# **Algorithm Notes**

All code is essentially an algorithm.

It's a sequence of well-defined instructions to get the computer to do the thing you want it to do.

Understanding the difference between code and an algorithm is a matter of context and implementation.

Programming Algorithms can be defined as concrete implementations using STL functions to solve problems

or blocks of code that perform a series of functions to achieve an output.

Algorithms is also a generic term that defines a series of steps to complete a task (like baking a cake).

Algorithm v Design Patterns

An algorithm can be expressed in code as a concrete sequence of operations that solve a specific problem. s

# STL Algorithms

#include<algorithm>

void AverageHeap()

{

std::vector<int> heap = { 3,2,1,5,6 };

auto fn = [](int& a, int& b) {return a > b; };

make\_heap(heap.begin(), heap.end(), fn);

heap.push\_back(9);

push\_heap(heap.begin(), heap.end(), fn);

pop\_heap(heap.begin(), heap.end(), fn);

heap.pop\_back();

bool bIsHeap = is\_heap(heap.begin(), heap.end(), fn);

(bIsHeap) ? std::cout << "Is a Heap!\n" : std::cout << "Is NOT a Heap!\n";

PrintVector(heap);

}

void BasicHeap()

{

std::vector<int> heap = { 3, 2, 1, 5, 6 };

//6

// Max\_heap tree. Left to right, top to bottom //5 //1

// make\_heap turns vector into a tree data structure (heap) //3 //2

make\_heap(heap.begin(), heap.end());

PrintVector(heap); // {6,5,1,3,2};

// Push

heap.push\_back(9);

push\_heap(heap.begin(), heap.end()); // add 9 to tree and shifts it.

PrintVector(heap); // {9,5,6,3,2,1};

// Pop

pop\_heap(heap.begin(), heap.end());

// printVector(heap); // {6,5,1,3,2,9};

heap.pop\_back(); // back to original

PrintVector(heap); // {6,5,1,3,2};

// Is this container a heap? is\_heap

(is\_heap(heap.begin(), heap.end()))

? std::cout << "This is a Heap!" : std::cout << "This is NOT a heap!";

cout << "\n";

heap.push\_back(92);

heap.push\_back(11);

heap.push\_back(19);

PrintVector(heap); // {6,5,1,3,2,92,11.19}

std::vector<int>::iterator it = is\_heap\_until(heap.begin(), heap.end());

std::cout << "Is Heap until => " << \*it << endl;

}

void Copy()

{

// Copy

std::vector<int> vs = { 1, 2, 3, 4, 5, /\* end(), end()+1 \*/ };

std::vector<int> vs2;

vs2.resize(6); // copy: vs.end() is up to, but not including end(), as end is one past the bounds.

copy(vs.begin(), vs.end() + 1, vs2.begin()); // copy needs to use same type of container.

for (int x : vs2)

cout << x << std::endl;

cout << "\n--------------------------------------\n";

// Copy\_If

std::vector<int> vs3;

vs3.resize(6);

copy\_if(vs.begin(), vs.end(), vs3.begin(), [](int& x) {return x % 2 == 0; });

for (int x : vs3)

cout << x << endl;

cout << endl;

}

void Count()

{

std::vector<int> vs = { 1, 1, 1, 1, 2, 3, 12, 1, 4, 41, 1, 1 };

// count(begin, end, value);

std::cout << "Number of times the element value of 1 appears: "

<< count(vs.begin(), vs.end(), 1) << std::endl;

}

void Count\_If()

{

std::vector<int> vs = { 1, 1, 1, 1, 2, 3, 12, 1, 4, 41, 1, 1 };

// count\_if algorithm using lambda as an inline parameter

cout << count\_if(vs.begin(), vs.end(), [](const int& a) { return a < 10; });

// a is passed in as a vector element.

cout << endl;

// Lambda function can be aware of other things that exist in your function.

// Use a '[]' as a capture list.

int b = 4;

int c = 5; // & captures b + c and all v other variables outside of the lamba function

cout << count\_if(vs.begin(), vs.end(), [&](const int& a) {return a > b + c; }) << "\n";

}

void Find()

{

* The find() function in C++ returns the iterator to the first element matched based on value.
* If the value is not matched, then the function returns the iterator to the last element of the range.
* So, vector, list, ect have a default end() iterator associated with that container.
* The find() function will look for the value, and if not found, the end() iterator is returned.

std::vector<int> vs = { 11, 41, 27, 52, 33 };

// find() returns an iterator

std::vector<int>::iterator it = find(vs.begin(), vs.end(), 11); // Same as vs.begin();

// std::cout << \*it << std::endl; Output: return 11;

std::cout << \*(++it) << std::endl; // Output: //return 41;

if (find(vs.begin(), vs.end(), 34) == vs.end()) // end() is one-past last element, like '/O' of a string.

cout << "Not Found" << endl; // if after searching values, if left with end iterator, not found.

else

cout << "Found" << endl;

}

void ForEach()

{

std::vector<int> vs = { 1, 2, 3, 4, 5 };

for\_each(vs.begin(), vs.end(), [](int& a) { cout << a \* a << endl; }); // no return on this lamba

// how to use an external predictors on this function (userDefinedCondition())?

}

void Generate()

{

std::vector<int> data(10);

generate(data.begin(), data.end(),

[]() { return rand() % 100; });

PrintVector(data);

}

void Lambdas()

{

// Lambda Function | needs #include<functional>

// Functions that are declared inline.

// Uses a function ptr variable to store the lambda function.

// To declare a lambda function, the syntax starts with '[]' brackets, then params.

// lambda function syntax: <return value (param) >

function<bool(int)> cond = [](int a) {return a < 20; };

// Problem: Find a value that is less than 20?

std::vector<int> vs = { 11, 41, 27, 52, 33 };

if (find\_if(vs.begin(), vs.end(), cond) == vs.end()) // if it gets to the end iterator, then its not found

std::cout << "Not Found!" << std::endl;

else

std::cout << "Found!" << std::endl;

}

void MaxMinSort()

{

function<bool(const Student, const Student)> SrtLowestGrade

= [](Student s1, Student s2) { return s1.getGrade() < s2.getGrade(); };

function<bool(const Student, const Student)> SrtGreatestGrade

= [](Student s1, Student s2) { return s1.getGrade() > s2.getGrade(); };

function<bool(const Student, const Student)> SrtByName

= [](Student s1, Student s2) { return s1.getName() < s2.getName(); };

const int size = 5;

Student sArray[size];

sArray[0] = { "John", 20, 'm', 9.0f };

sArray[1] = { "Bob", 21, 'm', 8.0f };

sArray[2] = { "Alice", 19, 'f', 9.2f };

sArray[3] = { "Eve", 20, 'f', 8.5f };

sArray[4] = { "Saldina", 22, 'f', 7.2f };

// MAX\_ELEMENT() & MIN\_ELEMENT()

// Max\_Element & Min\_Element need to have list sorted in ascending order, lowest first.

