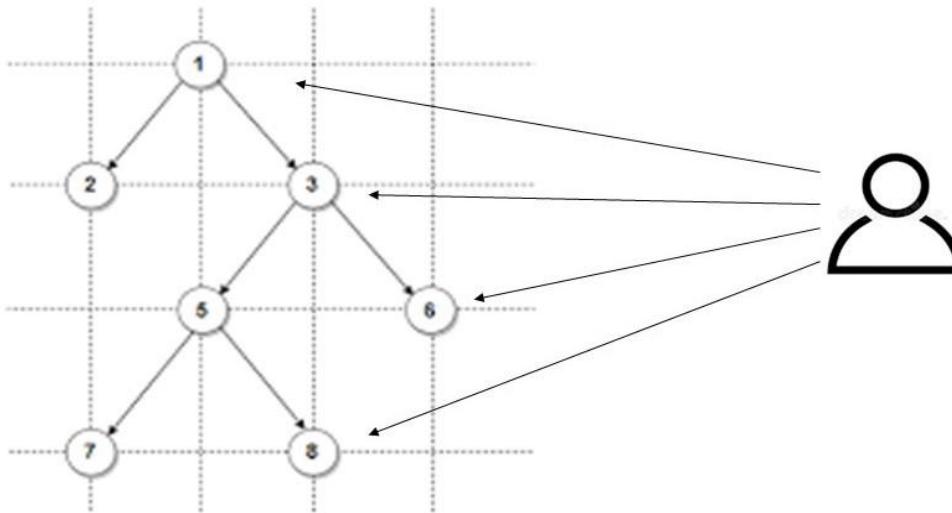


Output: 1 2 4 7

Time Complexity: O(n)
Space Complexity: O(h)

n: number of nodes in the BT
h: height of the BT

Right view of a Binary Tree



Right view : 1 3 6 8

Right view of a Binary Tree

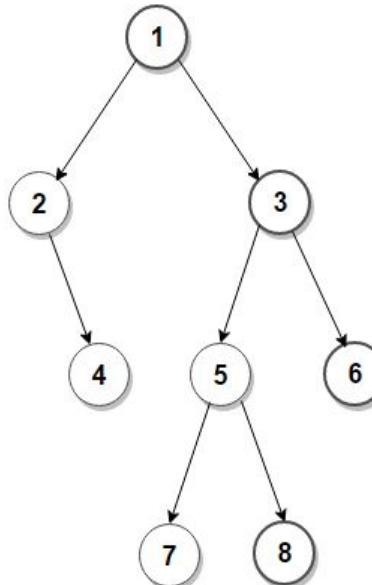
Approach

We can also solve this problem by using hashing.

The idea is to traverse the tree in a preorder fashion and pass level information in function arguments.

- For every node encountered, insert the node and level information into the map.

Finally, when all nodes are processed, traverse the map and print the right view.



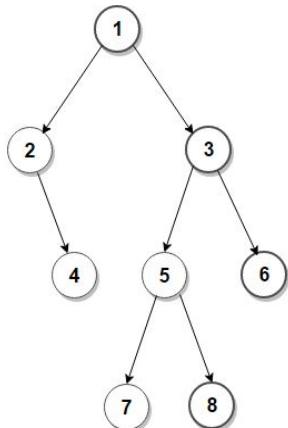
Right view of a Binary Tree



Program: Right view of a BT (Using Hashing)

rightview1.java

Sample IO
Input



Output: 1 3 6 8

Time Complexity: O(n)
Space Complexity: O(n)

n: size of the BT

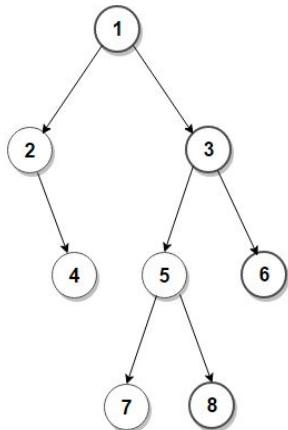
Right view of a Binary Tree



Program: Right view of a BT

rightview2.java

Sample IO
Input



Output: 1 3 6 8

Time Complexity: O(n)
Space Complexity: O(h)

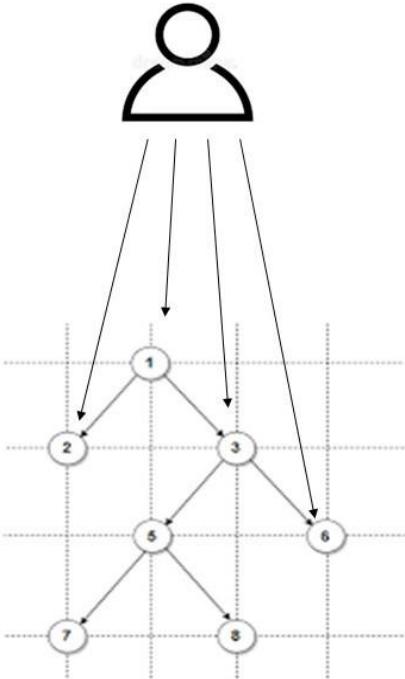
n: number of nodes
h: height of the BT



Top view of a Binary Tree



Explore | Expand | Enrich



Top view : 2 1 3 6





Top view of a Binary Tree



Approach

We can easily solve this problem with the help of hashing.

The idea is to create an empty map where each key represents the relative horizontal distance of the node from the root node, and the value in the map maintains a pair containing the node's value and its level number.

Then perform preorder traversal on the tree.

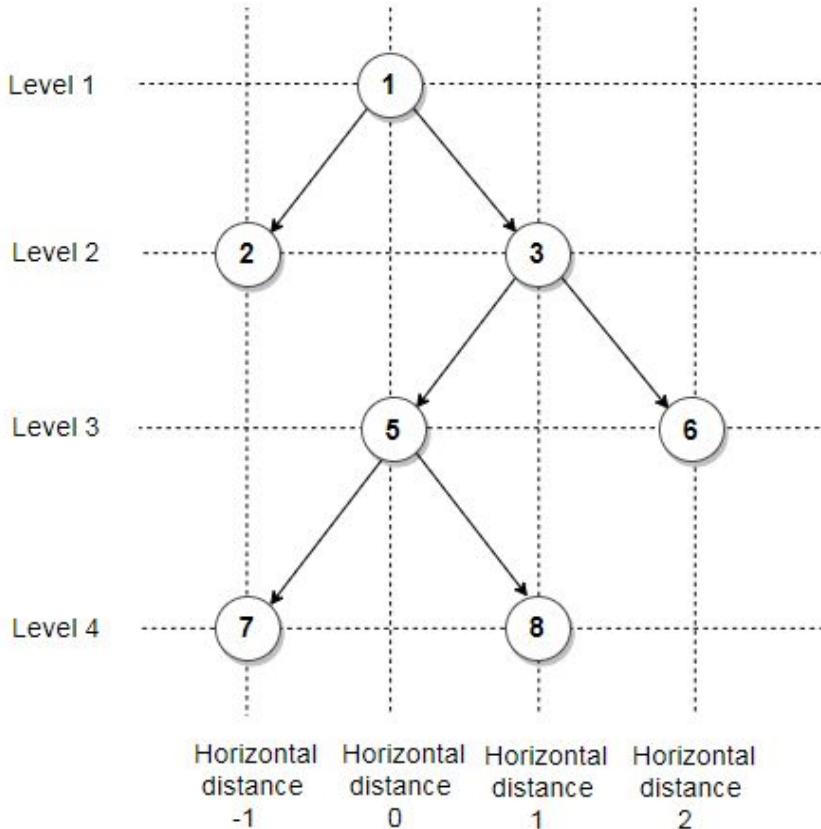
- Suppose the current node's level is less than the maximum level seen so far for the same horizontal distance as the current node or current horizontal distance is seen for the first time.

In that case, update the value and the level for the current horizontal distance in the map.

For each node, recur for its left subtree by decreasing horizontal distance and increasing level by one, and recur for right subtree by increasing both level and horizontal distance by one.



Top view of a Binary Tree



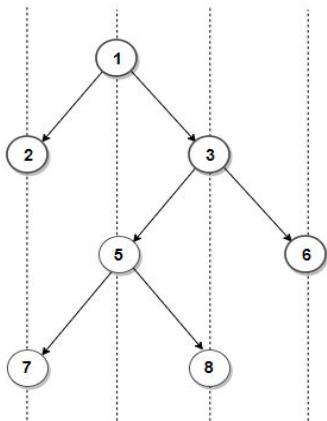
horizontal distance \rightarrow (node's value, node's level)

- 1 \rightarrow (2, 2)
- 0 \rightarrow (1, 1)
- 1 \rightarrow (3, 2)
- 2 \rightarrow (6, 3)

Program: Top view of a BT

topview1.java

Sample IO Input



Output: 2 1 3 6

Time Complexity: O(nlogn)

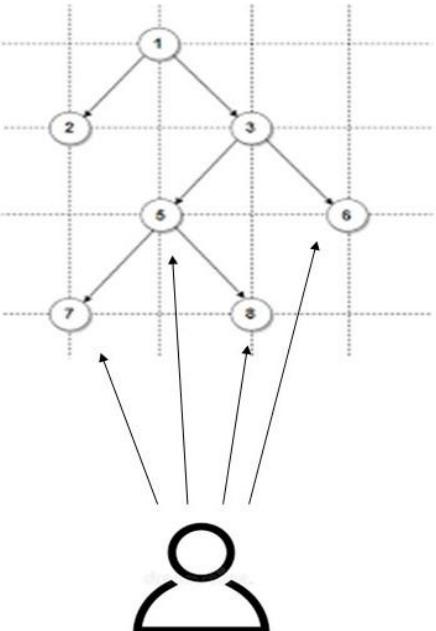
Space Complexity: O(n)



Bottom view of a Binary Tree



Explore | Expand | Enrich



Bottom view : 7586





Bottom view of a Binary Tree



Approach

We can easily solve this problem with the help of hashing.

The idea is to create an empty map where each key represents the relative horizontal distance of the node from the root node, and the value in the map maintains a pair containing the node's value and its level number.

Then perform preorder traversal on the tree.

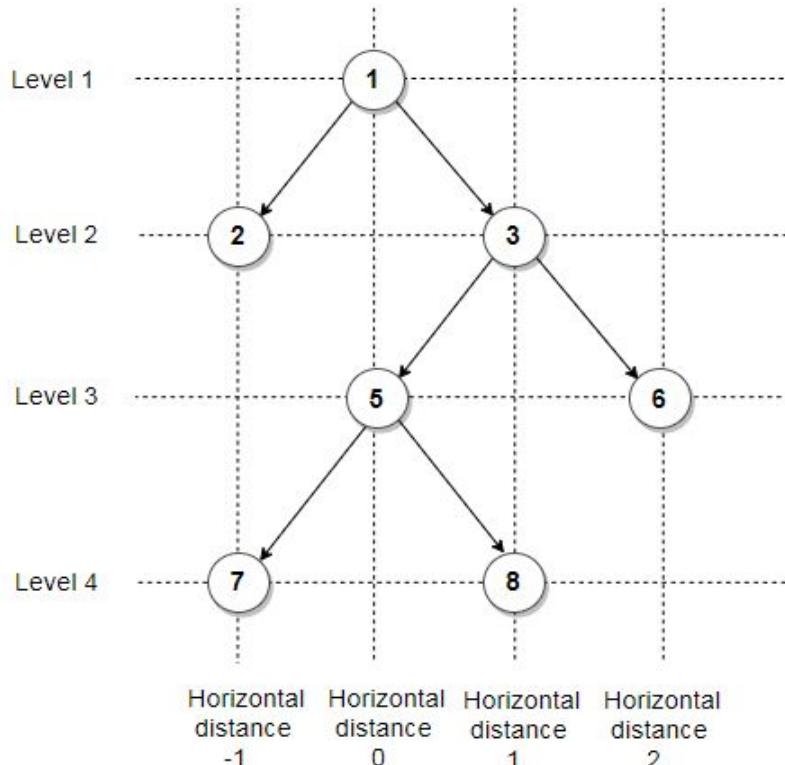
- Suppose the current node's level is more than or equal to the maximum level seen so far for the same horizontal distance as the current node's or current horizontal distance is seen for the first time.

In that case, update the value and the level for the current horizontal distance in the map.

For each node, recur for its left subtree by decreasing horizontal distance and increasing level by one, and recur for right subtree by increasing both level and horizontal distance by one.



Bottom view of a Binary Tree



(horizontal distance \rightarrow
(node's value, node's level))
-1 \rightarrow (7, 4)
0 \rightarrow (5, 3)
1 \rightarrow (8, 4)
2 \rightarrow (6, 3)

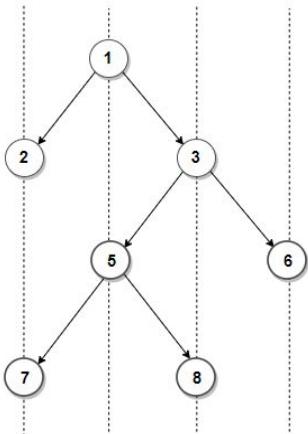
Bottom view of a Binary Tree



Program: Bottom view of a BT

bottomview1.java

Sample IO
Input



Output: 7 5 8 6

Time Complexity: $O(n \log n)$
Space Complexity: $O(n)$

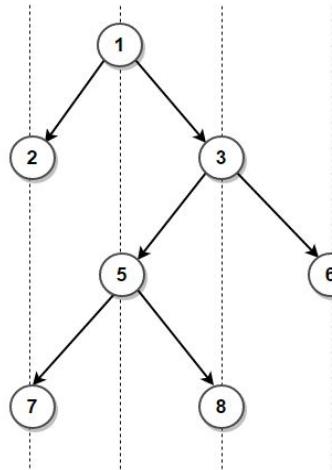
Introduction

Given a binary tree, perform vertical traversal on it. In a vertical traversal, nodes of a binary tree are processed in vertical order from left to right.

Assume that the left and right child makes a 45-degree angle with the parent.

For example the output for the following tree is

- 2, 7
- 1, 5
- 3, 8
- 6



Vertical order traversal



Approach - Using Preorder traversal

The idea is to create an empty map where each key represents the relative horizontal distance of a node from the root node, and the value in the map maintains all nodes present at the same horizontal distance.

Then perform a preorder traversal of the tree, and update the map.

For each node, recur for its left subtree by decreasing horizontal distance by 1 and recur for

- the right subtree by increasing horizontal distance by 1

For the BT given at the right, the final values of the map will be

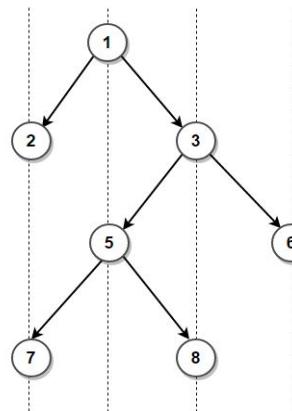
(horizontal distance \rightarrow Nodes)

-1 \rightarrow [2, 7]

0 \rightarrow [1, 5]

1 \rightarrow [3, 8]

2 \rightarrow [6]





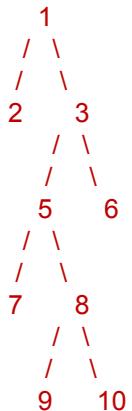
Vertical order traversal



Program: Vertical order traversal

vot1.java

Sample IO
Input



Output:

[2, 7]

[1, 5, 9]

[3, 8]

[10, 6]

Time Complexity: O(nlogn)

Space Complexity: O(n)

n is the size of the BT



Approach - Using Level order traversal

Since the previous solution uses preorder traversal to traverse the tree, the nodes might not get processed in the same order as they appear in the binary tree from top to bottom.

For instance, node 10 is printed before node 6 in the above solution.

We can perform a level order traversal to ensure that nodes are processed in the same order as they appear in the binary tree.

The idea remains the same as the previous approach – create an empty map whose each key represents the relative horizontal distance of a node from the root node, and the value in the map maintains all nodes present at the same horizontal distance.

The only difference is that the binary tree is traversed using level order traversal instead of the preorder traversal.



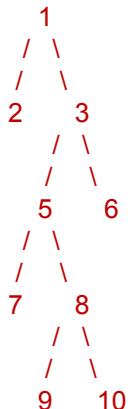
Vertical order traversal



Program: Using levelorder traversal

vot2.java

Sample IO
Input



Output:

[2, 7]

[1, 5, 9]

[3, 8]

[6, 10]

Time Complexity: O(nlogn)

Space Complexity: O(n)

n is the size of the BT



Boundary traversal



Introduction

The boundary traversal of the binary tree consists of the left boundary, leaves, and right boundary without duplicate nodes as the nodes may contain duplicate values.

There are two types of boundary, i.e., left boundary, right boundary and bottom boundary

The left boundary can be defined as the path from the root to the left-most node, whereas the right boundary can be defined as the path from the root to the right-most node. The bottom

- boundary is the bottom elements of the BT

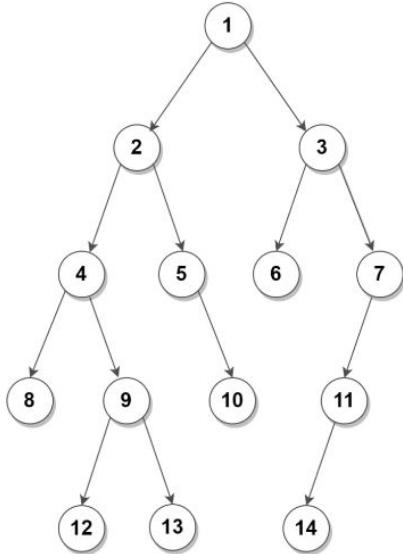
If the root node does not contain any left and right subtree, then the root node itself would be considered as the left boundary and right boundary.

The solution should print the boundary nodes starting from the tree's root, in an anti-clockwise direction, without any duplicates.



Boundary traversal

Example



The above binary tree's boundary traversal is
1, 2, 4, 8, 12, 13, 10, 6, 14, 11, 7, 3

Approach

The idea is to split the problem into 3 parts:

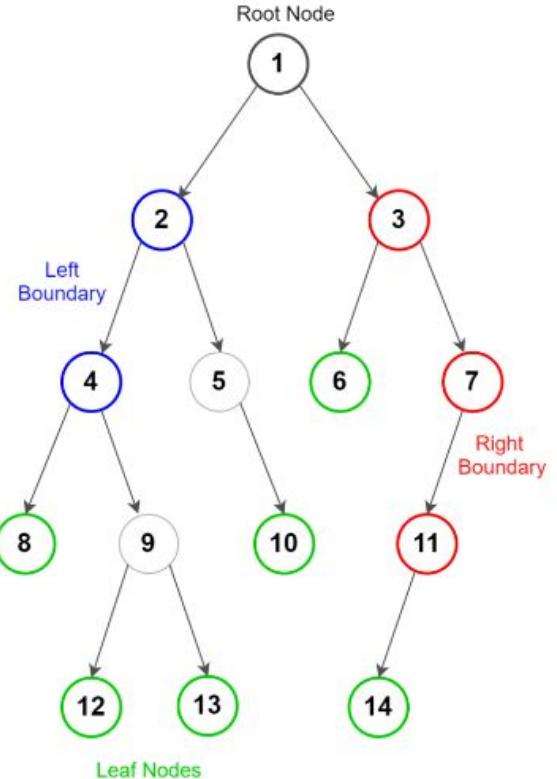
1. Print the left boundary in a top-down fashion.
2. Print the leaf nodes in the same order as in the inorder traversal.
3. Print the right boundary in a bottom-up fashion.

This approach looks simple, but there are several edge cases the solution should handle to

- avoid printing duplicates:
- Both the left boundary and the right boundary traversal prints the root node. We can avoid this by printing the root node first and performing the left boundary traversal on the root's left child and the right boundary traversal on the root's right child.
- The leftmost and the rightmost node of the binary tree are leaf nodes. They will get printed while performing the left boundary and the right boundary traversal while printing the leaf nodes. To avoid it, exclude the leaf nodes while doing the left boundary and the right boundary traversal.

Boundary traversal

Approach



Explore | Expand | Enrich

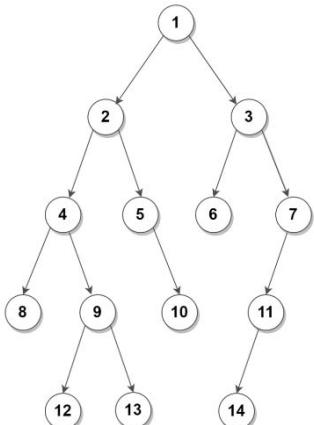


Boundary traversal

Program: Boundary traversal of a Binary Tree

boundarytraversal1.java

Sample IO
Input



Output:

1, 2, 4, 8, 12, 13, 10, 6, 14, 11, 7, 3

Time Complexity: $O(n)$
Space Complexity: $O(h)$

n is the number of nodes
 h is the height of the tree



Topic: Graphs

Introduction

Unlike tree traversal, graph traversal may require that some vertices be visited more than once, since it is not necessarily known before transitioning to a vertex that it has already been explored.

As graphs become more dense, this redundancy becomes more prevalent, causing computation time to increase; as graphs become more sparse, the

- opposite holds true.

Thus, it is usually necessary to remember which vertices have already been explored by the algorithm, so that vertices are revisited as infrequently as possible (or in the worst case, to prevent the traversal from continuing indefinitely).



Introduction

There are two Graph Traversal Algorithms: Breadth First Search and Depth First Search called BFS and DFS respectively

A breadth-first search (BFS) is a technique for traversing a finite graph. BFS visits the sibling vertices before visiting the child vertices.

- A stack is used in the search process in DFS

DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth.

A queue is used in the search process in BFS

Introduction

Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard BFS implementation puts each vertex of the graph into one of two categories:

- 1. Visited
- 2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.





