alg

Linear Search and Binary Search

Algorithms

Linear Search

```
Algorithm SeqSearch(a, x, n)
// Search for x in a[1 : n]. a[0] is used as additional space.
{
    i := n; a[0] := x;
    while (a[i] ≠ x) do i := i − 1;
    return i;
}
```

Algorithm 1.17 Sequential search

Binary Search

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                 return BinSrch(a, i, mid − 1, x);
             else return BinSrch(a, mid + 1, l, x);
    }
}
```

Algorithm 3.2 Recursive binary search

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// order, n ≥ 0, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low + high)/2⌋;
        if (x < a[mid]) then high := mid − 1;
        else if (x > a[mid]) then low := mid + 1;
             else return mid;
    }
    return 0;
}
```

Algorithm 3.3 Iterative binary search

Pseudocode

Linear Search

```
linear_search(array, query, loc)
begin
    If (loc ⩾ array.length) {
        Return “Not Found”
    }
        If (array[loc] = query) {
        Return loc
    }
    Else {
        Return linear_search(array, query, loc + 1)
end
```

Binary Search

```
binary_search(array, query, low, high)
begin
    If (low > high) {
        Return “Not Found”
    }
    mid ← (low + high) / 2
    If (array[mid] = query) {
        Return mid
    }
    If (array[mid] < query) {
```

```
        Return binary_search(array, query, mid + 1, high)
    }
    If (array[mod] > query) {
        Return binary_search(array, query, low, mid - 1)
    }
```

Code

```c
// 1. Implement recursive linear and binary search and determine the time taken
// to search an element. Repeat the experiment for different values of n, the
// number of elements in the list to be searched and plot a graph of the time
// taken versus n

// Included Libraries
#include <limits.h>  // Maximum value of int (constant)
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Random number generation
#include <time.h>    // Time based operations

// Macro definitions
#define MAX_LENGTH   20          // Array length
#define MAX_NUM      USHRT_MAX   // Maximum element in array
#define ITERATIONS   20000       // Maximum number of searches
#define NOT_FOUND    -1          // Element not found in array
#define LINEAR_INDEX 1  // Command line argument index for linear search file
#define BINARY_INDEX 2  // Command line argument index for binary search file

// Function definitions
void gen_rand_arr(int*, int);  // Generates random array
void print_arr(int*, int);     // Prints array
void sort_arr(int*, int);      // Sorts array in increasing order
int  random_query(int*, int);  // Gets a random element from array
int  linear_search(int*, int, int, int, int*);       // Recursive linear search
int  binary_search(int*, int, int, int, int, int*);  // Recursive binary search

int
main(int argc, char** argv) {
    int   array[MAX_LENGTH];  // Array in use in the program
    int   len = MAX_LENGTH;   // Length of array
    int   query;              // Element to be searched for
    int   index_found_at;     // Index location of element
    int   steps;              // Steps taken to find element
    FILE* linear_file;  // Text file to store steps taken in linear searching
    FILE* binary_file;  // Text file to store steps taken in binary searching

    srand(time(NULL));  // Takes current time as seed for rand()

    gen_rand_arr(array,
                 len);     // Generates an array of random integers `len` long
    sort_arr(array, len);  // Sorts an array `length` long in increasing order

    linear_file = fopen(argv[LINEAR_INDEX],
                        "w");  // Opens file to store linear search steps
    binary_file = fopen(argv[BINARY_INDEX],
                        "w");  // Opens file to store binary search steps

    for ( int i = 0; i < ITERATIONS; i++ ) {
        int lin_steps = 0;  // Steps taken to linear search `iter` times
        int bin_steps = 0;  // Steps taken to binary search `iter` times
        int iter      = 0;  // Number of searches in each iteration

        while ( iter <= i ) {
            query = random_query(array, len);

            steps          = 0;
            index_found_at = linear_search(array, len, query, 0, &steps);
            lin_steps += steps;

            steps = 0;
            index_found_at =
                binary_search(array, len, query, 0, len - 1, &steps);
            bin_steps += steps;

            iter++;
        }

        if ( iter % 1000 == 0 ) {
            // Prints number of iterations, number of linear and number of
            // binary steps every 1000 iterations
            fprintf(stdout, "iter: %7d, lin: %7d, bin: %7d\n", iter, lin_steps,
                    bin_steps);
        }
```

```c
        // Adds number of steps in iteration to file
        fprintf(linear_file, "%d ", lin_steps);
        fprintf(binary_file, "%d ", bin_steps);
    }

    fclose(linear_file);
    fclose(binary_file);

    return 0;
}

// Adds `len` numbers less than MAX_NUM to array
void
gen_rand_arr(int* array, int len) {
    for ( int i = 0; i < len; i++ ) {
        array[i] = rand() % MAX_NUM;
    }
}

// Prints first `len` elements of array
void
print_arr(int* array, int len) {
    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%d\t", array[i]);
    }

    fprintf(stdout, "\n");
}

// Sorts array in ascending order via bubble sort
void
sort_arr(int* array, int len) {
    int temp;

    for ( int i = 0; i < len - 1; i++ ) {
        for ( int j = 0; j < len - 1; j++ ) {
            if ( array[j] > array[j + 1] ) {  // Swap elements if predecessor is
                                              // greater than successor
                temp         = array[j];
                array[j]     = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

// Gets a random index from the array and returns element at that index
int
random_query(int* array, int len) {
    int random_index = rand() % len;

    return array[random_index];
}

// Recursively forward linear searches the array for `query`
int
linear_search(int* array, int len, int query, int loc, int* steps) {
    *steps += 1;  // Increment number of steps

    if ( loc >= len ) {  // If location is more than length of array, query does
                         // not exist in array
        return NOT_FOUND;
    } else if ( array[loc] == query ) {  // If location is the same as the
                                         // query, location is returned as index
        return loc;
    }

    return linear_search(array, len, query, loc + 1, steps);
}

// Recursively binary searches a sorted array for 'query'
int
binary_search(int* array, int len, int query, int low, int high, int* steps) {
    *steps += 1;  // Increments number of steps

    if ( low > high ) {  // If higher index is greater than lower index, query
                         // does not exist in array
        return NOT_FOUND;
    }

    int mid = (low + high) / 2;  // Midpoint of current subarray

    if ( array[mid] == query ) {  // If element exists at current midpoint of
```

```
                                    // subarray, return the midpoint as index
        return mid;
    }

    if ( array[mid] < query ) {  // If midpoint element is lower than query, use
                                 // the second half of subarray as new subarray
        return binary_search(array, len, query, mid + 1, high, steps);
    } else {  // If midpoint element is lower than query, use the first half of
              // subarray as new subarray
        return binary_search(array, len, query, low, mid - 1, steps);
    }
}
```

Analysis

Linear Search

Linear search algorithm searches across the array in a linear manner, element-by-element to find the required index.
Consider an array of length n. The element being searched can either be at any index in the array, i.e., at indices 0 to $n-1$, or it could not be in the array.
In the worst-case scenario, either the element exists at the end of the array, or it does not exist in the array. Thus, either the element is found after n comparisons, or it is not found after n comparisons. Therefore, we can say that the algorithm has a worst-case time complexity of $O(n)$.

Binary Search

Binary search algorithm works by dividing the array into subarrays successively, until the query element is found or not.
Consider an array of length n. The element being searched can either be at any index in the array, i.e., at indices 0 to $n-1$, or it could not be in the array.
In the worst-case scenario, either the element exists at the end of the last search in the array, or it does not exist in the array. Therefore, $\frac{n}{2}$ elements are searched first, then $\frac{n}{4}$, and so on.

$$comparisons = n + \frac{n}{2} + \frac{n}{4} + \ldots = \log_2 n$$

Therefore, since about $\log_2 n$ comparisons are needed, the worst-case time complexity is $O(\log_2 n)$.

Selection, Bubble and Insertion Sort

Algorithms

Selection Sort

```
Algorithm SelectionSort(a, n)
// Sort the array a[1 : n] into nondecreasing order.
{
    for i := 1 to n do
    {
        j := i;
        for k := i + 1 to n do
            if (a[k] < a[j]) then j := k;
        t := a[i]; a[i] := a[j]; a[j] := t;
    }
}
```

Algorithm 1.2 Selection sort

Bubble Sort

```
begin BubbleSort(arr)
    for all array elements
        if arr[i] > arr[i+1]
            swap(arr[i], arr[i+1])
        end if
    end for
    return arr
end BubbleSort
```

Insertion Sort

```
Algorithm InsertionSort(a, n)
// Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
{
    for j := 2 to n do
    {
        // a[1 : j − 1] is already sorted.
        item := a[j]; i := j − 1;
        while ((i ≥ 1) and (item < a[i])) do
        {
            a[i + 1] := a[i]; i := i − 1;
        }
        a[i + 1] := item;
    }
}
```

Algorithm 3.9 Insertion sort

Pseudocode

Selection Sort

```
selection_sort(array)
begin
    For i = (0 to array.length - 1) {
        min_index = i
        For j = (i + 1 to array.length) {
            If (array[j] < array[min_index]) {
                min_index = j
            }
        }
        Swap(array[i], array[min_index])
    }
end
```

Bubble Sort

```
bubble_sort(array)
begin
    For i = (0 to array.length) {
        For j = (0 to array.length - i) {
            If (array[j] > array[j + 1]) {
                Swap(array[j], array[j + 1])
            }
        }
    }
end
```

Insertion Sort

```
insertion_sort(array)
begin
    For i = (1 to array.length) {
        element = array[i]
        For j = (i to 0) {
            If (array[j - 1] > array[j]) {
                Swap(array[j - 1], array[j])
            }
        }
    }
end
```

Code

```c
// 2. Write a program to implement Selection, Bubble and Insertion Sorting
// Algorithms

// Included Libraries
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Random number generation
#include <time.h>    // Time based operations

// Macro definitions
#define MAX_LENGTH     20   // Maximum length of array
#define MAX_ELEM       100  // Maximum element in array
#define BUBBLE_SORT    1    // Menu driven program options
#define SELECTION_SORT 2    // Menu driven program options
#define INSERTION_SORT 3    // Menu driven program options
#define RESEED_ARRAY   4    // Menu driven program options
#define EXIT           5    // Menu driven program options

// Macro for swapping elements of array
#define SWAP(TYPE, A, B) \
    TYPE temp = A;       \
    A         = B;       \
    B         = temp;

// Function declarations
void gen_rand_arr(int*, int);     // Generates random array
void print_arr(int*, int);        // Prints array
void selection_sort(int*, int);  // Sorts array via selection sort algorithm
void bubble_sort(int*, int);      // Sorts array via bubble sort algorithm
void insertion_sort(int*, int);  // Sorts array via insertion sort algorithm

int
main(int argc, char** argv) {
    int choice;  // Variable used to declare choice in menu driven program

    srand(time(NULL));  // Takes current time as seed for rand()
```

```c
    do {
        int array[MAX_LENGTH];  // Array in use in the program
        int len = MAX_LENGTH;   // Length of array

        gen_rand_arr(array,
                     len);  // Generates an array of random integers `len` long
        fprintf(stdout, "Base Array:\n");
        print_arr(array, len);

        // Menu driven program
        fprintf(stdout, "1. Bubble Sort\n");
        fprintf(stdout, "2. Selection Sort\n");
        fprintf(stdout, "3. Insertion Sort\n");
        fprintf(stdout, "4. Reseed Array\n");
        fprintf(stdout, "5. Exit\n");
        fprintf(stdout, "Enter Choice: ");
        fscanf(stdin, "%d", &choice);

        switch ( choice ) {
        case BUBBLE_SORT: {
            bubble_sort(array, len);
        } break;

        case SELECTION_SORT: {
            selection_sort(array, len);
        } break;

        case INSERTION_SORT: {
            insertion_sort(array, len);
        } break;

        case RESEED_ARRAY: {
            srand(time(NULL));
        } break;

        case EXIT: {
        } break;

        default: {
        } break;
        }
    } while ( choice != 5 );

    return 0;
}

// Adds `len` numbers less than MAX_NUM to array
void
gen_rand_arr(int* array, int len) {
    for ( int i = 0; i < len; i++ ) {
        array[i] = rand() % MAX_ELEM;
    }
}

// Prints first `len` elements of array
void
print_arr(int* array, int len) {
    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%3d ", array[i]);
    }

    fprintf(stdout, "\n");
}

// Sorts array in ascending order via selection sort
void
selection_sort(int* array, int len) {
    int min_index;  // Index of minimum element of array

    for ( int i = 0; i < len - 1; i++ ) {
        min_index = i;  // Initialize minimum element as first element of array

        for ( int j = i + 1; j < len; j++ ) {
            if ( array[j] < array[min_index] ) {
                min_index = j;  // Change index if smaller element is found
            }
        }

        SWAP(int, array[min_index],
             array[i]);  // Swap first element with lowest element

        fprintf(stdout, "Pass %3d: ", (i + 1));
        print_arr(array, len);
```

```c
    }
}

// Sorts array in ascending order via bubble sort
void
bubble_sort(int* array, int len) {
    for ( int i = 0; i < len; i++ ) {
        for ( int j = 0; j < len - i; j++ ) {
            if ( array[j] >
                 array[j + 1] ) {  // Find if element is larger than successor
                SWAP(int, array[j],
                     array[j + 1]);  // Swaps elements to be in order one-by-one
            }
        }

        fprintf(stdout, "Pass %3d: ", (i + 1));
        print_arr(array, len);
    }
}

// Sorts array in ascending order via insertion sort algorithm
void
insertion_sort(int* array, int len) {
    int element;

    for ( int i = 1; i < len; i++ ) {
        element = array[i];  // Element being examined

        for ( int j = i; j > 0; j-- ) {
            if ( array[j - 1] >
                 array[j] ) {  // If element is larger than successor
                SWAP(int, array[j - 1], array[j]);  // Swaps elements to get to
                                                    // their correct position
            }
        }

        fprintf(stdout, "Pass %3d: ", (i + 1));
        print_arr(array, len);
    }
}
```

Analysis

Selection Sort

Selection sort algorithm iterates through the array to find the minimum or maximum element and swaps it to its proper location.
Consider an array of length n. The minimum/maximum element can be at any index in the array.
To place every element at its proper place, a nested loop is set up to find the lowest element in the complete array. After the lowest element is found, a subarray is considered which contains all elements besides the one found previously, from which the lowest element is found.
Therefore, we can see that number of searches are as follows:

$$\text{searches} = (n) + (n-1) + \ldots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Thus, the required complexity is $O(n^2)$.

Bubble Sort

Bubble sort algorithm iterates through the array to find out of order pairs of elements and swaps them to the right order repeatedly until the array is sorted.
Consider an array of length n. The minimum/maximum element can be at any index in the array.
The array is iterated and all pairs out of order are found and swapped. This leads to the maximum/minimum element being moved to the end of the array. Following this, a subarray excluding the last element is iterated upon, and the process repeats until the array is in order.
Therefore, we can see that the number of searches is as follows:

$$\text{searches} = (n) + (n-1) + \ldots + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

Thus, the required complexity is $O(n^2)$.