Student\* maxGradeStudent = max\_element(sArray, sArray + size, SrtGreatestGrade); // func passed in sorts.

std::cout << "Best Student: " << maxGradeStudent->getName() << std::endl;

Student\* minGradeStudent = min\_element(sArray, sArray + size, SrtLowestGrade);

std::cout << "Worst Student: " << minGradeStudent->getName() << std::endl;

for (int i = 0; i < size; i++)

{

if (sArray[i].getName() == "Alice")

sArray[i].setGrade(10.0f);

}

// STUDENT SORTING

std::sort(sArray, sArray + size, SrtLowestGrade); // passing in a func doesn't require '()'

std::cout << "Sorting By Lowest Grade" << std::endl;

PrintStudentArray(sArray, size);

std::sort(sArray, sArray + size, SrtByName); // passing in a func doesn't require '()'

std::cout << "Sorting By Name" << std::endl;

PrintStudentArray(sArray, size);

std::cout << "Sorting By Name In Reverse" << std::endl;

std::reverse(sArray, sArray + 5);

PrintStudentArray(sArray, size);

}

void MergeSort()

{

// Requires that both vector's be pre-sorted or Exception Error!

std::vector<int> a = { 1, 4, 6, 8, 9 };

std::vector<int> b = { 1, 2, 3, 4, 5 };

std::vector<int> c(20);

merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());

for (int x : c)

cout << x << endl;

}

void Mismatch()

{

// Mismatch: using mismatch we can find the first mismatching elements from two containers.

// mismatch(.,.,.,.) returns a pair<iterator, iterator> iterators.

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 }; // {4:6} mismatch from a and b. Also true for 2nd mismatch().

std::vector<int> c = { 1, 2, 3, 3, 7 }; // {5:7} 4>=3, so we skip index[4]. 5>=7 is false, so that's returned.

std::vector<int> d(20);

std::pair<vector<int>::iterator, vector<int>::iterator> pit;

pit = mismatch(a.begin(), a.end(), b.begin(), b.end()); // Output: 4,6

std::cout << \*(pit.first) << "," << \*(pit.second) << "\n---------------\n";

pit = mismatch(a.begin(), a.end(), b.begin(), b.end(), [](int a, int b) {

return a >= b; // a >= b considered match, so we skip. Return pair on FALSE, not true.

}); // Output: 5,7 : a < b is a mismatch, so we return first pair of mismatch. First false condition we return.

std::cout << \*(pit.first) << "," << \*(pit.second) << std::endl;

}

void Move()

{

/\*

std::move() is a function used to convert an lvalue reference into the rvalue reference.

Used to move the resources from a source object for efficient transfer of resources.

The Purpose of a move constructor is to steal or move as many resources

as it can from the source (original) object, as fast as possible.

The source does not need to have a meaningful value anymore,

because it is going to be destroyed in a moment anyway.

std::move avoids unnecessarily creating copies of an object

// des = (string&&)str

// cast str with a string&& rvalue forces move constructor to be called.

\*/

std::vector<int> vs = { 1, 2, 3, 4, 5 };

std::string apple = "Apple";

std::string dest;

std::cout << "Apple: " << apple << std::endl;

std::cout << "Dest: " << dest << std::endl;

std::cout << "Using std::move to move apple into dest: " << std::endl;

dest = std::move(apple); // assignment operator casting rvalue ref

std::cout << "Dest: " << dest << std::endl;

}

void Partition()

{

std::vector<int> data(10);

random\_device rd;

mt19937 rng(rd());

uniform\_int\_distribution<unsigned int> dist(1, 100);

generate(data.begin(), data.end(), [&]() {

return dist(rng);

});

PrintVector(data);

auto it = partition(data.begin(), data.end(), [](int x)

{ return x % 2; });

std::cout << "Partition Point: " << \*it << std::endl;

PrintVector(data);

}

void PartitionPoint()

{

std::vector<int> data(10);

random\_device rd;

mt19937 rng(rd());

uniform\_int\_distribution<unsigned int> dist(1, 100);

generate(data.begin(), data.end(), [&]() {

return dist(rng);

});

auto it = partition(data.begin(), data.end(), [](int x)

{ return x % 2; });

vector<int>::iterator it2 = partition\_point(data.begin(),

data.end(), [](int x) { return x % 2; });

PrintVector(data);

std::cout << "Partition Point: " << \*it2 << std::endl;

}

void NextPermutation()

{

std::cout << "---Next Permutation---\n";

std::vector<int> data = { 1,2,3,4,5,6 };

for (int i = 0; i < 10; i++)

{

next\_permutation(data.begin(), data.end());

PrintVector(data);

}

std::cout << "---Prev Permutation---\n";

for (int i = 0; i < 10; i++)

{

prev\_permutation(data.begin(), data.end());

PrintVector(data);

}

std::cout << "-------Reverse--------\n";

for (int i = 0; i < 10; i++)

{

std::reverse(data.begin(), data.end());

PrintVector(data);

}

}

void NextPermutationStr()

{

std::string str = "umbrella";

std::string strCpy = str;

for (int i = 0; i < str.length(); i++)

{

next\_permutation(str.begin(), str.end());

std::cout << str << endl;

}

}

void PrintStudentArray(Student\* sArr, int arrSize)

{

for (int i = 0; i < arrSize; i++)

std::cout << (\*(sArr + i)).getName() << std::endl; // Need to put '\*' after parentheses (\*sArr).

// cout << (sArr+i)->getName() << endl;

std::cout << "----------\n";

}

void PrintVector(std::vector<int> v)

{

for (int x : v)

{

std::cout << x << ", ";

}

std::cout << "\n";

}

void RandomShuffle\_Algorithm()

{

std::vector<int> vs = { 1, 2, 3, 4, 5 };

random\_shuffle(vs.begin(), vs.end());

for (int x : vs)

cout << x << endl;

}

void RandomGenerate()

{

std::vector<int> data(10);

random\_device rd;

mt19937 rng(rd());

uniform\_int\_distribution<unsigned int> dist(1, 100);

generate(data.begin(), data.end(), [&]() {

return dist(rng);

}

);

PrintVector(data);

}

void Reverse()

{

std::vector<int> vs = { 1, 2, 3, 4, 5 };

PrintVector(vs); // Outputs: 1,2,3,4,5

std::reverse(vs.begin(), vs.end());

PrintVector(vs); // Outputs: 5,4,3,2,1

std::cout << "--------------\n";

std::vector<int> u(5);

PrintVector(u);

reverse\_copy(vs.begin(), vs.end(), u.begin()); // reverse the reversed.

PrintVector(vs); // reversed original: 5,4,3,2,1

PrintVector(u); // Outputs: 1,2,3,4,5

}

void Rotate()

{

// std::rotate

// template <typename FwdIt>

// FwdIt rotate (FwdIt first, FwdIt middle, FwdIt last);

// Time: O(n): shifts elements between first and last,

// such that middle element is now first.

std::vector<int> vs = { 1, 2, 3, 4, 5 };

PrintVector(vs); // Outputs: 1,2,3,4,5

rotate(vs.begin(), vs.begin() + 2, vs.end());

PrintVector(vs); // Outputs: 3,4,5,1,2

std::vector<int> u(5);

PrintVector(u); // Outputs: 0,0,0,0

rotate\_copy(vs.begin(), vs.begin() + 2, vs.end(), u.begin());

PrintVector(u); // Output: 5,1,2,3,4

}

void SetUnion()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 };

std::vector<int> c(10); // set union containers DOES NOT guarantee that duplicates won't be added to new container.

set\_union(a.begin(), a.end(), b.begin(), b.end(), c.begin());

for (int x : c)

cout << x << endl;

}

void SetIntersection()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 };

std::vector<int> c(10); // set union containers DOES NOT guarantee that duplicates won't be added to new container.