Breadth-First Search

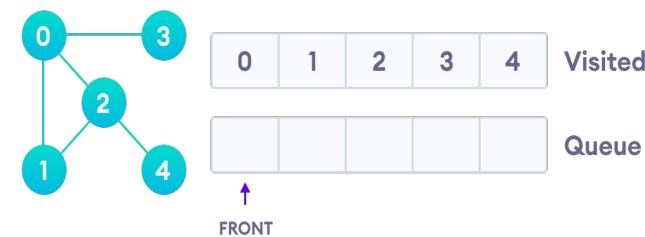
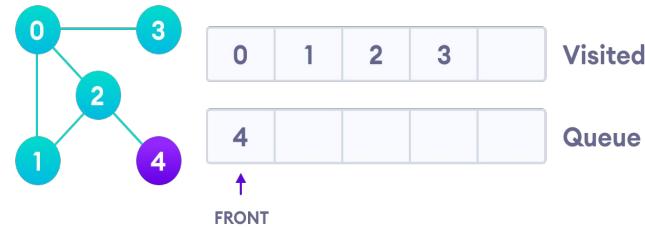
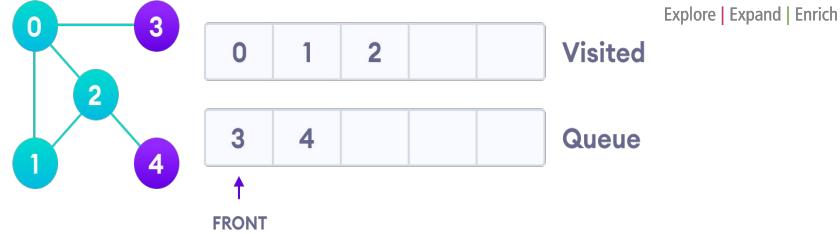
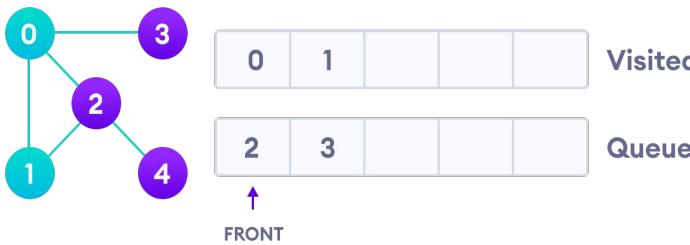
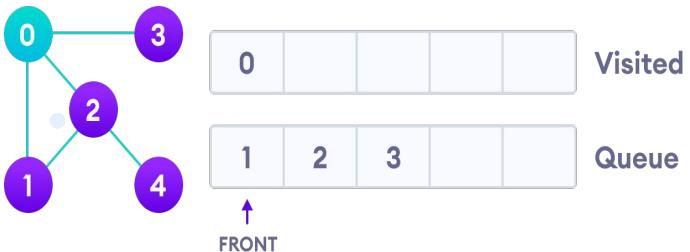
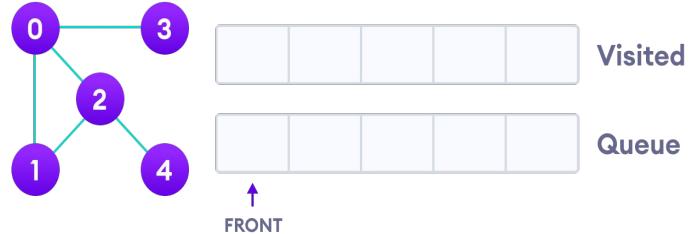


Explore | Expand | Enrich

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which are not in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.
5. The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

Breadth-First Search





Breadth-First Search



Explore | Expand | Enrich

Psuedocode

```
create a queue Q  
mark v as visited and put v into Q  
while Q is non-empty  
    remove the head u of Q  
    mark and enqueue all (unvisited) neighbours of u
```



Breadth-First Search



Explore | Expand | Enrich

Program: BFS Implementation

bfs.java

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

- where V is the number of nodes and E is the number of edges



Applications of BFS

1. BFS can be used to find the neighboring locations from a given source location.
2. In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes.
3. BFS can be used in web crawlers to create web page indexes where, every web page is considered as a node in the graph.
4. BFS is used to determine the shortest path and minimum spanning tree.
5. BFS is also used in Cheney's technique to duplicate the garbage collection.
6. It can be used in Ford-Fulkerson method to compute the maximum flow in a flow network.
7. It can be used in cycle detection in an undirected graph and also in navigation

Introduction

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure

A standard DFS implementation puts each vertex of the graph into one of two categories:

- 1. Visited
- 2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.



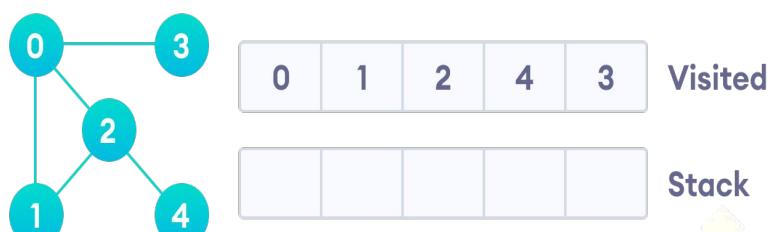
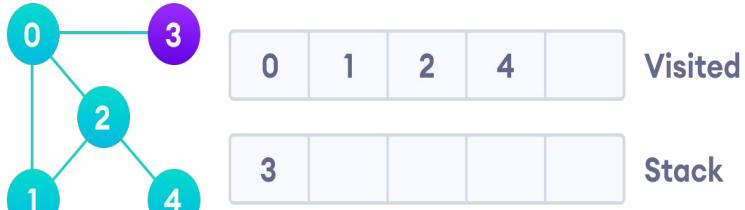
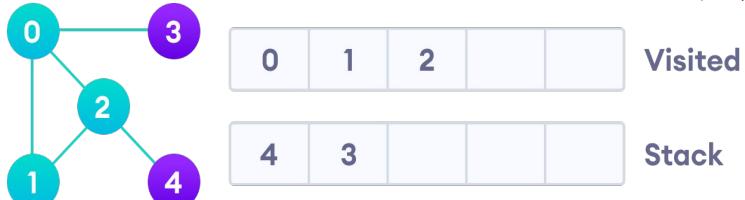
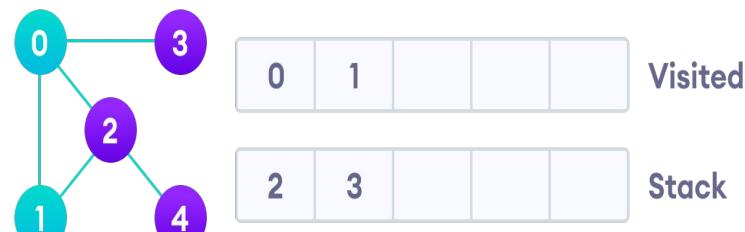
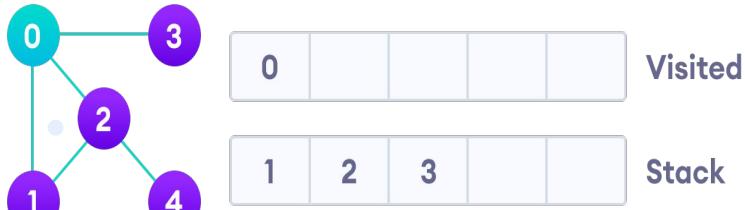
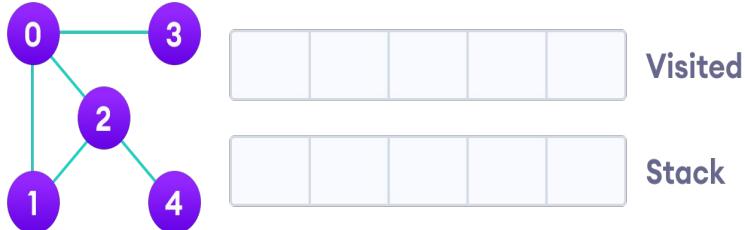
Depth-First Search



Algorithm

1. Start by putting any one of the graph's vertices on top of a stack.
 2. Take the top item of the stack and add it to the visited list.
 3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
..
 4. Keep repeating steps 2 and 3 until the stack is empty.
- 

Depth-First Search





Depth-First Search



Explore | Expand | Enrich

Psuedocode

```
DFS( $G, u$ )
   $u.visited = true$ 
  for each  $v \in G.Adj[u]$ 
    if  $v.visited == false$ 
      DFS( $G, v$ )
```

```
init() {
  For each  $u \in G$ 
     $u.visited = false$ 
  For each  $u \in G$ 
    DFS( $G, u$ )
}
```



Depth-First Search



Explore | Expand | Enrich

Program: DFS Implementation

dfs.java

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

- where V is the number of nodes and E is the number of edges



Applications of DFS

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree.
2. DFS can be used to detect a cycle in the graph
3. We can specialize the DFS algorithm to find a path between the two given vertices u and z.
4. DFS can be used in topological sorting. Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs
5. DFS can be used to test if a graph is bipartite
6. DFS can be used to finding Strongly connected components of a graph
7. DFS can be used to solving puzzles with only one solution

Introduction

Dijkstra algorithm is a single-source shortest path algorithm.

Single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

We generate a SPT (shortest path tree) with a given source as a root.

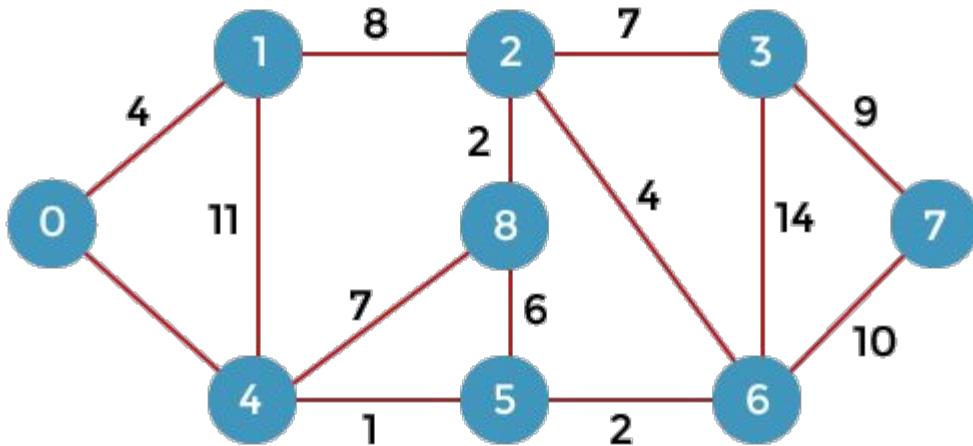
We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

Dijkstra's Algorithm

Introduction

Consider the following graph.



Dijkstra's Algorithm

Introduction

Let's assume that the vertex 0 is the source vertex is represented by 'x' and the vertex 1 is represented by 'y'.

Initially we do not know the distances. Distance to source vertex would be 0 and distance to other vertices would be ∞

The distance between the vertices can be calculated by using the below formula:

$$d(x, y) = d(x) + c(x, y) < d(y)$$

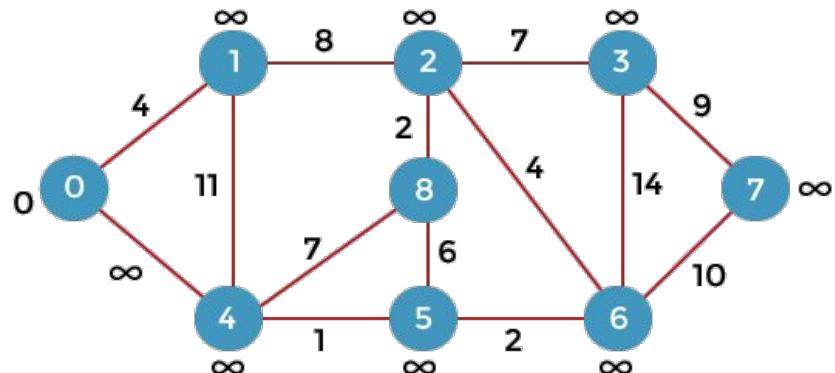
- $= (0 + 4) < \infty$

$$= 4 < \infty$$

Since $4 < \infty$ so we will update $d(v)$ from ∞ to 4.

Hence the formula is

$$\text{if } (d(u) + c(u, v) < d(v)) \\ \text{then } d(v) = d(u) + c(u, v)$$





Dijkstra's Algorithm

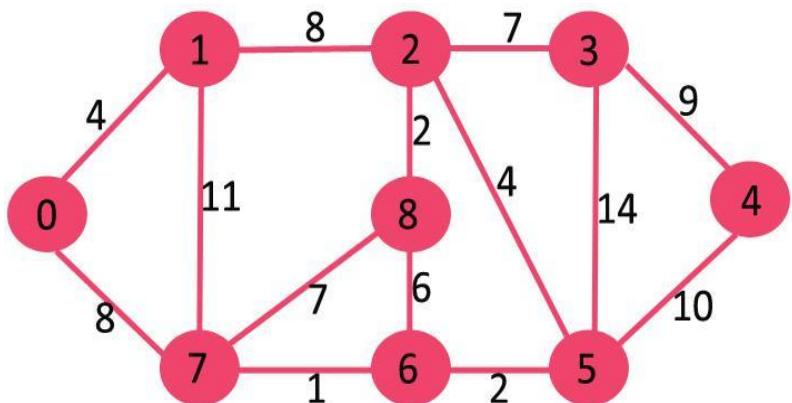


Introduction

- 1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- 2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
- 3) While sptSet doesn't include all vertices
 - a) Pick a vertex u which is not there in sptSet and has a minimum distance value.
 - b) Include u to sptSet.
 - c) Update distance value of all adjacent vertices of u . To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v , if the sum of distance value of u (from source) and weight of edge $u-v$, is less than the distance value of v , then update the distance value of v .

Dijkstra's Algorithm

Another Example



The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.

Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}.

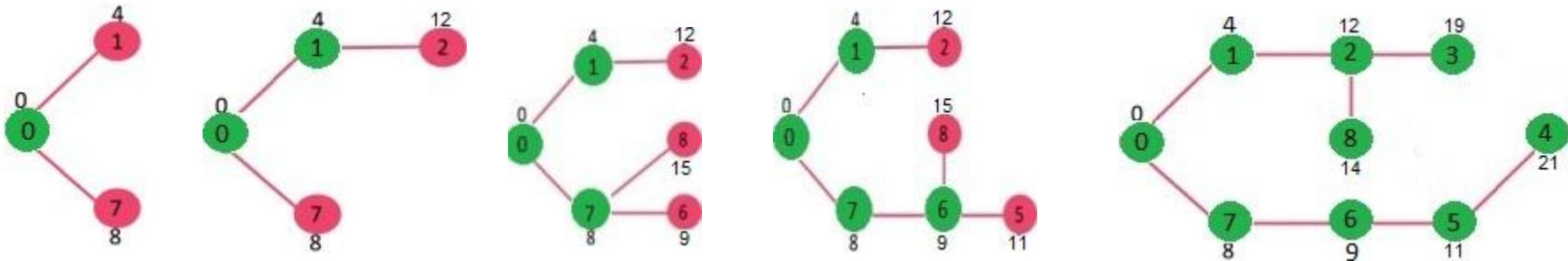
After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7.

The distance values of 1 and 7 are updated as 4 and 8. T

he subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

Dijkstra's Algorithm

Another Example

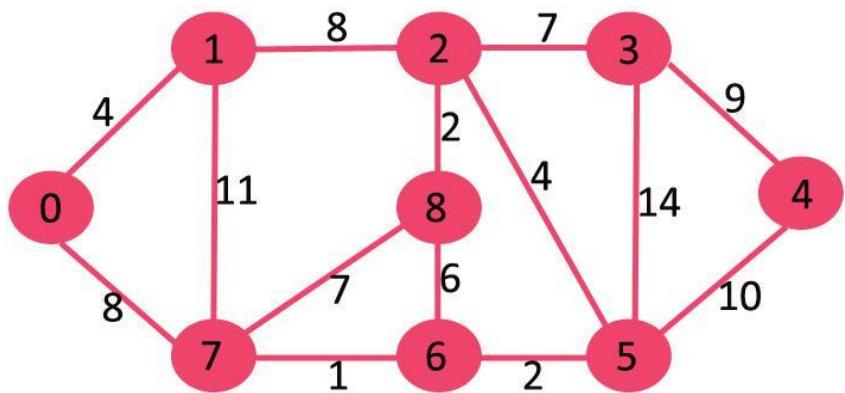


Dijkstra's Algorithm

Program: Dijkstra's Algorithm

dijsktra1.java

Input



Output

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14



Dial's Algorithm



Explore | Expand | Enrich

Introduction

Dijkstra's shortest path algorithm runs in $O(E \log V)$ time when implemented with adjacency list representation.

Can we optimize Dijkstra's shortest path algorithm to work better than

- $O(E \log V)$ if maximum weight is small (or range of edge weights is small)?



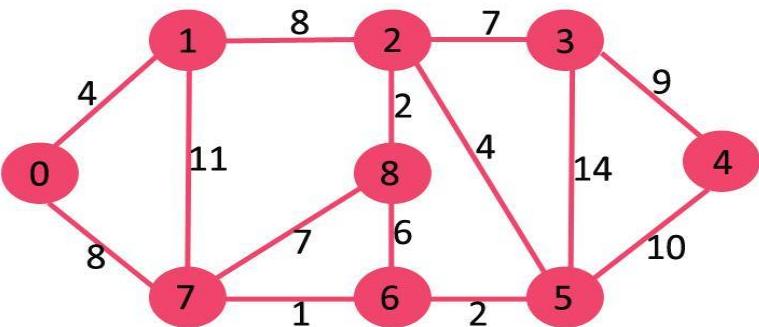
Dial's Algorithm

Introduction

Can we optimize Dijkstra's shortest path algorithm to work better than $O(E \log V)$ if maximum weight is small (or range of edge weights is small)?

Ans: Yes. In the given diagram, maximum weight is 14. Many a times the range of weights on edges in is in small range (i.e. all edge weight can be mapped to 0, 1, 2.. w where w is a small number).

In this case, Dijkstra's algorithm can be modified by using different data structure, buckets, which is called dial implementation of dijkstra's algorithm.



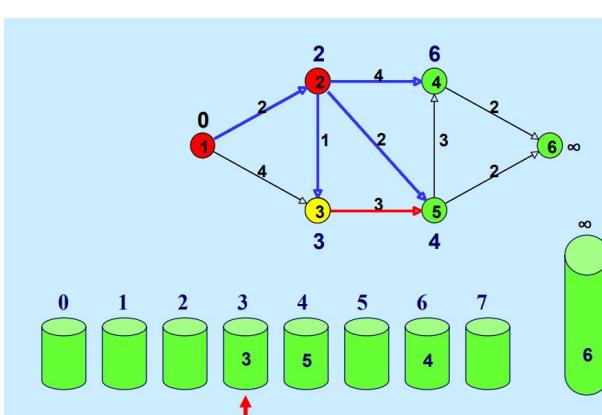
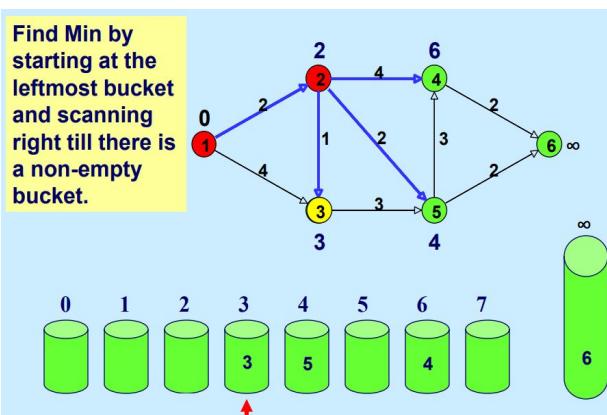
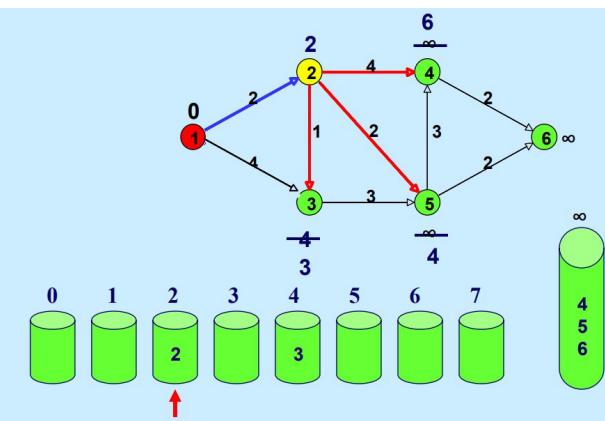
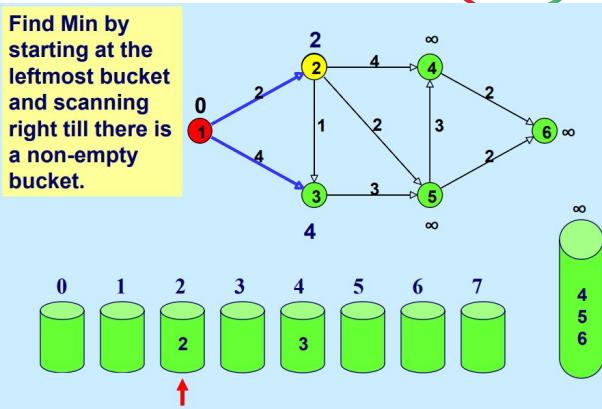
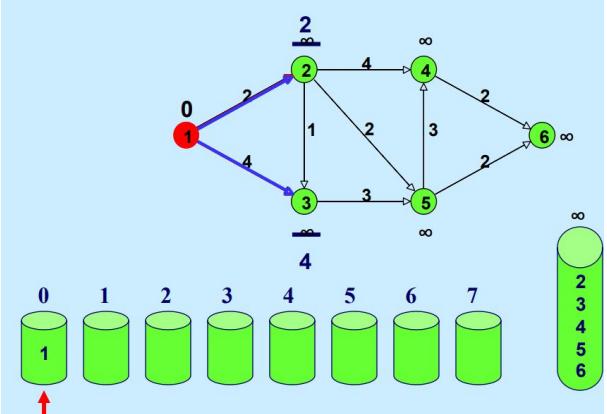
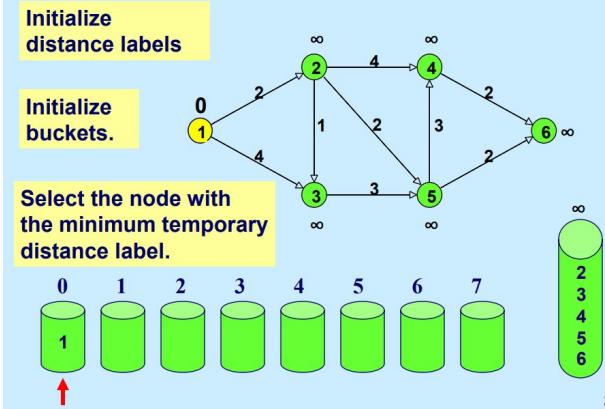
Observation

- Maximum distance between any two node can be at max $w(V - 1)$ (w is maximum edge weight and we can have at max $V-1$ edges between two vertices).
- In Dijkstra algorithm, distances are finalized in non-decreasing, i.e., distance of the closer (to given source) vertices is finalized before the distant vertices.