Insertion Sort

Insertion sort algorithm iterates through the array considering it element by element and swaps the element to its correct location until the array is fully sorted.
Consider an array of length n.
The array is divided into two subarrays, one sorted and one unsorted. One by one, elements from the unsorted subarrays are considered, and then they are inserted into the sorted subarray at their correct spot. Hence, we can understand that the sorted subarray grows in size from 1 element, to 2 elements, until it is n elements long. Therefore, the number of searches needed to insert the element at the correct spot in the worst-case scenario (i.e., the end of the array) will be:

$$\text{searches} = 1 + 2 + \ldots + (n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Thus, the required complexity is $O(n^2)$

Quicksort and Mergesort

Algorithms

Quicksort

```
Algorithm Partition(a, m, p)
// Within a[m], a[m+1], ..., a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j - 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

Mergesort

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

Algorithm 3.7 Merge sort

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h]; h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

Pseudocode

Quicksort

```
quick_sort(array, start, end)
begin
    If (start < end) {
        index = partition(array, start, end)
        quick_sort(array, start, index - 1)
        quick_sort(array, index + 1, end)
```

```
    }
end

partition(array, start, end)
begin
    pivot_index = end
    i = start - 1;
    For j = (start to end) {
        If (array[j] < array[pivot_index]) {
            i = i + 1
            Swap(array[i], array[j])
        }
    }
    Swap(array[i + 1], array[end]
    Return i + 1
```

Mergesort

```
merge_sort(array, start, end)
begin
    If (start < end) {
        mid = (start + end) / 2

        merge_sort(array, start, mid)
        merge_sort(array, mid + 1, end)
        merge(array, start, mid, end)
    }
end


merge(array, start, mid, end)
begin
    left_len = mid - start + 1
    right_len = end - mid
    left_arr[left_len]
    right_arr[right_len]
    For i = (0 to left_len) {
        left_arr[i] = array[start + i]
    }
    For i = (0 to right_len) {
        right_arr[i] = array[mid + I + 1]
    }
    left_index = 0
    right_index = 0
    For i = (start to end + 1) {
        If (left_index < left_len and right_index < right_len) {
            If (left_arr[left_index] ⩽ right_arr[right_index]) {
                array[i] = left_arr[left_index]
                left_index = left_index + 1
            }
            Else if (left_arr[left_index] > right_arr[right_index]) {
                array[i] = right_arr[right_index]
                right_index = right_index + 1
            }
        }
        Else if (left_index < left_len) {
            array[i] = left_arr[left_index]
            left_index = left_index + 1
        }
        Else if (right_index < right_len) {
            array[i] = right_arr[right_index]
            right_index = right_index + 1
        }
    }
end
```

Code

Quicksort

```
// 3. Write a program to implement QuickSort Algorithm

// Included Libraries
#include <stdio.h>   // IO and other operations
```

```c
#include <stdlib.h>  // Random number generation
#include <time.h>    // Time based operations

#define MAX_LENGTH   20    // Maximum length of array
#define MAX_ELEM     1000  // Maximum element in array
#define SORT_ARRAY   1     // Menu driven program options
#define RESEED_ARRAY 2     // Menu driven program options
#define EXIT         3     // Menu driven program options

// Macro for swapping elements of array
#define SWAP(TYPE, A, B) \
    TYPE TEMP = A;       \
    A         = B;       \
    B         = TEMP;

// Function Declarations
void gen_rand_arr(int*, const int);     // Generates random array
void print_arr(const int*, const int);  // Prints Array
int  partition(int*,
               const int,
               const int,
               const int,
               int*);  // Partitions array according to pivot
void quick_sort(int*,
                const int,
                const int,
                const int,
                int*);  // Sorts array via QuickSort algorithm

int
main(int argc, char** argv) {
    int       array[MAX_LENGTH];  // Array in use in the program
    const int len = MAX_LENGTH;   // Length of array
    int       pass;               // Number of passes of algorithm
    int       choice;  // Variable used to declare choice in menu driven program

    srand(time(NULL));  // Takes current time as seed for `rand()`

    do {
        pass = 0;  // At start of iteration, number of passes is set to zero

        fprintf(stdout, "\nBase Array:\n");
        gen_rand_arr(array, len);  // Generates an array of random integers
        print_arr(array, len);

        // Menu driven program
        fprintf(stdout, "1. Sort Array\n");
        fprintf(stdout, "2. Reseed Array\n");
        fprintf(stdout, "3. Exit\n");
        fprintf(stdout, "Enter Choice: ");
        fscanf(stdin, "%d", &choice);

        switch ( choice ) {
        case SORT_ARRAY: {
            quick_sort(array, len, 0, len - 1, &pass);

            fprintf(stdout, "Sorted Array:\n");
            print_arr(array, len);
        } break;

        case RESEED_ARRAY: {
            srand(time(NULL));
        } break;

        case EXIT: {
        } break;
        }
    } while ( choice != 3 );

    return 0;
}

// Adds `len` numbers less than MAX_NUM to array
void
gen_rand_arr(int* array, const int len) {
    for ( int i = 0; i < len; i++ ) {
        array[i] = rand() % MAX_ELEM;
    }
}

// Prints first `len` elements of array
void
print_arr(const int* array, const int len) {
    for ( int i = 0; i < len; i++ ) {
```

```
        fprintf(stdout, "%4d", array[i]);
    }

    fprintf(stdout, "\n");
}

// Divides array into two subarrays around a pivot, with first subarray being
// elements lower than pivot, and second subarray being elements greater than
// pivot
int
partition(int*      array,
          const int len,
          const int start,
          const int end,
          int*      pass) {
    int pivot_index = end;          // Pivot location
    int i           = (start - 1);  // Running index of elements

    for ( int j = start; j < end; j++ ) {
        if ( array[j] <
             array[pivot_index] ) {  // If element is lower than pivot, it is
                                     // swapped with the running element
            i++;

            SWAP(int, array[i], array[j]);
        }
    }

    SWAP(int, array[i + 1],
         array[end]);  // Swap pivot index to its correct location

    (*pass) += 1;
    fprintf(stdout, "Pass %2d:", *pass);
    print_arr(array, len);

    return i + 1;  // Return location of pivot
}

// Sorts array in ascending order via QuickSort
void
quick_sort(int*      array,
           const int len,
           const int start,
           const int end,
           int*      pass) {
    int index;

    if ( start < end ) {
        index =
            partition(array, len, start, end, pass);  // Gets partition index

        quick_sort(array, len, start, index - 1,
                   pass);  // Sorts subarray lower than pivot
        quick_sort(array, len, index + 1, end,
                   pass);  // Sorts subarray greater than pivot
    }
}
```

Mergesort

```
// 3-2. Write a program to implement Merge Sort Algorithm

// Included Libraries
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Random number generation
#include <time.h>    // Time based operations

#define MAX_LENGTH   20    // Maximum length of array
#define MAX_ELEM     1000  // Maximum element in array
#define SORT_ARRAY   1     // Menu driven program options
#define RESEED_ARRAY 2     // Menu driven program options
#define EXIT         3     // Menu driven program options

// Macro for swapping elements of array
#define SWAP(TYPE, A, B) \
    TYPE TEMP = A;       \
    A         = B;       \
    B         = TEMP;

// Function declarations
void gen_rand_arr(int*, const int);      // Generates random array
void print_arr(const int*, const int);  // Prints array
void merge_sort(int*,
```

```
                const int,
                const int,
                const int,
                int*);  // Sorts array via merge sort algorithm
void merge(int*,
           const int,
           const int,
           const int,
           int*);  // Merges sorted subarrays

int
main(int argc, char** argv) {
    int       array[MAX_LENGTH];  // Array in use in the program
    const int len = MAX_LENGTH;   // Length of array
    int       pass;               // Number of passes of algorithm
    int       choice;  // Variable used to declare choice in menu driven program

    srand(time(NULL));

    do {
        pass = 0;  // At the start of iteration, number of passes is set to zero

        fprintf(stdout, "\nBase Array:\n");
        gen_rand_arr(array, len);  // Generates an array of random integers
        print_arr(array, len);

        // Menu driven program
        fprintf(stdout, "1. Sort Array\n");
        fprintf(stdout, "2. Reseed Array\n");
        fprintf(stdout, "3. Exit\n");
        fprintf(stdout, "Enter Choice: ");
        fscanf(stdin, "%d", &choice);

        switch ( choice ) {
        case SORT_ARRAY: {
            merge_sort(array, len, 0, len - 1, &pass);

            fprintf(stdout, "Sorted Array:\n");
            print_arr(array, len);
        } break;

        case RESEED_ARRAY: {
            srand(time(NULL));
        } break;

        case EXIT: {
        } break;
        }
    } while ( choice != 3 );

    return 0;
}

// Adds `len` numbers less than MAX_NUM to array
void
gen_rand_arr(int* array, const int len) {
    for ( int i = 0; i < len; i++ ) {
        array[i] = rand() % MAX_ELEM;
    }
}

// Prints first `len` elements of array
void
print_arr(const int* array, const int len) {
    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%4d", array[i]);
    }

    fprintf(stdout, "\n");
}

// Successively subdivides array into subarrays half the size of the previous,
// until the subarray is one element long. After this, subarrays are
// successively merged in a sorted manner, giving the sorted array.
void
merge_sort(int*      array,
           const int len,
           const int start,
           const int end,
           int*      pass) {
    if ( start < end ) {
        int mid = (start + end) / 2;  // Gets middle element of subarray

        merge_sort(
```

```
            array, len, start, mid,
            pass);  // Executes merge sort algorithm on subarray before middle
        merge_sort(
            array, len, mid + 1, end,
            pass);  // Executes merge sort algorithm on subarray after middle
        merge(array, start, mid, end,
              pass);  // Merges sorted subarray to form larger array

        (*pass) += 1;

        fprintf(stdout, "Pass %3d: ", *pass);
        print_arr(array, len);
    }
}

// Merges subarrays in a sorted manner
void
merge(int* array, const int start, const int mid, const int end, int* pass) {
    int left_len  = mid - start + 1;  // Length of left subarray
    int right_len = end - mid;        // Length of right subarray

    int* left_arr;   // Left subarray
    int* right_arr;  // Right subarray

    left_arr  = ( int* ) (malloc(
         (left_len) * sizeof(int)));  // Allocates memory for the left subarray
    right_arr = ( int* ) (malloc(
        (right_len) * sizeof(int)));  // Allocates memory for the right subarray

    // Populating left subarray
    for ( int i = 0; i < left_len; i++ ) {
        left_arr[i] = array[start + i];
    }

    // Populating right subarray
    for ( int i = 0; i < right_len; i++ ) {
        right_arr[i] = array[mid + i + 1];
    }

    int left_index  = 0;  // Running index of left array elements
    int right_index = 0;  // Running index of right array elements

    // Loop which generates array with left and right subarray elements arranged
    // in a sorted manner
    for ( int i = start; i < end + 1; i++ ) {
        if ( left_index < left_len && right_index < right_len ) {
            if ( left_arr[left_index] <= right_arr[right_index] ) {
                array[i] = left_arr[left_index++];
            } else if ( left_arr[left_index] > right_arr[right_index] ) {
                array[i] = right_arr[right_index++];
            }
        } else if ( left_index < left_len ) {
            array[i] = left_arr[left_index++];
        } else if ( right_index < right_len ) {
            array[i] = right_arr[right_index++];
        }
    }

    // Frees allocated memory for the left and right subarrays
    free(left_arr);
    free(right_arr);
}
```

Analysis

Quicksort

QuickSort algorithm works on a pivot-based approach. It first picks a pivot element, and rearranges the array such that it the elements lower than the pivot are on one side of it, and elements greater than the pivot are on the other side of it, in other words, placing the pivot element at its correct location in the array. This algorithm continues to sort the subarrays on either side of the pivot in the same way until the complete array is sorted.
Consider an array of length n. The pivot is considered to be the last element of the subarray.
Say $T(n)$ is the worst-case time taken by the complete algorithm. To find the time taken by the quicksort algorithm, we must find the time taken by the `partition` subroutine. Say we input an array of n elements into the subroutine, all elements of the array are split around the pivot, giving two subproblems with the total size of $n-1$ elements. The subroutine also has its own time taken of order $\Theta(n)$, as it has to iterate all elements of the array. Therefore, the algorithm has an overall worst-case time of:

$$T(n) = \max\{T(q) + T(n-1-q) | 0 \leq q \leq n-1\} + \Theta(n)$$

Assuming $T(n) \leq cn^2$ for some $c > 0$.

$$\begin{aligned} T(n) &\leq \max\{cq^2 + c(n-1-q)^2 | 0 \leq q \leq n-1\} + \Theta(n) \\ &= c \cdot \max\{q^2 + (n-1-q)^2 | 0 \leq q \leq n-1\} + \Theta(n) \end{aligned}$$

Consider:

$$q^2 + (n-1+q)^2 = q^2 + (n-1)^2 + q^2 - 2q(n-1)$$
$$= (n-1)^2 + 2q(q-(n-1))$$

Since $0 \leq q \leq n-1$, $q-(n-1) \leq 0$, which implies $2q(q-(n-1)) \leq 0$. Therefore,

$$q^2 + (n-1-q)^2 \leq (n-1)^2$$

Therefore,

$$\begin{aligned} T(n) &\leq c(n-1)^2 + \Theta(n) \\ &\leq cn^2 - c(2n-1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

Thus, the required worst-case time complexity is $\Theta(n^2)$.

Mergesort

Merge Sort algorithm works on the divide and conquer algorithmic design. It successively subdivides the array into successively smaller subarrays, until a subarray with a length of one is reached. Following this, the subarrays are successively merged maintaining the order of the elements, giving the sorted array.
Consider an array of length n.
Merge Sort algorithm consists of three steps; Divide, Conquer and Combine. In the divide step, the middle of the subarray is computed, which can be done in constant time. In the conquer step, two subproblems are recursively solved, each of size $\frac{n}{2}$. Finally, in the combine step, the merge subroutine is executed, which takes linear time as it is iterating and populating the array in one pass. Therefore, the algorithm has an overall time complexity of:

$$\begin{aligned} T(n) &= \Theta(1) + 2T(\frac{n}{3}) + \Theta(n) \\ &= 2T(\frac{n}{3}) + \Theta(n) \end{aligned}$$

By the master theorem, we can conclude that:

$$T(n) = \Theta(n \log(n))$$

Thus, the required time complexity is $T(n) = \Theta(n \log(n))$.

Kruskal's and Prim's

Algorithms

Kruskal's

> Start $\text{MST\_KRUSKAL}(G, w)$
> $A = \emptyset$
> For each vertex $v \in G.V$
> $\text{MAKE\_SET}(v)$
> Create a single list of the edges in $G.E$
> Sort the list of edges into monotonically increasing order by weight w
> If $\text{FIND\_SET(u)} \neq FIND\_SET(v)$
> $A = A \cup \{(u, v)\}$
> $\text{UNION}(u, v)$
> Return A
> Stop

Prim's

> Start $\text{MST\_PRIM}(G, w, r)$
> For each vertex $u \in G.V$
> $u.key = \infty$
> $u.\pi = \text{NULL}$
> $r.key = 0$
> $Q = \emptyset$
> For each vertex $u \in G.V$
> $\text{INSERT}(Q, u)$
> While $Q \neq \emptyset$
> $U = \text{EXTRACT\_MIN}(Q)$
> For each vertex $v \in G.Adj[u]$
> If $v \in Q$ and $w(u, v) < v.key$
> $v.\pi = u$
> $v.key = w(u, v)$
> $\text{DECREASE\_KEY}(Q, v, w(u, v))$
> Stop

Pseudocode

Kruskal's

```
mst_kruskal(graph)
    begin
    sort(graph.edges)
    allocate(min_span_tree, rows = graph.node_count, cols = graph.node_count)
    sum = 0

    For i = (0 to graph.node_count - 1) {
        For j = (0 to graph.node_count - 1) {
```

```
            min_span_tree[i][j] = 0
        }
    }

    For i = (0 to graph.node_count - 1) {
        allocate(vertex_sets[i], i)
    }

    For i = (0 to graph.node_count - 1) {
        start     = graph.edges[i].start
        end       = graph.edges[i].end

        union(vertex_sets[start], vertex_sets[end])

        min_span_tree[vertex_sets[start]][vertex_sets[end]] = graph.edges[i].weight
        min_span_tree[vertex_sets[end]][vertex_sets[start]] = graph.edges[i].weight
    }

    sum = 0

    For i = (0 to graph.node_count - 1) {
        For j = (0 to graph.node_count - 1) {
            sum += min_span_tree[i][j]
        }
    }

    return sum
end
```

Prim's

```
mst_prim(graph) {
    allocate(min_span_tree, rows = graph.node_count, cols = graph.node_count)

    For i = (0 to graph.node_count - 1) {
        For j = (0 to graph.node_count - 1) {
            min_span_tree[i][j] = 0
        }
    }

    For i = (0 to graph.node_count - 1) {
        vertices[i].vertex = i
        vertices[i].key    = INFINITY
        vertices[i].parent = NULL
    }

    vertices[root].key = 0

    while((vertex_u = extract_min(vertices, graph.node_count))) {
        For i = (0 to graph.node_count - 1) {
            If (vertex_u has adjacent vertex_v) {
                weight_u_v = weight(vertex_u, vertex_v)