set\_intersection(a.begin(), a.end(), b.begin(), b.end(), c.begin());

for (int x : c)

cout << x << endl;

}

void SetSymmetricDifference()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 };

std::vector<int> c(10);

// returns values NOT in a while in B && NOT in b while in a. Not intersected values.

set\_symmetric\_difference(a.begin(), a.end(), b.begin(), b.end(), c.begin()); // different values of a, not in b. So, 4,5 but not 6,7 from b.

for (int x : c)

cout << x << endl;

}

void SetDifference()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 };

std::vector<int> c(10);

set\_difference(a.begin(), a.end(), b.begin(), b.end(), c.begin()); // different values of a, not in b. So, 4,5 but not 6,7 from b.

for (int x : c)

cout << x << endl;

}

void Sort()

{

// TODO: std::sort(map.begin(), map.end());

// std::sort = IntroSort()

sort() is a std algorithm that comes with the <algorithm> header.

sort() takes 3 parameters: (begin() ptr\*, end() ptr\*, custom method)

sort() custom method is a function used to sort the array.

int arr[5] = { 1, 4, 2, 5, 3 };

// sort(\*begin, \*end) // sorts in ascending order by default.

std::sort(arr, arr + 5); // arr+5 is one past the last element in array arr[5]

// arr + 5 == arr[5]

for (int x : arr)

std::cout << x << std::endl;

//// VECTOR ////

std::vector<int> myVec = { 1, 4, 2, 5, 3 };

//std::sort(myVec.begin(), myVec.end());

std::sort(begin(myVec), end(myVec)); // sort can be used with any 2 iterators on any DS.

for (int x : myVec)

std::cout << x << std::endl;

cout << "\n";

// greater, less, less\_equal, greater\_equal optional function ptr argument.

std::sort(begin(myVec), end(myVec), greater<int>()); // greater = descending order, large first

for (int x : myVec)

std::cout << x << std::endl;

cout << "\n";

// Predictor: User-Defined Condition

// userDefinedCondition() predictor : condition sorts by first decimal place instead of overall

std::vector<int> myVec2 = { 11, 44, 27, 52, 33 };

std::sort(myVec2.begin(), myVec2.end(), UserDefinedCondition); // return a%10 < b%10

for (int x : myVec2)

std::cout << x << std::endl;

}

void STL\_BinarySearch()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

bool isPresent = binary\_search(a.begin(), a.end(), 22);

std::cout << (isPresent ? "Found" : "Not Found") << std::endl;

}

void Swap()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 4, 6, 8, 9 };

std::vector<int> c(20);

swap(a, b);

cout << "------- Printing A (used to vector B) ------- " << endl;

for (int x : a)

cout << x << endl;

cout << "------- Printing B (used to vector A) ------- " << endl;

for (int x : b)

cout << x << endl;

}

void Swap\_Ranges()

{

std::vector<int> a = { 1, 2, 3, 4, 5 };

std::vector<int> b = { 1, 2, 3, 6, 7 };

std::vector<int> c(10);

swap\_ranges(a.begin(), a.begin() + 3, b.begin() + 2);

cout << "------- Printing A (used to vector B) ------- " << endl;

for (int x : a)

cout << x << endl;

cout << "------- Printing B (used to vector A) ------- " << endl;

for (int x : b)

cout << x << endl;

}

bool UserDefinedCondition(int a, int b)

{

return (a % 10) < (b % 10); // while %10 of two number is lower, put lowest %10 first.

}

void VectorMinMax()

{

// max, min, minmax return iterators, need to de-reference

std::vector<int> myVec = { 1,2,3,4,5 };

std::cout << "Vector: "; PrintVector(myVec);

std::cout << "Vector Max: " << \*(max(myVec.begin(), myVec.end() - 1)) << std::endl;

std::cout << "Vector Min: " << \*(min(myVec.begin(), myVec.end() - 1)) << std::endl;

std::cout << "minmax first: " << \*(minmax(myVec.begin(), myVec.end() - 1).first) << std::endl;

std::cout << "minmax second: " << \*(minmax(myVec.begin(), myVec.end() - 1).second) << std::endl;

// minmax return a pair of iterators where first in min and second is max.

}

# Searching-Algorithms

// SearchAlgorithms.cpp | https://www.geeksforgeeks.org/searching-algorithms/

// Linear Search, Sentinetal-Linear, Binary Search, Ternary Search

// Meta Binary Search (one-sided), Interpolation Search, Jump Search

//

// OTHER Search algorithms NOT COVERED: exponential, fibonacci, & ubiquitous binary.

#include <iostream>

#include <algorithm>

#include <string>

#include <cmath>

#include <vector>

#include <windows.h>

using namespace std;

void LinearSearch();

int search(int arr[], int N, int x);

void SentinelLinearSearch();

void sentinelSearch(int arr[], int n, int key);

void BinarySearch();

int binarySearch(int\* arr, int low, int high, int key);

int binarySearchRecursion(int\* arr, int low, int high, int key);

void TernarySearch();

int ternarySearch(int l, int r, int key, int ar[]);

void MetaBinary();

int bsearch(std::vector<int> A, int key\_to\_search);

void InterpolationSearch();

int interpolationSearch(int arr[], int lo, int hi, int x);

void JumpSearch();

int jumpSearch(int arr[], int x, int n);

// ### SEARCH EXAMPLES ###

int main()

{

// LinearSearch();

// SentinelLinearSearch();

// BinarySearch();

// TernarySearch();

// MetaBinary();

// InterpolationSearch();

// JumpSearch();

system("pause");

}

void LinearSearch()

{

// Linear Search is defined as a sequential search algorithm that starts at one end

// and goes through each element of a list until the desired element is found,

// otherwise the search continues till the end of the data set.

int arr[] = { 2, 3, 4, 10, 40 };

int x = 10;

int N = sizeof(arr) / sizeof(arr[0]);

// Function call

int result = search(arr, N, x);

if (result == -1)

std::cout << "Element is not present in array" << std::endl;

else

std::cout << "Element is present at index " << result << std::endl;

}

void SentinelLinearSearch()

{

/\*

The basic idea of Sentinel Linear Search is to add an extra element at the end of the array

(i.e., the sentinel value) that matches the search key. By doing so, we can avoid the conditional check

for the end of the array in the loop and terminate the search early, as soon as we find the sentinel element.

This eliminates the need for a separate check for the end of the array, resulting in a slight improvement.

When and Where:

The sentinel linear search algorithm is useful for arrays with a large number of elements where the target

value may be located towards the end of the array.

\*/

int arr[] = { 10, 20, 180, 30, 60, 50, 110, 100, 70 };

int n = sizeof(arr) / sizeof(arr[0]);

int key = 180;

sentinelSearch(arr, n, key);

}

void BinarySearch()

{

// \*\*\* BINARY SEARCH NOTES \*\*\*

// Works on sorted array. Data needs to be sorted before searching!

// Divide & Conquer - Each iteration divides array in half.

//

// Steps

// 1. Find middle element of array and check if that is the value you are looking for

// 2. If yes, exit the algorithm and return value

// 3. If no, check if that middle element is larger than the value we want.

// 4. If element is bigger, we want to shrink array to the left half of current array

// 5. If element is smaller, we want to shrink array to the right half of current array

// 6. Repeat until value is found.

//

// Complexity

// Time: O(log2(n))

// Best Case: O(1)

// Average Case: O(log2(n))

// ### QUESTIONS ###

// If half'n an array with an even number of elements, which side do you select as mid?