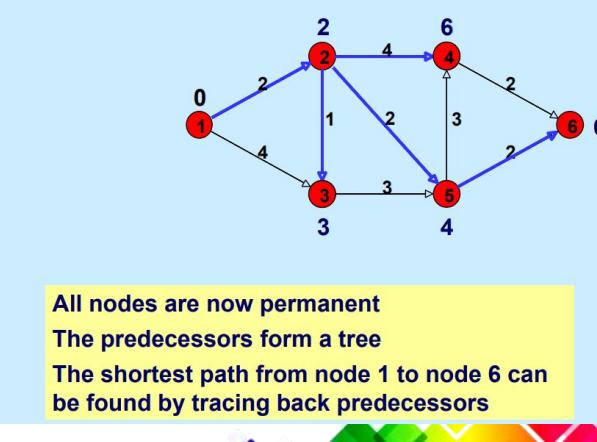
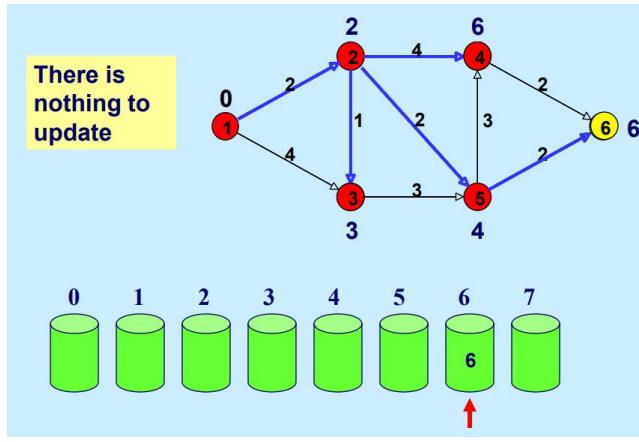
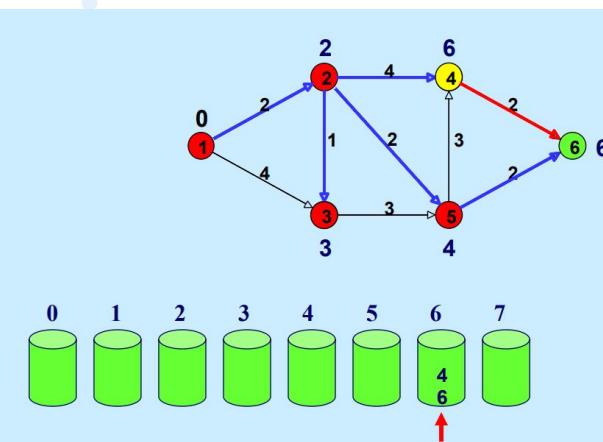
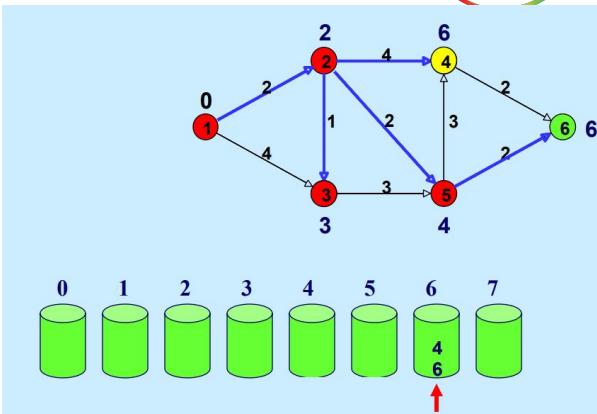
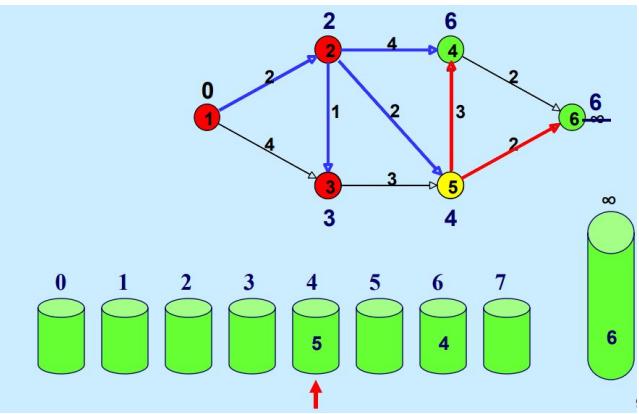
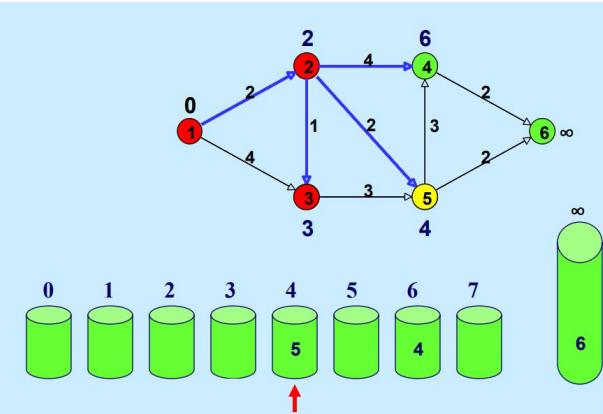
Algorithm

1. Maintains some buckets, numbered 0, 1, 2,...,wV.
2. Bucket k contains all temporarily labeled nodes with distance equal to k.
3. Nodes in each bucket are represented by list of vertices.
4. Buckets 0, 1, 2,..wV are checked sequentially until the first non-empty bucket is found. Each node contained in the first non-empty bucket has the minimum distance label by definition.
5. One by one, these nodes with minimum distance label are permanently labeled and deleted from the bucket during the scanning process.
6. Thus operations involving vertex include:
 1. Checking if a bucket is empty
 2. Adding a vertex to a bucket
 3. Deleting a vertex from a bucket.
7. The position of a temporarily labeled vertex in the buckets is updated accordingly when the distance label of a vertex changes.
8. Process repeated until all vertices are permanently labeled (or distances of all vertices are finalized).

Dial's Algorithm



Dial's Algorithm





Bellman-Ford Algorithm



Introduction

Bellman-Ford algorithm is used to find the shortest path from the source vertex to every vertex in a weighted graph.

Unlike Dijkstra's algorithm, the bellman ford algorithm can also find the shortest distance to every vertex in the weighted graph even with the negative edges.

The only difference between the Dijkstra algorithm and the bellman ford algorithm is that Dijkstra's algorithm just visits the neighbour vertex in each iteration but the bellman ford algorithm visits each vertex through each edge in every iteration.

Apart from Bellman-Ford Algorithm and Dijkstra's Algorithm, Floyd Warshall Algorithm is also the shortest path algorithm.

But Bellman-Ford algorithm is used to compute the shortest path from the single source vertex to all other vertices whereas Floyd-Warshall algorithms compute the shortest path from each node to every other node.



Bellman-Ford Algorithm



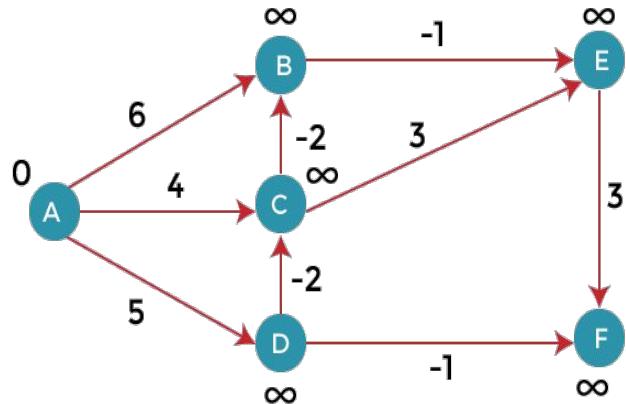
Principle of Bellman-Ford algorithm

We will go on relaxing all the edges $(n - 1)$ times where,
 n = number of vertices

As we can observe in the graph that some of the weights are negative.

The graph contains 6 vertices so we will go on relaxing till the 5 vertices.

Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times then also the answer will be the same, i.e., there would be no change in the distance between the vertices.



Relaxing:

If $(d(u) + c(u, v) < d(v))$
 $d(v) = d(u) + c(u, v)$

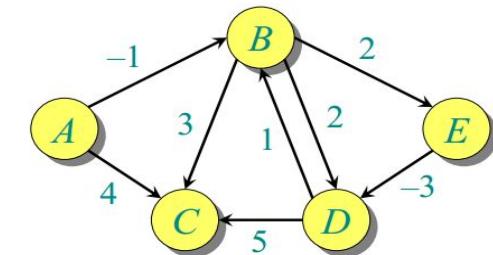
Bellman-Ford Algorithm

Problem Statement

Given a source vertex s from a set of vertices V in a weighted directed graph where its edge weights $w(u, v)$ can be negative, find the shortest path weights $d(s, v)$ from source s for all vertices v present in the graph. If the graph contains a negative-weight cycle, report it.

For example, in the given graph

Let source vertex = 0,



The distance of vertex 1 from vertex 0 is -1. Its path is $[0 \rightarrow 1]$

The distance of vertex 2 from vertex 0 is 2. Its path is $[0 \rightarrow 1 \rightarrow 2]$

The distance of vertex 3 from vertex 0 is -2. Its path is $[0 \rightarrow 1 \rightarrow 4 \rightarrow 3]$

The distance of vertex 4 from vertex 0 is 1. Its path is $[0 \rightarrow 1 \rightarrow 4]$



Bellman-Ford Algorithm



Idea

The idea is to use the Bellman–Ford algorithm to compute the shortest paths from a single source vertex to all the other vertices in a given weighted digraph.

If a graph contains a “negative cycle” that is reachable from the source, then there is no shortest path. Any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle.

- The algorithm initializes the distance to the source to 0 and all other nodes to INFINITY. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value.

At each iteration i that the edges are scanned, the algorithm finds all shortest paths of at most length i edges. Since the longest possible path without a cycle can be $V-1$ edges, the edges must be scanned $V-1$ times to ensure that the shortest path has been found for all nodes. A final scan of all the edges is performed, and if any distance is updated, then a path of length $|V|$ edges have been found, which can only occur if at least one negative cycle exists in the graph.



Bellman-Ford Algorithm

Pseudocode

```
function BellmanFord(list vertices, list edges, vertex source) is
    // This implementation takes in a graph, represented as
    // lists of vertices (represented as integers [0..n-1]) and edges,
    // and fills two arrays (distance and predecessor) holding
    // the shortest path from the source to each vertex

    distance := list of size n
    predecessor := list of size n

    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf           // Initialize the distance to all vertices to infinity
        predecessor[v] := null       // And having a null predecessor

    distance[source] := 0          // The distance from the source to itself is, of course, zero

    // Step 2: relax edges repeatedly

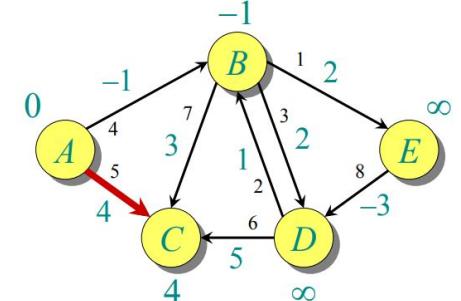
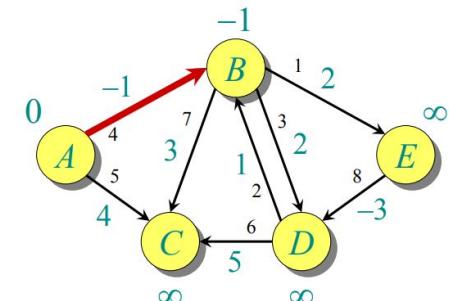
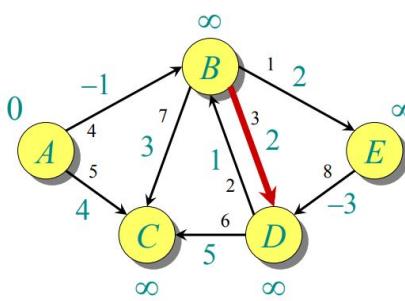
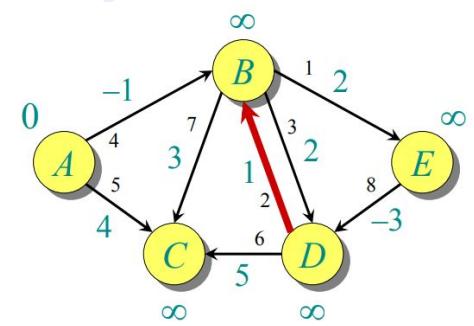
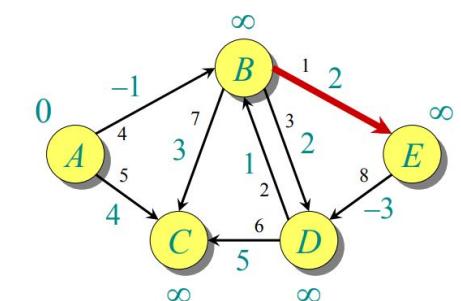
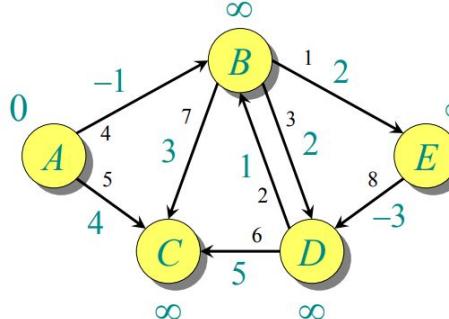
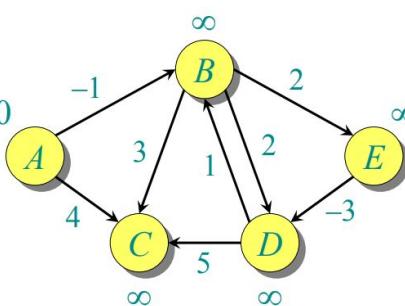
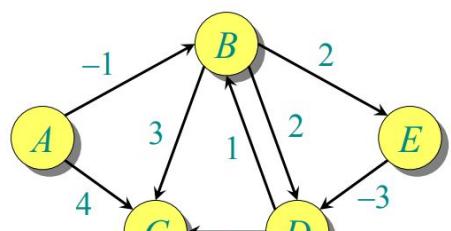
    repeat |V|-1 times:
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance, predecessor
```

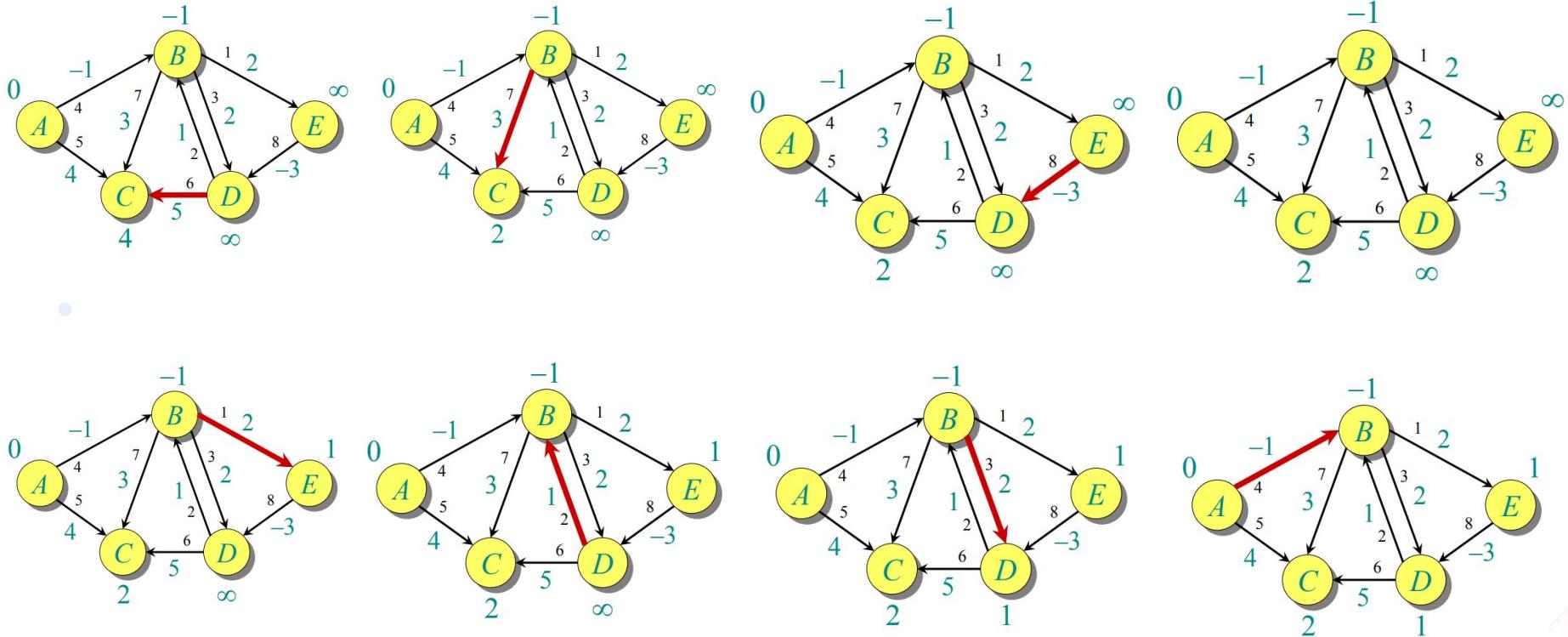
Bellman-Ford Algorithm

Example



Bellman-Ford Algorithm

Example

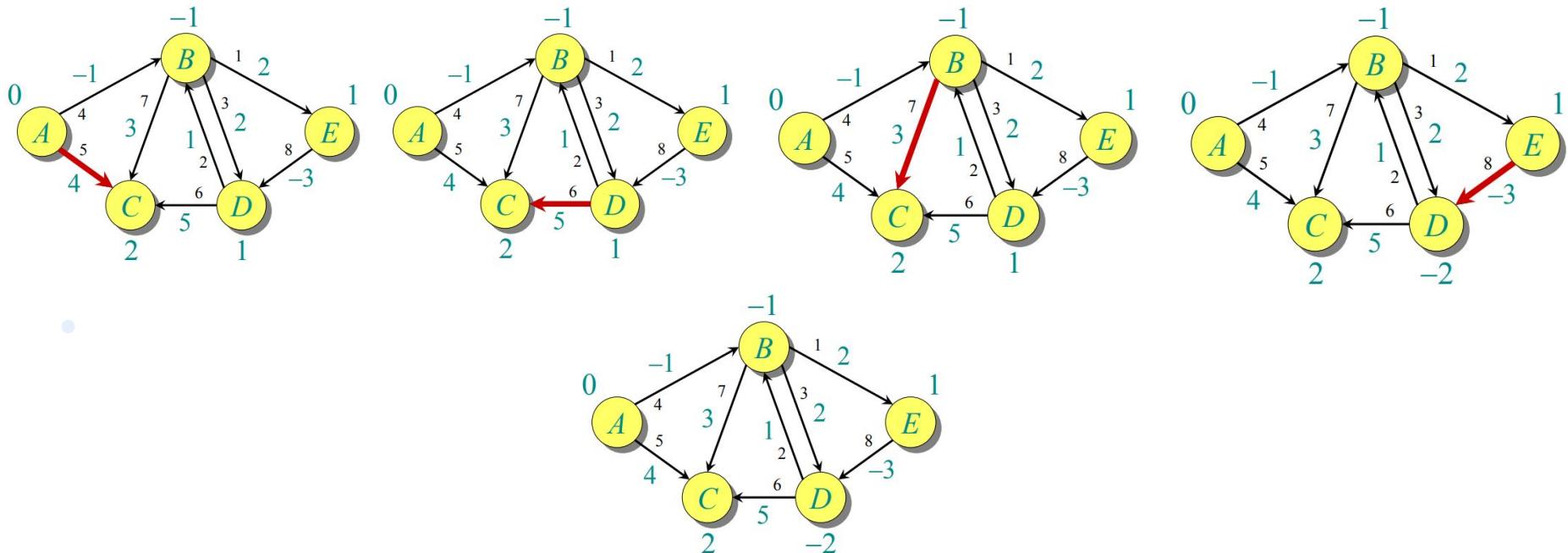


Explore | Expand | Enrich



Bellman-Ford Algorithm

Example



Explore | Expand | Enrich



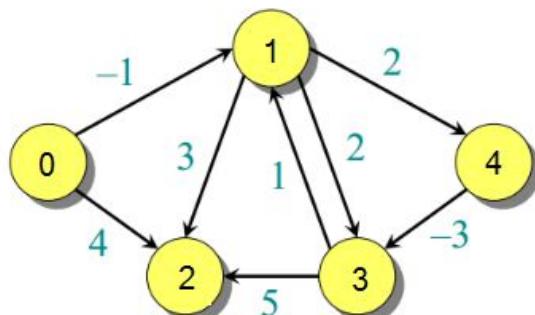
Bellman-Ford Algorithm

Program: Bellman Ford

bellmanford.java

Time Complexity: $O(V * E)$
Space Complexity: $O(V)$

Input



Output

The distance of vertex 1 from vertex 0 is -1. Its path is [0, 1]
The distance of vertex 2 from vertex 0 is 2. Its path is [0, 1, 2]
The distance of vertex 3 from vertex 0 is -2. Its path is [0, 1, 4, 3]
The distance of vertex 4 from vertex 0 is 1. Its path is [0, 1, 4]
The distance of vertex 2 from vertex 1 is 3. Its path is [1, 2]
The distance of vertex 3 from vertex 1 is -1. Its path is [1, 4, 3]
The distance of vertex 4 from vertex 1 is 2. Its path is [1, 4]
The distance of vertex 1 from vertex 3 is 1. Its path is [3, 1]
The distance of vertex 2 from vertex 3 is 4. Its path is [3, 1, 2]
The distance of vertex 4 from vertex 3 is 3. Its path is [3, 1, 4]
The distance of vertex 1 from vertex 4 is -2. Its path is [4, 3, 1]
The distance of vertex 2 from vertex 4 is 1. Its path is [4, 3, 1, 2]
The distance of vertex 3 from vertex 4 is -3. Its path is [4, 3]



Topological Sort



Introduction

The topological sort algorithm takes a DAG and returns an array of the nodes where each node appears before all the nodes it points to.

In computer science and mathematics, a directed acyclic graph (DAG) refers to a directed graph which has no directed cycles.

A DAG is used to represent a conceptual representation of a series of activities. The order of the activities is depicted by a graph

- The ordering of the nodes in the array is called a topological ordering

For every directed edge $u \rightarrow v$, vertex u comes before v in the ordering

Topological Sorting for a graph is not possible if the graph is not a DAG



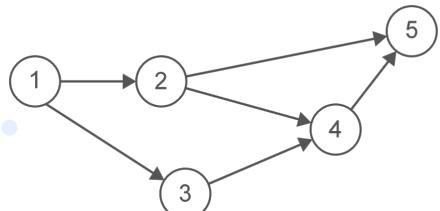


Topological Sort



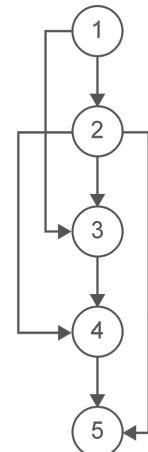
Explore | Expand | Enrich

Example



Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering. And, since nodes 2 and 3 both point to node 4, they appear before it in the ordering.