                If (vertices[i].key > weight_u_v) {
                    vertices[i].parent = vertex_u
                    vertices[i].key    = weight_u_v
                }
            }
        }
    }

    sum = 0

    For i = (0 to graph.node_count - 1) {
        sum += vertices[i].key
    }

    return sum
}
```

Code

Kruskal's

```c
// 4. Write a program to implement Kruskal's Algorithm

// Included Libraries
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Memory operations and atoi()
#include <string.h>  // memcpy() function

// Macro Definitions
#define NOT_FOUND      -1  // Element not found in set
#define PRINT_BLANKS   1   // Print blanks to show no edge in adjacency matrix
#define PRINT_ZEROS    0   // Print zeros to show no edge in adjacency matrix
#define NODE_COUNT_STR argv[1]  // Command line argument for number of nodes
#define EDGE_COUNT_STR argv[2]  // Command line argument for number of edges
/**
 * @brief Macro to Swap Edges
 * @param A First Edge
 * @param B Second Edge
 */
#define EDGESWAP(A, B)    \
    struct Edge temp = A; \
    A                = B; \
    B                = temp;

/** @brief Structure to implement a set data structure */
struct Set {
    int* data;  // Array storing elements in Set
    int  size;  // Number of elements stored in Set
};

/** @brief Structure to implement a graph edge */
struct Edge {
    int start;   // Source Node of edge
    int end;     // Destination Node of edge
    int weight;  // Weight/Cost of edge
};

/** @brief Structure to implement a graph data structure */
struct Graph {
    int          node_count;  // Number of nodes in graph
    int          edge_count;  // Number of edges in graph
    struct Edge* edges;       // Array holding all edges associated with graph
    int**        adj;         // Adjacency matrix of graph
};

/**
 * @brief Searches the @p array for @p query using recursive linear search
 * @param array Array to be searched
 * @param len Length of @p array
 * @param query Element to be searched for
 * @param loc Location currently being searched for
 * @return Location where element is found
 */
int           linear_search(int const* array,
                            int const  len,
                            int const  query,
                            int const  loc);
/**
 * @brief Initializes the set data structure with elements in @p array
 * @param array Array of elements being input to the set
 * @param len Length of @p array
 * @return Set containing elements in @p array
 */
struct Set*   set_init(int const* array, int const len);
/**
 * @brief Appends @p element to array
 * @param set Set to which @p element is to be appended
 * @param element Element which is appended to @p set
 */
void          set_append(struct Set* set, int const element);
/**
 * @brief Displays the specified set
 * @param set Set to be displayed
 */
void          set_display(struct Set const* set);
/**
 * @brief Performs union operation on @p set1 and @p set2
 * @param set1 First set
 * @param set2 Second set
 * @return Union of @p set1 and @p set2
 */
struct Set*   set_union(struct Set const* set1, struct Set const* set2);
/**
 * @brief Prints a specified square matrix
```

```c
 * @param matrix Square matrix to be printed
 * @param len Length of @p matrix
 */
void          print_matrix(int** matrix, int const len);
/**
 * @brief Prints all edges of a specified graph
 * @param graph Graph whose edges are to be printed
 */
void          print_edges(struct Graph const* graph);
/**
 * @brief Initializes the graph data structure with @p node_count nodes
 * @param node_count Number of nodes of the graph
 * @return Graph with @p node_count nodes and no edges
 */
struct Graph* graph_init(int const node_count);
/**
 * @brief Populates the @p graph using the @p edges between the nodes
 * @param graph Graph to be populated
 * @param edges Collection of edges to be added to @p graph
 * @param edge_count Number of @p edges to be added
 */
void          populate_graph(struct Graph*      graph,
                             struct Edge const* edges,
                             int const          edge_count);
/**
 * @brief Sorts the edges in non-decreasing order using bubble sort.
 * @param graph Graph whose edges are to be sorted
 */
void          edge_sort(struct Graph* graph);
/**
 * @brief Finds the minimum spanning tree of the @p graph using Kruskal's
 * algorithm
 * @param graph Graph whose minimum spanning tree is to be found
 */
void          mst_kruskal(struct Graph const* graph);

int
main(int argc, char** argv) {
    // If no nodes or edges are mentioned, skip execution of program
    if ( ! NODE_COUNT_STR || ! EDGE_COUNT_STR || argc != 3 ) {
        return 0;
    }

    // Initialize and populate graph
    struct Graph*     graph   = graph_init(atoi(NODE_COUNT_STR));
    const struct Edge edges[] = {
        {.start = 0, .end = 1,  .weight = 4},
        {.start = 0, .end = 7,  .weight = 8},
        {.start = 1, .end = 2,  .weight = 8},
        {.start = 1, .end = 7, .weight = 11},
        {.start = 2, .end = 3,  .weight = 7},
        {.start = 3, .end = 4,  .weight = 9},
        {.start = 3, .end = 5, .weight = 14},
        {.start = 4, .end = 5, .weight = 10},
        {.start = 5, .end = 6,  .weight = 2},
        {.start = 6, .end = 7,  .weight = 1},
        {.start = 6, .end = 8,  .weight = 6},
        {.start = 7, .end = 8,  .weight = 7}
    };

    populate_graph(graph, edges, atoi(EDGE_COUNT_STR));

    fprintf(stdout, "Adjacency Matrix of Graph:\n");
    print_matrix(graph->adj, graph->node_count);

    edge_sort(graph);

    fprintf(stdout, "Edges: \n");
    print_edges(graph);

    // Execute Prim's Algorithm and print Minimum Spanning Tree
    mst_kruskal(graph);

    {  // Free memory occupied by graph
        free(graph->edges);

        for ( int i = 0; i < graph->edge_count; i++ ) {
            free(graph->adj[i]);
        }

        free(graph->adj);
        free(graph);
    }
```

```c
    return 0;
}

int
linear_search(int const* array, int const len, int const query, int const loc) {
    if ( loc < 0 ) {  // If the location being searched for goes out of bounds,
                      // this means that the element does not exist
        return NOT_FOUND;
    } else if ( array[loc] == query ) {
        return loc;
    }

    return linear_search(array, len, query, loc - 1);
}

struct Set*
set_init(int const* array, int const len) {
    // Allocate and define an empty set
    struct Set* set = ( struct Set* ) malloc(sizeof(struct Set));
    set->data       = NULL;
    set->size       = 0;

    // If elements are specified in initialization
    if ( array != NULL ) {
        // Allocate and define set elements
        set->data = ( int* ) malloc(sizeof(int));

        for ( int i = 0; i < len; i++ ) {
            set_append(set, array[i]);
        }
    }

    return set;
}

void
set_append(struct Set* set, int const element) {
    // If element does not already exist in set, increase the size of the set,
    // reallocate it to more memory and define it to include the element
    if ( linear_search(set->data, set->size, element, set->size - 1) == -1 ) {
        set->size++;

        set->data = ( int* ) realloc(set->data, set->size * sizeof(int));
        set->data[set->size - 1] = element;
    }
}

void
set_display(struct Set const* set) {
    fprintf(stdout, "%d: ", set->size);

    for ( int i = 0; i < set->size; i++ ) {
        fprintf(stdout, "%d\t", set->data[i]);
    }

    fprintf(stdout, "\n");
}

struct Set*
set_union(struct Set const* set1, struct Set const* set2) {
    // Create result array as duplicate of set1, and then append all elements of
    // set2 to it
    struct Set* result = set_init(set1->data, set1->size);

    for ( int i = 0; i < set2->size; i++ ) {
        set_append(result, set2->data[i]);
    }

    return result;
}

void
print_matrix(int** matrix, int const len) {
    fprintf(stdout, "+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|   |");
```

```c
    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%2d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%2d ", i);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == 0 ) {
                if ( PRINT_BLANKS ) {
                    fprintf(stdout, "   ");
                } else if ( PRINT_ZEROS ) {
                    fprintf(stdout, "%2d ", matrix[i][j]);
                }
            } else {
                fprintf(stdout, "%2d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n");
}

void
print_edges(struct Graph const* graph) {
    for ( int i = 0; i < graph->edge_count; i++ ) {
        fprintf(stdout, "%d--%d-->%d\n", graph->edges[i].start,
                graph->edges[i].weight, graph->edges[i].end);
    }
}

struct Graph*
graph_init(int const node_count) {
    // Allocate and define an empty graph with only nodes and no edges
    struct Graph* graph = ( struct Graph* ) malloc(sizeof(struct Graph));
    graph->node_count   = node_count;
    graph->adj          = ( int** ) malloc(graph->node_count * sizeof(int*));
    graph->edge_count   = 0;
    graph->edges        = NULL;

    for ( int i = 0; i < graph->node_count; i++ ) {
        graph->adj[i] = ( int* ) malloc(graph->node_count * sizeof(int));
    }

    for ( int i = 0; i < graph->node_count; i++ ) {
        for ( int j = 0; j < graph->node_count; j++ ) {
            graph->adj[i][j] = 0;
        }
    }

    return graph;
}
```

```c
void
populate_graph(struct Graph*      graph,
               struct Edge const* edges,
               int const          edge_count) {
    // Copy the edges to be a part of the graph object, and then add the edges
    // to the graph by updating the adjacency matrix
    graph->edge_count = edge_count;

    graph->edges =
        ( struct Edge* ) malloc(graph->edge_count * sizeof(struct Edge));

    memcpy(graph->edges, edges, edge_count * sizeof(struct Edge));

    for ( int i = 0; i < edge_count; i++ ) {
        graph->adj[graph->edges[i].start][graph->edges[i].end] =
            graph->edges[i].weight;
        graph->adj[graph->edges[i].end][graph->edges[i].start] =
            graph->edges[i].weight;
    }
}

void
edge_sort(struct Graph* graph) {
    for ( int i = 0; i < graph->edge_count; i++ ) {
        for ( int j = 0; j < graph->edge_count - i - 1; j++ ) {
            if ( graph->edges[j].weight > graph->edges[j + 1].weight ) {
                EDGESWAP(graph->edges[j], graph->edges[j + 1]);
            }
        }
    }
}

void
mst_kruskal(struct Graph const* graph) {
    // Allocate an empty adjacency matrix to represent an empty minimum spanning
    // tree
    int** min_span_tree = ( int** ) malloc(graph->node_count * sizeof(int*));
    int   sum           = 0;

    for ( int i = 0; i < graph->node_count; i++ ) {
        min_span_tree[i] = ( int* ) malloc(graph->node_count * sizeof(int));

        for ( int j = 0; j < graph->node_count; j++ ) {
            min_span_tree[i][j] = 0;
        }
    }

    // Create sets for each vertex
    struct Set** vertex_sets =
        ( struct Set** ) malloc(graph->node_count * sizeof(struct Set*));

    for ( int i = 0; i < graph->node_count; i++ ) {
        vertex_sets[i] = set_init(&i, 1);
    }

    fprintf(stdout, "\nVertex Sets: \n");

    for ( int i = 0; i < graph->node_count; i++ ) {
        set_display(vertex_sets[i]);
    }

    // Iterate through the edges. If the endpoints are in different sets, make a
    // union of these sets, and add the edge to the MST. If the endpoints are in
    // the same set, a cycle will be formed, and this edge should be ignored.
    // After all the edges are examined, we get the adjacency matrix of the MST.
    for ( int i = 0; i < graph->edge_count; i++ ) {
        int start     = graph->edges[i].start;
        int end       = graph->edges[i].end;
        int start_set = -1;
        int end_set   = -1;

        fprintf(stdout, "\nEdge Under Consideration: %d--%d-->%d\n",
                graph->edges[i].start, graph->edges[i].weight,
                graph->edges[i].end);

        for ( int j = 0; j < graph->node_count; j++ ) {
            if ( vertex_sets[j] == NULL ) {
                continue;
            }

            start_set =
                (linear_search(vertex_sets[j]->data, vertex_sets[j]->size,
                               start, vertex_sets[j]->size - 1) == -1)
                    ? start_set
```

```c
                    : j;
            end_set = (linear_search(vertex_sets[j]->data, vertex_sets[j]->size,
                                     end, vertex_sets[j]->size - 1) == -1)
                        ? end_set
                        : j;
        }

        if ( start_set != end_set ) {
            fprintf(stdout,
                    "Sets containing start and end of edge are different, "
                    "performing union\n");
            vertex_sets[start_set] =
                set_union(vertex_sets[start_set], vertex_sets[end_set]);

            vertex_sets[end_set] = NULL;

            min_span_tree[start_set][end_set] = graph->edges[i].weight;
            min_span_tree[end_set][start_set] = graph->edges[i].weight;
        } else {
            fprintf(stdout,
                    "Sets containing start and end are the same, continuing\n");
        }

        fprintf(stdout, "\nVertex Sets: \n");

        for ( int j = 0; j < graph->node_count; j++ ) {
            if ( vertex_sets[j] != NULL ) {
                set_display(vertex_sets[j]);
            }
        }
    }

    fprintf(stdout, "\nAdjacency Matrix of Minimum Spanning Tree: \n");

    print_matrix(min_span_tree, graph->node_count);

    for ( int i = 0; i < graph->node_count; i++ ) {
        for ( int j = 0; j < graph->node_count; j++ ) {
            sum += min_span_tree[i][j];
        }
    }

    sum /= 2;

    fprintf(stdout, "Total Weight: %d\n", sum);

    for ( int i = 0; i < graph->node_count; i++ ) {
        free(vertex_sets[i]->data);
        free(vertex_sets[i]);
    }

    free(vertex_sets);

    for ( int i = 0; i < graph->node_count; i++ ) {
        free(min_span_tree[i]);
    }

    free(min_span_tree);
}
```

Prim's

```c
// 4-2. Write a program to implement Prim's Algorithm

// Included Libraries
#include <limits.h>  // INT_MAX
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Memory operations and atoi()
#include <string.h>  // memcpy()

// Macro Definitions
#define NOT_FOUND      -1  // Element not found in set
#define PRINT_BLANKS   1   // Print blanks to show no edge in adjacency matrix
#define PRINT_ZEROS    0   // Print zeros to show no edge in adjacency matrix
#define NODE_COUNT_STR argv[1]  // Command line argument for number of nodes
#define EDGE_COUNT_STR argv[2]  // Command line argument for number of edges
#define ROOT_STR       argv[3]  // Command line argument for root node

/** @brief Structure to implement a graph edge */
struct Edge {
    int start;   // Source Node of edge
    int end;     // Destination Node of edge
    int weight;  // Weight/Cost of edge
```

```c
};

/** @brief Structure to implement a graph data structure */
struct Graph {
    int          node_count;  // Number of nodes in graph
    int          edge_count;  // Number of edges in graph
    struct Edge* edges;       // Array holding all edges associated with graph
    int**        adj;         // Adjacency matrix of graph
};

/** @brief Structure to implement a graph vertex */
struct Vertex {
    int vertex;   // Name of the current vertex
    int key;      // Key value of the current vertex
    int parent;   // Parent of the current vertex
    int in_list;  // Whether the vertex is in the vertices list
};

/**
 * @brief Prints a specified square matrix
 * @param matrix Square matrix to be printed
 * @param len Length of @p matrix
 * @endcode
 */
void          print_matrix(int** matrix, int const len);
/**
 * @brief Prints all edges of a specified graph
 * @param graph Graph whose edges are to be printed
 */
void          print_edges(struct Graph const* graph);
/**
 * @brief Initializes the graph data structure with @p node_count nodes
 * @param node_count Number of nodes of the graph
 * @return Graph with @p node_count nodes and no edges
 */
struct Graph* graph_init(int const node_count);
/**
 * @brief Populates the @p graph using the @p edges between the nodes
 * @param graph Graph to be populated
 * @param edges Collection of edges to be added to @p graph
 * @param edge_count Number of @p edges to be added
 */
void          populate_graph(struct Graph*      graph,
                             struct Edge const* edges,
                             int const          edge_count);
/**
 * @brief Extracts the minimum vertex from @p vertices
 * @param vertices Vertex array from which minimum vertex is extracted
 * @param len Length of @p vertices
 * @return Minimum vertex from @p vertices
 */
int           vertex_extract_min(struct Vertex const* vertices, int const len);
/**
 * @brief Finds the minimum spanning tree of the @p graph using Prim's algorithm
 * @param graph Graph whose minimum spanning tree is to be found
 */
void          mst_prim(struct Graph const* graph, struct Vertex const* root);

int
main(int argc, char** argv) {
    // If no nodes or edges are mentioned, skip execution of program
    if ( argc != 4 ) {
        return 0;
    }