// How is that decided in code?

// How do you run Binary search on something other than int numbers?

int arr[] = { 1, 2, 3, 5, 7, 8, 9, 10 };

std::cout << binarySearch(arr, 0, 7, 2) << std::endl; // returns index 1

std::cout << binarySearchRecursion(arr, 0, 7, 2) << std::endl; // returns index 1

std::cout << binarySearch(arr, 0, 7, 6) << std::endl; // returns -1, NOT found

std::cout << binarySearch(arr, 0, 7, 10) << std::endl; // returns index 7

}

void TernarySearch()

{

/\*

TS principle of dividing the array into three parts instead of two, as in binary search.

STEPS:

Below are the step-by-step explanation of working of Ternary Search:

Initialization:

Begin with a sorted array.

Set two pointers, left and right, initially pointing to the first and last elements of the array.

Divide the Array:

Calculate two midpoints, mid1 and mid2, dividing the current search space into three roughly equal parts:

mid1 = left + (right – left) / 3

mid2 = right – (right – left) / 3

The array is now effectively divided into [left, mid1], (mid1, mid2), and [mid2, right].

Comparison with Target:.

If the target is equal to the element at mid1 or mid2, the search is successful, and the index is returned

If the target is less than the element at mid1, update the right pointer to mid1 – 1.

If the target is greater than the element at mid2, update the left pointer to mid2 + 1.

If the target is between the elements at mid1 and mid2, update the left pointer to mid1 + 1

and the right pointer to mid2 – 1.

Repeat or Conclude:

Repeat the process with the reduced search space until the target is found or the search space becomes empty.

If the search space is empty and the target is not found, return a value indicating that the target

is not present in the array.

WHEN AND WHERE:

Sorted Array.

When you have a large ordered array or list and need to find the position of a specific value.

When you need to find the maximum or minimum value of a function.

Ternary search can find maxima/minima for unimodal functions, where binary search is not applicable.

Time: O(2 \* log3n)

DISADVANTAGES:

Ternary Search takes more time to find maxima/minima of monotonic functions as compared to Binary Search.

\*/

int l, r, p, key;

// Get the array

// Sort the array if not sorted

int ar[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Starting index

l = 0;

// end element index

r = 9;

// Checking for 5

// Key to be searched in the array

key = 5;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

std::cout << "Index of " << key

<< " is " << p << std::endl;

// Checking for 50

// Key to be searched in the array

key = 50;

// Search the key using ternarySearch

p = ternarySearch(l, r, key, ar);

// Print the result

std::cout << "Index of " << key

<< " is " << p << std::endl;

}

void MetaBinary() // incrementally constructs the index of the target value in the array

{

/\*

The Meta Binary (One-Sided) algorithm uses a heuristic to determine the size of the next interval.

It computes the difference between the value of the middle element and the value of the target element,

and divides the difference by a predetermined constant, usually 2.

This result is then used as the size of the new interval.

The algorithm continues until it finds the target element or determines that it is not in the list.

WHEN AND WHERE:

The advantage of Meta Binary Search over binary search is that it can perform fewer comparisons in some cases,

particularly when the target element is close to the beginning of the list.

Time: O(log n)

\*/

std::vector<int> A = { -2, 10, 100, 250, 32315 };

std::cout << bsearch(A, 10) << std::endl;

}

void InterpolationSearch()

{

/\* INTERPOLATION SEARCH

. Interpolation constructs new data points within the range of a discrete set

of known data points. Binary Search always goes to the middle element to check.

On the other hand, interpolation search may go to different locations

according to the value of the key being searched.

\*/

// Array of items on which search will

int arr[] = { 10, 12, 13, 16, 18, 19, 20, 21,

22, 23, 24, 33, 35, 42, 47 };

int n = sizeof(arr) / sizeof(arr[0]);

// Element to be searched

int x = 18;

int index = interpolationSearch(arr, 0, n - 1, x);

// If element was found

if (index != -1)

std::cout << "Element found at index " << index;

else

std::cout << "Element not found.";

std::cout << "\n";

}

void JumpSearch()

{

/\*

Jump Search is an algorithm for finding a specific value in a sorted array by jumping

through certain steps in the array. The steps are determined by the sqrt of the length of the array.

\*/

int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,

34, 55, 89, 144, 233, 377, 610 };

int x = 55;

int n = sizeof(arr) / sizeof(arr[0]);

// Find the index of 'x' using Jump Search

int index = jumpSearch(arr, x, n);

// Print the index where 'x' is located

std::cout << "\nNumber " << x << " is at index " << index << std::endl;

}

// ### SEARCH FUNCTIONS ###

int search(int arr[], int N, int x)

{

for (int i = 0; i < N; i++)

if (arr[i] == x)

return i;

return -1;

}

void sentinelSearch(int arr[], int n, int key)

{

// Last element of the array

int last = arr[n - 1];

// Element to be searched is

// placed at the last index

arr[n - 1] = key;

int i = 0;

while (arr[i] != key)

i++;

// Put the last element back

arr[n - 1] = last;

if ((i < n - 1) || (arr[n - 1] == key))

std::cout << key << " is present at index " << i << std::endl;

else

std::cout << "Element Not found" << std::endl;

}

int binarySearch(int\* arr, int low, int high, int key) // returns the index

{

while (low <= high)

{

int mid = (low + high) / 2;

if (arr[mid] < key) // top half, change low

low = mid + 1;

else if (arr[mid] > key) // bottom half, change high

high = mid - 1;

else

return mid;

}

return -1;

}

int binarySearchRecursion(int\* arr, int low, int high, int key)

{

if (low > high)

return -1;

int mid = (low + high) / 2;

if (arr[mid] == key)

return mid;

else if (arr[mid] > key)

high = mid - 1;

else

low = mid + 1;

binarySearchRecursion(arr, low, high, key);

}

int bsearch(std::vector<int> A, int key\_to\_search)

{

int n = (int)A.size();

// Set number of bits to represent largest array index

int lg = log2(n - 1) + 1;

//while ((1 << lg) < n - 1)

//lg += 1;

int pos = 0;

for (int i = lg; i >= 0; i--) {

if (A[pos] == key\_to\_search)

return pos;

// Incrementally construct the

// index of the target value

int new\_pos = pos | (1 << i);

// find the element in one

// direction and update position

if ((new\_pos < n) && (A[new\_pos] <= key\_to\_search))

pos = new\_pos;

}

// if element found return pos otherwise -1

return ((A[pos] == key\_to\_search) ? pos : -1);

/\* META BINARY SEARCH PSEUDOCODE

function meta\_binary\_search(A, target):

n = length(A)

interval\_size = n

while interval\_size > 0:

index = min(n - 1, interval\_size / 2)

mid = A[index]

if mid == target:

return index

elif mid < target:

interval\_size = (n - index) / 2

else:

interval\_size = index / 2

return -1

\*/

}

int ternarySearch(int l, int r, int key, int ar[])

{

if (r >= l) {

// Find the mid1 and mid2

int mid1 = l + (r - l) / 3;

int mid2 = r - (r - l) / 3;

// Check if key is present at any mid

if (ar[mid1] == key) {

return mid1;

}

if (ar[mid2] == key) {

return mid2;

}

// Since key is not present at mid,

// check in which region it is present

// then repeat the Search operation

// in that region

if (key < ar[mid1]) {

// The key lies in between l and mid1

return ternarySearch(l, mid1 - 1, key, ar);