[1, 2, 3, 4, 5] would be a topological ordering of the graph.



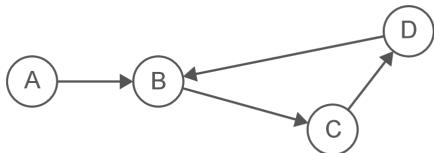


Topological Sort



Explore | Expand | Enrich

Counter-example



The cycle creates an impossible set of constraints—B has to be before and after D in the ordering.

As a rule, cyclic graphs don't have valid topological orderings.



Topological Sort



Explore | Expand | Enrich

Algorithm

1. Identify a node with no incoming edges.
2. Add that node to the ordering.
3. Remove it from the graph.
4. Repeat.

- We'll keep looping until there aren't any more nodes with indegree zero. This could happen for two reasons:
 - There are no nodes left. We've taken all of them out of the graph and added them to the topological ordering.
 - There are some nodes left, but they all have incoming edges. This means the graph has a cycle, and no topological ordering exists.



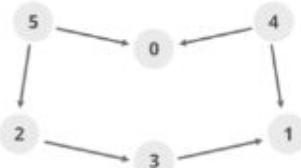


Topological Sort



Explore | Expand | Enrich

Steps involved



Adjac ent list (G)
0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

0	1	2	3	4	5
false	false	false	false	false	false

Stack(empty)

Step 1:

Topological Sort(0), visited[0] = true

List is empty. No more recursion call.

Stack

0	
---	--

Step 2:

Topological Sort(1), visited[1] = true

List is empty. No more recursion call.

Stack

0	1	
---	---	--

Step 3:

Topological Sort(2), visited[2] = true

Topological Sort(3), visited[3] = true

'1' is already visited. No more recursion call

Stack

0	1	3	2
---	---	---	---

Step 4:

Topological Sort(4), visited[4] = true

'0', '1' are already visited. No more recursion call

Stack

0	1	3	2	4
---	---	---	---	---

Step 5:

Topological Sort(5), visited[5] = true

'2', '0' are already visited. No more recursion call

Stack

0	1	3	2	4	5
---	---	---	---	---	---

Step 6:

Print all elements of stack from top to bottom



Topological Sort



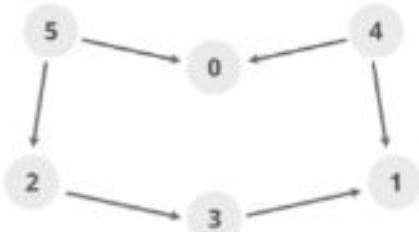
Explore | Expand | Enrich

Program

topologicalsort1.java

Time Complexity: $O(V + E)$
Space Complexity: $O(V)$

- Input



- Output

5 4 2 3 1 0





Topic: Heaps

Introduction

A heap is a complete binary tree

A complete binary tree is a binary tree in which all the levels except the last level

That is, the leaf node should be completely filled, and all the nodes should be left-justified

Two kinds of Heaps

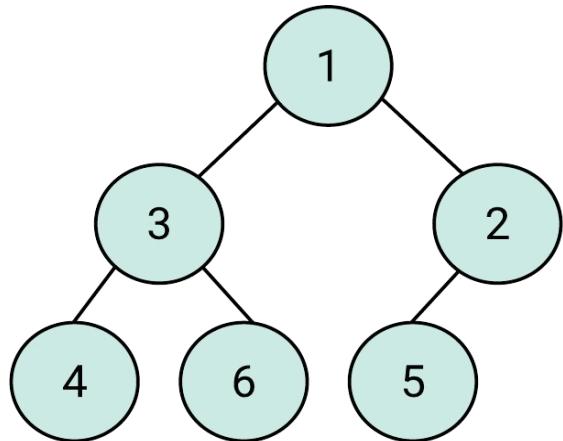
- Max Heap
 - In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all subtrees in that Binary Tree.
- Min Heap
 - In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all subtrees in that Binary Tree

Heap Data Structure

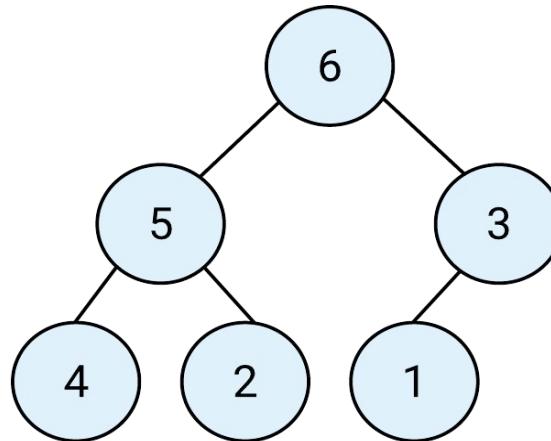
Max and Min Heaps



Explore | Expand | Enrich



Min heap



Max Heap



Heap Sort



Introduction

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array.

Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.





Heap Sort



Introduction

The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list

Heap sort is the in-place sorting algorithm

Heap sort algorithm consist of two phases

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.



Heap Sort



Algorithm:

```
heapSort(arr)
    buildMaxHeap(arr)
    for i = length(arr) to 2
        swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
    End
```

```
buildMaxHeap(arr)
    heap_size(arr) = length(arr)
    for i = length(arr)/2 to 1
        maxHeapify(arr,i)
    End
```

```
maxHeapify(arr,i)
    L = left(i)
    R = right(i)
    if L ? heap_size[arr] and arr[L] > arr[i]
        largest = L
    else
        largest = i
    if R ? heap_size[arr] and arr[R] > arr[largest]
        largest = R
    if largest != i
        swap arr[i] with arr[largest]
        maxHeapify(arr,largest)
    End
```





Heap Sort

Procedure

First, we have to construct a heap from the given array and convert it into max heap.

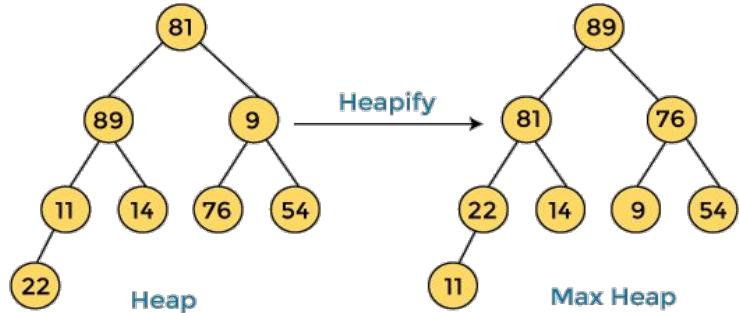
81	89	9	11	14	76	54	22
----	----	---	----	----	----	----	----

Heap Sort

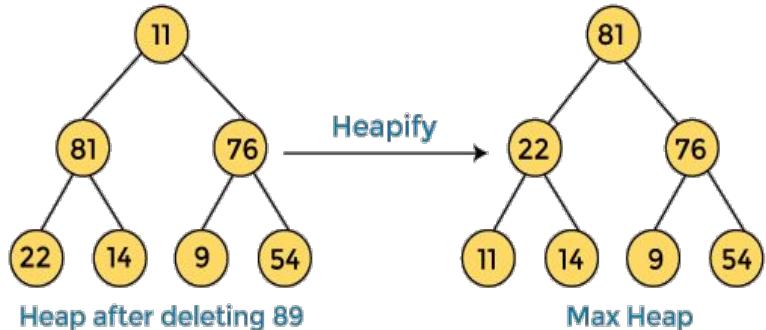
Procedure



Explore | Expand | Enrich



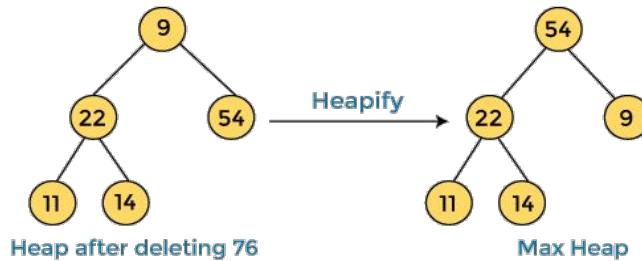
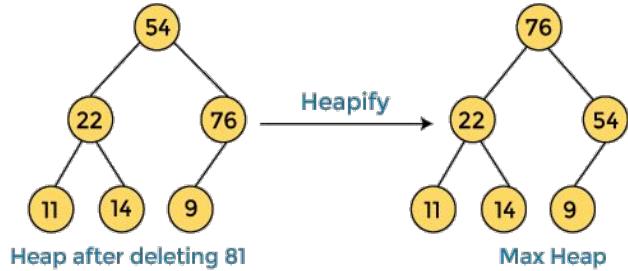
89	81	76	22	14	9	54	11
----	----	----	----	----	---	----	----



81	22	76	11	14	9	54	89
----	----	----	----	----	---	----	----

Heap Sort

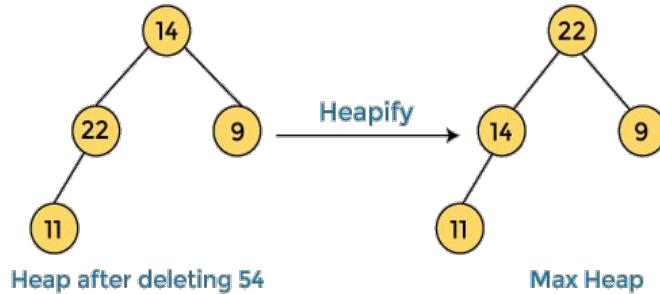
Procedure



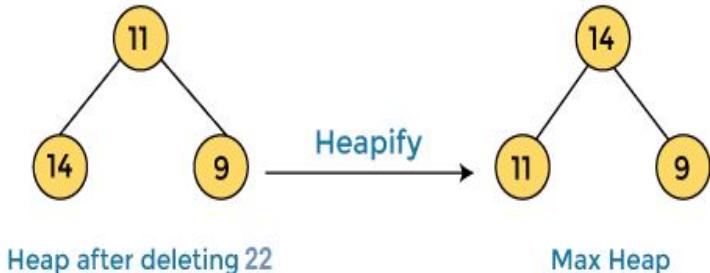
Heap Sort



Procedure



22	14	9	11	54	76	81	89
----	----	---	----	----	----	----	----

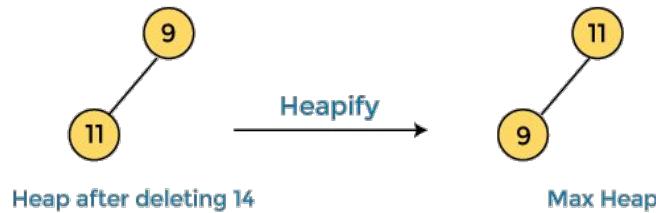


14	11	9	22	54	76	81	89
----	----	---	----	----	----	----	----

Heap Sort



Procedure



11	9	14	22	54	76	81	89
----	---	----	----	----	----	----	----



9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----

Heap Sort



Procedure



After completion of sorting, the array elements are

9	11	14	22	54	76	81	89
---	----	----	----	----	----	----	----



Heap Sort



Program

heapsort1.java

Time Complexity: $O(n \log n)$

Space Complexity: $O(1)$

Input

17 12 11 13 9 18

Output

9 11 12 13 17 18



Binomial Heap



Explore | Expand | Enrich

Introduction

Binomial Heap is an extension of Binary Heap

It provides faster union or merge operation together with other operations provided by Binary Heap.

Binary heap is used to implement priority queue, as seen in the previous sections

“A Binomial Heap is a collection of Binomial Trees”



Introduction

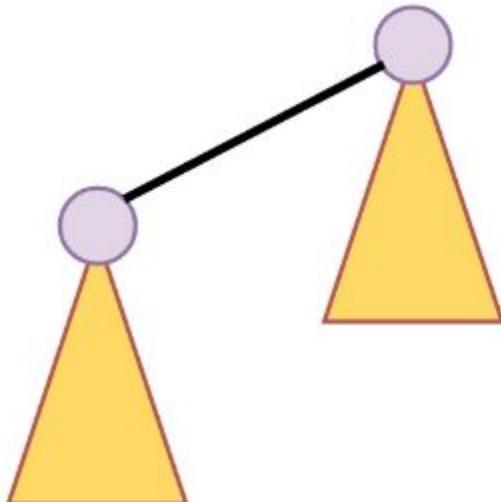
A binomial Heap is a collection of Binomial Trees.

A binomial tree B_k is an ordered tree defined recursively.

A binomial Tree B_0 is consists of a single node.

- A binomial tree B_k is consisting of k binomial trees B_{k-1} that are linked together

The root of one is the left most child of the root of the other

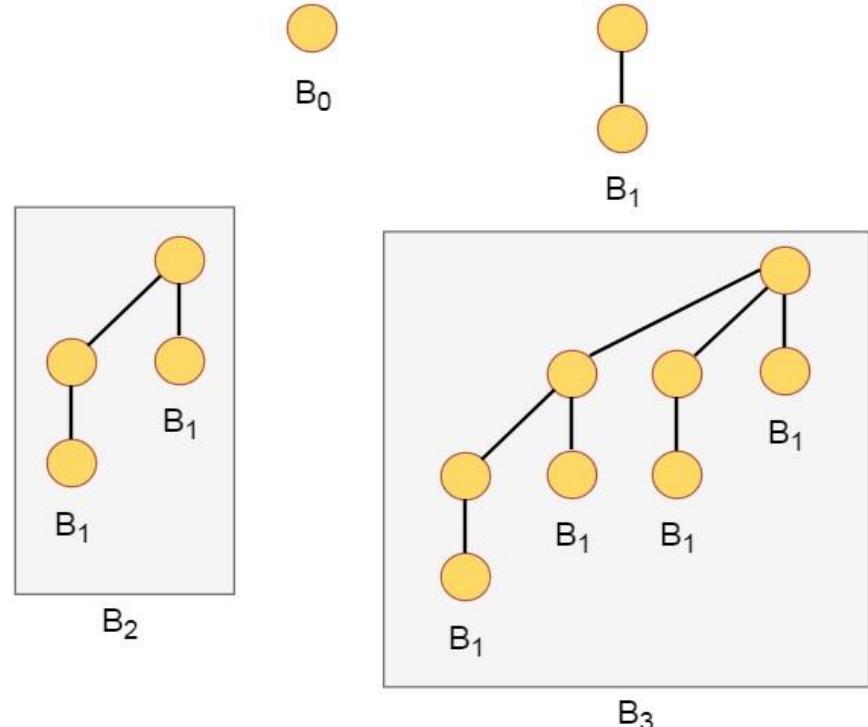


Binomial Tree

Properties and an example

A Binomial Tree of order k has following properties.

- It has exactly 2^k nodes.
- It has depth as k.
- There are exactly $\binom{k}{j}$ nodes at depth i for $i = 0, 1, \dots, k$.
- The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,.. 0 from left to right.





Binomial Heap



Explore | Expand | Enrich

Introduction

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows Min Heap property

There can be at most one Binomial Tree of any degree.

Properties

Each binomial tree in H is heap-ordered.

So the key of a node is greater than or equal to the key of its parent.

There is at most one binomial tree in H, whose root has a given degree.



Binomial Heap



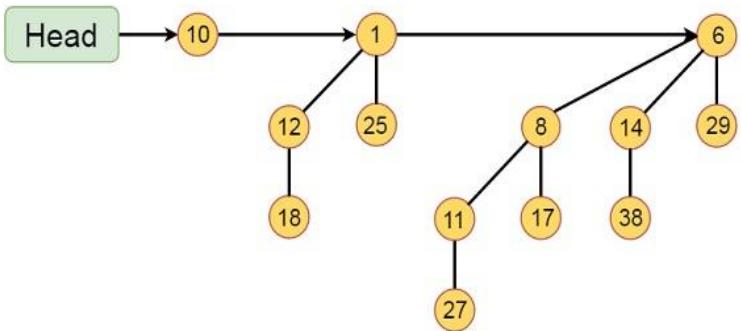
Example

This binomial Heap H consists of binomial trees B0, B2 and B3.

They have 1, 4 and 8 nodes respectively.

- In total $n = 13$ nodes.

The root of binomial trees are linked by a linked list in order of increasing degree



Operations

The main operation in Binomial Heap is union()

All other operations mainly use this operation.

The union() operation is to combine two Binomial Heaps into one



Operations

- **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
- **getMin(H):** A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key.
- **extractMin(H):** We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call union() on H and the newly created Binomial Heap.
- **delete(H):** Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().
- **decreaseKey(H):** decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node.

Union Operation

Given two Binomial Heaps H1 and H2, $\text{union}(H1, H2)$ creates a single Binomial Heap.

The first step is to simply merge the two Heaps in non-decreasing order of degrees.

After the simple merge, we need to make sure that there is at most one Binomial Tree of any order.

To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, prev, x and next-x. There can be following 4 cases when we traverse the list of roots.

- Orders of x and next-x are not same, we simply move ahead. In following 3 cases orders of x and next-x are same.
- If the order of next-next-x is also same, move ahead.
- If the key of x is smaller than or equal to the key of next-x, then make next-x as a child of x by linking it with x.
- If the key of x is greater, then make x as the child of next.

Binomial Heap

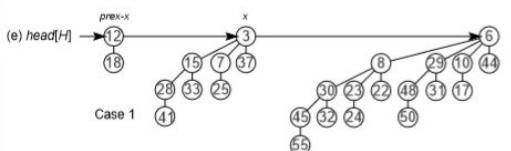
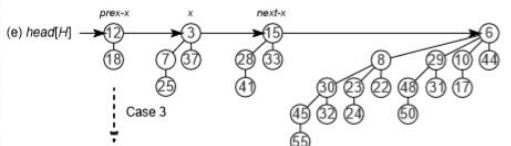
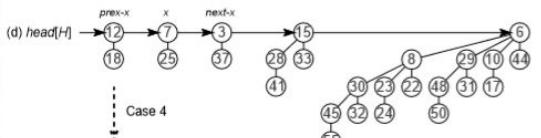
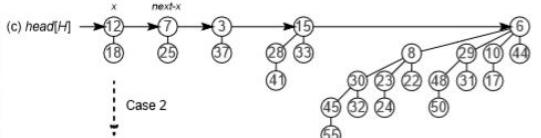
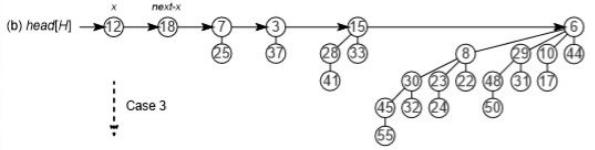
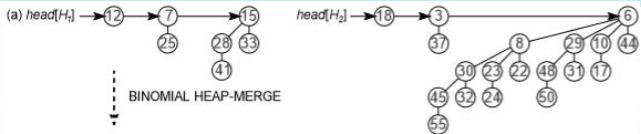


Diagram: Result after merging



Representation

A Binomial Heap is a set of Binomial Trees.

A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling. We need this in and extractMin() and delete().

- The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.



Binomial Heap



Program

binomialHeap.java

Operations are as follows

1. Insert(K): Insert an element K into the binomial heap
2. Delete(k): Deletes the element k from the heap
3. getSize(): Returns the size of the heap
4. makeEmpty(): Makes the binomial heap empty by deleting all the elements
5. checkEmpty(): Check if the binomial heap is empty or not
6. displayHeap(): Prints the binomial heap

Output

Size of the binomial heap is 7

Heap : 9 7 2 12 8 15 5

Size of the binomial heap is 5

Heap : 7 9 5 12 2

true



Introduction

The d-ary heap or d-heap or k-ary heap is a priority queue data structure, a generalization of the binary heap in which the nodes have d children instead of 2.

Properties

- Nearly complete binary tree, with all levels having maximum number of nodes except the last, which is filled in left to right manner.

It can be divided into two categories:

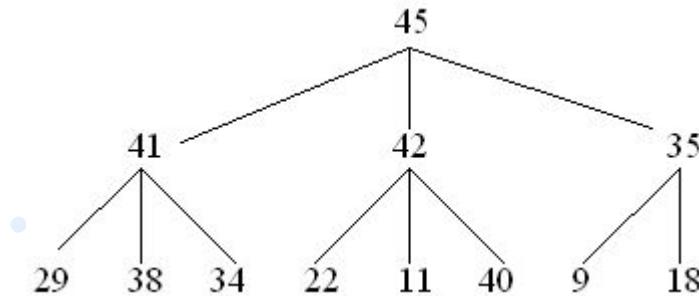
(a) Max k-ary heap: Key at root is greater than all descendants and same is recursively true for all nodes

(b) Min k-ary heap: Key at root is lesser than all descendants and same is recursively true for all nodes

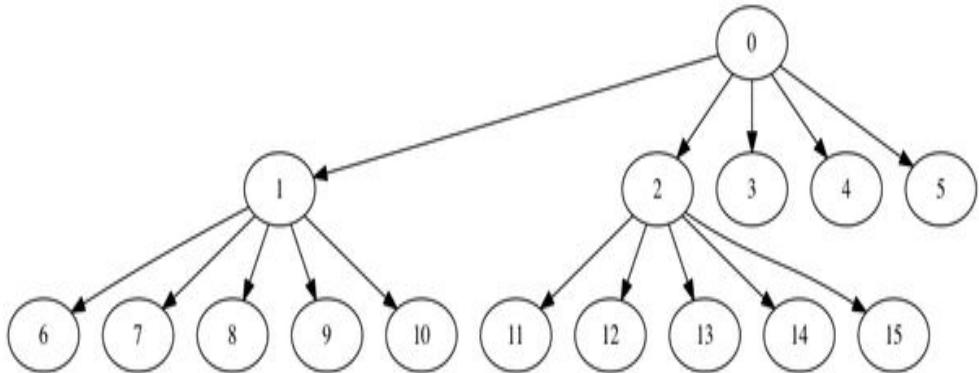
k-ary heap or d-ary heap

Examples

3-ary max heap



5-ary min heap



Implementation

Assuming 0 based indexing of array, an array represents a K-ary heap such that for any node we consider:

- Parent of the node at index i (except root node) is located at index $(i-1)/k$
- Children of the node at index i are at indices $(k*i)+1$, $(k*i)+2$ $(k*i)+k$
- The last non-leaf node of a heap of size n is located at index $(n-2)/k$

Implementation

`insert()` : Inserting a node into the heap

`buildHeap()` : Builds a heap from an input array.