    // Initialize and populate graph
    struct Graph*     graph = graph_init(atoi(NODE_COUNT_STR));
    struct Vertex*    root  = ( struct Vertex* ) malloc(sizeof(struct Vertex));
    const struct Edge edges[] = {
        {.start = 0, .end = 1,  .weight = 4},
        {.start = 0, .end = 7,  .weight = 8},
        {.start = 1, .end = 2,  .weight = 8},
        {.start = 1, .end = 7, .weight = 11},
        {.start = 2, .end = 3,  .weight = 7},
        {.start = 3, .end = 4,  .weight = 9},
        {.start = 3, .end = 5, .weight = 14},
        {.start = 4, .end = 5, .weight = 10},
        {.start = 5, .end = 6,  .weight = 2},
        {.start = 6, .end = 7,  .weight = 1},
        {.start = 6, .end = 8,  .weight = 6},
        {.start = 7, .end = 8,  .weight = 7}
    };

    populate_graph(graph, edges, atoi(EDGE_COUNT_STR));
```

```c
    fprintf(stdout, "Adjacency Matrix of Graph:\n");
    print_matrix(graph->adj, graph->node_count);

    fprintf(stdout, "Edges:\n");
    print_edges(graph);

    // Initialize root vertex
    root->vertex = atoi(ROOT_STR);
    root->parent = -1;
    root->key    = INT_MAX;

    // Execute Prim's Algorithm and print Minimum Spanning Tree
    mst_prim(graph, root);

    {  // Free memory occupied by graph
        free(graph->edges);

        for ( int i = 0; i < graph->edge_count; i++ ) {
            free(graph->adj[i]);
        }

        free(graph->adj);
        free(graph);
        free(root);
    }

    return 0;
}

void
print_matrix(int** matrix, int const len) {
    fprintf(stdout, "+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|   |");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%2d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%2d ", i);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == 0 ) {
                if ( PRINT_BLANKS ) {
                    fprintf(stdout, "   ");
                } else if ( PRINT_ZEROS ) {
                    fprintf(stdout, "%2d ", matrix[i][j]);
                }
            } else {
                fprintf(stdout, "%2d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }
```

```c
    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n");
}

void
print_edges(struct Graph const* graph) {
    for ( int i = 0; i < graph->edge_count; i++ ) {
        fprintf(stdout, "%d--%d-->%d\n", graph->edges[i].start,
                graph->edges[i].weight, graph->edges[i].end);
    }
}

struct Graph*
graph_init(int const node_count) {
    // Allocate and define an empty graph with only nodes and no edges
    struct Graph* graph = ( struct Graph* ) malloc(sizeof(struct Graph));
    graph->node_count   = node_count;
    graph->adj = ( int** ) malloc(( size_t ) graph->node_count * sizeof(int*));
    graph->edge_count = 0;
    graph->edges      = NULL;

    for ( int i = 0; i < graph->node_count; i++ ) {
        graph->adj[i] =
            ( int* ) malloc(( size_t ) graph->node_count * sizeof(int));
    }

    for ( int i = 0; i < graph->node_count; i++ ) {
        for ( int j = 0; j < graph->node_count; j++ ) {
            graph->adj[i][j] = 0;
        }
    }

    return graph;
}

void
populate_graph(struct Graph*      graph,
               struct Edge const* edges,
               int const          edge_count) {
    // Copy the edges to be a part of the graph object, and then add the edges
    // to the graph by updating the adjacency matrix
    graph->edge_count = edge_count;

    graph->edges = ( struct Edge* ) malloc(( size_t ) graph->edge_count *
                                           sizeof(struct Edge));

    memcpy(graph->edges, edges, ( size_t ) edge_count * sizeof(struct Edge));

    for ( int i = 0; i < edge_count; i++ ) {
        graph->adj[graph->edges[i].start][graph->edges[i].end] =
            graph->edges[i].weight;
        graph->adj[graph->edges[i].end][graph->edges[i].start] =
            graph->edges[i].weight;
    }
}

int
vertex_extract_min(struct Vertex const* vertices, int const len) {
    struct Vertex min_vertex;
    int           min_vertex_index = -1;

    min_vertex.key = INT_MAX;

    for ( int i = 0; i < len; i++ ) {
        if ( vertices[i].in_list == 1 ) {
            if ( min_vertex.key > vertices[i].key ) {
                min_vertex       = vertices[i];
                min_vertex_index = i;
            }
        }
    }

    return min_vertex_index;
}
```

```c
void
mst_prim(struct Graph const* graph, struct Vertex const* root) {
    // Allocate an empty adjacency matrix to represent an empty minimum spanning
    // tree
    int** min_span_tree =
        ( int** ) malloc(( size_t ) graph->node_count * sizeof(int*));

    for ( int i = 0; i < graph->node_count; i++ ) {
        min_span_tree[i] =
            ( int* ) malloc(( size_t ) graph->node_count * sizeof(int));

        for ( int j = 0; j < graph->node_count; j++ ) {
            min_span_tree[i][j] = 0;
        }
    }

    // Create list of vertices from which minimum is extracted
    int            vertex_u_index;
    struct Vertex* vertices = ( struct Vertex* ) malloc(
        ( size_t ) graph->node_count * sizeof(struct Vertex));

    for ( int i = 0; i < graph->node_count; i++ ) {
        vertices[i].vertex  = i;
        vertices[i].key     = INT_MAX;
        vertices[i].parent  = -1;
        vertices[i].in_list = 1;
    }

    vertices[root->vertex].key = 0;

    // Iterate through the vertices in the list, and keep updating the keys of
    // the vertices which are adjacent to be lower, and update the parent of the
    // vertices accordingly. Do until all the vertices in the list are iterated
    // through. Then use the parents of the vertices to make the Minimum
    // Spanning Tree.
    while ( (vertex_u_index =
                 vertex_extract_min(vertices, graph->node_count)) != -1 ) {
        fprintf(stdout, "\nVertex under examination: (Name: %2d, Key: %2d, ",
                vertices[vertex_u_index].vertex, vertices[vertex_u_index].key);

        if ( vertices[vertex_u_index].parent == -1 ) {
            fprintf(stdout, "Parent:  NULL)\n");
        } else {
            fprintf(stdout, "Parent: %2d)\n", vertices[vertex_u_index].parent);
        }

        fprintf(stdout, "Neighbours of %d: ", vertex_u_index);

        for ( int i = 0; i < graph->node_count; i++ ) {
            if ( graph->adj[i][vertex_u_index] != 0 ) {
                fprintf(stdout, "%d ", i);
            }
        }

        fprintf(stdout, "\n");

        for ( int i = 0; i < graph->node_count; i++ ) {
            if ( graph->adj[vertices[vertex_u_index].vertex][i] != 0 ) {
                int weight_u_v = graph->adj[vertices[vertex_u_index].vertex][i];

                if ( vertices[i].key > weight_u_v &&
                     vertices[i].in_list == 1 ) {
                    vertices[i].parent = vertices[vertex_u_index].vertex;
                    vertices[i].key    = weight_u_v;

                    fprintf(
                        stdout,
                        "Vertex Updated: (Name: %2d, Key: %2d, Parent: %2d)\n",
                        vertices[i].vertex, vertices[i].key,
                        vertices[i].parent);
                }

                vertices[vertex_u_index].in_list = 0;
            }
        }
    }

    fprintf(stdout, "\nFinal Vertices:\n");

    for ( int i = 0; i < graph->node_count; i++ ) {
        fprintf(stdout, "Name: %2d, Key: %2d, ", vertices[i].vertex,
                vertices[i].key);

        if ( vertices[i].parent == -1 ) {
```

```
            fprintf(stdout, "Parent:  NULL\n");
        } else {
            fprintf(stdout, "Parent: %2d\n", vertices[i].parent);
        }
    }

    int sum = 0;

    for ( int i = 0; i < graph->node_count; i++ ) {
        sum += vertices[i].key;
    }

    for ( int i = 0; i < graph->node_count; i++ ) {
        if ( vertices[i].parent != -1 ) {
            min_span_tree[vertices[i].vertex][vertices[i].parent] =
                graph->adj[vertices[i].vertex][vertices[i].parent];
            min_span_tree[vertices[i].parent][vertices[i].vertex] =
                graph->adj[vertices[i].parent][vertices[i].vertex];
        }
    }

    fprintf(stdout, "\nAdjacency Matrix of Minimum Spanning Tree:\n");

    print_matrix(min_span_tree, graph->node_count);

    fprintf(stdout, "Sum: %d", sum);

    for ( int i = 0; i < graph->node_count; i++ ) {
        free(min_span_tree[i]);
    }

    free(min_span_tree);
    free(vertices);
}
```

Analysis

Kruskal's

Kruskal's algorithm is an edge-based approach to finding the minimum spanning tree of a given graph. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) with the lowest weight. Kruskal's algorithm qualifies as a greedy algorithm because at each step if adds to the forest an edge with the lowest possible weight.
The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the specific implementation of the disjoint-set data structure. The asymptotically fastest known implementation is the disjoint-set-forest implementation. Initializing the set A takes $O(1)$ time, creating a single list of edges takes $O(V + E)$ time (which is $O(E)$ because G is connected), and the time to sort the edges is $O(E \log E)$. The for loop performs $O(E)$ FIND_SET $(UNION)$ and $(UNION)$ operations on the disjoint-set forest. Along with the $|V|$ MAKE_SET operations, these disjoint-set operations take a total of $O((V + E)\alpha(V))$ time, where α is a very slowly growing function. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\log V) = O(\log E)$, the total running time of Kruskal's algorithm is $O(E \log E)$. Observing that $|E| < |V|^2$, we have $\log |E| = O(\log E)$, and so we can restate the running time of Kruskal's algorithm as $O(E \log V)$.

Prim's

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap, including a way to build a map between vertices and their corresponding heap elements. The BUILD_MIN_HEAP procedure can perform in $O(V)$ time. In fact, there is no need to call BUILD_MIN_HEAP. One can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the while loop executes $|V|$ times, and since each EXTRACT_MIN operation takes $O(\log V)$ time, the total time for all calls to EXTRACT_MIN is $O(V \log V)$. The for loop $O(E)$ times altogether since the sum of the lengths of all adjacency lists is $2|E|$. Within the for loop, the test for membership in Q can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE_KEY takes $O(\log V)$ time. Thus, the total time for Prim's algorithm is $O(V \log V + E \log V) = O(E \log V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

Greedy Fractional Knapsack

Algorithm

> Start $\text{FRACTIONAL\_KNAPSACK}(V, W, W_0)$
> For each item value and weight pair $(v, w) \in (V, W)$
> $P[i] = \frac{v}{w}$
> $\text{SORT\_DESCENDING}(P)$
> $i = 1$
> While $W_0 > 0$
> $\text{amount} = \min(W_0, W[i])$
> $\text{solution}[i] = \text{amount} \times P[i]$
> $W_0 = W_0 - \text{amount}$
> $i = i + 1$
> Return solution
> Stop

Pseudocode

```
fractional_knapsack(items, capacity)
    begin
    For i = (0 to len(items) - 1) {
```

```
        items[i].price = items[i].value / items[i].weight
    }

    profit = 0
    i      = 0

    Sort_Items(items, descending)

    While (capacity > 0) {
        amount          = min(capacity, items[i].weight)
        items_in_bag[i] = item
        capacity        = capacity - amount
        profit          = profit + items[i].price
        i               = i + 1

        if (i > len(items)) {
            break;
        }
    }

    return items_in_bag, profit
end
```

Code

```c
// 5. Write a program to solve the Fractional Knapsack problem using Greedy
// Approach.

// Included Libraries
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Memory operations and atoi()

// Macro Definitions
#define ITEM_COUNT_STR   argv[1]  // Command line argument for number of items
#define BAG_CAPACITY_STR argv[2]  // Command line argument for bag capacity
/**
 * @brief Macro to Swap @p A and @p B
 * @param A First Item
 * @param B Second Item
 */
#define ITEMSWAP(A, B)    \
    struct Item temp = A; \
    A                = B; \
    B                = temp;

/** @brief Structure to model a Knapsack problem item */
struct Item {
    int   name;    // Name of the item
    int   weight;  // Weight of the item
    int   value;   // Total value of the item
    float price;   // Price of the value per unit weight
};

/** @brief Structure to model an Item in a Bag */
struct Item_In_Bag {
    struct Item item;           // Item details
    int         weight_in_bag;  // Weight of the item stored in bag
};

/**
 * @brief Print a summary table for the @p items
 * @param items Array of items
 * @param len Number of @p items
 * @param print_prices Whether to print prices or not. Takes values 1 (Print
 * prices of each item) or 0 (Do not print prices of each item)
 * @param in_bag Whether the item is in the bad or not. Used only if @p
 * print_prices is 1. Takes values 1 (Print amount of each item in the bag) or 0
 * (Do not print amount of each item in the bag)
 * @param amounts_in_bag Amount of @p items in the bag. Works only if @p in_bag
 * is 1.
 */
void  print_items(struct Item* items,
                  int          len,
                  int          print_prices,
                  int          in_bag,
                  int*         amounts_in_bag);
/**
 * @brief Sort the @p items by their respective prices in decreasing order via
 * bubble sort
 * @param items Array of items
```

```c
 * @param len Number of @p items
 */
void  sort_items(struct Item* items, int len);
/**
 * @brief Solves the Fractional Knapsack Problem via greedy approach on the @p
 * items array
 * @param items Array of items
 * @param len Number of @p items
 * @param capacity Total capacity of the bag
 * @return Maximum profit achievable from the @p items array
 */
float fractional_knapsack(struct Item* items, int len, int capacity);

int
main(int argc, char** argv) {
    // If no items or bag capacity are mentioned, skip execution of program
    if ( ! ITEM_COUNT_STR || ! BAG_CAPACITY_STR || argc != 3 ) {
        return 0;
    }


    // Initialize items
    struct Item items[] = {
        {.name = 1,  .weight = 5,  .value = 30, .price = 0.0},
        {.name = 2, .weight = 10,  .value = 20, .price = 0.0},
        {.name = 3, .weight = 20, .value = 100, .price = 0.0},
        {.name = 4, .weight = 30,  .value = 90, .price = 0.0},
        {.name = 5, .weight = 40, .value = 160, .price = 0.0}
    };

    fprintf(stdout, "Items Given:\n");
    print_items(items, atoi(ITEM_COUNT_STR), 0, 0, NULL);

    fprintf(stdout, "Bag Capactiy: %d\n", atoi(BAG_CAPACITY_STR));

    // Get the maximum profit possible from the list of items by solving the
    // Fractional Knapsack Problem
    fprintf(stdout, "Total Profit Accumulated: %.2f",
            fractional_knapsack(items, atoi(ITEM_COUNT_STR),
                                atoi(BAG_CAPACITY_STR)));

    return 0;
}

void
print_items(struct Item* items,
            int          len,
            int          print_prices,
            int          in_bag,
            int*         amounts_in_bag) {
    if ( ! print_prices ) {
        fprintf(stdout, "+------+--------+-------+\n");
        fprintf(stdout, "| Item | Weight | Value |\n");
        fprintf(stdout, "+------+--------+-------+\n");

        for ( int i = 0; i < len; i++ ) {
            fprintf(stdout, "| %3d  |  %3d   |  %3d  |\n", items[i].name,
                    items[i].weight, items[i].value);
        }

        fprintf(stdout, "+------+--------+-------+\n");
    } else if ( ! in_bag ) {
        fprintf(stdout, "+------+--------+-------+-------+\n");
        fprintf(stdout, "| Item | Weight | Value | Price |\n");
        fprintf(stdout, "+------+--------+-------+-------+\n");

        for ( int i = 0; i < len; i++ ) {
            fprintf(stdout, "| %3d  |  %3d   |  %3d  |  %2.2f |\n",
                    items[i].name, items[i].weight, items[i].value,
                    items[i].price);
        }

        fprintf(stdout, "+------+--------+-------+-------+\n");
    } else {
        fprintf(
            stdout,
            "+------+--------+-------+-------+---------------+--------------"
            "-------+\n");
        fprintf(stdout,
                "| Item | Weight | Value | Price | Weight In Bag |  Profit "
                "Accumulated |\n");
        fprintf(
            stdout,
            "+------+--------+-------+-------+---------------+--------------"
```