}

else if (key > ar[mid2]) {

// The key lies in between mid2 and r

return ternarySearch(mid2 + 1, r, key, ar);

}

else {

// The key lies in between mid1 and mid2

return ternarySearch(mid1 + 1, mid2 - 1, key, ar);

}

}

// Key not found

return -1;

}

int interpolationSearch(int arr[], int lo, int hi, int x)

{

int pos;

// Since array is sorted, an element present

// in array must be in range defined by corner

if (lo <= hi && x >= arr[lo] && x <= arr[hi]) {

// Probing the position with keeping

// uniform distribution in mind.

pos = lo

+ (((double)(hi - lo) / (arr[hi] - arr[lo]))

\* (x - arr[lo]));

// Condition of target found

if (arr[pos] == x)

return pos;

// If x is larger, x is in right sub array

if (arr[pos] < x)

return interpolationSearch(arr, pos + 1, hi, x);

// If x is smaller, x is in left sub array

if (arr[pos] > x)

return interpolationSearch(arr, lo, pos - 1, x);

}

return -1;

}

int jumpSearch(int arr[], int x, int n)

{

// Finding block size to be jumped

int step = sqrt(n);

// Finding the block where element is

// present (if it is present)

int prev = 0;

while (arr[min(step, n) - 1] < x)

{

prev = step;

step += sqrt(n);

if (prev >= n)

return -1;

}

// Doing a linear search for x in block

// beginning with prev.

while (arr[prev] < x)

{

prev++;

// If we reached next block or end of

// array, element is not present.

if (prev == min(step, n))

return -1;

}

// If element is found

if (arr[prev] == x)

return prev;

}

# Sorting-Algorithms

// Bubble Sort(), Selection Sort(), Merge Sort()

// Insertion Sort(), Quick Sort(), Counting Sort(), Heap Sort()

// IntroSort, QuickSort(), HeapSort(), then InsertionSort()

// IntroSort is implemented using hybrid of QuickSort, HeapSort and InsertionSort.By default,

// it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N\*logN time,

// it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort

#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <utility>

#include <stdio.h>

void SelectionSort();

void BubbleSort();

void mergeSort(int A[], int low, int high);

void merge(int A[], int low, int high, int mid);

void MergeSortExample();

void InsertionSortExample();

void insertion\_sort(int a[], int size);

void QuickSortExample();

void quicksort(int array[], int length);

void quicksort\_recursion(int array[], int low, int high);

int partition(int array[], int low, int high);

void CountingSortExample();

void countingSort(int arr[], int n, int max);

void HeapSortExample();

void heapSort(vector<int>& heap);

void heapify(vector<int>& heap, int current, int size);

void printVector(std::vector<int>& v);

void SelectionSort()

{

// Selecting the smallest element again and again, and putting them in appropriate place.

// Take a pass for all element still unsorted, track smallest value, and put the smallest value at beginning of unsorted.

// add 1 to sorted, subtract one from unsorted, start again at beginning of unsorted tracking min value again.

// https://www.youtube.com/watch?v=4oqjcKenCH8

// From 0 to size-1

// Find smallest number between numbers[i] and numbers[size-1] // outer array sorted

// Swap number with numbers[i] // inner array unsorted

std::vector<int> numbers = { 1,5,2,3,7,4,6 };

int min;

int minIndex;

for (int i = 0; i < numbers.size() - 1; i++)

{

min = numbers[i];

minIndex = i;

for (int j = i; j < numbers.size(); j++) // Mistake: int j = 0, instead of j=i;

{

if (numbers[j] < min)

{

min = numbers[j];

minIndex = j;

}

}

swap(numbers[i], numbers[minIndex]); // Mistake: numbers[j] instead of numbers[minIndex]

}

printVector(numbers); // 1,2,3,4,5,6,7 | 5,2,3,7,4,1,6

}

void BubbleSort()

{

// https://www.youtube.com/watch?v=4oqjcKenCH8

// Repeat n-1 times

// For i from 0 to n-2

// if numbers[i] and numbers[i+1] are out of order

// Swap numbers

std::vector<int> numbers = { 1,5,2,3,7,4,6 };

for (int i = 0; i < numbers.size() - 2; i++)

{

for (int j = i + 1; j < numbers.size() - 1; j++)

{

if (numbers[i] > numbers[j])

swap(numbers[i], numbers[j]);

}

}

printVector(numbers);

}

void MergeSortExample()

{

// https://www.youtube.com/watch?v=RZK6KVpaT3I&t=205s

// O(n\*log(n))

// LIFO recursion

// Divide and Conquer

// High memory usage for faster sorting.

// Continous divides and array into sub arrays recursively until only one element in each sub-Array.

// Then, recursively merge arrays by sorting them until one array is left.

// Keeps a low and high index to indentify what subArray values need sorted.

const int arraySize = 4;

int A[arraySize] = { 5,1,3,2 };

int low = 0; // [0:1] - merge(A, low, high, mid); m(A, 0, 1, 0);

int high = arraySize - 1; // [1:1] - mS[1,1] - returns, both mS calls for [0:1] finishes, now merge[0:1]

// finishes previous call - // [0:1] - ms(mid+1, high) mS[0,1]

//

// [0:0] - ms(low, mid) mS[0,0] - returns, pops off

// [0:1] - ms(low, mid) mS[0,1] - push

// [0:3] - ms(low, mid) mS[0,3] - push

// [0:6] - mS(low, mid) mS[0,6] - pushes to bottom of call stack

mergeSort(A, low, high); // // A[0:6] first entry

for (int i = 0; i < arraySize; i++)

{

std::cout << A[i] << ", ";

}

std::cout << "\n";

}

void mergeSort(int A[], int low, int high)

{

if (low < high) // checks if only one element remains. if it does, the subarray that made this call is already sorted.

{

int mid;

mid = (low + high) / 2;

mergeSort(A, low, mid); // sorts left side

mergeSort(A, mid + 1, high); // sorts right side

// merges 2 sub-arrays.

merge(A, low, high, mid); // last valid subArray values(low,mid,high) before returning on calls above.

// [0:1] - merge(A, low, high, mid); m(A, 0, 1, 0); mid is 0 because of int truncation: (0+1)/2 = 0.5 = 0

// [1:1] - mS[1,1] - returns | Both mergeSort calls for [0:1] low and high finish, now merge[0:1]

// [0:1] - ms(mid+1, high) mS[0,1]

//

// [0:0] - ms(low, mid) mS[0,0] - returns, pops off

// [0:1] - ms(low, mid) mS[0,1] - push (merge(0,1) left half, merge(0,1) right half)

// [0:3] - ms(low, mid) mS[0,3] - push (merge(whole list, comparing 0,3).

// [0:6] - mS(low, mid) mS[0,6] - pushes to bottom of call stack

}

}

void merge(int A[], int low, int high, int mid)

{

// Looks at 2 pseudo sub-arrays |

// defined with index variables passed in

// merge combines 2 arrays in sorted order.

int i, j, k;

int c[50];

i = low;

j = mid + 1;

k = low;

while (i <= mid && j <= high)

{

if (A[i] < A[j])

{

c[k] = A[i];

i++;

k++;

}

else

{

c[k] = A[j];

j++;

k++;

}

}

while (i <= mid)

{

c[k] = A[i];

i++;

k++;

}

while (j <= high)

{

c[k] = A[j];

j++;

k++;

}

for (i = low; i < k; i++)

{

A[i] = c[i];

}

}

void InsertionSortExample()

{

// https://www.youtube.com/watch?v=PEhwXHEPbmI&t=17s

// builds sorted array one element at a time.