- `restoreDown()` /`maxHeapify()` : Used to maintain heap property

`extractMax()` : Extracting the root node.



k-ary heap or d-ary heap



Explore | Expand | Enrich

Program

[karyheap.java](#)

For the actual Java program

[karyheapIO.txt](#)

For sample IO operations



Introduction

A tournament tree comes from the idea of a tournament or a competition

If there are p players in a tournament, then the tournament tree

- Will have p leaf nodes which can also be called external nodes
- Will have $p-1$ internal nodes

The tournament trees are also known as selection trees

The tournament tree is an application of the binary heap data structure

Introduction

The tournament tree is an application of the binary heap data structure

Hence it is a complete binary tree

Two Types of Tournament Trees

- 1. Winner Trees
 2. Looser Trees

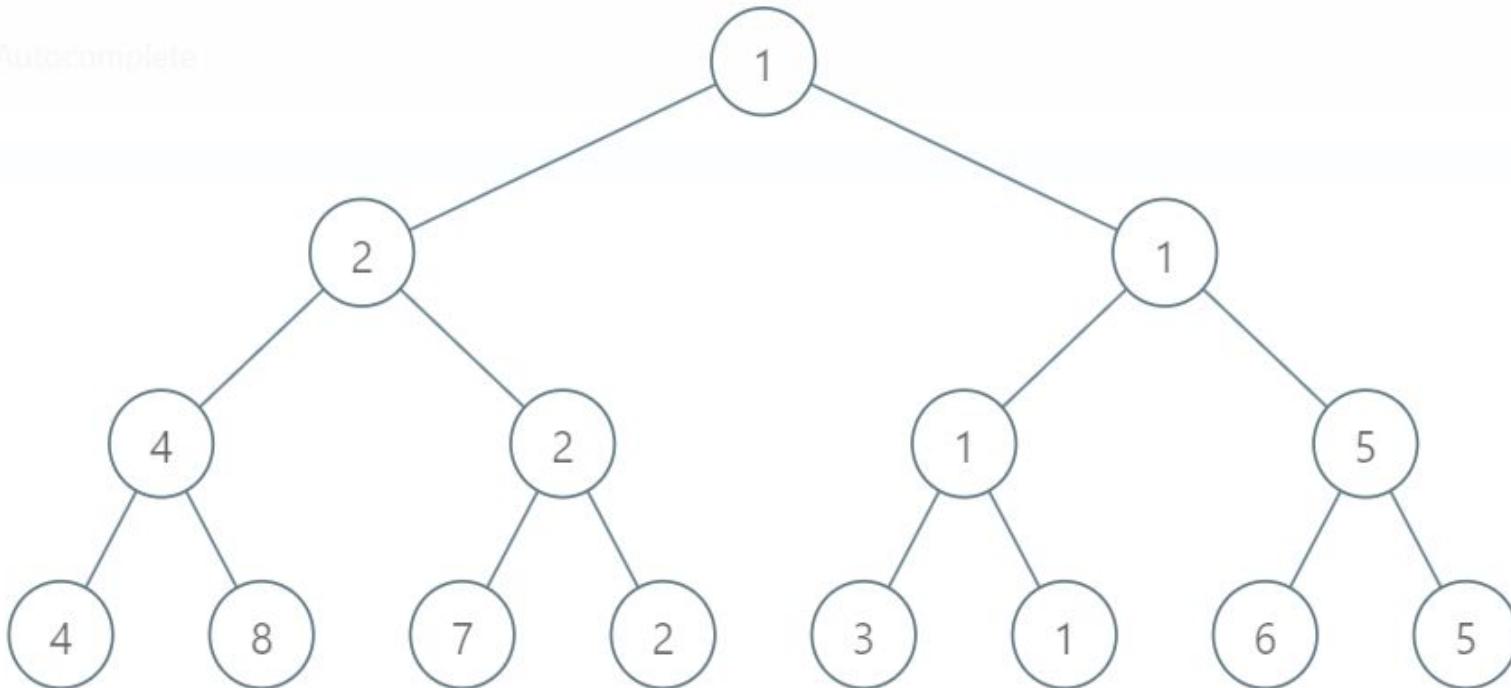
Introduction

The idea here is that in a winner tree, the nodes at a particular level store the winner of the two nodes at the level below it.

Hence, the tree's root node stores the overall winner of the competition.

- For instance, if we consider that in a tournament there are a total of eight players (1, 2, 3, 4, 5, 6, 7, 8) and in a match between any two players among these, the player with numerically smaller value wins the match (minimum winner tree).

Winner Trees



ich

Introduction

The loser tree is another type of tournament tree in which the nodes at a particular level store the loser of the two nodes at the level below it.

In a Loser tree, the tournament's winner is stored at the top, and the second place (runner-up) is the child of it.

So, if we are considering a numerically smaller value as the winner, then each node's value in a Loser Tree will be the greater value among both of its children.

Introduction

One advantage of loser trees compared to winner trees is that it is sufficient to examine the nodes on the path from leaf nodes to the root node for restructuring the tree in loser trees.

- To form a loser tree, we have to create winner tree at first. We will store looser of the match in each internal node.

Loser Trees can be used to find the runner up of a tournament

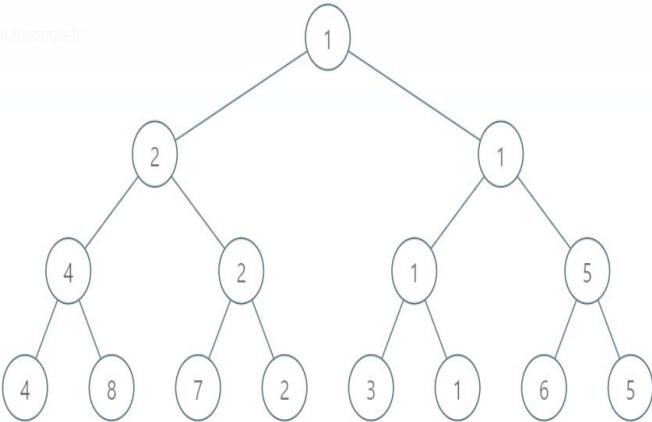
For each of the matches, the time complexity is $O(1)$, and in total, there are $n-1$ nodes (internal nodes); hence the time complexity to initialize the tree is $O(n)$.

Loser Trees

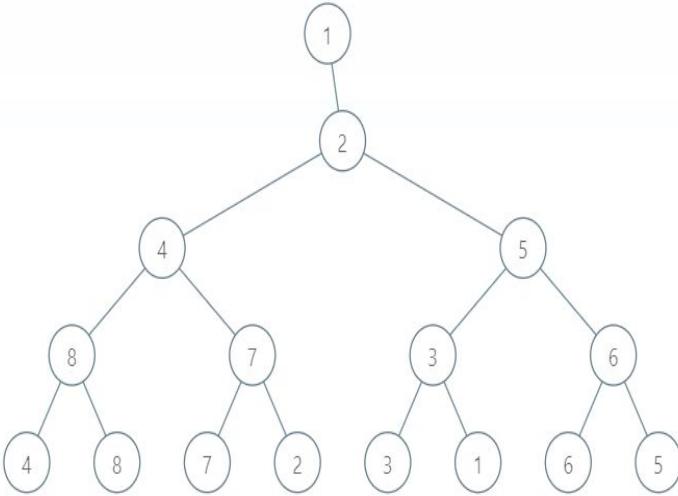


Explore | Expand | Enrich

* Autocomplete

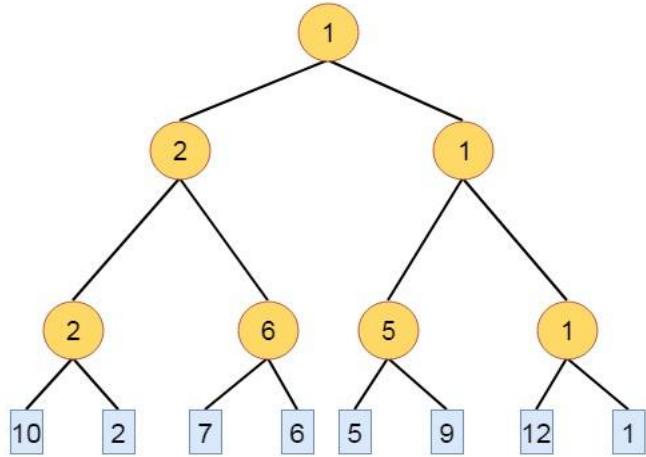


For this winner tree

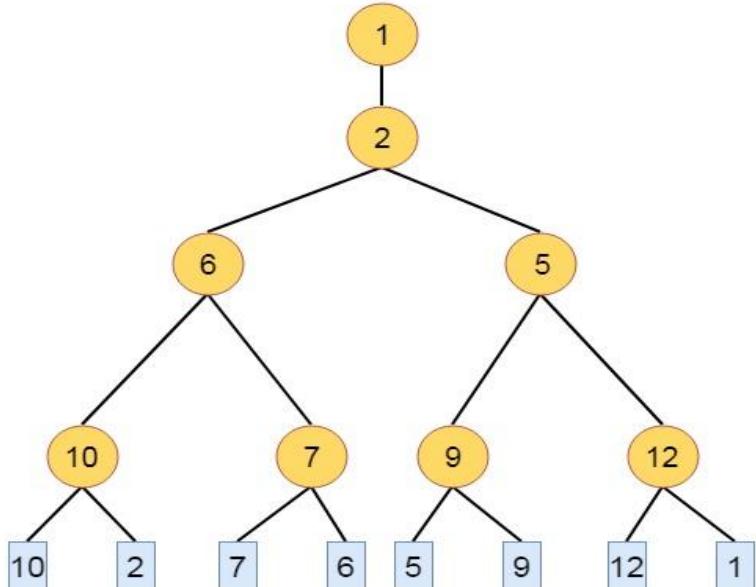


This is the loser tree (2 is the runner up)

Loser Trees



For this winner tree for the elements 10, 2, 7, 6, 5, 9, 12, 1.



This is the loser tree



Topic: Maps

Introduction

HashMap is implemented using a Hash Table

TreeMap is implemented using Red-Black Tree

HashMap is much faster than Treemap for inserting and searching

- That is, HashMap = constant time and TreeMap = Logarithmic time

However TreeMap guarantees ordering and HashMap does not.

Introduction

In a TreeMap, the map is ordered according to the natural ordering of its keys or a specified Comparator in the TreeMap's constructor.

If we want near-HashMap performance and insertion-order iteration, we can use LinkedHashMap.

- There are a few ways to convert HashMap to TreeMap in Java:
 1. Using Collectors (valid in Java 8 and above)
 2. Using vanilla Java
 3. Using Guava library
 4. Manual conversion

Introduction

To sum it up, Hashmap stores data in (Key, Value) pair

To access a value in HashMap, one must know its key

HashMap is known as HashMap because it uses a technique Hashing for storage of data

The TreeMap in Java is used to implement Map interface and NavigableMap along with the Abstract Class

The map is sorted according to the natural ordering or it can be done using a comparator provided while map creation

Method: Using Vanilla Java (Direct Method)

Pass HashMap instance to the TreeMap constructor OR to putAll() method.

This will directly create the TreeMap from the HashMap

Algorithm

- Get the HashMap to be converted
- Create a new TreeMap
- Pass the hashMap to putAll() method of treeMap
- Return the formed TreeMap



Hashmap to Treemap



Program: Direct Method

hashMapToTreeMap1.java



Hashmap to Treemap



Method: Using Stream.collect() in Java 8+

This method includes converting the HashMap to a Stream and collects elements of a stream in a TreeMap

This is done using Stream.collect() method which accepts a collector.

Algorithm for the method using Stream.collect()

- Get the HashMap to be converted
- Get the entries from the HashMap
- Convert the map entries into stream
- Using Collectors, collect the entries and convert it into TreeMap
- Now collect the TreeMap
- Return the formed TreeMap



Hashmap to Treemap



Explore | Expand | Enrich

Program: Using Stream.collect()

hashMapToTreeMap2.java



Method: Using Guava library

Guava also provides a TreeMap implementation which can be used to create an empty TreeMap instance.

Algorithm

- Get the HashMap to be converted
- Create a new TreeMap using `Maps.newTreeMap()` of Guava library
- Pass the hashMap to `putAll()` method of treeMap
- Return the formed TreeMap



Hashmap to Treemap



Explore | Expand | Enrich

Program: Using Guava

hashMapToTreeMap3.java



Method: Conversion between incompatible types

This method can be used if the required TreeMap is of the different type than the HashMap.

In this, the conversion needs to be done manually

Algorithm

- Get the HashMap to be converted
- Create a new TreeMap
- For each entry of the hashMap:
 - Convert the Key and the Value into the desired type by casting
 - Insert the converted pair by put() method of treeMap
- Return the formed TreeMap



Hashmap to Treemap



Program: Using Conversion

hashMapToTreeMap4.java



Topic: Sets

Introduction

- 1. Empty set
- 2. Singleton set
- 3. Finite set
- 4. Infinite set
- 5. Equal sets
- 6. Equivalent sets
- 7. Universal set
- 8. Subset
- 9. Proper subset
- 10. Superset
- 11. Proper superset
- 12. Power set

Types of Sets



Explore | Expand | Enrich

Finite Set

A set which contains a definite number of elements is called a finite set.

Example: $S = \{ x \mid x \in N \text{ and } 70 > x > 50 \}$

Infinite Set

A set which contains infinite number of elements is called an infinite set.

Example: $S = \{ x \mid x \in N \text{ and } x > 10 \}$

Subset

A set X is a subset of set Y (Written as $X \subseteq Y$) if every element of X is an element of set Y.

Example 1: Let, $X = \{ 1, 2, 3, 4, 5, 6 \}$ and $Y = \{ 1, 2 \}$. Here set Y is a subset of set X as all the elements of set Y is in set X. Hence, we can write $Y \subseteq X$.

Example 2 : Let, $X = \{ 1, 2, 3 \}$ and $Y = \{ 1, 2, 3 \}$. Here set Y is a subset (Not a proper subset) of set X as all the elements of set Y is in set X. Hence, we can write $Y \subseteq X$.

Proper Subset

The term “proper subset” can be defined as “subset of but not equal to”. A Set X is a proper subset of set Y (Written as $X \subset Y$) if every element of X is an element of set Y and $|X| < |Y|$.

Example: Let, $X = \{ 1, 2, 3, 4, 5, 6 \}$ and $Y = \{ 1, 2 \}$. Here set $Y \subset X$ since all elements in X are contained in X too and X has at least one element more than set Y.

Universal Set

It is a collection of all elements in a particular context or application. All the sets in that context or application are essentially subsets of this universal set. Universal sets are represented as U.

Example: We may define U as the set of all animals on earth. In this case, set of all mammals is a subset of U, set of all fishes is a subset of U, set of all insects is a subset of U, and so on



Types of Sets



Empty Set or Null Set

An empty set contains no elements. It is denoted by \emptyset . As the number of elements in an empty set is finite, empty set is a finite set. The cardinality of empty set or null set is zero.

Example: $S = \{ x \mid x \in N \text{ and } 7 < x < 8 \} = \emptyset$

Singleton Set or Unit Set

Singleton set or unit set contains only one element. A singleton set is denoted by $\{ s \}$.

Example: $S = \{ x \mid x \in N, 7 < x < 9 \} = \{ 8 \}$

Equal Set

If two sets contain the same elements they are said to be equal.

Example: If $A = \{ 1, 2, 6 \}$ and $B = \{ 6, 1, 2 \}$, they are equal as every element of set A is an element of set B and every element of set B is an element of set A.





Types of Sets



Equivalent Set

If the cardinalities of two sets are same, they are called equivalent sets.

Example: If $A = \{ 1, 2, 6 \}$ and $B = \{ 16, 17, 22 \}$, they are equivalent as cardinality of A is equal to the cardinality of B. i.e. $|A| = |B| = 3$

Overlapping Set

Two sets that have at least one common element are called overlapping sets.

In case of overlapping sets :

$$n(A \cup B) = n(A) + n(B) - n(A \cap B)$$

$$n(A \cup B) = n(A - B) + n(B - A) + n(A \cap B)$$

$$n(A) = n(A - B) + n(A \cap B)$$

$$n(B) = n(B - A) + n(A \cap B)$$

Example: Let, $A = \{ 1, 2, 6 \}$ and $B = \{ 6, 12, 42 \}$. There is a common element '6', hence these sets are overlapping sets.





Types of Sets



Disjoint Set

Two sets A and B are called disjoint sets if they do not have even one element in common. Therefore, disjoint sets have the following properties:

$$n(A \cap B) = \emptyset$$

$$n(A \cup B) = n(A) + n(B)$$

Example: Let, A = { 1, 2, 6 } and B = { 7, 9, 14 }, there is not a single common element, hence these sets are overlapping sets.

Power Set

The power set is a set which includes all the subsets including the empty set and the original set itself. Power set's cardinality depends on the number of subsets formed for a given set.

Example: If set A = {x, y, z} and its subsets {x}, {y}, {z}, {x, y}, {y, z}, {x, z}, {x, y, z} and {}

Power set of A, denoted using P(A) = { {x}, {y}, {z}, {x, y}, {y, z}, {x, z}, {x, y, z}, {} }





Topic: Hashmap

Introduction

You have been given N sweets of many different types

There are k customers and one customer won't make the same type of sweet more than 2 pieces

The task is to find if it is possible to distribute all, then print true or otherwise false

The array arr represents the array of sweets and consequently, $\text{arr}[i]$ is the type of sweet

There are two methods to solve the problem: Brute Force Method and Hashing Method



Distributing Sweets



Examples

Input

arr = [1, 1, 2, 3, 1]

k = 2

Output

true

Explanation

There are three pieces of sweet type 1, one piece of type 2 and one piece of type 3. Two customers can distribute sweets under given constraints.

Input

arr = [2, 3, 3, 5, 3, 3]

k = 2

Output

true

Input

arr = [2, 3, 3, 5, 3, 3, 3]

k = 2

Output

false



Brute Force Method

1. Traverse array for each element
 2. Count occurrences of each element in the array
 3. Check if the result of each element must be less than or equal to 2^*k
- ..



Distributing Sweets



Explore | Expand | Enrich

Program: Brute Force Method

distSweets1.java

Input

arr = [2, 3, 3, 5, 3, 3]
k = 2

Time Complexity

$O(n^2)$

Output

- true

Input

arr = [2, 3, 3, 5, 3, 3, 3]
k = 2

Output

false



Hash Table Method

1. Maintain a hash for 32 different type of sweets.
2. Traverse an array and check for every $\text{arr}[i]$

• $\text{hash}[\text{arr}[i]] \leq 2^*k$



Distributing Sweets



Explore | Expand | Enrich

Program: Hash Table Method

distSweets2.java

Input

arr = [2, 3, 3, 5, 3, 3]
k = 2

Time Complexity

$O(n)$

Output

- true

Input

arr = [2, 3, 3, 5, 3, 3, 3]
k = 2

Output

false





Topic: Dynamic Programming

Introduction

Dynamic Programming (DP) is an algorithmic paradigm that solves a given complex problem by breaking it into sub-problems and stores the results of sub-problems to avoid computing the same results again.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used.

- Mostly, these algorithms are used for optimization.

Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.

The solutions of sub-problems are combined in order to achieve the best solution.



Introduction

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such

- optimal substructures are usually described by means of recursion.

Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.



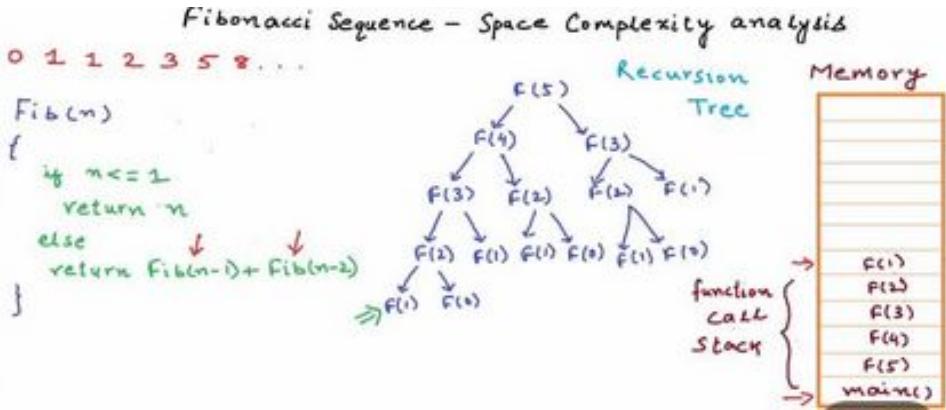
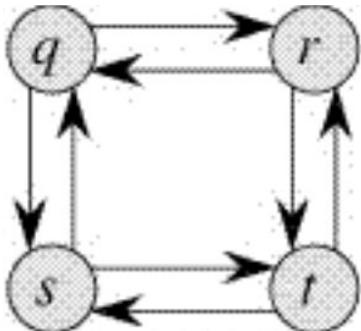
Properties

Dynamic Programming Properties

1) Overlapping Subproblems

- a) Memoized (Top Down)
- b) Tabulation (Bottom Up)

2) Optimal Substructure



- Optimal Substructure
 - An optimal solution to a problem contains within it an optimal solution to subproblems
 - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
 - If a recursive algorithm revisits the same subproblems over and over \Rightarrow the problem has overlapping subproblems

Introduction

Two approaches to solve Dynamic Programming problems

1. Top-Down Approach
 2. Bottom-Up Approach
- 

Top Down Approach

This is the direct fall-out of the recursive formulation of any problem.

If the solution to any problem can be formulated recursively using the solution to its sub-problems, and if its sub-problems are overlapping, then one can easily memorize or store the solutions to the sub-problems in a table.

- Whenever we attempt to solve a new sub-problem, we first check the table to see if it is already solved.

If a solution has been recorded, we can use it directly, otherwise we solve the sub-problem and add its solution to the table.

Bottom-up approach

Once we formulate the solution to a problem recursively as in terms of its sub-problems, we can try reformulating the problem in a bottom-up fashion: try solving the sub-problems first and use their solutions to build-on and arrive at solutions to bigger sub-problems.

This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub-problems by using the solutions to small sub-problems

- Top Down and Bottom Up approach are also known as Memoization and Tabulation respectively

Dynamic Programming



Explore | Expand | Enrich

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Examples of Problems solved using Dynamic programming

1. Fibonacci number series
2. Knapsack problem
3. Tower of Hanoi
4. All pair shortest path by Floyd-Warshall
5. Shortest path by Dijkstra
6. Project scheduling

Examples: Fibonacci Number series

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F(n) = F(n)-1 + F(n)-2, \text{ where } F(0) = 0 \text{ and } F(1) = 1$$



Dynamic Programming



Explore | Expand | Enrich

Program - Fibonacci Series: Without Dynamic Programming

`fibowithoutdp.java`

Fibonacci Series: Top Down approach

In this example, we initialize a lookup array with all initial values as -1.