```
            "-------+\n");

        for ( int i = 0; i < len; i++ ) {
            int j;

            for ( j = 0; j < len; j++ ) {
                if ( j + 1 == items[i].name ) {
                    break;
                }
            }
            fprintf(stdout,
                    "| %3d  |  %3d   |  %3d  |  %3.2f |      %3d      |        "
                    "%6.2f       |\n",
                    items[i].name, items[i].weight, items[i].value,
                    items[i].price, amounts_in_bag[j],
                    items[i].price * ( float ) amounts_in_bag[j]);
        }

        fprintf(
            stdout,
            "+------+--------+-------+-------+---------------+--------------"
            "-------+\n");
    }
}

void
sort_items(struct Item* items, int len) {
    for ( int i = 0; i < len; i++ ) {
        for ( int j = 0; j < len - i - 1; j++ ) {
            if ( items[j].price < items[j + 1].price ) {
                ITEMSWAP(items[j], items[j + 1]);
            }
        }
    }
}

float
fractional_knapsack(struct Item* items, int len, int capacity) {
    // Initialise array of items in the bag and the profit
    struct Item_In_Bag* items_in_bag = NULL;
    float               profit       = 0;

    // Get prices per unit weight for each item
    for ( int i = 0; i < len; i++ ) {
        items[i].price = ( float ) items[i].value / ( float ) items[i].weight;
    }

    sort_items(items, len);

    fprintf(stdout, "Items after sorting:\n");
    print_items(items, len, 1, 0, NULL);

    int i = 0;

    int base_capacity = capacity;

    // Iterate through the array of items until the bag has no capacity left. In
    // each iteration, add as much as possible of the maximum price item to the
    // bag and update the profit accordingly.
    while ( capacity > 0 ) {
        fprintf(stdout, "Capacity left in bag: %d\n", capacity);
        fprintf(stdout,
                "Item being Considered: (Name: %3d, Weight: %3d, Value: %3d, "
                "Price: %3.2f)\n",
                items[i].name, items[i].weight, items[i].value, items[i].price);

        int amount = (capacity < items[i].weight) ? capacity : items[i].weight;

        if ( amount == items[i].weight ) {
            fprintf(stdout, "Item Completely Placed in bag\n");
        } else {
            fprintf(stdout, "%d / %d fraction of item placed in bag\n", amount,
                    items[i].weight);
        }

        items_in_bag = ( struct Item_In_Bag* ) realloc(
            items_in_bag, ( size_t ) (i + 1) * sizeof(struct Item_In_Bag));

        items_in_bag[i].item          = items[i];
        items_in_bag[i].weight_in_bag = amount;

        capacity -= amount;

        profit += ( float ) amount * items[i].price;
```

```
        fprintf(stdout, "Total profit accumulated so far: %.2f + %.2f = %.2f\n",
                profit - items[i].price * ( float ) amount,
                items[i].price * ( float ) amount, profit);

        i++;

        if ( i >= len ) {
            break;
        }

        fprintf(stdout, "\n");
    }

    fprintf(stdout, "Capacity left in bag: %d\n\n", capacity);

    fprintf(stdout, "Summary of Items in Bag:\n");

    int* amounts_in_bag = ( int* ) malloc(( size_t ) len * sizeof(int));

    for ( int j = 0; j < len; j++ ) {
        amounts_in_bag[j] = 0;
    }

    for ( int j = 0; j < i; j++ ) {
        amounts_in_bag[items_in_bag[j].item.name - 1] =
            items_in_bag[j].weight_in_bag;
    }

    print_items(items, len, 1, 1, amounts_in_bag);

    fprintf(stdout, "\nKnapsack Diagram:\n");

    int solution_item_count = i;

    for ( int j = 0; j < base_capacity; j++ ) {
        fprintf(stdout, "-");
    }

    fprintf(stdout, "\n");

    for ( int l = 0; l < 3; l++ ) {
        int amount_filled = 0;
        int k             = 0;

        for ( int j = 0; j < base_capacity; j++ ) {
            if ( j == 0 || j == base_capacity - 1 ) {
                fprintf(stdout, "|");
            } else if ( items_in_bag[k].weight_in_bag + amount_filled == j &&
                        k < solution_item_count ) {
                fprintf(stdout, "|");
                amount_filled += items_in_bag[k].weight_in_bag;
                k++;
            } else {
                if ( l == 1 ) {
                    fprintf(stdout, "%d", items_in_bag[k].item.name);
                } else {
                    fprintf(stdout, " ");
                }
            }
        }

        fprintf(stdout, "\n");
    }

    for ( int j = 0; j < base_capacity; j++ ) {
        fprintf(stdout, "-");
    }

    fprintf(stdout, "\n");

    free(amounts_in_bag);

    return profit;
}
```

Analysis

The fractional knapsack problem solution used here starts with a for loop in which every item's price is calculated via division, all of which can be completed in $O(n)$ time, where n is the number of items. Sorting the items takes at least $O(n \log n)$ time, and placing the items in the bag takes $O(n)$ time. Therefore, the time complexity of the algorithm is:

$$O(n) + O(n \log n) + O(n) = O(n \log n)$$

Therefore, the algorithm has a time complexity of $O(n \log n)$

Dynamic 0/1 Knapsack

Algorithm

> Start $\text{ZERO\_ONE\_KNAPSACK}(v, w, n, M)$
> For $w = 0$ to M do
> $c[0, w] = 0$
> For $i = 1$ to n do
> If $w_i \leq w$ and $v_i + c[i-1, w-w_i] > c[i-1, w]$
> Then $c[i, w] = v_i + c[i-1, w-w_i]$
> $keep[i, w] = 1$
> Else
> $c[i, w] = c[i-1, w]$
> $keep[i, w] = 0$
> $k = M$
> For $i = n$ to 1
> If $keep[i, k] = 1$
> Print i
> $k = k - w_i$
> Return $c[n, w]$
> Stop

Pseudocode

```
zero_one_knapsack(items, capacity)
begin
    For i = (0 to len(items)) {
        For j = (0 to capacity) {
            If (i == 0 or j == 0) {
                C[i][j] = 0
                Keep[i][j] = 0
            }
        }
    }

    For i = (1 to len(items)) {
        For j = (1 to capacity) {
            If (items[i - 1].weight <= j and items[i - 1].value + C[i - 1][j - items[i - 1].weight] > C[i - 1][j]) {
                C[i][j] = items[i - 1].value + C[i - 1][j - items[i - 1].weight]
                Keep[i][j] = 1
            } Else {
                C[i][j] = C[i - 1][j]
                Keep[i][j] = 0
            }
        }
    }

    K = capacity

    For i = (len to 1) {
        If (Keep[i][k] == 1) {
            K = K - items[i - 1].weight
        }
    }

    Return C[len(items)][capacity]

end
```

Code

```
// Write a program to solve the Zero-One Knapsack Problem using Dynamic
// Programming Approach

// Included Libraries
#include <stdio.h>    // IO and other operations
#include <stdlib.h>   // Memory operations

// Macro Definitions
#define ITEM_COUNT_STR   argv[1]  // Command line argument for number of items
#define BAG_CAPACITY_STR argv[2]  // Command line argument for bag capacity
#define PRINT_LINE(COLS)                    \
    fprintf(stdout, "\n+");                 \
    for ( int i = 0; i < cols * 5; i++ ) { \
        if ( (i + 1) % 5 == 0 ) {           \
```

```c
            fprintf(stdout, "+");           \
        } else {                            \
            fprintf(stdout, "-");           \
        }                                   \
    }                                       \
    fprintf(stdout, "\n");

/** @brief Structure to model a Knapsack problem item */
struct Item {
    int name;     // Name of the item
    int weight;   // Weight of the item
    int value;    // Total value of the item
};

/** @brief Structure containing solution matrices for the execution */
struct Solution {
    int C;  // Contains maximum value of any of the subset of items {1, 2, ...,
            // i} of at most capacity
    int keep;  // Contains information of which items are kept in the knapsack
};

/**
 * @brief Prints given matrix of @p solution of `struct Solution` type
 * @param solution Given matrix
 * @param rows Number of rows in @p solution
 * @param cols Number of columns in @p solution
 */
void  print_matrix(struct Solution** solution, int const rows, int const cols);
/**
 * @brief Print a summary table for the @p items
 * @param items Array of items
 * @param len Number of @p items
 */
void  print_items(struct Item* items, int len);
/**
 * @brief Solves the Zero-One Knapsack Problem via Dynamic Programming approach
 * on the @p items array
 * @param items Array of items
 * @param len Number of @p items
 * @param capacity Total capacity of the bag
 * @return Maximum profit achievable from the @p items array
 */
float zero_one_knapsack(struct Item* items, int len, int capacity);

int
main(int argc, char** argv) {
    // If no items or bag capacity are mentioned, skip execution of program
    if ( ! ITEM_COUNT_STR || ! BAG_CAPACITY_STR || argc != 3 ) {
        return 0;
    }

    // Number of items and capacity of bag
    int len      = atoi(ITEM_COUNT_STR);
    int capacity = atoi(BAG_CAPACITY_STR);

    // Initialize array of items
    struct Item items[] = {
        {.name = 1, .weight = 5, .value = 10},
        {.name = 2, .weight = 4, .value = 40},
        {.name = 3, .weight = 6, .value = 40},
        {.name = 4, .weight = 3, .value = 50}
    };

    fprintf(stdout, "Given Items:\n");
    print_items(items, len);

    fprintf(stdout, "\n");

    // Get the maximum profit possible form the list of items by solving the
    // Zero-One Knapsack Problem
    fprintf(stdout, "\nTotal Profit Earned: %.2f",
            zero_one_knapsack(items, len, capacity));

    return 0;
}

void
print_matrix(struct Solution** solution, int rows, int cols) {
    fprintf(stdout, "C Matrix:");

    PRINT_LINE(cols);

    for ( int i = 0; i < rows; i++ ) {
        fprintf(stdout, "|");
```

```
        for ( int j = 0; j < cols; j++ ) {
            if ( solution[i][j].C == -1 ) {
                fprintf(stdout, "    |");
            } else {
                fprintf(stdout, " %2d |", solution[i][j].C);
            }
        }

        PRINT_LINE(cols);
    }

    fprintf(stdout, "\nKeep Matrix:");

    PRINT_LINE(cols);

    for ( int i = 0; i < rows; i++ ) {
        fprintf(stdout, "|");

        for ( int j = 0; j < cols; j++ ) {
            if ( solution[i][j].keep == -1 ) {
                fprintf(stdout, "    |");
            } else {
                fprintf(stdout, " %2d |", solution[i][j].keep);
            }
        }

        PRINT_LINE(cols);
    }

    fprintf(stdout, "\n");
}

void
print_items(struct Item* items, int len) {
    fprintf(stdout, "+------+--------+-------+\n");
    fprintf(stdout, "| Item | Weight | Value |\n");
    fprintf(stdout, "+------+--------+-------+\n");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "| %3d  |  %3d   |  %3d  |\n", items[i].name,
                items[i].weight, items[i].value);
    }

    fprintf(stdout, "+------+--------+-------+\n");
}

float
zero_one_knapsack(struct Item* items, int len, int capacity) {
    // Allocating C and Keep Matrices
    struct Solution** solution =
        ( struct Solution** ) malloc((len + 1) * sizeof(struct Solution*));

    for ( int i = 0; i < (len + 1); i++ ) {
        solution[i] = ( struct Solution* ) malloc((capacity + 1) *
                                                  sizeof(struct Solution));
    }

    // Initializing C and Keep Matrices
    for ( int i = 0; i < (len + 1); i++ ) {
        for ( int j = 0; j < (capacity + 1); j++ ) {
            if ( i == 0 || j == 0 ) {
                solution[i][j].C    = 0;
                solution[i][j].keep = 0;
            } else {
                solution[i][j].C    = -1;
                solution[i][j].keep = -1;
            }
        }
    }

    print_matrix(solution, len + 1, capacity + 1);

    // Running Zero-One Knapsack Algorithm on the given item list
    for ( int i = 1; i < (len + 1); i++ ) {
        fprintf(stdout,
                "Item under consideration:\nName: %2d, Value: %2d, Weight: "
                "%2d\n",
                items[i - 1].name, items[i - 1].value, items[i - 1].weight);
        for ( int j = 1; j < (capacity + 1); j++ ) {
            if ( (items[i - 1].weight <= j) &&
                 (items[i - 1].value +
                      solution[i - 1][j - items[i - 1].weight].C >
                  solution[i - 1][j].C) ) {
```

```
                solution[i][j].C = items[i - 1].value +
                                   solution[i - 1][j - items[i - 1].weight].C;
                solution[i][j].keep = 1;
            } else {
                solution[i][j].C    = solution[i - 1][j].C;
                solution[i][j].keep = 0;
            }
        }

        print_matrix(solution, len + 1, capacity + 1);
    }

    int k = capacity;

    fprintf(stdout, "Items kept in knapsack: ");

    // Determining Items Kept in Bag
    for ( int i = len; i >= 1; i-- ) {
        if ( solution[i][k].keep == 1 ) {
            fprintf(stdout, "%d ", i);
            k = k - items[i - 1].weight;
        }
    }

    // Returning Total Capacity of Bag
    return solution[len][capacity].C;
}
```

Analysis

Say the solution to the zero-one knapsack problem takes $T(n)$ time. The zero-one knapsack problem solution used here starts with initializing the matrices C and $Keep$ of size $(\text{No. of items} + 1) \times (\text{Capacity} + 1)$. Both matrices are assigned values of 0 in the first row and column, a $\Theta(n)$ operation. Populating the matrices is of $\Theta(n^2)$ complexity, as for each cell of the matrix we need to perform an operation of $\Theta(1)$ to find the value to be populated. Finally, finding the list of items which are to be placed in the knapsack is a $\Theta(n)$ operation. Therefore, we can conclude:

$$T(n) = \Theta(n) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$$

Therefore, the algorithm has a time complexity of $\Theta(n^2)$.

Breadth First and Depth First Search

Algorithms

Breadth First Search

> Start $\text{BREADTH\_FIRST\_SEARCH}(G, s)$
> For each vertex $u \in G.V - s$ do
> $u.\text{color} = \text{WHITE}$
> $u.d = \infty$
> $u.\pi = \text{NIL}$
> $s.\text{color} = \text{GRAY}$
> $s.d = 0$
> $s.\pi = \text{NIL}$
> $Q = \emptyset$
> $\text{ENQUEUE}(Q, s)$
> While $Q \neq \emptyset$
> $u = \text{DEQUEUE}(Q)$
> For each vertex $v \in G.Adj[u]$
> If $v.\text{color} = \text{WHITE}$
> $v.\text{color} = \text{GRAY}$
> $v.d = u.d + 1$
> $v.\pi = u$
> $\text{ENQUEUE}(Q, v)$
> $u.\text{color} = \text{BLACK}$
> Stop

Depth First Search

> Start $\text{DEPTH\_FIRST\_SEARCH}(G, s)$
> For each vertex $u \in G.V$ do
> $u.\text{color} = \text{WHITE}$
> $u.\pi = \text{NIL}$
> $time = 0$
> For each vertex $u \in G.V$
> If $u.\text{color} = \text{WHITE}$
> $\text{DFS\_VISIT}(G, u)$
> Stop

> Start $\text{DFS\_VISIT}(G, u)$
> $time = time + 1$
> $u.d = 0$

> $u.\text{color} = \text{GRAY}$
> For each vertex $v \in G.Adj[u]$
> If $v.\text{color} = \text{WHITE}$
> $v.\pi = u$
> $\text{DFS\_VISIT}(G, u)$
> $\text{time} = \text{time} + 1$
> $u.f = \text{time}$
> $u.\text{COLOR} = \text{BLACK}$
> Stop

Pseudocode

Breadth First Search