// building up a sorted array on left, and an unsorted right portion of the array.

// first unsorted moves left checking if it's less than the value and keeps moving left until it's greater than left value.

// opposite bubble sort, values sink to the begining of the array, then values to the right of current index gets shifted right.

/\*int a[] = { 8, 2, 7, 4, 5, 1, 9, 3 };

int length = sizeof(a) / sizeof(int);

insertion\_sort(a, length);\*/

int arr[] = { 8,1,7,6,5,0,11,13 };

int size = sizeof(arr) / sizeof(int);

insertion\_sort(arr, size);

for (int x : arr)

std::cout << x << ", ";

cout << std::endl;

}

void insertion\_sort(int a[], int size)

{

// int arr[] = { 8,1,7,6,5,0,11,13 }; scanning left to right of unsorted, use a key value, find the index, shift everything right, insert

// i is unsorted, j is sorted. We are inserting key in the outer portion. j is leftmost sorted, so, j+1 will be the index added to sorted.

for (int i = 1; i < size; i++)

{

int key = a[i];

int j = i - 1;

while (j >= 0 && a[j] > key) // while index value of j is greater than current key, keep going. stop, when a[j] < key.

{

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = key;

}

// a[] = {8,2,7,4,5,1,9,3};

// i looks at each unsorted portion of the array.

//for (int i = 1; i < size; i++)

//{

// int key = a[i]; // key is the value in the outer loop that needs to be sorted.

// int j = i - 1; // keeps track of position in the sorted portion of array as we shift values.

// while (j >= 0 && a[j] > key) // until j gets to 0 and the array's value at j > current key value (a[i]). keep looping

// {

// a[j + 1] = a[j]; // shifts all values right.

// j = j - 1; // keeps track of the index in left hand sorted portion of array.

// } // j = -1 exits the inner loop, because it's reached the beginning of array.

// a[j + 1] = key; // key value inserted into correct position.

//}

}

void QuickSortExample()

{

// https://www.youtube.com/watch?v=vhSLT3a-t-A&t=811s

int a[] = { 10, 11, 23, 44, 8, 15, 3, 9, 12, 45, 56, 45, 45 };

int length = sizeof(a) / sizeof(int);

quicksort(a, length);

for (int i = 0; i < length; i++)

{

std::cout << a[i] << ", ";

}

std::cout << endl;

}

void quicksort(int array[], int length)

{

// our quicksort algorithm randomly selects the pivot, so we seed the random

// number generator to ensure the randomization of our random numbers

srand(time(NULL));

quicksort\_recursion(array, 0, length - 1);

}

void quicksort\_recursion(int array[], int low, int high)

{

// stop recursion when low >= high

if (low < high)

{

// partition the array by a pivot, and return the pivot element's index

int pivot\_index = partition(array, low, high);

// apply quicksort to the left side subarray

quicksort\_recursion(array, low, pivot\_index - 1);

// apply quicksort to the right side subarray

quicksort\_recursion(array, pivot\_index + 1, high);

}

}

int partition(int array[], int low, int high)

{

// index of the pivot after partition. int return

// https://www.youtube.com/watch?v=vhSLT3a-t-A&t=811s

// randomly select a pivot value between low-high by randomly selecting an

// index in the range low-high... we do this as opposed to just selecting

// the last element each time because it's technically possible that always

// selecting the same pivot will have poor performance if the array happens

// to contain values in a way that is suboptimal (e.g. if the array is sorted

// already before quicksort is applied)

int pivot\_index = low + (rand() % (high - low + 1)); // int pivot\_value = array[high]; int i = low;

// swap the element at the pivot\_index with the element at index high (i.e.

// the last element in this portion of the array), doing so allows us to

// apply the below partitioning algorithm

if (pivot\_index != high)

{

swap(array[pivot\_index], array[high]);

}

// the pivot value is now whatever is in the high index

int pivot\_value = array[high];

// the partitioning algorithm will shift elements that are less than the pivot

// value to the front portion of the low - high array indexes, with i keeping

// track of where the elements that are greater than the pivot value begin

int i = low;

// increment j from low up until but not including the pivot value at high

for (int j = low; j < high; j++)

{

// if the array value at j is less than the pivot value, perform a swap with

// the value at the array at index i... this effectively moves this "less

// than the pivot values" to the front portion, and we increment i to

// reflect where the values that are greater than the pivot now begin

if (array[j] <= pivot\_value)

{

swap(array[i], array[j]);

i++;

}

}

// we now know that the value at index i is greater than or equal to the pivot

// so we swap it with the pivot value to complete the partition

swap(array[i], array[high]);

// index i now contains the pivot value, so return this so that the

// quicksort\_recursion function knows where to split the array when applying

// the algorithm to the resulting subarrays

return i;

}

void CountingSortExample()

{

// O(n)

// NOTE: Counting Sort is efficient sort if range(largest value)

// is similar to number of total element in initial array.

// Analyze data before applying an algorithm to see which algorithm works best.

// When will this algorithm be more efficient than a O(n\*log(n))?

https://www.youtube.com/watch?v=vhSLT3a-t-A&t=811s

int i = 0;

int max = 0;

const int arrSize = 10;

int arr[arrSize] = { 0 };

for (int i = 0; i < arrSize; i++)

{

printf("Enter Element: ");

if (scanf("%d", &arr[i])) // Needs an error check, while(not a bad value)...

{

if (arr[i] > max)

max = arr[i];

}

}

countingSort(arr, arrSize, max);

}

void countingSort(int arr[], int n, int max)

{

// Increase the count by 1 for the value in the intial array to the corresponding index of the count array.

// [3,0,2,7,2,1,4] initial array

// [0,0,0,0,0,0,0,0] initial count array (size based on largest value)

// [1,1,2,1,1,0,0,1] final count array // add a +1 to index 3 for the first element in intial array.

// sorted:[0,1,2,2,3,4,7] final sorted array | skip the 0s of counted array.

// count array

int count[50] = { 0 }; // dynamic setup here would be better code than magic number

int i, j;

for (i = 0; i < n; i++) // +1 value to the indexed location in count associated with value of initial array.

count[arr[i]] = count[arr[i]] + 1;

std::cout << "Elements: ";

for (i = 0; i <= max; ++i)

{

// prints final sorted values - but doesn't create sorted array

for (j = 1; j <= count[i]; ++j) // non-zero element because if count array has a zero value for that index

{ // the final sorted array doesn't get that array index added.

printf("%d, ", i);

}

}

std::cout << std::endl;

}

void HeapSortExample()

{

// Heap sort is | https://www.youtube.com/watch?v=kU4KBD4NFtw

std::vector<int> heap = { 9,6,8,2,1,4,3 };

heapSort(heap);

std::cout << "Heap in Ascending Order: \n";

for (int i = 0; i < heap.size(); i++)

std::cout << heap[i] << ", ";

cout << "\n";

system("pause");

}

void heapSort(vector<int>& heap)

// Time: O(nlog(n)), Space: O(log(n))

// Build\_Max\_Heap

// for (i = size()-1; i > 0; i++)

// Swap(a[0], a[i]);

// Max\_Heapify(a, 0);

// 9

{ //6 //8

// Make a max heap from array - create a tree [9, 6, 8, //2 //1 //4 //3

// Given max heap, sort elements in ascending order

// After copying last element to the root, the last edge to that last element is removed. reducing heap size by 1.