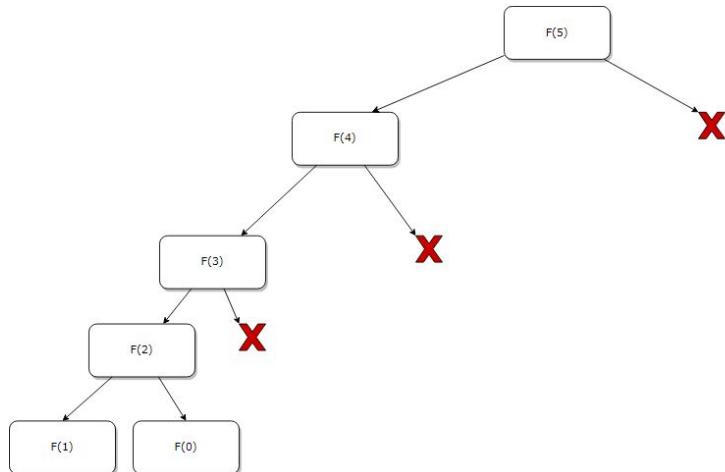
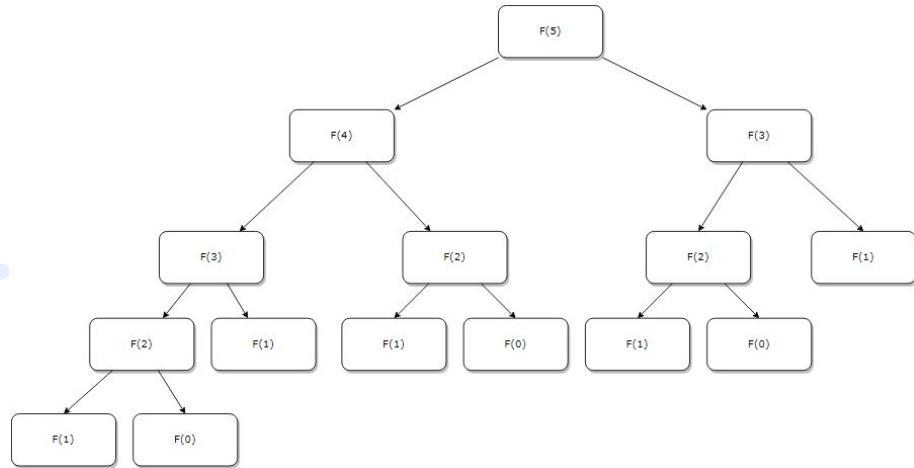
Whenever we need solution to a subproblem, we first look into the lookup array.

If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup array so that it can be reused later.



Dynamic Programming

Fibonacci Series: Top Down approach



Program - Fibonacci Series: Top Down approach

fibotopdown.java

Fibonacci Series: Bottom-Up approach

In the bottom-up dynamic programming approach, we'll reorganize the order in which we solve the subproblems.

We'll compute $F(0)$, then $F(1)$, then $F(2)$, and so on

- This will allow us to compute the solution to each problem only once, and we'll only need to save two intermediate results at a time.

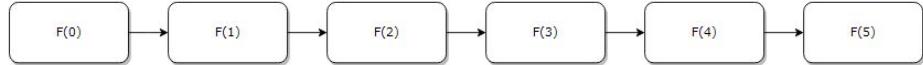
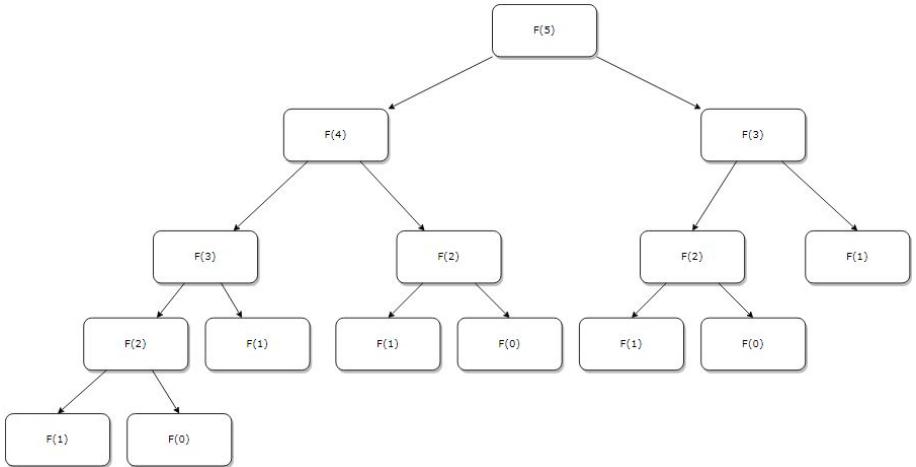
For example, when we're trying to find $F(2)$, we only need to have the solutions to $F(1)$ and $F(0)$ available. Similarly, for $F(3)$, we only need to have the solutions to $F(2)$ and $F(1)$.



Dynamic Programming



Fibonacci Series: Bottom Up approach





Dynamic Programming



Program - Fibonacci Series: Bottom Up approach

fibobottomup.java



Longest Common Subsequence



Introduction

The Longest Common Subsequence (LCS) problem is finding the longest subsequence present in given two sequences in the same order

Find the longest sequence which can be obtained from the first original sequence by deleting some items and from the second original sequence by deleting other items

- Unlike substrings, subsequences are not required to occupy consecutive positions within the original string





Longest Common Subsequence



Subsequence vs Substring vs Subarray vs Subset

A subarray is a slice from a contiguous array and inherently maintains the order of elements.

The subarrays of array {1, 2, 3} are {1}, {1, 2}, {1, 2, 3}, {2}, {2, 3}, and {3}.

A substring of a string s is a string s' that occurs in s.

The substrings of string 'apple' are 'apple', 'appl', 'pple', 'app', 'ppl', 'ple', 'ap', 'pp', 'pl', 'le', 'a', 'p', 'l', 'e', ''

A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.

For example, {B, D} is a subsequence of sequence { B, C, D, E} obtained after removing {C} and {E}

A subset is any possible combination of the original set. The term subset is often used for subsequence, but that's not right.

A subsequence always maintains the relative order of the array elements (i.e., increasing index), but there is no such restriction on a subset.

For example, {3, 1} is a valid subset of {1, 2, 3, 4, 5}, but it is neither a subsequence nor a subarray





Longest Common Subsequence



Explore | Expand | Enrich

Example

Consider two strings X and Y

Where,

X = ABCBDAB

Y = BDCABA

The length of the LCS is 4

LCS are BDAB, BCAB, and BCBA



Brute Force solution

A Brute force solution is to check if every subsequence of $X[1\dots m]$ to see if it is also a subsequence of $Y[1\dots n]$.

As there are 2^m subsequences possible of X , the time complexity of this solution would be

- $O(n * 2^m)$

where m is the length of the first string and n is the length of the second string

We can exploit the optimal substructure nature of the problem to reduce the time complexity



Longest Common Subsequence



Top Down Solution

Let us consider two sequences, X and Y, of length m and n that both end in the same element

To find their LCS, shorten each sequence by removing the last element, find the LCS of the shortened sequences, and that LCS append the removed element

$$LCS(X[1 \dots m], Y[1 \dots n]) = LCS(X[1 \dots m-1], Y[1 \dots n-1]) + X[m] \quad \text{if } X[m] = Y[n]$$



Longest Common Subsequence



Top Down Solution

Now suppose that the two sequences does not end in the same symbol

Then the LCS of X and Y is the longer of the two sequences $\text{LCS}(X[1\dots m-1], Y[1\dots n])$ and $\text{LCS}(X[1\dots m], Y[1\dots n-1])$.

To understand this property, let's consider the two following sequences:

X: ABCBDAB (*n elements*)
Y: BDCABA (*m elements*)



Longest Common Subsequence



Explore | Expand | Enrich

Top Down Solution

The LCS of these two sequences either ends with B (the last element of the sequence X) or does not

Case 1: If LCS ends with B, then it cannot end with A, and we can remove A from the sequence Y, and the problem reduces to $\text{LCS}(X[1\dots m], Y[1\dots n-1])$

Case 2: If LCS does not end with B, then we can remove B from sequence X and the problem reduces to $\text{LCS}(X[1\dots m-1], Y[1\dots n])$. An example of this is shown below.

- $\text{LCS}(ABCBDAB, BDCABA) = \text{maximum } (\text{LCS}(ACBDA, BDCABA), \text{LCS}(ABCDBAB, BDCAB))$

$$\text{LCS}(ACBDA, BDCABA) = \text{LCS}(ACBD, BDCAB) + A$$

$$\text{LCS}(ABCDBAB, BDCAB) = \text{LCS}(ACBDA, BDCA) + B$$

$$\text{LCS}(ACBD, BDCAB) = \text{maximum } (\text{LCS}(ACB, BDCAB), \text{LCS}(ACBD, BDCA))$$

$$\text{LCS}(ACBDA, BDCA) = \text{LCS}(ACBD, BDC) + A$$





Longest Common Subsequence



Program: Basic

`lcs1.java`

Input

ABCBDA
BDCABA

Output

4

Time Complexity

$O(2^{m+n})$

where m and n are the length of the strings



Longest Common Subsequence

Overlapping Subproblems

The LCS problem exhibits overlapping subproblems.

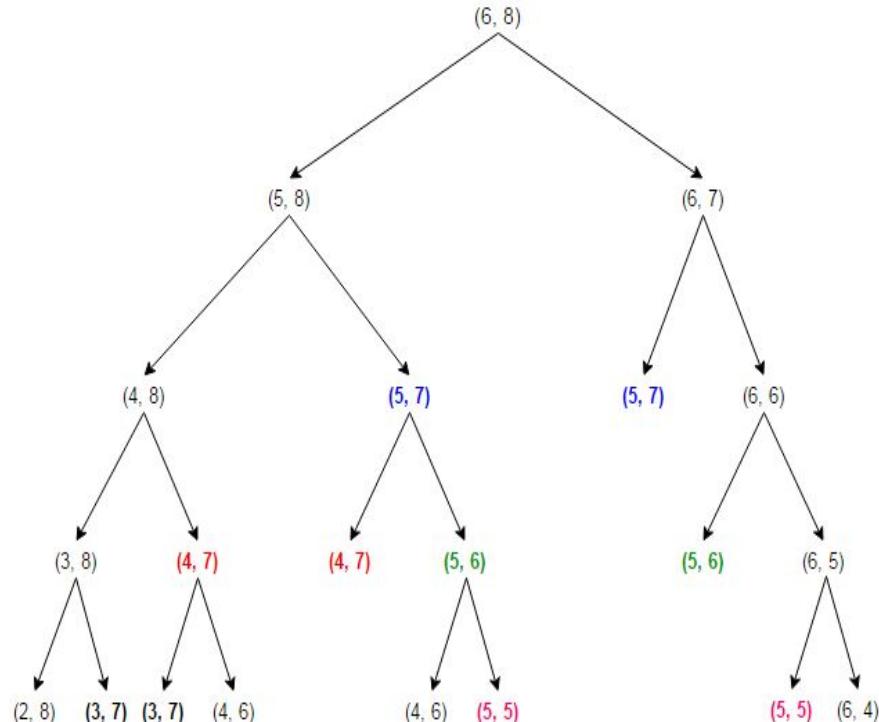
A problem is said to have overlapping subproblems if the recursive algorithm for the problem solves the same subproblem repeatedly rather than generating new subproblems

A recursion tree for two sequences of length 6 and

- 8 looks like the following

As we can see, the same subproblems (highlighted in the same color) are getting computed repeatedly.

We know that problems having optimal substructure and overlapping subproblems can be solved by dynamic programming





Longest Common Subsequence



Program: With memoization

Ics2.java

Input

ABCBDA
BDCABA

Output

4

Time and space Complexity

$O(mn)$

where m and n are the length of the strings



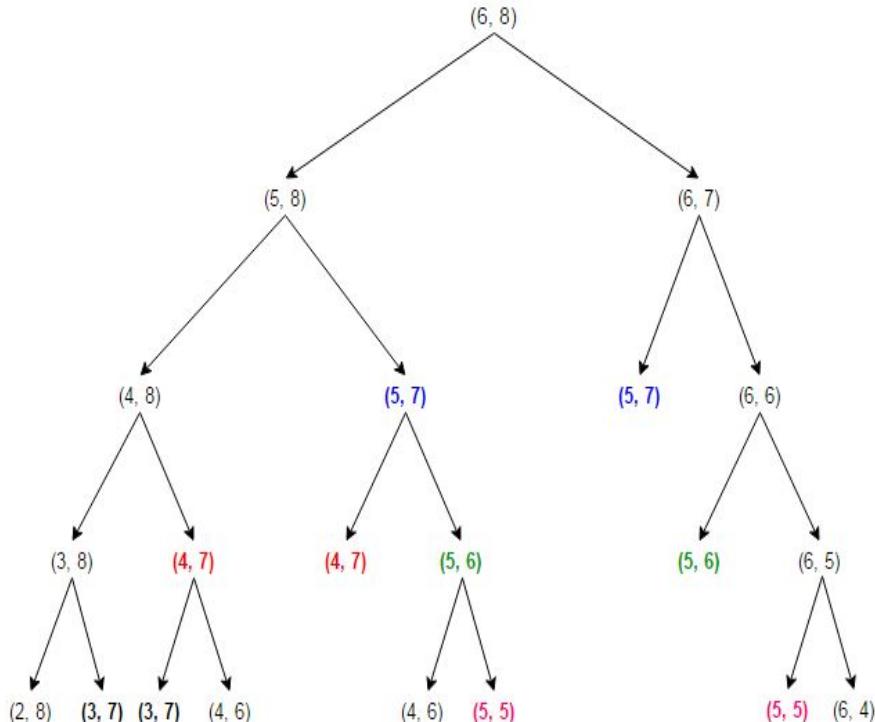
Longest Common Subsequence

Overlapping Subproblems

We have memoized and thus prevented repeated computation

The memoized version follows the top-down approach since we first break the problem into subproblems and then calculate and store values.

- We can also solve this problem in a bottom-up manner.





Longest Common Subsequence



Explore | Expand | Enrich

Bottom Up Solution

In the bottom-up approach, we calculate the smaller values of $LCS(i, j)$ first, then build larger values from them. The rules are given as follows

$$LCS[i][j] = \begin{cases} 0 & \text{if } i == 0 \text{ or } j == 0 \\ LCS[i - 1][j - 1] + 1 & \text{if } X[i-1] == Y[j-1] \\ \text{longest}(LCS[i - 1][j], LCS[i][j - 1]) & \text{if } X[i-1] != Y[j-1] \end{cases}$$

- Let X be XMJYAUZ, and Y be MZJAWXU. The longest common subsequence between X and Y is MJAU. The table is generated by the function in the code.

The function shows the LCS's length between prefixes of X and Y . The i 'th row and j 'th column show the LCS's length of substring $X[0\dots i-1]$ and $Y[0\dots j-1]$

	0	1	2	3	4	5	6	7
Ø	Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1
2	M	0	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2
4	Y	0	1	1	2	2	2	2
5	A	0	1	1	2	3	3	3
6	U	0	1	1	2	3	3	4
7	Z	0	1	2	2	3	3	4





Longest Common Subsequence



Program: With bottom up method

Ics3.java

Input

ABCBDA
BDCABA

Output

4

Time and space Complexity

$O(mn)$

where m and n are the length of the strings



Introduction

The longest increasing subsequence problem is to find a subsequence of a given sequence

The subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible

- This subsequence is not necessarily contiguous or unique

Subsequences are not required to occupy consecutive positions within the original sequences



Longest Increasing Subsequence



Explore | Expand | Enrich

Example

Input

[0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]

Output

[0, 2, 6, 9, 11, 15]

Explanation

The longest increasing subsequence of [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15] is [0, 2, 6, 9, 11, 15].

This subsequence has length 6; the input sequence has no 7-member increasing subsequences. The longest increasing subsequence in this example is not unique.



Solution: Using recursion

The idea is to use recursion to solve this problem.

For each item, there are two possibilities

1. Include the current item in LIS if it is greater than the previous element in LIS and recur for the remaining items
2. Exclude the current item from LIS and recur for the remaining items

Finally, return the maximum value we get by including or excluding the current item.

The base case of the recursion would be when no items are left



Longest Increasing Subsequence



Program: Using Recursion
`lis1.java`

Input

10, 22, 9, 33, 21, 50, 41, 60

Output

5

Time Complexity

$O(2^n)$, where n is the size of the array

Space Complexity

$O(n)$



Bottom Up Approach

Bottom-up approach computes $L[i]$, for each $0 \leq i < n$, which stores the length of the longest increasing subsequence of subarray $\text{arr}[0\dots i]$ that ends with $\text{arr}[i]$.

To calculate $L[i]$, consider LIS of all smaller values of i (say j) already computed and pick maximum $L[j]$, where $\text{arr}[j]$ is less than the current element $\text{arr}[i]$.

It has the same asymptotic runtime as Memoization but no recursion overhead

Optimal Substructure

The problem has an optimal substructure. That means the problem can be broken down into smaller, simple “subproblems”

The above solution also exhibits overlapping subproblems.

- If we draw the solution’s recursion tree, we can see that the same subproblems are repeatedly computed

Thus, we can use dynamic programming. We can use memoization due to its overlapping subproblems



Longest Increasing Subsequence



Program: Using Recursion
`lis2.java`

Input

10, 22, 9, 33, 21, 50, 41, 60

Output

5

Time Complexity

$O(n^*n)$, where n is the size of the array

Space Complexity

$O(n^*n)$



Longest Increasing Subsequence



Printing LIS

The above-discussed methods only print the length of LIS but do not print LIS itself.

To print the LIS, we have to store the longest increasing subsequence in

- the lookup table instead of storing just the LIS length.

Example: Consider array arr = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], the LIS of subarray arr[0...i] that ends with arr[i] are given in the right

LIS[0] — 0
LIS[1] — 0 8
LIS[2] — 0 4
LIS[3] — 0 8 12
LIS[4] — 0 2
LIS[5] — 0 8 10
LIS[6] — 0 4 6
LIS[7] — 0 8 12 14
LIS[8] — 0 1
LIS[9] — 0 4 6 9
LIS[10] — 0 4 5
LIS[11] — 0 4 6 9 13
LIS[12] — 0 2 3
LIS[13] — 0 4 6 9 11
LIS[14] — 0 4 6 7
LIS[15] — 0 4 6 9 13 15



Longest Increasing Subsequence



Explore | Expand | Enrich

Program: Printing Elements of LIS

`lis3.java`

Input

0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

Output

0, 4, 6, 9, 13, 15

Time Complexity

$O(n^*n)$, where n is the size of the array

Space Complexity

$O(n^*n)$



Introduction

A sequence is called Bitonic if it is first increasing, then decreasing

In other words, an array $\text{arr}[0..n-i]$ is Bitonic if there exists an index i where $0 \leq i \leq n-1$ such that

- $x_0 \leq x_1 \dots \leq x_i \text{ and } x_i \geq x_{i+1} \dots \geq x_{n-1}$

Examples for Bitonic sequence would be the sequences $(1, 4, 6, 8, 3, -2)$, $(9, 2, -4, -10, -5)$, and $(1, 2, 3, 4)$ are bitonic, but $(1, 3, 12, 4, 2, 10)$ is not bitonic



Longest Bitonic Subsequence



Explore | Expand | Enrich

Problem

The longest bitonic subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are first sorted in increasing order, then in decreasing order, and the subsequence is as long as possible

- Example

Input

4, 2, 5, 9, 7, 6, 10, 3, 1

Output

4, 5, 9, 7, 6, 3, 1





Longest Bitonic Subsequence



Explore | Expand | Enrich

Introduction

For sequences sorted in increasing or decreasing order, the output is the same as the input sequence, i.e.,

- [1, 2, 3, 4, 5] —> [1, 2, 3, 4, 5]
[5, 4, 3, 2, 1] —> [5, 4, 3, 2, 1]

Unlike subarrays, subsequences are not required to occupy consecutive positions within the original array





Longest Bitonic Subsequence



Solution

The idea is to maintain two arrays, $I[]$ and $D[]$

- $I[i]$ store the length of the longest increasing subsequence, ending at $\text{nums}[i]$
- $D[i]$ stores the length of the longest decreasing subsequence, starting from $\text{nums}[i]$
-

Finally, the length of longest bitonic subsequence is maximum among all $I[i] + D[i] - 1$

For example, consider the sequence [4, 2, 5, 9, 7, 6, 10, 3, 1]



Longest Bitonic Subsequence



Solution

The contents of the LIS and DIS array, which have been discussed previously, are as follows

	I[i]	D[i]	
(i = 0)	1	3	
(i = 1)	1	2	
(i = 2)	2	3	
(i = 3)	3	5	
(i = 4)	3	4	
(i = 5)	3	3	
(i = 6)	4	3	
(i = 7)	2	3	
(i = 8)	1	1	

Longest Bitonic Subsequence



Solution

The longest bitonic subsequence length is 7

The contents of the longest bitonic sequence is as follows [4, 5, 9, 7, 6, 3, 1]

The longest bitonic subsequence is formed by $I[3] + D[3] - 1$

	I[i]	D[i]	
(i = 0)	1	3	
(i = 1)	1	2	
(i = 2)	2	3	
(i = 3)	3	5	
(i = 4)	3	4	
(i = 5)	3	3	
(i = 6)	4	3	
(i = 7)	2	3	
(i = 8)	1	1	



Longest Bitonic Subsequence



Program

lbs1.java

Sample IO

Input

4, 2, 5, 9, 7, 6, 10, 3, 1

•

Output

7



Longest Bitonic Subsequence



Printing the subsequence

The idea remains the same, except instead of storing the length of the LIS and LDS, we store LIS and LDS itself.

- For example, consider the sequence [4, 2, 5, 9, 7, 6, 10, 3, 1].

The contents of the LIS and LDS list shown to the right

	I[i]	D[i]
(i = 0)	4	4 3 1
(i = 1)	2	2 1
(i = 2)	4 5	5 3 1
(i = 3)	4 5 9	9 7 6 3 1
(i = 4)	4 5 7	7 6 3 1
(i = 5)	4 5 6	6 3 1
(i = 6)	4 5 9 10	10 3 1
(i = 7)	2 3	3 1
(i = 8)	1	1



Longest Bitonic Subsequence



Explore | Expand | Enrich

Program

lbs2.java

Sample IO

Input

4, 2, 5, 9, 7, 6, 10, 3, 1



Output

[4, 5, 9][7, 6, 3, 1]



Introduction

Given a string S, find the common palindromic sequence

Palindromic sequence is a sequence that does not need to be contiguous and is a palindrome, which is common in itself.

- You need to return the length of the longest palindromic subsequence in A.

Unlike substrings, subsequences are not required to occupy consecutive positions within the original string.