```
breadth_first_search(graph, source)
begin
    graph.vertices[source].color = GRAY
    graph.vertices[source].distance = 0
    graph.vertices[source].parent = NULL

    queue = Queue()
    visit_index = 0

    enqueue(queue, graph.vertices[source])

    While (len(queue) > 0) {
        u = dequeue(queue)

        For i = (0 to graph.node_count) {
            If (graph.adj[u][i] != 0 and graph.vertices[i].color == WHITE) {
                graph.vertices[i].color = GRAY
                graph.vertices[i].distance = u.distance + 1
                graph.vertices[i].parent = u

                enqueue(queue, graph.vertices[i])
            }
        }

        u.color = BLACK
        visit_order[visit_index++] = u;
    }

    Display visit_order
end
```

Depth First Search

```
depth_first_search(graph)
begin
    time = 0

    For i = (0 to graph.node_count) {
        If (graph.vertices[i].color == WHITE) {
            depth_first_search_visit(graph, i, time)
        }
    }
end

depth_first_search_visit(graph, vertex, time)
begin
    time = 0

    graph.vertices[vertex].discovery = time
    graph.vertices[vertex].color = GRAY

    For i = (0 to graph.node_count) {
        If (graph.adj[graph.vertices[vertex]][i] != 0 and graph.vertices[i].color = WHITE) {
            graph.vertices[i].parent = graph.vertices[vertex]
            depth_first_search_visit(graph, i, time)
        }
    }

    time++

    graph.vertices[vertex].finish = time
```

```
    graph.vertices[vertex].color = BLACK
end
```

Code

Breadth First Search

```
// 7. Write a program to implement Breadth-First Search on a Graph

// Included Libraries
#include <assert.h>  // assert()
#include <limits.h>  // INT_MAX
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Memory operations
#include <string.h>  // memcpy()

// Macro Definitions
#define PRINT_ZEROS    0  // Print zeros to show no edge in adjacency matrix
#define PRINT_BLANKS   1  // Print blanks to show no edge in adjacency matrix
#define NODE_COUNT_STR argv[1]  // Command line argument for number of nodes
#define EDGE_COUNT_STR argv[2]  // Command line argument for number of edges

/** @brief Available vertex colors */
enum Color {
    WHITE = 1,
    GRAY  = 2,
    BLACK = 3
};

/** @brief Structure to implement a graph edge */
struct Edge {
    int start;  // Source Node of edge
    int end;    // Destination Node of edge
};

/** @brief Structure to implement a graph vertex */
struct Vertex {
    int            vertex;    // Name of the current vertex
    enum Color     color;     // Color of the current vertex
    int            distance;  // Distance of vertex from source
    struct Vertex* parent;    // Parent of vertex
};

/** @brief Structure to implement a graph data structure */
struct Graph {
    int            node_count;  // Number of nodes in graph
    int            edge_count;  // Number of edges in graph
    struct Edge*   edges;       // Array holding all edges associated with graph
    struct Vertex* vertices;    // Array holding all vertices in graph
    int**          adj;         // Adjacency matrix of graph
};

/** @brief Structure to implement a node in a linked queue */
struct Node {
    struct Vertex* vertex;  // Element at node
    struct Node*   next;    // Link to next node
};

/** @brief Structure to implement a linked queue */
struct Queue {
    struct Node* front;  // Front of queue
    struct Node* rear;   // End of queue
    int          len;    // Length of queue
};

/**
 * @brief Prints a specified square matrix
 * @param matrix Square matrix to be printed
 * @param len length of @p matrix
 */
void           print_matrix(int** matrix, int const len);
/**
 * @brief Prints all edges of a specified graph
 * @param graph Graph whose edges are to be printed
 */
void           print_edges(struct Graph const* graph);
/**
 * @brief Prints all vertices with their colors
 * @param graph Graph whose vertices are to be printed
 */
void           print_vertices(struct Graph const* graph);
/**
 * @brief Initializes a queue
```

```c
 * @return Empty queue
 */
struct Queue*  queue_init();
/**
 * @brief Enqueue @p vertex to @p queue
 * @param queue Queue to which element is added
 * @param vertex Vertex to enqueue
 */
void           enqueue(struct Queue* queue, struct Vertex* vertex);
/**
 * @brief Dequeue vertex from @p queue and return it
 * @param queue Queue to which element is added
 * @return Dequeued element
 */
struct Vertex* dequeue(struct Queue* queue);
/**
 * @brief Initializes the graph data structure with @p node_count nodes
 * @param node_count Number of nodes of the graph
 * @return Graph with @p node_count nodes and no edges
 */
struct Graph*  graph_init(int const node_count);
/**
 * @brief Clear the memory allocated for graph
 * @param graph Graph to be cleared
 */
void           graph_dealloc(struct Graph* graph);
/**
 * @brief Populates the @p graph using the @p edges between the nodes
 * @param graph Graph to be populated
 * @param edges Collection of edges to be added to @p graph
 * @param edge_count Number of @p edges to be added
 */
void           graph_populate(struct Graph*      graph,
                              struct Edge const* edges,
                              int const          edge_count);
/**
 * @brief Perform Breadth First Search on @p graph
 * @param graph Graph to be searched
 * @param source Node to be used as the source for the search
 */
void breadth_first_search(struct Graph const* graph, int const source);

int
main(int argc, char** argv) {
    // If no nodes are mentioned, skip execution of program
    if ( argc != 3 ) {
        return 0;
    }

    // Initialize and populate graph
    struct Graph*     graph   = graph_init(atoi(NODE_COUNT_STR));
    struct Edge const edges[] = {
        {.start = 0, .end = 1},
        {.start = 0, .end = 7},
        {.start = 1, .end = 2},
        {.start = 1, .end = 7},
        {.start = 2, .end = 3},
        {.start = 3, .end = 4},
        {.start = 3, .end = 5},
        {.start = 4, .end = 5},
        {.start = 5, .end = 6},
        {.start = 6, .end = 7},
        {.start = 6, .end = 8},
        {.start = 7, .end = 8}
    };

    graph_populate(graph, edges, atoi(EDGE_COUNT_STR));

    fprintf(stdout, "Adjacency Matrix of Graph:\n");
    print_matrix(graph->adj, graph->node_count);

    fprintf(stdout, "Edges:\n");
    print_edges(graph);

    breadth_first_search(graph, 0);

    graph_dealloc(graph);

    return 0;
}

void
print_matrix(int** matrix, int const len) {
    fprintf(stdout, "+");
```

```
    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|   |");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%2d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%2d ", i);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == 0 ) {
                if ( PRINT_BLANKS ) {
                    fprintf(stdout, "   ");
                } else if ( PRINT_ZEROS ) {
                    fprintf(stdout, "%2d ", matrix[i][j]);
                }
            } else {
                fprintf(stdout, "%2d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n");
}

void
print_edges(struct Graph const* graph) {
    for ( int i = 0; i < graph->edge_count; i++ ) {
        fprintf(stdout, "%d--->%d\n", graph->edges[i].start,
                graph->edges[i].end);
    }
}

void
print_vertices(struct Graph const* graph) {
    for ( int i = 0; i < graph->node_count; i++ ) {
        fprintf(stdout, "Vertex %2d: Color %2d Distance %2d\n",
                graph->vertices[i].vertex, graph->vertices[i].color,
                graph->vertices[i].distance);
    }
}

struct Queue*
queue_init() {
    struct Queue* queue = ( struct Queue* ) malloc(sizeof(struct Queue));
```

```c
    queue->front        = NULL;
    queue->rear         = NULL;
    queue->len          = 0;

    return queue;
}

void
enqueue(struct Queue* queue, struct Vertex* vertex) {
    struct Node* new_node = ( struct Node* ) malloc(sizeof(struct Node));

    new_node->vertex = vertex;
    new_node->next   = NULL;

    if ( queue->front == NULL ) {
        queue->front = new_node;
        queue->rear  = new_node;
    } else {
        queue->rear->next = new_node;
        queue->rear       = new_node;
    }

    queue->len++;
}

struct Vertex*
dequeue(struct Queue* queue) {
    struct Node* temp = ( struct Node* ) malloc(sizeof(struct Node));

    // Quit if queue is empty
    assert(queue->front != NULL);

    struct Vertex* vertex = queue->front->vertex;

    temp         = queue->front;
    queue->front = queue->front->next;

    if ( queue->front == NULL ) {
        queue->rear = NULL;
    }

    free(temp);

    queue->len--;

    return vertex;
}

struct Graph*
graph_init(int const node_count) {
    // Allocate and define an empty graph with only nodes and no edges
    struct Graph* graph = ( struct Graph* ) malloc(sizeof(struct Graph));
    graph->node_count   = node_count;
    graph->edge_count   = 0;
    graph->edges        = NULL;
    graph->adj          = ( int** ) malloc(graph->node_count * sizeof(int*));
    graph->vertices =
        ( struct Vertex* ) malloc(node_count * sizeof(struct Vertex));

    for ( int i = 0; i < graph->node_count; i++ ) {
        graph->adj[i] =
            ( int* ) malloc(( size_t ) graph->node_count * sizeof(int));
        graph->vertices[i].vertex   = i;
        graph->vertices[i].color    = WHITE;
        graph->vertices[i].distance = INT_MAX;
        graph->vertices[i].parent   = NULL;
    }

    for ( int i = 0; i < graph->node_count; i++ ) {
        for ( int j = 0; j < graph->node_count; j++ ) {
            graph->adj[i][j] = 0;
        }
    }

    return graph;
}

void
graph_dealloc(struct Graph* graph) {
    for ( int i = 0; i < graph->node_count; i++ ) {
        free(graph->adj[i]);
    }

    free(graph->adj);
```

```c
    free(graph->edges);
    free(graph->vertices);
    free(graph);
}

void
graph_populate(struct Graph*      graph,
               struct Edge const* edges,
               int const          edge_count) {
    // Copy the edges to be a part of the graph object, and then add the edges
    // to the graph by updating the adjacency matrix
    graph->edge_count = edge_count;

    graph->edges =
        ( struct Edge* ) malloc(graph->edge_count * sizeof(struct Edge));

    memcpy(graph->edges, edges, edge_count * sizeof(struct Edge));

    for ( int i = 0; i < edge_count; i++ ) {
        graph->adj[graph->edges[i].start][graph->edges[i].end] = 1;
        graph->adj[graph->edges[i].end][graph->edges[i].start] = 1;
    }
}

void
print_queue(struct Queue* queue) {
    struct Node* node = queue->front;

    fprintf(stdout, "%d: ", queue->len);

    while ( node != NULL ) {
        fprintf(stdout, "%d ", node->vertex->vertex);

        node = node->next;
    }

    fprintf(stdout, "\n");
}

void
breadth_first_search(struct Graph const* graph, int const source) {
    // Assign node as source
    graph->vertices[source].color    = GRAY;
    graph->vertices[source].distance = 0;
    graph->vertices[source].parent   = NULL;

    struct Queue* queue = queue_init();
    int* visit_order    = ( int* ) malloc(graph->node_count * sizeof(int));
    int  visit_index    = 0;

    // Add node to queue
    enqueue(queue, &graph->vertices[source]);

    fprintf(stdout, "Queue Status:\n");

    // Until all nodes are encountered, iterate through them using a queue, thus
    // in a breadth wise manner
    while ( queue->len > 0 ) {
        print_queue(queue);
        struct Vertex* u = dequeue(queue);

        for ( int i = 0; i < graph->node_count; i++ ) {
            if ( graph->adj[u->vertex][i] != 0 &&
                 graph->vertices[i].color == WHITE ) {
                graph->vertices[i].color    = GRAY;
                graph->vertices[i].distance = u->distance + 1;
                graph->vertices[i].parent   = u;

                enqueue(queue, &graph->vertices[i]);
            }
        }

        u->color = BLACK;

        visit_order[visit_index++] = u->vertex;
    }

    fprintf(stdout, "\nBreadth First Visit Order:\n");

    for ( int i = 0; i < graph->node_count; i++ ) {
        fprintf(stdout, "%d\t", visit_order[i]);
    }
}
```

Depth First Search

```
// 7-2. Write a program to implement Depth-First Search on a Graph

// Included Libraries
#include <stdio.h>   // IO and other operations
#include <stdlib.h>  // Memory operations
#include <string.h>  // Memcpy

// Macro Definitions
#define PRINT_ZEROS    0  // Print zeros to show no edge in adjacency matrix
#define PRINT_BLANKS   1  // Print blanks to show no edge in adjacency matrix
#define NODE_COUNT_STR argv[1]  // Command line argument for number of nodes
#define EDGE_COUNT_STR argv[2]  // Command line argument for number of nodes

/** @brief Available vertex colors */
enum Color {
    WHITE = 1,
    GRAY  = 2,
    BLACK = 3
};

/** @brief Structure to implement a graph edge */
struct Edge {
    int start;   // Source Node of edge
    int end;     // Destination Node of edge
    int weight;  // Weight/Cost of edge
};

/** @brief Structure to implement a graph vertex */
struct Vertex {
    int            vertex;     // Name of the current vertex
    enum Color     color;      // Color of the current vertex
    struct Vertex* parent;     // Parent of vertex
    int            discovery;  // Discovery time of vertex
    int            finish;     // Finish time of vertex
};

/** @brief Structure to implement a graph data structure */
struct Graph {
    int            node_count;  // Number of nodes in graph
    int            edge_count;  // Number of edges in graph
    struct Edge*   edges;       // Array holding all edges associated with graph
    struct Vertex* vertices;    // Array holding all vertices in graph
    int**          adj;         // Adjacency matrix of graph
};

/**
 * @brief Prints a specified square matrix
 * @param matrix Square matrix to be printed
 * @param len length of @p matrix
 * @endcode
 */
void          print_matrix(int** matrix, int const len);
/**
 * @brief Prints all edges of a specified graph
 * @param graph Graph whose edges are to be printed
 */
void          print_edges(struct Graph const* graph);
/**
 * @brief Prints all vertices with their colors
 * @param graph Graph whose vertices are to be printed
 */
void          print_vertices(struct Graph const* graph);
/**
 * @brief Initializes the graph data structure with @p node_count nodes
 * @param node_count Number of nodes of the graph
 * @return Graph with @p node_count nodes and no edges
 */
struct Graph* graph_init(int const node_count);
/**
 * @brief Clear the memory allocated for graph
 * @param graph Graph to be cleared
 */
void          graph_dealloc(struct Graph* graph);
/**
 * @brief Populates the @p graph using the @p edges between the nodes
 * @param graph Graph to be populated
 * @param edges Collection of edges to be added to @p graph
 * @param edge_count Number of @p edges to be added
 */
void          graph_populate(struct Graph*      graph,
                             struct Edge const* edges,
```

```
                              int const          edge_count);
/**
 * @brief Perform Depth First Search on @p graph
 * @param graph Graph to be searched
 */
void          depth_first_search(struct Graph const* graph);
/**
 * @brief Visit suboperation for depth first search
 * @param graph Graph to be searched
 * @param vertex Vertex from which search is continued
 * @param time Current timestamp of search
 */
void          depth_first_search_visit(struct Graph const* graph,
                                       int const           vertex,
                                       int*                time);

int
main(int argc, char** argv) {
    // If no nodes are mentioned, skip execution of program
    if ( argc != 3 ) {
        return 0;
    }

    // Initialize and populate graph
    struct Graph*     graph = graph_init(atoi(NODE_COUNT_STR));
    struct Edge const edges[] = {
        {.start = 0, .end = 1},
        {.start = 0, .end = 7},
        {.start = 1, .end = 2},
        {.start = 1, .end = 7},
        {.start = 2, .end = 3},
        {.start = 3, .end = 4},
        {.start = 3, .end = 5},
        {.start = 4, .end = 5},
        {.start = 5, .end = 6},
        {.start = 6, .end = 7},
        {.start = 6, .end = 8},
        {.start = 7, .end = 8}
    };

    graph_populate(graph, edges, atoi(EDGE_COUNT_STR));

    fprintf(stdout, "Adjacency Matrix of Graph:\n");
    print_matrix(graph->adj, graph->node_count);

    fprintf(stdout, "Edges:\n");
    print_edges(graph);

    depth_first_search(graph);

    fprintf(stdout, "\nDepth First Vertex Status:\n");
    print_vertices(graph);

    graph_dealloc(graph);

    return 0;
}

void
print_matrix(int** matrix, int const len) {
    fprintf(stdout, "+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n|   |");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%2d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
```