// Extract max element, put it aside, max = 9;

// last element in heap gets copied to the root 3 to arr[0]

// last link to 3 from 9 is broken, reducing the heap size.

// Reduce heapsize each iteration before doing a max heapify.

// Builds Initial Max\_Heap

for (int i = (heap.size() / 2) - 1; i >= 0; i--) // if already a heap, skip.

{

heapify(heap, i, heap.size());

}

for (int i = heap.size() - 1; i > 0; --i)

{

int max = heap[0]; // store the max variable temporarily.

heap[0] = heap[i]; // swap heap root with last element

heap[i] = max;

heapify(heap, 0, i); // heapify root with heapsize = i;

}

}

void heapify(vector<int>& heap, int current, int size)

{

// starts at root, checks bottom 2 below that

// which is greater between those is swapped.

// happens again only to the lower value that was swapped.

// repeats the process until bottom of tree.

int largest = current;

int left = (2 \* current) + 1; // left child

int right = (2 \* current) + 2; // right child

if (left < size && heap[left] > heap[largest]) // find the largest between parent and left child.

largest = left;

// finds the largest between parent and right child (largest could be updated to left child before this check).

if (right < size && heap[right] > heap[largest]) // ultimately if the right child is largest, it will be swapped with parent.

largest = right;

if (largest != current) // recursively calls (like a while loop here) heapify until current root = largest.

{

int temp = heap[current];

heap[current] = heap[largest];

heap[largest] = temp;

heapify(heap, largest, size); // tree traversal recursively swapping.

}

}

void printVector(std::vector<int>& v)

{

for (int x : v)

std::cout << x << ", ";

std::cout << endl;

}

# Breadth-First & Depth-First Searches

class Graph

{

public:

void insertNode(char node, char neighbors)

{

myMap[node].push\_back(neighbors);

}

void breadthFirstSearch(Graph& g, char source)

{

std::queue<char> myQueue;

myQueue.push(source);

while (myQueue.size() > 0)

{

char currentFront = myQueue.front();

myQueue.pop();

if (currentFront != ' ')

std::cout << currentFront << std::endl;

for (auto& neighbor : g.myMap[currentFront])

{

myQueue.push(neighbor);

}

}

std::cout << "-----" << std::endl;

}

void depthFirstSearch(Graph& g, char source) // traverses different first path depending on which DFS used

{

std::stack<char> searchStack;

searchStack.push(source);

while (!searchStack.empty())

{

auto current = searchStack.top();

searchStack.pop();

std::cout << current << std::endl;

for (auto neighbor : g.myMap[current]) // for each element in the list, associated

{

if (neighbor == ' ') continue;

searchStack.push(neighbor);

}

}

std::cout << "-----" << std::endl;

}

void depthFirstRecursive(Graph& g, char source) // traverses different first path depending on which DFS used

{

if (source != ' ') // if array is empty, don't print it out, skip to next recursive call.

std::cout << source << std::endl;

for (auto neighbor : g.myMap[source])

{

depthFirstRecursive(g, neighbor);

}

}

void printAdjList()

{

for (auto& element : myMap)

{

std::cout << "Key: " << element.first << " Neighbors ";

for (auto& nbr : element.second)

{

if (nbr == ' ') continue;

std::cout << nbr << ", ";

}

std::cout << "\n";

}

std::cout << "-----" << std::endl;

}

private:

std::unordered\_map<char, std::list<char>> myMap;

std::set<char> visitedSet;

int largestCount = 0;

};

int main()

{

Graph g;

g.insertNode('a', 'c');

g.insertNode('a', 'b');

g.insertNode('b', 'd');

g.insertNode('c', 'e');

g.insertNode('d', 'f');

g.insertNode('e', ' ');

g.insertNode('f', ' ');

// g.printAdjList(); // print a key and it's list of neighbors

g.breadthFirstSearch(g, 'a'); // a, c, b, e, d, f

/\*

Breadth first: FIFO queue.  Remove from the front queue.front() queue.pop

and add to the back queue.push\_back()

\*/

system("pause");

}

// Breadth First Search (BFS) Notes

//

// Queue data structure

// Finds all neighbors first

// Used in unweighted graphs

// Better than DFS for searching vertices closer to source.

// GOOD for shortest path

// NOT Good for decision making trees.

// NOT Good for memory

// Time: O(N+E) | Nodes + Edges

# Dijkstra’s Algorithm

Sdsd

// Djikstras\_Algorithm.cpp | https://www.youtube.com/watch?v=a1Z1GmKzcPs

// Not sure if this works.

#include <iostream>

using namespace std;

#define INF 999

int cost[100][100]; // graph

int dist[100]; // distance metric for all the nodes

int n; // # of vertices

int src; // source node

bool visited[100] = { 0 };

int parent[100];

void init()

{

for (int i = 0; i < n; i++)

{

dist[i] = INF;

parent[i] = i;

}

dist[src] = 0;

}

int getMin(int dist[], bool visited[]) {

int key = 0;

int min = INT\_MAX;

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

if (!visited[i] && dist[i] < min) {

min = dist[i];

key = i;

}

}

return key;

}

void display(int dist[], int par[]) {

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

int temp = par[i];

cout << i << " <- ";

while (temp != -1)

{

cout << temp << " <- ";

temp = par[temp];

}

cout << endl;

cout << "::::Distance = " << dist[i];

cout << endl;

}

}

int getNearest()

{

// Calculates minimum value of distance

int minvalue = INF;

int minnode = 0;

for (int i = 0; i < n; i++)

{

if (!visited[i] && dist[i] < minvalue) // see if node is already visited or not

{

minvalue = dist[i];

//visited[i] = true;

minnode = i;

}

}

return minnode;

}

void dijkstra2()

{

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

{

int nearest = getNearest(); // find nearest unvisited node.

visited[nearest] = true; // mark it as visited.

// UPDATE distances of all adjacent nodes!

// For all adjacent nodes, return nodes that are NOT 999

for (int adj = 0; adj < n; adj++)

{

// check if adj of my nearest adjacent node's distance is not INF(999).

if (cost[nearest][adj] != INF)

{

// if previous distance of adjacent node is NOT already minimum, update the minimum node.