Longest Palindromic Subsequence



Example

Consider S = “ABBDCACB”

The length of the longest palindromic subsequence is 5

The longest palindromic subsequence is BCACB

S = “BEBEEED”

The longest common palindromic subsequence is “EEEE”, which has a length of 4



Solution

The idea is to use recursion to solve this problem. The idea is to compare the last character of the string $X[i\dots j]$ with its first character. There are two possibilities

1. If the string's last character is the same as the first character, include the first and last characters in palindrome and recur for the remaining substring $X[i+1, j-1]$.
2. If the last character of the string is different from the first character, return the maximum of the two values we get by
 1. Removing the last character and recursing for the remaining substring $X[i, j-1]$.
 2. Removing the first character and recursing for the remaining substring $X[i+1, j]$.

Longest Palindromic Subsequence



Solution

This yields the following recursive relation to finding the length of the longest repeated subsequence of a sequence X:

$$\text{LPS}[i..j] = \begin{cases} 1 & (\text{if } i = j) \\ \text{LPS}[i+1..j-1] + 2 & (\text{if } X[i] = X[j]) \\ \max(\text{LPS}[i+1..j], \text{LPS}[i..j-1]) & (\text{if } X[i] \neq X[j]) \end{cases}$$



Longest Palindromic Subsequence



Explore | Expand | Enrich

Program

lps1.java

Sample IO

Input

ABBDCCACB



Output

5

Time Complexity

$O(2^n)$, where n is the length of the input string





Longest Palindromic Subsequence



Explore | Expand | Enrich

Improvement

In the previous attempt, the worst case happens when there is no repeated character present in X (i.e., LPS length is 1)

Also each recursive call will end up in two recursive calls. This also requires additional space for the call stack.

However, the LPS problem has an optimal substructure and also exhibits overlapping subproblems.

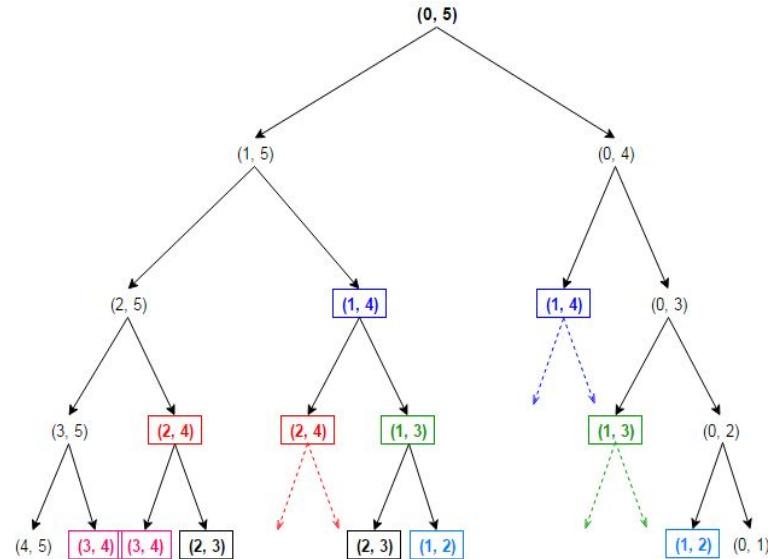


Longest Palindromic Subsequence



Improvement

Let's consider the recursion tree for a sequence of length 6 having all distinct characters, say ABCDEF, whose LPS length is 1.

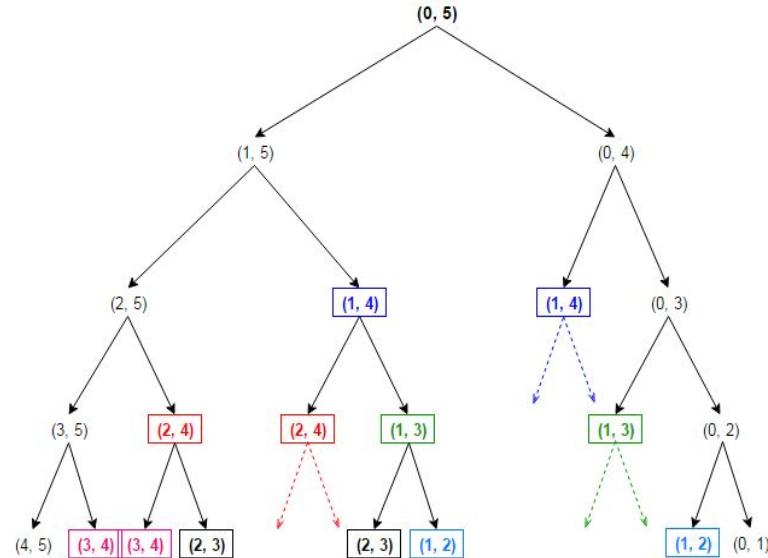


Longest Palindromic Subsequence

Improvement

As we can see, the same subproblems (highlighted in the same color) are getting computed repeatedly. We know that problems with

- optimal substructure and overlapping subproblems can be solved by dynamic programming, where subproblem solutions are memoized rather than computed and again.1.





Longest Palindromic Subsequence



Explore | Expand | Enrich

Program

lps2.java

Sample IO

Input

ABBDCCACB



Output

5

Time Complexity

$O(n^*n)$, where n is the length of the input string



Printing the subsequence

The discussed methods only find the longest palindromic subsequence length but does not print the longest palindromic subsequence itself

The longest palindromic subsequence problem is a classic variation of the Longest Common Subsequence (LCS) problem

The idea is to find LCS of the given string with its reverse

Call $\text{LCS}(X, \text{reverse}(X))$ and the longest common subsequence will be the longest palindromic subsequence.



Longest Palindromic Subsequence



Explore | Expand | Enrich

Program

Ips3.java

Sample IO

Input

ABBCDCACB

Output

5

BCACB

Time Complexity

$O(n^*n)$, where n is the length of the input string





Subset sum problem



Explore | Expand | Enrich

Introduction

Given an integer array A of size N.

You are also given an integer B, you need to find whether there exist a subset in A whose sum equal B.

If there exist a subset then return true else return false.





Subset sum problem



Explore | Expand | Enrich

Examples

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True

Explanation: There is a subset (4, 5) with sum 9.



Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30

Output: False

Explanation: There is no subset that add up to 30.





Subset sum problem



Explore | Expand | Enrich

Approach: Recursion

For the recursive approach we will consider two cases.

Consider the last element and now the required sum = target sum – value of ‘last’ element and number of elements = total elements – 1



Leave the ‘last’ element and now the required sum = target sum and number of elements = total elements – 1





Subset sum problem



Explore | Expand | Enrich

Approach: Recursion

isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum)||isSubsetSum(set, n-1, sum-set[n-1])

- Base Cases

$\text{isSubsetSum}(\text{set}, \text{n}, \text{sum}) = \text{false}$, if $\text{sum} > 0$ and $\text{n} == 0$

$\text{isSubsetSum}(\text{set}, \text{n}, \text{sum}) = \text{true}$, if $\text{sum} == 0$



Subset sum problem

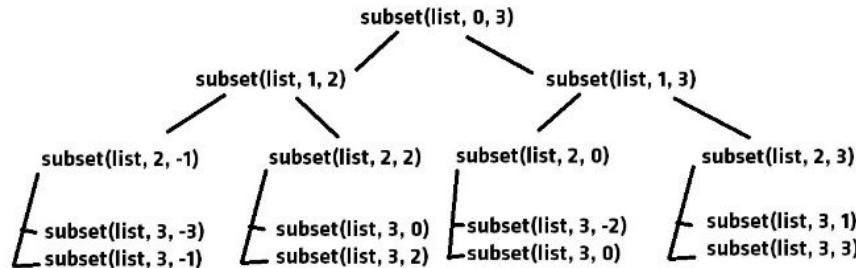
Approach: Recursion

```
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1])
```

Base Cases

isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0

isSubsetSum(set, n, sum) = true, if sum == 0





Subset sum problem



Explore | Expand | Enrich

Program

subsetsum1.java

Sample IO

Input

3, 34, 4, 12, 5, 2

9

Output

true

Time Complexity

Exponential





Subset sum problem



Explore | Expand | Enrich

Optimization

The problem has an optimal substructure.

That means the problem can be broken down into smaller, simple “subproblems”, which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial.

- The above solution also exhibits overlapping subproblems. If we draw the solution’s recursion tree, we can see that the same subproblems are getting computed repeatedly.

We know that problems with optimal substructure and overlapping subproblems can be solved using dynamic programming, where subproblem solutions are memoized rather than computed again and again.





Subset sum problem



Explore | Expand | Enrich

Program

subsetsum2.java

Sample IO

Input

7, 3, 2, 5, 8

14

Output

true

Time Complexity

$O(n \times \text{sum})$, where n is the size of the input and sum is the sum of all elements in the input





Subset sum problem



Bottom Up Approach

We can also solve this problem in a bottom-up manner.

In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them.

- The following bottom-up approach computes $T[i][j]$, for each $1 \leq i \leq n$ and $1 \leq j \leq \text{sum}$, which is true if subset with sum j can be found using items up to first i items.

It uses the value of smaller values i and j already computed.

It has the same asymptotic runtime as Memoization but no recursion overhead.



Subset sum problem



Explore | Expand | Enrich

Program

subsetsum3.java

Sample IO

Input

7, 3, 2, 5, 8

18

Output

true

Time Complexity

$O(n \times \text{sum})$, where n is the size of the input and sum is the sum of all elements in the input



Introduction

Fractional Knapsack problem is also called 0-1 Knapsack problem which is solved using Greedy Approach

This is not to be confused with 0/1 Knapsack problem for which the solution involves dynamic programming approach

- Given a set of N items each having value V with weight W and the total capacity of a knapsack.

The task is to find the maximal value of fractions of items that can fit into the knapsack.



Fractional Knapsack Problem



Examples

Input:

Items as (value, weight) pairs

A[] = {{60, 10}, {100, 20}, {120, 30}}

Total_capacity = 50;

- #### Output:

240

Explanation

Maximum possible value = 240 by taking items of weight 10 and 20 kg and 2/3 fraction of 30 kg. Hence total price will be $60+100+(2/3)(120) = 240$



Fractional Knapsack Problem



Examples

Input: A[] = {{60, 20}, {100, 50}, {120, 30}}, Total_capacity = 50

Output: 180.00

Explanation: Take the first item and the third item. Total value = $60 + 120 = 180$ with a total capacity of $20 + 30 = 50$

Input: A[] = {{500, 30}}, Total_capacity = 10

Output: 166.67

Explanation: Since the total capacity of the knapsack is 10, consider 1/3rd of the item

Brute Force Approach

The most basic approach is to try all possible subsets and possible fractions of the given set and find the maximum value among all such fractions.

- The time complexity will be exponential, as you need to find all possible combinations of the given set.
- 

Greedy Approach

In Fractional Knapsack, we can break items for maximizing the total value of knapsack

The Fractional Knapsack problem can be solved efficiently using the greedy algorithm, where you need to sort the items according to their value/weight ratio

-

Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can

Fractional Knapsack Problem

Greedy Approach



Value	Weights	Value/Weight
60	20	$60/20 = 3$
100	50	$100/50 = 2$
120	30	$120/30 = 4$

After sorting with respect to
Value/Weight

Value	Weights	Value/Weight
120	30	4
60	20	3
100	50	2

Fractional Knapsack Problem



Greedy Approach - Algorithm

Total capacity = 50

Value	Weights (W)	X	W * X	Curr_Cap	Max_Val
120	30	1	$30 * 1 = 30$	$50 - 30 = 20$	120
60	20	1	$20 * 1 = 20$	$20 - 20 = 0$	60
100	50	0	Knapsack filled	0	0

Tot_val =
 $120 + 60 = 180$



Fractional Knapsack Problem



Explore | Expand | Enrich

Program

fracknapsack.java

Sample IO

Input

wt ={10, 40, 20, 30}

val = {60, 40, 100, 120}

capacity = 50

Output

240

Time Complexity

$O(n \log n)$



Introduction

In 0/1 or 0–1 Knapsack problem, we CANNOT break items for maximizing the total value of knapsack

- In the 0–1 Knapsack problem, we are given a set of items, each with a weight and a value, and we need to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Introduction

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In other words, given two integer arrays $\text{val}[0..n-1]$ and $\text{wt}[0..n-1]$ which represent values and weights associated with n items respectively.

Also given an integer W which represents knapsack capacity, find out the maximum value subset of $\text{val}[]$ such that sum of the weights of this subset is smaller than or equal to W .

You cannot break an item, either pick the complete item or don't pick it (0-1 property)



0/1 Knapsack Problem



Explore | Expand | Enrich

Examples

```
value[] = {60, 100, 120};  
weight[] = {10, 20, 30};  
W = 50;
```

- Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50



0/1 Knapsack Problem



Examples

Input

value = [20, 5, 10, 40, 15, 25]

weight = [1, 2, 3, 8, 7, 4]

W = 10

- Output

60



Brute Force approach

The idea is to use recursion to solve this problem. For each item, there are two possibilities:

1. Include the current item in the knapsack and recur for remaining items with knapsack's decreased capacity. If the capacity becomes negative, do not recur or return -INFINITY.
2. Exclude the current item from the knapsack and recur for the remaining items.

Finally, return the maximum value we get by including or excluding the current item. The base case of the recursion would be when no items are left, or capacity becomes 0.



Program

01knapsack1.java

Sample IO

Input

wt = {1, 2, 3, 8, 7, 4}

val = {20, 5, 10, 40, 15, 25}

capacity = 10

Output

60

Time Complexity

Exponential

DP approach

The above solution has an optimal substructure, i.e., the optimal solution can be constructed efficiently from optimal solutions of its subproblem as shown below.

- Case 1: The item is included in the optimal subset.
- Case 2: The item is not included in the optimal set.

It also has overlapping subproblems, i.e., the problem can be broken down into subproblems, and each subproblem is repeated several times.

To reuse the subproblem solutions, we can apply dynamic programming, in which subproblem solutions are memoized rather than computed over and over again.



DP approach

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

- Maximum value obtained by $n-1$ items and W weight (excluding n th item).
- Value of n th item plus maximum value obtained by $n-1$ items and W minus the weight of the n th item (including n th item).

If the weight of n th item is greater than ‘ W ’, then the n th item cannot be included



Program

01knapsack2.java

Sample IO

Input

wt = {1, 2, 3, 8, 7, 4}

val = {20, 5, 10, 40, 15, 25}

capacity = 10

Output

60

Time and Space Complexity

$O(n \cdot W)$, n is the total number of items in the input and W is the knapsack's capacity

0/1 Knapsack Problem



Another bottom up approach

$V[i, w]$	w=0	1	2	3	W
i = 0	0	0	0	0	0
1							
2							
:							
n							

bottom

up

Another bottom up approach

First row and the first column are always zero.

Why? If you have no items, you can not achieve any value. If your knapsack size is zero, you can not take any items.

All other fields will be filled up based on the formula below

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

0/1 Knapsack Problem



Another bottom up approach

Example

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Another bottom up approach

Example

In the second row

[1,0] -> 0 Maximum value should be zero since knapsack size is 0.

[1,1] -> 0 Max value should 0 since knapsack size is 1 but first items weight is 5. You can not fit 5 in a 1kg bag.

[1,2],[1,3],[1,4] -> 0 should be above same reason.

[1,5] -> 10 because item weight is 5 and knapsack size is also 5.

[1,6] -> Now your knapsack size is 6, take your first item 5kg. Now you have 1kg available space in the knapsack. Now check what is the maximum gain for 1kg space([0,1]=0). Now 10+0=0.

.

[1,10] -> Now your knapsack size is 10, take your first item 5kg. Now you have 5kg available space in the knapsack. Now check what is the maximum gain for 1kg space([0,5]=0). Now 10+0=0.

Program

01knapsack3.java

Sample IO

Input

wt = {5, 4, 6, 3}

val = {10, 40, 30, 50}

W = 10

Output

90

Time and Space Complexity

O(n.W), n is the total number of items in the input and W is the knapsack's capacity



Coin Change problem



Introduction

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If

- that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.





Coin Change problem



Explore | Expand | Enrich

Examples

Input: $S = \{ 1, 3, 5, 7 \}$, target = 8

The total number of ways is 6

- { 1, 7 }
- { 3, 5 }
- { 1, 1, 3, 3 }
- { 1, 1, 1, 5 }
- { 1, 1, 1, 1, 1, 3 }
- { 1, 1, 1, 1, 1, 1, 1, 1 }

Input: $S = \{ 1, 2, 3 \}$, target = 4

The total number of ways is 4

- { 1, 3 }
- { 2, 2 }
- { 1, 1, 2 }
- { 1, 1, 1, 1 }

Brute Force approach

The idea is to use recursion to solve this problem.

We recur to see if the total can be reached by choosing the coin or not for each coin of given denominations.

- If choosing the current coin results in the solution, update the total number of ways.

Program

coinchange1.java

Sample IO

Input

1, 2, 3

4

Output

4

Explanation

The first line of input is the coins of the given denominations. The second line is the total change required. This approach has exponential space complexity

DP Approach

Optimal Substructure

To count the total number of solutions, we can divide all set solutions into two sets.

- 1) Solutions that do not contain mth coin (or S_m).
- 2) Solutions that contain at least one S_m .

Let $\text{count}(S[], m, n)$ be the function to count the number of solutions, then it can be written as sum of $\text{count}(S[], m-1, n)$ and $\text{count}(S[], m, n-S_m)$.

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.



DP Approach

Here each coin of a given denomination can come an infinite number of times.
(Repetition allowed), this is called **Unbounded Knapsack**.

We have 2 choices for a coin of a particular denomination, either

- to include, or
- to exclude

But here, the inclusion process is not for just once; we can include any denomination any number of times until $N < 0$



DP Approach

If we are at $s[m-1]$, we can take as many instances of that coin (unbounded inclusion) i.e. $\text{count}(S, m, n - S[m-1])$; then we move to $s[m-2]$.

After moving to $s[m-2]$, we can't move back and can't make choices for $s[m-1]$ i.e $\text{count}(S, m-1, n)$.

Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e $\text{count}(S, m, n - S[m-1]) + \text{count}(S, m-1, n)$; which will be our required answer



Coin Change Problem



Program

coinchange2.java

Sample IO

Input

1, 2, 3

4

Output

4

Time and Space Complexity

$O(mn)$ and $O(n)$ respectively



Program

coinchange3.java

This is a simplified version of the previous approach

Sample IO

Input

1, 2, 3

4

Output

4

Time and Space Complexity

$O(mn)$ and $O(n)$ respectively

Program

coinchange4.java

This is a top-down approach of dynamic programming

Sample IO

Input

- 1, 2, 3
- 4

Output

4

Time and Space Complexity

$O(mn)$ and $O(mn)$ respectively

Introduction

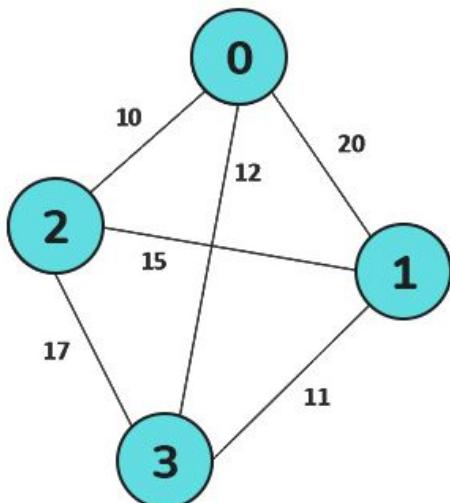
Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point

- The problem is a famous NP hard problem

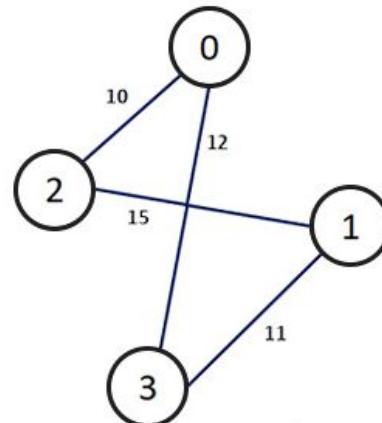
A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP- problem (nondeterministic polynomial time) problem

Travelling Salesman Problem

Example



The Shortest Path
Covering All The Nodes



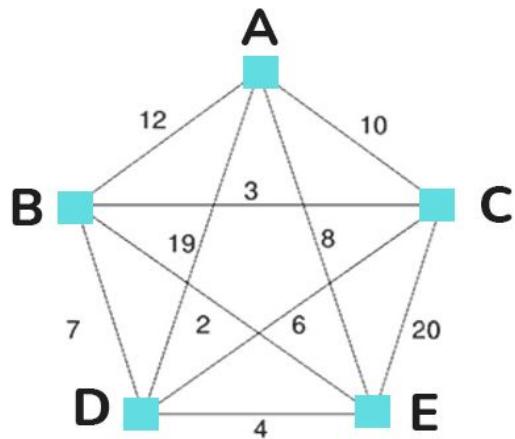


Travelling Salesman Problem



Explore | Expand | Enrich

Example



Minimum weight Hamiltonian Cycle:
 $EACBDE = 32$

Naive Approach

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point
2. Now, we will generate all possible permutations of cities which are $(n-1)!$
3. Find the cost of each permutation and keep track of the minimum cost permutation
4. Return the permutation with minimum cost



Travelling Salesman Problem



Program

tsp1.java

Sample IO

Input

```
 {{0, 10, 15, 20},  
  {10, 0, 35, 25},  
  {15, 35, 0, 30},  
  {20, 25, 30, 0}};
```

Output

80

Time and Space Complexity

$O(N!)$, Where N is the number of cities.

$O(1)$



Using DP

- In this algorithm, we take a subset N of the required cities that need to be visited, the distance among the cities dist, and starting city s as inputs
- Each city is identified by a unique city id which we say like 1,2,3,4,5.....n
- Here we use a dynamic approach to calculate the cost function Cost()
- Using recursive calls, we calculate the cost function for each subset of the original problem
- To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Using DP

- To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems
- We start with all subsets of size 2 and calculate $C(S, i)$ for all subsets where S is the subset, then we calculate $C(S, i)$ for all subsets S of size 3 and so on
- There are at most $O(n2^n)$ subproblems, and each one takes linear time to solve
- The total running time is, therefore, $O(n^22^n)$
- The time complexity is much less than $O(n!)$ but still exponential
- The space required is also exponential





Travelling Salesman Problem



Program

tsp2.java

Sample IO

Input

```
0, 0, 0, 8, 0, 0  
0, 0, 0, 0, 0, 12  
0, 0, 0, 0, 4, 0  
0, 0, 6, 0, 0, 0  
0, 2, 0, 0, 0, 0  
10, 0, 0, 0, 0, 0
```



Output

Tour: [0, 3, 2, 4, 1, 5, 0]

Tour cost: 42.0

Time Complexity

$O(N^{22^N})$



Introduction

The Shortest Common Supersequence (SCS) is finding the shortest supersequence Z of given sequences X and Y such that both X and Y are subsequences of Z

The problem differs from the problem of finding the shortest common substring

- Unlike substrings, subsequences are not required to occupy consecutive positions within the original string.