```c
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%2d ", i);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == 0 ) {
                if ( PRINT_BLANKS ) {
                    fprintf(stdout, "   ");
                } else if ( PRINT_ZEROS ) {
                    fprintf(stdout, "%2d ", matrix[i][j]);
                }
            } else {
                fprintf(stdout, "%2d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "---");
        }
    }

    fprintf(stdout, "+\n");
}

void
print_edges(struct Graph const* graph) {
    for ( int i = 0; i < graph->edge_count; i++ ) {
        fprintf(stdout, "%d--->%d\n", graph->edges[i].start,
                graph->edges[i].end);
    }
}

void
print_vertices(struct Graph const* graph) {
    for ( int i = 0; i < graph->node_count; i++ ) {
        fprintf(stdout, "Vertex %d: Discovery Time: %2d Finish Time: %2d\n",
                graph->vertices[i].vertex, graph->vertices[i].discovery,
                graph->vertices[i].finish);
    }
}

struct Graph*
graph_init(int const node_count) {
    // Allocate and define an empty graph with only nodes and no edges
    struct Graph* graph = ( struct Graph* ) malloc(sizeof(struct Graph));
    graph->node_count   = node_count;
    graph->edge_count   = 0;
    graph->edges        = NULL;
    graph->adj          = ( int** ) malloc(graph->node_count * sizeof(int*));
    graph->vertices =
        ( struct Vertex* ) malloc(node_count * sizeof(struct Vertex));

    for ( int i = 0; i < graph->node_count; i++ ) {
        graph->adj[i] =
            ( int* ) malloc(( size_t ) graph->node_count * sizeof(int));
        graph->vertices[i].vertex    = i;
        graph->vertices[i].color     = WHITE;
        graph->vertices[i].parent    = NULL;
        graph->vertices[i].discovery = -1;
        graph->vertices[i].finish    = -1;
    }

    for ( int i = 0; i < graph->node_count; i++ ) {
        for ( int j = 0; j < graph->node_count; j++ ) {
            graph->adj[i][j] = 0;
        }
    }
```

```c
    return graph;
}

void
graph_dealloc(struct Graph* graph) {
    for ( int i = 0; i < graph->node_count; i++ ) {
        free(graph->adj[i]);
    }

    free(graph->adj);
    free(graph->edges);
    free(graph->vertices);
    free(graph);
}

void
graph_populate(struct Graph*      graph,
               struct Edge const* edges,
               int const          edge_count) {
    // Copy the edges to be a part of the graph object, and then add the edges
    // to the graph by updating the adjacency matrix
    graph->edge_count = edge_count;

    graph->edges =
        ( struct Edge* ) malloc(graph->edge_count * sizeof(struct Edge));

    memcpy(graph->edges, edges, edge_count * sizeof(struct Edge));

    for ( int i = 0; i < edge_count; i++ ) {
        graph->adj[graph->edges[i].start][graph->edges[i].end] = 1;
        // graph->adj[graph->edges[i].end][graph->edges[i].start] = 1;
    }
}

void
depth_first_search(struct Graph const* graph) {
    // Start search with timestamp as 0
    int  temp = 0;
    int* time = &temp;

    // Visit each unvisited node
    for ( int i = 0; i < graph->node_count; i++ ) {
        if ( graph->vertices[i].color == WHITE ) {
            depth_first_search_visit(graph, i, time);
        }
    }
}

void
depth_first_search_visit(struct Graph const* graph,
                         int const           vertex,
                         int*                time) {
    // Increment time per visit
    (*time)++;

    // Set node as node currently being visited and set its discovery time
    graph->vertices[vertex].discovery = *time;
    graph->vertices[vertex].color     = GRAY;

    for ( int i = 0; i < graph->node_count; i++ ) {
        // Select adjacent unvisited nodes
        if ( graph->adj[graph->vertices[vertex].vertex][i] != 0 &&
             graph->vertices[i].color == WHITE ) {
            // Set parent of every adjacent vertex
            graph->vertices[i].parent = &graph->vertices[vertex];

            // Visit every adjacent vertex
            depth_first_search_visit(graph, i, time);
        }
    }

    // Increment time when all adjacent vertices are visited
    (*time)++;

    // Set node as visited and set its finish time
    graph->vertices[vertex].finish = *time;
    graph->vertices[vertex].color  = BLACK;
}
```

Analysis

Breadth First Search

Consider an input graph to the algorithm $G = (V, E)$. We use aggregate analysis. After initialization, breadth-first search never whitens a vertex, and thus the test ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all $|V|$ adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(V + E)$. The overhead for initialization is $O(V)$, and thus total running time of the BFS procedure is $O(V + E)$. Thus breadth-first search runs in time linear in size of the adjacency-list representation of G.

Depth First Search

Consider an input graph to the algorithm $G = (V, E)$. The loops of $\text{DEPTH\_FIRST\_SEARCH}$ take $\Theta(V)$ time, exclusive of the time to execute the calls to DFS_VISIT. We use aggregate analysis. The procedure DFS_VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS_VISIT is invoked must be WHITE and the first thing DFS_VISIT does is paint vertex u GRAY. During an execution of $\text{DFS\_VISIT}(G, v)$, the loop executes $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$ and DFS_VISIT is called once per vertex, the total cost of executing the loop of DFS_VISIT is $\Theta(V + E)$. The running time of DFS is therefore $\Theta(V + E)$.

Travelling Salesperson Branch and Bound

Algorithms

> Start $\text{TSP\_BRANCH\_AND\_BOUND}(root)$
> $\text{ROW\_COL\_REDUCE}(root)$
> $temp = root$
> While $temp.childcount \neq 0$
> $\text{ADD\_CHILDREN}(temp)$
> For $i = 0$ to $temp.childcount$ do
> $\text{ROW\_COL\_REDUCE}(temp.child[i])$
> For $i = 0$ to $temp.childcount$ do
> $temp.child[i].cost = temp.child[i].cost + temp.cost$
> $temp.child[i].cost = temp.child[i].cost + root.adj[temp - 1][temp.child[i] - 1]$
> $min\_index = 0$
> For $i = 0$ to $temp.childcount$ do
> If $temp.child[i].cost < temp.child[min\_index].cost$
> $min\_index = i$
> Return $root$
> End

Pseudocode

```
tsp_branch_and_bound(root)
begin
    row_col_reduce(root)
    temp = root

    While (temp.childcount ≠ 0) {
        add_children(temp)

        For i = (0 to temp.childcount) {
            row_col_reduce(temp.child[i])
        }

        For i = (0 to temp.childcount) {
            temp.child[i].cost += temp.cost
            temp.child[i].cost += root.adj[temp - 1][temp.child[i] - 1]
        }

        min_index = 0

        For i = (0 to temp.childcount) {
            If (temp.child[i].cost < temp.child[min_index].cost) {
                min_index = i
            }
        }

        temp = temp.child[min_index]
    }

    Return root
end
```

Code

```c
// 8. Write a program to solve Travelling Salesperson Problem using Branch and
// Bound Approach

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#define MATRIX_SIZE   5
#define PRINT_SPACING 3

struct Node {
    int           index;
    int           cost;
    int**         adj;
    int           adj_len;
    struct Node** children;
    struct Node*  parent;
    int           children_count;
    int           max_children;
};

void         matrix_print(struct Node* node, bool print_children);
struct Node* node_init(int**     adj,
                       int const len,
                       int const index,
                       int const children);
void         node_dealloc(struct Node* node);
void         node_add_children(struct Node* node);
void         node_infinitize(struct Node* node, int const row, int const col);
void         node_row_col_reduce(struct Node* node);
void         tsp_branch_and_bound(struct Node* root);

int
main(int argc, char** argv) {
    int* adj[] = {
        (int[]) {INT_MAX,      20,      30,      10,      11},
         (int[]) {     15, INT_MAX,      16,       4,       2},
        (int[]) {      3,       5, INT_MAX,       2,       4},
         (int[]) {     19,       6,      18, INT_MAX,       3},
        (int[]) {     16,       4,       7,      16, INT_MAX}
    };

    struct Node* root = node_init(adj, MATRIX_SIZE, 1, MATRIX_SIZE - 1);

    tsp_branch_and_bound(root);

    node_dealloc(root);

    return 0;
}

void
matrix_print(struct Node* node, bool print_children) {
    fprintf(stdout, "+");

    for ( int i = -2; i < node->adj_len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|    |");

    for ( int i = 0; i < node->adj_len; i++ ) {
        fprintf(stdout, "%3d ", i + 1);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < node->adj_len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < node->adj_len; i++ ) {
        for ( int j = -2; j < node->adj_len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%3d ", i + 1);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }
```

```c
            if ( node->adj[i][j] == INT_MAX ) {
                fprintf(stdout, "%3c ", 'I');
            } else {
                fprintf(stdout, "%3d ", node->adj[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < node->adj_len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n");

    fprintf(stdout, "Cost: %3d\n", node->cost);

    if ( print_children == true ) {
        for ( int i = 0; i < node->children_count; i++ ) {
            fprintf(stdout, "\n(%d, %d)\n", node->index,
                    node->children[i]->index);

            matrix_print(node->children[i], true);
        }
    }
}

struct Node*
node_init(int** adj, int const len, int const index, int const children) {
    struct Node* node = ( struct Node* ) malloc(sizeof(struct Node));

    node->index          = index;
    node->adj_len        = len;
    node->cost           = 0;
    node->children       = NULL;
    node->parent         = NULL;
    node->children_count = 0;
    node->max_children   = children;
    node->adj            = ( int** ) malloc(node->adj_len * sizeof(int*));

    for ( int i = 0; i < node->adj_len; i++ ) {
        node->adj[i] = ( int* ) malloc(node->adj_len * sizeof(int));
        memcpy(node->adj[i], adj[i], node->adj_len * sizeof(int));
    }

    return node;
}

void
node_dealloc(struct Node* node) {
    if ( node == NULL ) {
        return;
    }

    for ( int i = 0; i < node->children_count; i++ ) {
        node_dealloc(node->children[i]);
    }

    for ( int i = 0; i < node->adj_len; i++ ) {
        free(node->adj[i]);
    }

    if ( node->children != NULL ) {
        free(node->children);
        node->children = NULL;
    }

    free(node->adj);
    free(node);
}

void
node_add_children(struct Node* node) {
    node->children =
        ( struct Node** ) malloc(node->max_children * sizeof(struct Node*));
```

```c
    int          index_occurance[MATRIX_SIZE] = { 0 };
    struct Node* temp                         = node;

    while ( temp != NULL ) {
        index_occurance[temp->index - 1] = 1;
        temp                             = temp->parent;
    }

    for ( int i = 0; i < node->max_children; i++ ) {
        int index = 1;

        while ( index_occurance[index - 1] == 1 ) {
            index++;
        }

        index_occurance[index - 1] = 1;

        node->children[i] =
            node_init(node->adj, node->adj_len, index, node->max_children - 1);
        node->children[i]->parent = node;
        node->children_count++;

        index++;

        node_infinitize(node->children[i], node->index - 1,
                        node->children[i]->index - 1);
    }
}

void
node_infinitize(struct Node* node, int const row, int const col) {
    for ( int i = 0; i < node->adj_len; i++ ) {
        node->adj[row][i] = INT_MAX;
        node->adj[i][col] = INT_MAX;
    }

    node->adj[col][0] = INT_MAX;
}

void
node_row_col_reduce(struct Node* node) {
    int row_reductions = 0;
    int col_reductions = 0;

    for ( int i = 0; i < node->adj_len; i++ ) {
        int min_row = INT_MAX;

        for ( int j = 0; j < node->adj_len; j++ ) {
            min_row = (node->adj[i][j] < min_row) ? node->adj[i][j] : min_row;
        }

        if ( min_row != 0 ) {
            for ( int j = 0; j < node->adj_len; j++ ) {
                if ( node->adj[i][j] == INT_MAX ) {
                    continue;
                }

                node->adj[i][j] -= min_row;
            }
        }

        if ( min_row != INT_MAX ) {
            row_reductions += min_row;
        }
    }

    for ( int i = 0; i < node->adj_len; i++ ) {
        int min_col = INT_MAX;

        for ( int j = 0; j < node->adj_len; j++ ) {
            min_col = (node->adj[j][i] < min_col) ? node->adj[j][i] : min_col;
        }

        if ( min_col != 0 ) {
            for ( int j = 0; j < node->adj_len; j++ ) {
                if ( node->adj[j][i] == INT_MAX ) {
                    continue;
                }

                node->adj[j][i] -= min_col;
            }
        }

        if ( min_col != INT_MAX ) {
```

```c
            col_reductions += min_col;
        }
    }

    node->cost = row_reductions + col_reductions;
}

void
tsp_branch_and_bound(struct Node* root) {
    node_row_col_reduce(root);

    struct Node* temp = root;

    matrix_print(root, true);

    while ( temp->max_children != 0 ) {
        node_add_children(temp);

        for ( int i = 0; i < temp->max_children; i++ ) {
            node_row_col_reduce(temp->children[i]);
        }

        for ( int i = 0; i < temp->max_children; i++ ) {
            temp->children[i]->cost += temp->cost;
            temp->children[i]->cost +=
                root->adj[temp->index - 1][temp->children[i]->index - 1];
        }

        int min_index = 0;

        for ( int i = 0; i < temp->max_children; i++ ) {
            if ( temp->children[i]->cost < temp->children[min_index]->cost ) {
                min_index = i;
            }
        }

        temp = temp->children[min_index];
    }

    matrix_print(root, true);

    temp = root;

    fprintf(stdout, "Closed TSP Path: %d", temp->index);

    while ( temp->max_children != 0 ) {
        int min_index = 0;

        for ( int i = 0; i < temp->max_children; i++ ) {
            if ( temp->children[i]->cost < temp->children[min_index]->cost ) {
                min_index = i;
            }
        }

        temp = temp->children[min_index];

        fprintf(stdout, " -> %d", temp->index);
    }

    fprintf(stdout, " -> %d", root->index);
}
```

Analysis

Say the solution to the problem here takes $T(n)$ time. The algorithm used here begins with row and column reducing the root node, which is a $\Theta(n^2)$ operation. Then we have a loop, which runs for successively lower number of iterations, i.e. $n, n-1, n-2, \ldots, 1$ giving an additional time complexity of $\Theta(n^2)$ to each command in the loop. Within this loop, we add children to temp, which is a $\Theta(n^4)$ operation due to the outer loop. After this, we row and column reduce the child nodes, having a complexity $\Theta(n^5)$. Getting the minimum node and advancing in the tree is a $Theta(n^4)$ operation.

Therefore, we can conclude:

$$T(n) = \Theta(n^2) + \Theta(n^4) + \Theta(n^5) + \Theta(n^4) = \Theta(n^5)$$

Therefore, the algorithm has a time complexity of $\Theta(n^5)$.

Dijkstra's and Bellman-Ford's

Algorithms

Dijkstra's

> Start $\text{DIJKSTRAS\_ALGORITHM}(v, adj, dist, n)$
> For $i = 1$ to n do
> $S[i] = \text{FALSE}$

> $dist[i] = adj[v, i]$
> $S[v] = \text{TRUE}$
> $dist[v] = 0.0$
> For $num = 2$ to $n - 1$ do
> $u = \text{CHOOSE\_MIN\_VERTEX}(dist)$
> $S[u] = \text{TRUE}$
> For w adjacent to u and $S[w] = \text{FALSE}$ do
> If $dist[w] > dist[u] + adj[u, w]$
> $dist[w] = dist[u] + adj[u, w]$
> Stop

Bellman-Ford's

> Start $\text{BELLMAN\_FORDS\_ALGORITHM}(v, adj, dist, n)$
> For $i = 1$ to n do
> $dist[i] = adj[v, i]$
> For $k = 2$ to $n - 1$ do
> For each u such that $u \neq v$ and u has at least one incoming edge do
> For each (i, u) in the graph do
> If $dist[u] > dist[i] + adj[i, u]$
> $dist[u] = dist[i] + adj[i, u]$
> Stop

Pseudocode

Dijkstra's