// if new distance is lesser distance, then update.

if (dist[adj] > dist[nearest] + cost[nearest][adj])

{

dist[adj] = dist[nearest] + cost[nearest][adj]; // now keep track of parent node.

parent[adj] = nearest;

}

}

}

}

}

void dijkstra(int src) {

int par[100], dist[100];

bool visited[100] = { 0 };

fill(dist, dist + n, INT\_MAX);

dist[src] = 0;

par[src] = -1;

for (int g = 0; g < n - 1; g++) {

int u = getMin(dist, visited);

visited[u] = true;

cout << " min = " << u << endl;

for (int v = 0; v < n; v++) {

if (!visited[v] && (dist[u] + cost[u][v]) < dist[v] && cost[u][v] != 9999)

{

par[v] = u;

dist[v] = dist[u] + cost[u][v];

}

}

}

display(dist, par);

}

int main(void) {

cout << "Enter number of vertices : ";

cin >> n; // enter number of vertices.

cout << "Enter cost matrix : \n";

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

cin >> cost[i][j];

}

int src;

cout << "\nEnter source : "; cin >> src;

//dijkstra(src);

//init();

dijkstra(0);

//display(dist, parent);

system("pause");

}

// GRAPH SETUP:

//

// N(3) --7-- N(2) --8--- N(1)

// 9/ \14 4/ \2 | \4

// / \ / N(8) | N(0)

// N(4)--10--- N(5) |6 |8

// 2 \ | |

// N(6)--1--N(7)

//

// N=Node 0:N 1:N

// Node(0) | 0 4 999 999 999 999 999 8 999

// Node(1) | 4 0 8 999 999 999 999 8 999

// Node(2) | 999 8 0 7 999 4 999 999 2

// Node(3) | 999 999 7 0 9 14 999 999 999

// Node(4) | 999 999 999 9 0 10 999 999 999

// Node(5) | 999 999 4 14 10 0 2 999 999

// Node(6) | 999 999 999 999 999 2 0 1 6

// Node(7) | 8 11 999 999 999 999 1 0 7

// Node(8) | 999 999 2 999 999 999 6 7 0

// ADDITIONAL DJIKSTRA NOTES

//

// A\* GRAPH Example

//

// Node(B)----4----Node(D)

// 2 / | \ | \ 2

// Node(A) 1 2 3 Node(F)

// 4 \ | \ | / 2

// Node(C)---3---Node(E) -/

//

// Dijkstra Algorithm Steps:

//

// 1. Set all Nodes to being unvisited.

// 2. Set all distances of the Nodes to NULL.

// 3. Set origin Node to be current Node.

// 4. Set distance of current Node to be 0.

// 5. Set Current Node to visited.

//

// 1. Set all adjacent nodes label's to Current Node label + edge value (e.g. B = 2, C = 4 labels).

// 2. Check which adjacent node label contains lowest value and set current Node to that node (B is current).

// 3. Repeat steps 1 and 2 of iterative steps (Check labels of B's adjacent Nodes). B(C) = 1, B(E) = 2, B(D) = 6.

// - C becomes current Node, update C's label from 4 to 3 (A,B,C).

// 4. Update all labels of node accounting for values starting from A to dest Node, update label to shortest distance.

// - C (A,B,C) = 3, E (A,B,E) = 4, D (A,B,D) = 6

// - F = Node(E)->F = 4+2 = 6 or F = Node(D)->F = 6 + 2 = 8.

// - Lowest of A->F or 6 (A,B,E,F).

//

// PseudoCode

//

// wrap start and goal in Nodes

// insert start node into open list

// loop while open not empty

// set current to lowest node in open

// remove current from open

// store current adjacent nodes in adjacent linked list

// loop while adjacent not empty

// set adj to lowest weighted in adjacent linked list

// wrap adj in Node

// set distance to current's g value + DistanceBetween current and adj.

// if adj node's g is zero or distance is less than adj node's g

// set adj node's g to distance

// set adj node's parent to current

//

// if adj is goal

// return ReconstructPlan(adj's node)

//

// insert adj's node into open

// delete adj from adjancent list

// end loop

// end loop

// return

sss

# Dynamic Programming

// Dynamic\_Programming\_Fibonacci.cpp | https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

// YouTube search Dynamic Programming and watch explanation

// C++ program for Fibonacci Series

// using Dynamic Programming

#include <windows.h>

#include <iostream>

using namespace std;

class GFG {

public:

int\* fib(int n)

{

// Declare an array to store

// Fibonacci numbers.

// 1 extra to handle

// case, n = 0

int\* f = new int[n + 2];

int i;

// 0th and 1st number of the

// series are 0 and 1

\*f = 0;

\*(f+1) = 1;

for (i = 2; i <= n; i++) {

// Add the previous 2 numbers

// in the series and store it

\*(f+i) = \*(f + i - 1) + \*(f + i - 2);

}

return f;

}

};

// Driver code

int main()

{

GFG g;

int n = 9;

int\* fiba = g.fib(n);

std::cout << \*(fiba+n) << std::endl; // memory leak.

system("pause");

delete[] fiba;

return 0;

}

# Priority Scheduling

// Priority\_Scheduling.cpp |

// https://www.youtube.com/watch?v=MFa2NvJOI54&list=PLm0fFUL4gEt9QZ5gGl63ty4qSwrmzVvH\_&index=31&t=222s

// C++ program for implementation of FCFS scheduling

#include <iostream>

#include <windows.h>

#include <algorithm>

using namespace std;

struct Process {

int pid; // Process ID

int bt; // CPU Burst time required

int priority; // Priority of this process

};

// Function to sort the Process acc. to priority

bool comparison(Process a, Process b)

{

return (a.priority > b.priority);

}

// Function to find the waiting time for all

// processes

void findWaitingTime(Process proc[], int n, int wt[])

{

// waiting time for first process is 0

wt[0] = 0;

// calculating waiting time

for (int i = 1; i < n; i++)

wt[i] = proc[i - 1].bt + wt[i - 1];

}

// Function to calculate turn around time

void findTurnAroundTime(Process proc[], int n, int wt[],

int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

for (int i = 0; i < n; i++)

tat[i] = proc[i].bt + wt[i];

}

// Function to calculate average time

void findavgTime(Process proc[], const int n)

{

int\* wt = new int( (sizeof(int) \* n) );

int\* tat = new int( (sizeof(int) \* n) );

int total\_wt = 0, total\_tat = 0;

// Function to find waiting time of all processes

findWaitingTime(proc, n, wt);

// Function to find turn around time for all processes

findTurnAroundTime(proc, n, wt, tat);

// Display processes along with all details

std::cout << "\nProcesses "

<< " Burst time "

<< " Waiting time "

<< " Turn around time\n";

// Calculate total waiting time and total turn

// around time

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

total\_wt = total\_wt + \*(wt + i);

total\_tat = total\_tat + \*(tat + i);

std::cout << " " << proc[i].pid << "\t\t" << proc[i].bt

<< "\t " << wt[i] << "\t\t " << tat[i]

<< std::endl;

}

std::cout << "\nAverage waiting time = "

<< (float)total\_wt / (float)n;

std::cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

void priorityScheduling(Process proc[], const int n)

{

// Sort processes by priority

std::sort(proc, proc + n, comparison);

std::cout << "Order in which processes gets executed \n";

for (int i = 0; i < n; i++)

cout << proc[i].pid << " ";

findavgTime(proc, n);

}

// Driver code

int main()

{

Process proc[]

= { { 1, 10, 2 }, { 2, 5, 0 }, { 3, 8, 1 } };

size\_t a = sizeof(proc) / sizeof (proc[0]);

const int n = a;

priorityScheduling(proc, n);

system("pause");

}

# A\* Pathfinding

Dsd

Sds

Algorithm Types: (define)

// Algorithm Types (what) :

// Predicate checking

// Comparison

// Searching

// Binary Searching

// Max\_Min search

// Property Checking

// Copying

// Partitioning

// Sorting

// Populating

// Transforming

// Reordering

// Heap

Advanced Algorithms:

Advanced Algorithms:

Kruskal’s Algorithm

Floyd Warshall Algorithm

Dijkstra’s Algorithm

Bellman Ford Algorithm

Kadane’s Algorithm

Lee Algorithm

Flood Fill Algorithm

Floyd’s Cycle Detection Algorithm

Topological Sorting in a DAG

Union Find Algorithm