Shortest Common Supersequence



Example

For example, consider the two following sequences, X and Y:

X: ABCBDAB

Y: BDCABA

- The length of the SCS is 9
The SCS are ABCBDCABA, ABDCABDAB, and ABDCBDABA

Approach

- 1) Find Longest Common Subsequence (lcs) of two given strings. For example, lcs of “geek” and “eke” is “ek”
- 2) Insert non-lcs characters (in their original order in strings) to the lcs found above, and return the result. So “ek” becomes “geeke” which is shortest common supersequence

Let us consider another example, str1 = “AGGTAB” and str2 = “GXTXAYB”. LCS of str1 and str2 is “GTAB”. Once we find LCS, we insert characters of both strings in order and we get “AGXGTAXAYB”

Approach

- 1) Find Longest Common Subsequence (lcs) of two given strings.
- 2) Let us consider str1 = “AGGTAB” and str2 = “GXTXAYB”
- 3) In our example example, lcs of str1 and str2 is “GTAB”
 -
- 4) Once we find LCS, we insert non-LCS characters (in their original order in strings) to the LCS found above, and return the result.
- 5) So finally, we get “AGXGTAXAYB” as the answer

How does this work?

We need to find a string that has both strings as subsequences and is shortest such string.

If both strings have all characters different, then result is sum of lengths of two given strings.

If there are common characters, then we don't want them multiple times as the task is to minimize length.

Therefore, we first find the longest common subsequence, take one occurrence of this subsequence and add extra characters.



Shortest Common Supersequence



Explore | Expand | Enrich

How does this work?

Length of the shortest supersequence = (Sum of lengths of given two strings) - (Length of LCS of two given strings)





Shortest Common Supersequence



Explore | Expand | Enrich

Program

shortestcommonsuperseq1.java

Sample IO

Input

AGGTAB

GXTXAYB

Output

9



Another Approach

```
1 /* Let X[0..m - 1] and Y[0..n - 1] be two strings
2 and m and n be respective lengths*/
3
4 if(m == 0) return n;
5 if (n == 0) return m;
6
7 /* If last characters are same, then
8 add 1 to result and recur for X[] */
9 if (X[m - 1] == Y[n - 1])
10    return 1 + SCS(X, Y, m - 1, n - 1);
11
12 /* Else find shortest of following two
13     a) Remove last character from X and recur
14     b) Remove last character from Y and recur
15 */
16 else
17    return 1 + min(SCS(X, Y, m - 1, n), SCS(X, Y, m, n - 1));
```



Shortest Common Supersequence



Explore | Expand | Enrich

Program

shortestcommonsuperseq2.java

Sample IO

Input

AGGTAB

GXTXAYB

Output

9

Time Complexity

$O(2^{\min(m, n)})$

Here min is a function which returns the minimum of m and n

Since there are overlapping subproblems, we can efficiently solve this recursive problem using Dynamic Programming.





Shortest Common Supersequence



Explore | Expand | Enrich

DP Solution - Bottom Up

Since there are overlapping subproblems, we can efficiently solve this recursive problem using Dynamic Programming

Time complexity of the dynamic programming solution would reduce to quadratic i.e. $O(mn)$



Shortest Common Supersequence



Explore | Expand | Enrich

Program

shortestcommonsuperseq3.java

Sample IO

Input

AGGTAB

GXTXAYB

Output

9

Time Complexity

$O(m, n)$

Here min is a function which returns the minimum of m and n

Since there are overlapping subproblems, we can efficiently solve this recursive problem using Dynamic Programming.





Shortest Common Supersequence



Explore | Expand | Enrich

DP Solution - Top Down (Memoization)

The idea is to follow the simple recursive solution, use a lookup table to avoid recomputations.

- Before computing result for an input, we check if the result is already computed or not.

If already computed, we return that result.





Shortest Common Supersequence



Explore | Expand | Enrich

Program

shortestcommonsuperseq4.java

Sample IO

Input

AGGTAB

GXTXAYB

Output

9





Levenshtein Distance problem



Explore | Expand | Enrich

Introduction

This problem is also called the ‘Edit Distance Problem’

Given two strings A and B, your task is to find the minimum number of steps

- required to convert A to B

In this problem, each operation is counted as 1 step



Introduction - Permitted Operations

- Insert a character
- Delete a character
- Replace a character



Levenshtein Distance problem



Examples

Input: A = “abad”, B = “abac”

Output: 1

Explanation: Operation 1: Replace d with c.



Input: A = “horse”, B = “ro”

Output: 3

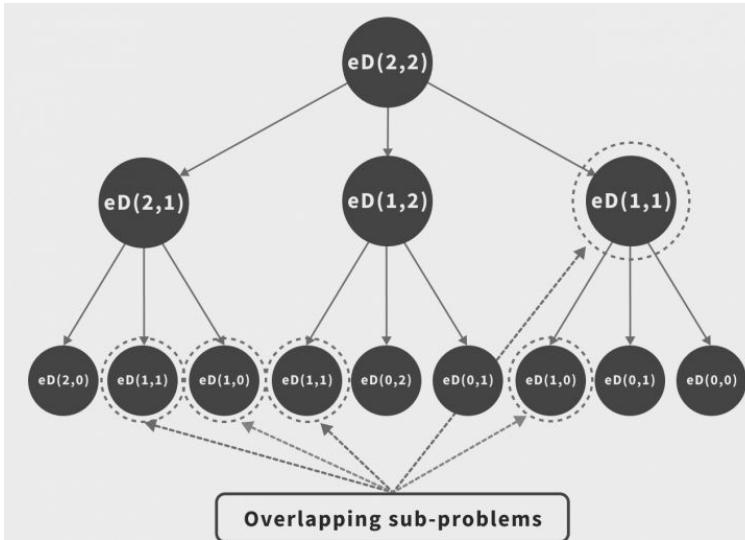


Levenshtein Distance problem

Approach: Using Recursion

The idea is to use a recursive approach to solve the problem.

All the characters of both the strings are traversed one by one either from the left or the right end and apply the given operations.



Algorithm: Using Recursion

- Consider two pointers i and j pointing the given string A and B
- If the current character, pointing in both the strings are same, then no changes are to be made. Therefore, recurse for lengths $i + 1$ and $j + 1$
- Otherwise, try to apply the free operations provided
 - Each of the given operations would cause 1 units
 - The character pointer i points to is $A[i]$ and pointer j is $B[j]$. Therefore, $F(i, j)$ is the edit distance of $A(0, i)$ and $B(0, j)$
 - For insertion: Recurse $i - 1$ to j
 - For deletion: Recurse i to $j - 1$
 - For replacement: Recurse $i - 1$ to $j - 1$
- After applying all the operations, $f(i, j) = 1 + \min(f(i - 1, j), f(i, j - 1), f(i - 1, j - 1))$



Shortest Common Supersequence



Explore | Expand | Enrich

Program

editdistance1.java

Sample IO

Input

abac

abad

Output

1

Time Complexity

$O(3^{(N * M)})$, where N and M is the length of the first and second string.

Space Complexity

$O(N + M)$, where N and M is the length of the first and second string.





Levenshtein Distance problem



Approach: Using DP

The idea is to use a dynamic programming approach here.

The tabulation method is the most efficient method to solve this problem.

A diagram showing the prefix "HORSE" enclosed in a circle, representing the string $\text{word1}[1..i]$.

word1[1..i]

A diagram showing the prefix "ROS" enclosed in a circle, representing the string $\text{word2}[1..j]$.

word2[1..j]

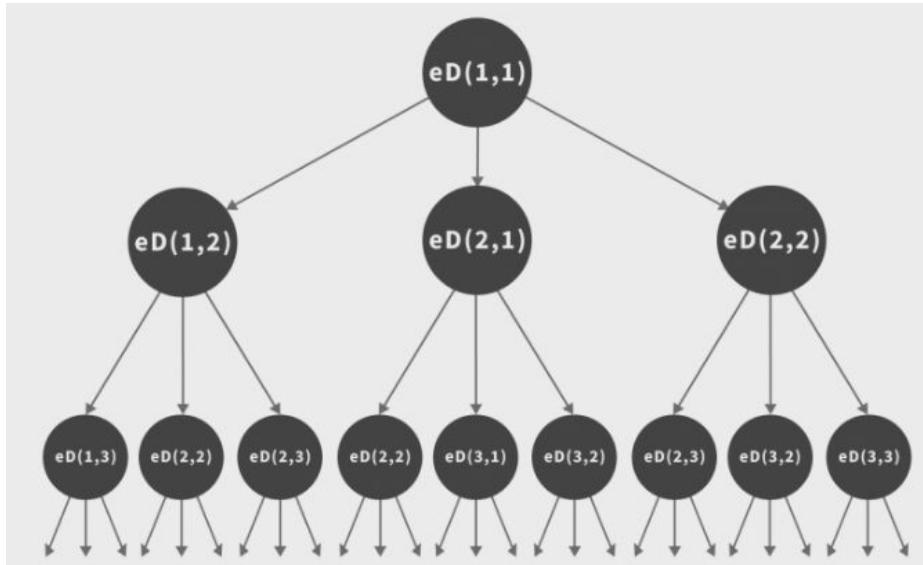
$D[i][j] = \text{the edit distance between } \text{word1}[1..i] \text{ and } \text{word2}[1..j]$
i. e. between HOR and RO



Levenshtein Distance problem

Approach: Using DP

Since the problem has overlapping subproblems, many of the calculations are repeated.



Levenshtein Distance problem



Approach: Using DP

Initialise a 2D dp, where $dp[i][j]$ denotes the edit distance of the length $(i + 1)$ th of A and $(j + 1)$ th length of B

$$dp[i][j] = \begin{cases} dp[i-1], & \text{if } A[i] == B[j] \\ 1 + \min \left(\begin{array}{l} dp[i-1][j], \\ dp[i][j-1], \\ dp[i-1][j-1] \end{array} \right) & \text{if } A[i] \neq B[j] \end{cases}$$

If current character of both the strings are same: $dp[i][j] = dp[i - 1][j - 1]$

If current character of both the strings are different: $dp[i][j] = 1 + \min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1])$



Levenshtein Distance problem



Approach: Using DP

This example dry runs all the possible edit distances of the two words HORSE and ROS.

HORSE

word1[1..i]

RO**S**

word2[1..j]

$D[i][j] = ?$ the edit distance between HOR and RO

$D[i][j] = 1 + \min(D[i-1][j], D[i][j-1], D[i-1][j-1])$, since R!=O

$D[i-1][j] = 1$ the edit distance between HO and RO

$D[i][j-1] = 2$ the edit distance between HOR and R

$D[i-1][j-1] = 2$ the edit distance between HO and R

$D[i][j] = 2$



Levenshtein Distance problem



Approach: Using DP

The Tabulations is done as follows

	#	I	N	T	E	N	T	I	O	N	
#	0										
E	1										
X	2										
E	3										
C	4										
U	5										
T	6										
I	7										
O	8										
N	9										

$$D(i,j) = \min \begin{cases} D(i,-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 1; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

	#	I	N	T	E	N	T	I	O	N	
#	0	1	2	3	4	5	6	7	8	9	
E	1	1	2	3	4	5	6	6	7	8	
X	2	2	2	3	4	5	6	7	7	7	
E	3	3	3	3	4	5	5	6	7	8	
C	4	3	4	3	4	5	6	6	7	8	
U	5	4	4	4	4	5	6	7	7	7	
T	6	5	5	5	5	5	5	6	7	8	
I	7	6	6	6	6	6	6	5	6	7	
O	8	7	7	7	7	7	7	6	5	6	
N	9	8	8	8	8	8	8	7	6	5	



Levenshtein Distance problem



Explore | Expand | Enrich

Program

editdistance2.java

Sample IO

Input

abac

abad

Output

1

Time Complexity

$O(N * M)$, where N and M is the length of the first and second string.



Introduction

“Given a rod of length n and a list of rod prices of length i , where $1 \leq i \leq n$, find the optimal way to cut the rod into smaller rods to maximize profit”

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n

The problem statement is to determine the maximum value obtainable by cutting up the rod and selling the pieces

Rod Cutting Problem



Example

Input

length[] = [1, 2, 3, 4]

price[] = [1, 5, 8, 9]

Rod length: 4

Output

10

Cut	Profit
4	9
1, 3	(1 + 8) = 9
2, 2	(5 + 5) = 10
3, 1	(8 + 1) = 9
1, 1, 2	(1 + 1 + 5) = 7
1, 2, 1	(1 + 5 + 1) = 7
2, 1, 1	(5 + 1 + 1) = 7
1, 1, 1, 1	(1 + 1 + 1 + 1) = 4

Explanation

Cut the rod into two pieces of length 2 each to gain revenue of $5 + 5 = 10$

The possibilities are given in the image

Brute Force Solution

We are given an array $\text{price}[]$, where the rod of length i has a value $\text{price}[i-1]$

The idea is simple – one by one, partition the given rod of length n into two parts: i and $n-i$

- Recur for the rod of length $n-i$ but don't divide the rod of length i any further

Finally, take the maximum of all values

The recursive relation for the same is given in the next slide



Rod Cutting Problem



Brute Force Solution

$$\text{rodcut}(n) = \max \{ \text{price}[i - 1] + \text{rodCut}(n - i) \} \text{ where } 1 \leq i \leq n$$



Rod Cutting Problem



Explore | Expand | Enrich

Program

rodcutting1.java

Sample IO

Input

1, 5, 8, 9, 10, 17, 17, 20

8

Output

22

Time Complexity

$O(n^n)$, where n is the length of the rod



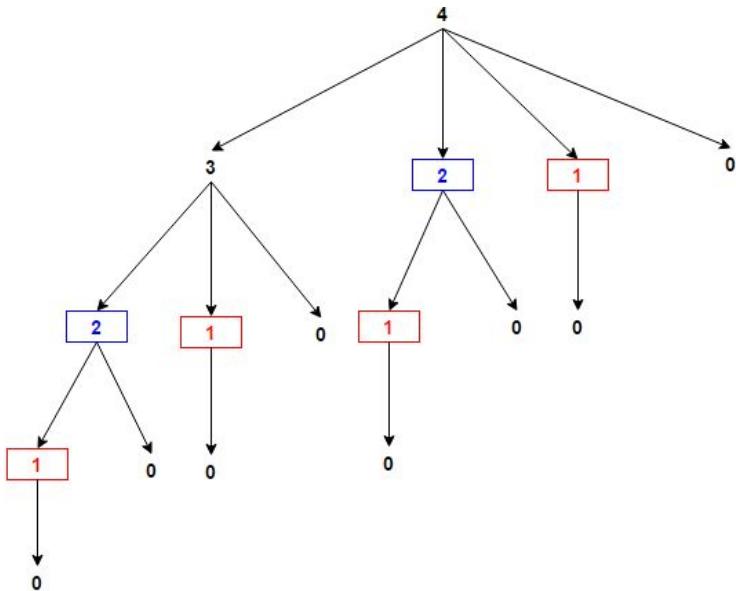
Rod Cutting Problem

Optimizing the solution

The problem has optimal substructure as shown in the figure

This recursion tree is for a rod length of 4

The highlighted boxes represent repeated computation





Rod Cutting Problem



Explore | Expand | Enrich

Optimizing the solution

We know that problems with optimal substructure and overlapping subproblems can be solved by dynamic programming

Subproblem solutions are memoized rather than computed and again

- We will solve this problem in a bottom-up manner.

In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them.



Optimizing the solution

Our bottom-up approach computes $T[i]$, which stores maximum profit achieved from the rod of length i for each $1 \leq i \leq n$

It uses the value of smaller values i already computed

The time complexity of the above bottom-up solution is $O(n^2)$ and requires $O(n)$ extra space, where n is the rod length



Rod Cutting Problem



Explore | Expand | Enrich

Program

rodcutting2.java

Sample IO

Input

1, 5, 8, 9, 10, 17, 17, 20

8

Output

22

Time Complexity

$O(n^2)$, where n is the length of the rod

Space Complexity

Linear



Introduction

Given a string and a pattern containing wildcard characters: * and ?

- ? can match to any single character in the string
- * can match to any number of characters including zero characters

Your task is to design an efficient algorithm to check if the pattern matches with the complete string or not

Wildcard Pattern Matching



Examples

If the text is **baaabab**, then

Wildcard *******ba*****ab** matches with it (* represents any no. of characters including no characters)

- Wildcard pattern **baaa?ab** matches the text (? represents a single character)

Wildcard pattern **ba*a?** matches the text

However, pattern **a*ab** does not match with the text

Input = ba aab ab
Pattern = *****ba*****ab
Output : true

No matching text
Input = baaabab
Pattern = a * ab
Output : false

Input = ba aab ab
Pattern = ba * a?
Output : true

Examples

Input: string = “xyxzzxy”, pattern = “x***y”

Output: Match

Input: string = “xyxzzxy”, pattern = “x***x”

Output: No Match

Input: string = “xyxzzxy”, pattern = “x***x?”

Output: Match

Input: string = “xyxzzxy”, pattern = “*”

Output: Match

Solution

If we carefully analyze the problem, we can see that it can easily be further divided into subproblems.

Top-down solution

For a given pattern[0...m] and word[0...n],

- If pattern[m] == '*', if '*' matches the current character in the input string, move to the next character in the string; otherwise, ignore '*' and move to the next character in the pattern.
- If pattern[m] == '?', ignore current characters of both string and pattern and check if pattern[0...m-1] matches word[0...n-1].
- If the current character in the pattern is not a wildcard character, it should match the current character in the input string.

Top-down solution

Special care has to be taken to handle base conditions:

- If both the input string and pattern reach their end, return true
- If the only pattern reaches its end, return false
- If only the input string reaches its end, return true only if the remaining characters in the pattern are all *



Wildcard Pattern Matching



Explore | Expand | Enrich

Program

wildcard1.java

Sample IO

Input

word[] = "xyxzzxy"
pattern[] = "x***x?"

Output

Match

Time Complexity

$O(m.n)$, where n is the length of the text and m is the length of the pattern.

Space Complexity

$O(m.n)$, where n is the length of the text and m is the length of the pattern.





Wildcard Pattern Matching



Program

wildcard2.java

Iterative version

Sample IO

Input

- word[] = "xyxzzxy"
pattern[] = "x***x?"

Output

Match

Time Complexity

$O(m.n)$, where n is the length of the text and m is the length of the pattern.

Space Complexity

$O(m.n)$, where n is the length of the text and m is the length of the pattern.



Introduction

There are two players, A and B, in Pots of gold game, and pots of gold arranged in a line, each containing some gold coins

The players can see how many coins are there in each gold pot, and

- each player gets alternating turns in which the player can pick a pot from either end of the line

The winner is the player who has a higher number of coins at the end.

The objective is to “maximize” the number of coins collected by A, assuming B also plays “optimally”, and A starts the game

Introduction

There are two players, A and B, in Pots of gold game, and pots of gold arranged in a line, each containing some gold coins

The players can see how many coins are there in each gold pot, and each player gets alternating turns in which the player can pick a pot

- from either end of the line

The winner is the player who has a higher number of coins at the end

The objective is to “maximize” the number of coins collected by A, assuming B also plays “optimally”, and A starts the game





Pots of gold game



Explore | Expand | Enrich

Examples

Player	Opponent
4, 6, 2, 3	3
4, 6, 2	4
6, 2	6
2	2
9 coins	6 coins





Pots of gold game



Examples

Player	Opponent
6, 1, 4, 9, 8, 5	6
1, 4, 9, 8, 5	5
1, 4, 9, 8	8
1, 4, 9	9
1, 4	4
1	1
18 coins	15 coins



Pots of gold game



Explore | Expand | Enrich

Optimal Strategy

The idea is to find an optimal strategy that makes the player wins, knowing that an opponent is playing optimally.

The player has two choices for $\text{coin}[i\dots j]$, where i and j denote the front and rear of the line, respectively.



Optimal Strategy

1. If the player chooses the front pot i , the opponent is left to choose from $[i+1, j]$.

If the opponent chooses front pot $i+1$, recur for $[i+2, j]$.

If the opponent chooses rear pot j , recur for $[i+1, j-1]$.

• 2. If the player chooses rear pot j , the opponent is left to choose from $[i, j-1]$.

If the opponent chooses front pot i , recur for $[i+1, j-1]$.

If the opponent chooses rear pot $j-1$, recur for $[i, j-2]$.

Since the opponent is playing optimally, he will try to minimize the player's points, i.e., the opponent will make a choice that will leave the player with minimum coins.





Pots of gold game



Explore | Expand | Enrich

Optimal Strategy

Hence the recursive definition of the problem is

```

| coin[i]           (if i = j)
optimalStrategy(i, j) = | max(coin[i], coin[j])   (if i + 1 = j)
| max (coin[i] + min(optimalStrategy(coin, i + 2, j),
                     optimalStrategy(coin, i + 1, j - 1)),
    coin[j] + min(optimalStrategy(coin, i + 1, j - 1),
                  optimalStrategy(coin, i, j - 2)))

```



Pots of gold game



Explore | Expand | Enrich

Program

potsofgold1.java

Sample IO

Input

```
coin = { 4, 6, 2, 3 };
```

Output

9





Better Solution

The time complexity of the current solution is exponential. Hence we need to find a better solution

The problem has **optimal substructure**. We have seen that the problem can be broken down into smaller subproblems, which can further be broken down into yet smaller subproblems, and so on.

The problem also exhibits **overlapping subproblems**, so we will end up solving the same subproblem over and over again.

Hence we can use DP, where subproblem solutions are memoized rather than computed and again.





Pots of gold game



Explore | Expand | Enrich

Program

potsofgold2.java

Sample IO

Input

```
coin = { 4, 6, 2, 3 };
```

Output

```
9
```

Time and Space Complexity

$O(n^*n)$, where 'n' is the total number of gold pots





Pots of gold game



Better Solution: Bottom-Up version

We can make use of the bottom-up version of the solution, which has a better space complexity when compared to the previous version.



Pots of gold game



Program

potsofgold3.java

Sample IO

Input

```
coin = { 4, 6, 2, 3 };
```

Output

```
9
```

Time Complexity

$O(n*n)$, where 'n' is the total number of gold pots

Space Complexity

$O(n)$, where 'n' is the total number of gold pots





/ethnuscodemithra



Ethnus Codemithra



/ethnus



/code_mithra

THANK YOU

<https://www.codemithra.com/>



Explore | Expand | Enrich



codemithra@ethnus.com



+91 7815 095 095



+91 9019 921 340