```
dijkstras_algorithm(adj, node_count, source)
begin
    For i = (0 to node_count) {
        set[i] = false
        distances[i] = INFINITY
        prev[i] = NULL
    }

    distances[source] = 0

    while (len(set) < node_count) {
        u = extract_min(distances, node_count, set)
        set[u] = true

        for j = (0 to node_count) {
            if (adj[u][j] ≠ INFINITY and adj[u][j] ≠ 0 and set[i] == false) {
                if (distances[j] > distances[u] + adj[u][j]) {
                    distances[j] = distances[u] + adj[u][j]
                    prev[j] = u
                }
            }
        }
    }

    print_paths(prev, distances, source, node_count)
end
```

Bellman-Ford's

```
bellman_fords_algorithm(adj, node_count, source)
begin
    For i = (0 to node_count) {
        distances[i] = INFINITY
        prev[i] = NULL
    }

    distances[source] = 0

    For i = (1 to node_count) {
        For j = (0 to node_count) {
            For u = (0 to node_count) {
                If (adj[j][u] ≠ 0 and adj[j][u] ≠ INFINITY and distances[u] > distances[j] + adj[j][u]) {
                    distances[u] = distances[j] + adj[j][u]
                    prev[u] = j
                }
            }
        }
    }
end
```

Code

Dijkstra's

```c
// 9. Write a program to implement Dijkstra's Algorithm

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>

#define NODE_COUNT_STR argv[1]
#define EDGE_COUNT_STR argv[2]

void matrix_print(int const** matrix, int const len);
int  extract_min(int const* array, int const len, bool const* set);
void print_paths(int const* prevs,
                 int const* distances,
                 int const  source,
                 int const  len);
void dijkstra(int const** adj, int const node_count, int const source);

int
main(int argc, char** argv) {
    if ( ! NODE_COUNT_STR || ! EDGE_COUNT_STR || argc != 3 ) {
        return 0;
    }

    int  node_count = atoi(NODE_COUNT_STR);
    int  source     = 0;
    int* adj[]      = {
             (int[]) {      0,      10, INT_MAX,       5, INT_MAX},
             (int[]) {INT_MAX,       0,       1,       2, INT_MAX},
             (int[]) {INT_MAX, INT_MAX,       0, INT_MAX,       4},
             (int[]) {INT_MAX,       3,       9,       0,       2},
             (int[]) {      7, INT_MAX,       6, INT_MAX,       0},
    };

    fprintf(stdout, "Adjacency Matrix:\n");
    matrix_print(( int const** ) adj, node_count);

    dijkstra(( int const** ) adj, node_count, source);

    return 0;
}

void
matrix_print(int const** matrix, int const len) {
    fprintf(stdout, "+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|    |");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%3d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%3d ", i);
                continue;
            } else if ( j == -1 ) {
```

```
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == INT_MAX ) {
                fprintf(stdout, "    ");
            } else {
                fprintf(stdout, "%3d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n");
}

int
extract_min(int const* array, int const len, bool const* set) {
    int min_index = -1;

    for ( int i = 0; i < len; i++ ) {
        if ( set[i] == false ) {
            min_index = i;
            break;
        }
    }

    for ( int i = 1; i < len; i++ ) {
        if ( array[min_index] > array[i] && set[i] == false ) {
            min_index = i;
        }
    }

    return min_index;
}

void
print_paths(int const* prevs,
            int const* distances,
            int const  source,
            int const  len) {
    for ( int i = 0; i < len; i++ ) {
        if ( i != source ) {
            fprintf(stdout, "Path from %d to %d (Cost: %d): ", source, i,
                    distances[i]);
            int temp = prevs[i];
            fprintf(stdout, "%d", i);

            while ( temp != -1 ) {
                fprintf(stdout, " <- %d", temp);
                temp = prevs[temp];
            }

            fprintf(stdout, "\n");
        }
    }
}

void
dijkstra(int const** adj, int const node_count, int const source) {
    int   set_size  = 0;
    bool* set       = ( bool* ) malloc(node_count * sizeof(bool));
    int*  distances = ( int* ) malloc(node_count * sizeof(int));
    int*  prev      = ( int* ) malloc(node_count * sizeof(int));

    for ( int i = 0; i < node_count; i++ ) {
        set[i]       = false;
        distances[i] = INT_MAX;
        prev[i]      = -1;
    }

    distances[source] = 0;
```

```
    fprintf(stdout, "\n");

    while ( set_size < node_count ) {
        int u  = extract_min(distances, node_count, set);
        set[u] = true;
        set_size++;

        for ( int j = 0; j < node_count; j++ ) {
            if ( adj[u][j] != INT_MAX && adj[u][j] != 0 && set[j] == false ) {
                if ( distances[j] > distances[u] + adj[u][j] ) {
                    distances[j] = distances[u] + adj[u][j];
                    prev[j]      = u;
                }
            }
        }
    }

    fprintf(stdout, "Source: %d\n", source);
    fprintf(stdout, "Distance Array: ");

    for ( int k = 0; k < node_count; k++ ) {
        if ( distances[k] == INT_MAX ) {
            fprintf(stdout, "%c\t", 'I');
        } else {
            fprintf(stdout, "%d\t", distances[k]);
        }
    }

    fprintf(stdout, "\nPrevious Array: ");

    for ( int k = 0; k < node_count; k++ ) {
        fprintf(stdout, "%d\t", prev[k]);
    }

    fprintf(stdout, "\n\n");
    print_paths(prev, distances, source, node_count);

    free(set);
    free(distances);
    free(prev);
}
```

Bellman-Ford's

```
// 9-2. Write a program to implement Bellman-Ford Algorithm

#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define NODE_COUNT_STR argv[1]  // Command line argument for number of nodes
#define EDGE_COUNT_STR argv[2]  // Command line argument for number of edges

void matrix_print(int const** matrix, int const len);
bool has_incoming(int const** adj, int const node_count, int const vertex);
void print_paths(int const* prevs,
                 int const* distances,
                 int const  source,
                 int const  len);
void bellman_ford(int const** adj, int const node_count, int const source);

int
main(int argc, char** argv) {
    if ( ! NODE_COUNT_STR || ! EDGE_COUNT_STR || argc != 3 ) {
        return 0;
    }

    int  node_count = atoi(NODE_COUNT_STR);
    int  source     = 0;
    int* adj[]      = {
            (int[]) {      0,       6, INT_MAX,       7, INT_MAX},
            (int[]) {INT_MAX,       0,       5,       8,      -4},
            (int[]) {INT_MAX,      -2,       0, INT_MAX, INT_MAX},
            (int[]) {INT_MAX, INT_MAX,      -3,       0,       9},
            (int[]) {      2, INT_MAX,       7, INT_MAX,       0},
    };

    fprintf(stdout, "Adjacency Matrix:\n");
    matrix_print(( int const** ) adj, node_count);

    bellman_ford(( int const** ) adj, node_count, source);
```

```c
    return 0;
}

void
matrix_print(int const** matrix, int const len) {
    fprintf(stdout, "+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|    |");

    for ( int i = 0; i < len; i++ ) {
        fprintf(stdout, "%3d ", i);
    }

    fprintf(stdout, "|\n+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n|");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = -2; j < len; j++ ) {
            if ( j == -2 ) {
                fprintf(stdout, "%3d ", i);
                continue;
            } else if ( j == -1 ) {
                fprintf(stdout, "|");
                continue;
            }

            if ( matrix[i][j] == INT_MAX ) {
                fprintf(stdout, "    ");
            } else {
                fprintf(stdout, "%3d ", matrix[i][j]);
            }
        }

        fprintf(stdout, "|\n|");
    }

    fprintf(stdout, "\b+");

    for ( int i = -2; i < len; i++ ) {
        if ( i == -1 ) {
            fprintf(stdout, "+");
        } else {
            fprintf(stdout, "----");
        }
    }

    fprintf(stdout, "+\n");
}

void
print_paths(int const* prevs,
            int const* distances,
            int const  source,
            int const  len) {
    for ( int i = 0; i < len; i++ ) {
        if ( i != source ) {
            fprintf(stdout, "Path from %d to %d (Cost: %d): ", source, i,
                    distances[i]);
            int temp = prevs[i];
            fprintf(stdout, "%d", i);

            while ( temp != -1 ) {
                fprintf(stdout, " <- %d", temp);
                temp = prevs[temp];
            }

            fprintf(stdout, "\n");
```

```
        }
    }
}

void
bellman_ford(int const** adj, int const node_count, int const source) {
    int* distances = ( int* ) malloc(node_count * sizeof(int));
    int* prev      = ( int* ) malloc(node_count * sizeof(int));

    for ( int i = 0; i < node_count; i++ ) {
        distances[i] = INT_MAX;
        prev[i]      = -1;
    }

    distances[source] = 0;

    for ( int i = 1; i <= node_count - 1; i++ ) {
        for ( int j = 0; j < node_count; j++ ) {
            for ( int u = 0; u < node_count; u++ ) {
                if ( adj[j][u] != 0 && adj[j][u] != INT_MAX &&
                     distances[u] > distances[j] + adj[j][u] ) {
                    distances[u] = distances[j] + adj[j][u];
                    prev[u]      = j;
                }
            }
        }
    }

    fprintf(stdout, "Source: %d\n", source);
    fprintf(stdout, "Distance Array: ");

    for ( int k = 0; k < node_count; k++ ) {
        if ( distances[k] == INT_MAX ) {
            fprintf(stdout, "%c\t", 'I');
        } else {
            fprintf(stdout, "%d\t", distances[k]);
        }
    }

    fprintf(stdout, "\nParent Array: ");

    for ( int k = 0; k < node_count; k++ ) {
        fprintf(stdout, "%d\t", prev[k]);
    }

    fprintf(stdout, "\n\n");

    print_paths(prev, distances, source, node_count);

    free(distances);
    free(prev);
}
```

Analysis

Dijkstra's

Say the algorithm takes $T(n)$ time. The implementation of Dijkstra's algorithm used here starts with initializing the set, $distances$ and $prev$ arrays in $\Theta(n)$ time. We set the distance of the source node to be 0 in $\Theta(1)$ time. We loop over the nodes twice in a nested manner to populate the $distance$ and $prev$ arrays, taking $\Theta(n^2)$ time. Therefore, we can conclude:

$$T(n) = \Theta(n) + \Theta(1) + \Theta(n^2) = \Theta(n^2)$$

Therefore, the algorithm has a time complexity of $\Theta(n^2)$.

Bellman-Ford's

Say the algorithm takes $T(n)$ time. The implementation of Bellman-Ford's algorithm used here starts with initializing the $distances$ and $prev$ arrays in $\Theta(n)$ time. The distance of the source node is set to be 0 in $\Theta(1)$ time. We then use a triple nested loop to set values $distances$ and $prev$ in which we repeatedly iterate through the edges, constantly updating $distances$ and $prev$, taking $\Theta(n^3)$ time. Therefore, we can conclude:

$$T(n) = \Theta(n) + \Theta(1) + \Theta(n^3) = \Theta(n^3)$$

Therefore, the algorithm has a time complexity of $\Theta(n^3)$.

N-Queens Backtracking

Algorithm

> Start $\mathrm{N\_QUEENS}(k, n)$
> For $i = 0$ to n do
> If $\mathrm{PLACE}(k, i)$
> Then $x[k] = i$
> If $(k = n)$

> Then write$(x[1:n])$
> Else
> N_QUEENS$(k+1, n)$
> Stop

> Start PLACE(k, i)
> For $j = 1$ to $k = 1$ do
> If $x[j] = i$ or $|x[j] - 1| = |j - k|$
> Then return FALSE
> Return $TRUE$
> Stop

Pseudocode

```
n_queen(positions, row)
begin
    For i = (0 to len(positions)) {
        positions[row].x = row
        positions[row].y = i
        positions[row].active = true

        if (valid(positions, row + 1)) {
            if (row < len(positions) - 1) {
            n_queen(positions, row + 1)
        } else {
            print(board_init(positions))
        }

        positions[row].active = false
    }
end
```

Code

```c
// 10. Write a program to solve the N-Queens Problem using Backtracking Approach

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N_STR argv[1]

struct Position {
    int  x;
    int  y;
    bool active;
};

int** board_init(struct Position const* positions, int const len);
void  board_dealloc(int** board, int const len);
void  matrix_print(int** board, int const len);
bool  place_check(int const i, int const j, int const k, int const l);
bool  check_valid(struct Position const* positions, int const len);
int   n_queen_init(int const len);
void  n_queen(struct Position* positions,
              int const        row,
              int const        len,
              int*             solncount);

int
main(int argc, char** argv) {
    if ( argc != 2 ) {
        return 0;
    }

    int len = atoi(N_STR);

    fprintf(stdout, "Solution boards for %d-queen problem:\n", len);
    int solncount = n_queen_init(len);
    fprintf(stdout, "Number of solutions with %d queens: %d\n", len, solncount);

    return 0;
}

int**
board_init(struct Position const* positions, int const len) {
    int** board = ( int** ) malloc(len * sizeof(int*));

    for ( int i = 0; i < len; i++ ) {
```

```c
        board[i] = ( int* ) malloc(len * sizeof(int));

        for ( int j = 0; j < len; j++ ) {
            board[i][j] = 0;
        }
    }

    for ( int i = 0; i < len; i++ ) {
        board[positions[i].x][positions[i].y] = 1;
    }

    return board;
}

void
board_dealloc(int** board, int const len) {
    for ( int i = 0; i < len; i++ ) {
        free(board[i]);
    }

    free(board);
}

void
matrix_print(int** board, int const len) {
    for ( int i = -1; i < len * 4; i++ ) {
        fprintf(stdout, "-");
    }

    fprintf(stdout, "\n");

    for ( int i = 0; i < len; i++ ) {
        for ( int j = 0; j < len; j++ ) {
            fprintf(stdout, "| %c ", (board[i][j] == 1) ? 'Q' : ' ');
        }

        fprintf(stdout, "|\n");

        for ( int i = -1; i < len * 4; i++ ) {
            fprintf(stdout, "-");
        }

        fprintf(stdout, "\n");
    }
}

bool
place_check(int const i, int const j, int const k, int const l) {
    if ( i == k ) {
        return false;
    } else if ( j == l ) {
        return false;
    } else if ( (i + j) == (k + l) ) {
        return false;
    } else if ( (i - j) == (k - l) ) {
        return false;
    }

    return true;
}

bool
check_valid(struct Position const* positions, int const elements) {
    for ( int i = 0; i < elements; i++ ) {
        struct Position p1 = positions[i];

        for ( int j = i + 1; j < elements; j++ ) {
            struct Position p2 = positions[j];

            if ( p2.active == false && elements == 1 ) {
                return true;
            }

            if ( place_check(p1.x, p1.y, p2.x, p2.y) == false ) {
                return false;
            }
        }
    }

    return true;
}

int
n_queen_init(int const len) {
```

```c
    int              row       = 0;
    int              solncount = 0;
    struct Position* positions =
        ( struct Position* ) calloc(len, sizeof(struct Position));

    n_queen(positions, row, len, &solncount);
    free(positions);

    return solncount;
}

void
n_queen(struct Position* positions,
        int const        row,
        int const        len,
        int*             solncount) {
    for ( int i = 0; i < len; i++ ) {
        positions[row].x      = row;
        positions[row].y      = i;
        positions[row].active = true;

        if ( check_valid(positions, row + 1) == true ) {
            if ( row < len - 1 ) {
                n_queen(positions, row + 1, len, solncount);
            } else {
                int** board = board_init(positions, len);
                matrix_print(board, len);

                (*solncount) += 1;

                board_dealloc(board, len);
            }
        }

        positions[row].active = false;
    }
}
```

Analysis

Say the solution to the n-queens problem takes $T(n)$ time. The n-queens problem solution used here is much more effective than the brute force approach. Consider an 8-queen problem. There are ${}^{64}C_8$ possible ways to place 8 pieces, or approximately 4.4 billion 8-tuples to examine. However, by allowing only placement of queens on distinct rows and columns, we require the examination of at most $8!$, or only 40320 8 tuples. Therefore, we can conclude:

$$T(n) = \Theta(n!)$$

Therefore, the algorithm has a worst case time complexity of $T(n) = \Theta(n!)$.