Faculty of Engineering and Technology

Department of Computer Science

COMP2421 || Data Structures and Algorithms

Prepared By: Lana Batnij __ 1200308

Instruction: Dr. Ahmed Abusnaina

Section: 1

Table of Contents:

1) Sorting Algorithms: 4

1.1: Sleep Sort algorithm: 4

1.2: Quick Sort Algorithm: 7

1.3: Gnome Sort Algorithm: 8

2) Dynamic Programming: 10

2.1: Fibonacci numbers: 10

2.2: Min Cost Path: 10

2.3: Count Number of Coins: 11

3) Appendixes: 12

3.1: Sleep Sort: 12

3.2: Quick Sort: 12

3.3: Gnome Sort: 13

3.4: Fibonacci numbers: 14

References 16

Table of Figures:

Figure 1: Step one put all elements to sleep 4
Figure 2: The smallest elements wake up first..... 4
Figure 3: The second smallest element and they add it to the sorted array 5
Figure 4: the wake process kepp going in ascending terms..... 5
Figure 5: the 7th element 5
Figure 6: almost finishing the sleep..... 6
Figure 7: the sorted array 6
Figure 8: How it works [3]..... 7
Figure 9: Example of quick Sort [3] 7
Figure 10: Example of Gnome Sort algorithm [4]..... 8
Figure 11: Example on the Min Cost Path.....11

1) Sorting Algorithms:

1.1: Sleep Sort algorithm:

It is also called the king of laziness or sorting while sleeping, In this approach, we generate separate threads for each element in the input array, and each thread naps for some time proportionate to the value of the associated array element. As a result, the thread with the smallest amount of sleeping time awakens up first and prints the number, followed by the second-least element, and so on. The largest element awakens after a long period, and the element is printed at the end. Thus, the output is sorted. All of this multithreading occurs in silent mode and at the core of the OS. We don't know what's going on in the background, hence this is a "mysterious" sorting process [1].

As an example, [2]:

We have an input array = [5,3,3,2,1,1,3,7,11]

Figure 1: Step one put all elements to sleep

Figure 2: The smallest elements wake up first

Figure 3: The second smallest element and they add it to the sorted array

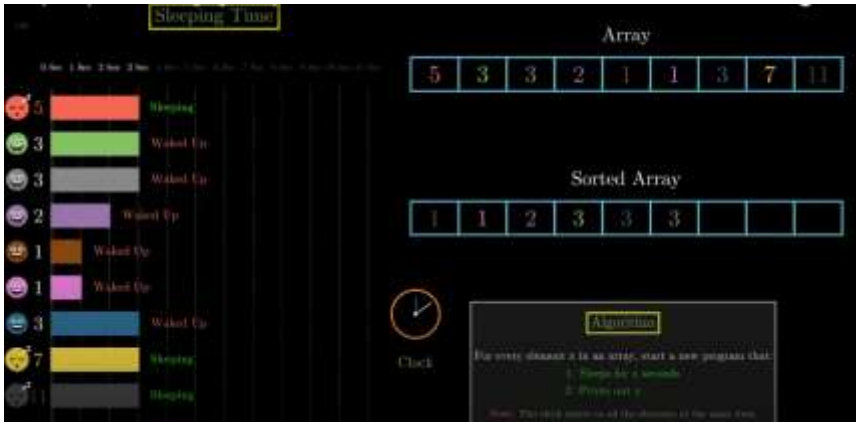

Figure 4: the wake process kepp going in ascending terms

Figure 5: the 7th element

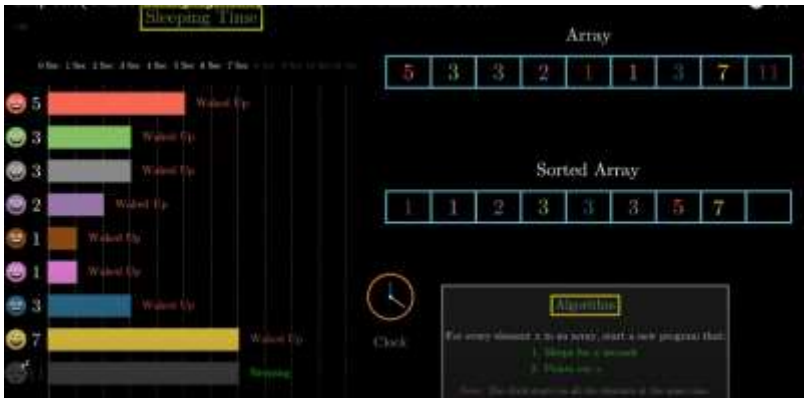

Figure 6: almost finishing the sleep

Figure 7: the sorted array

Now what about its time complexity in different situations:

- At a normal input array with random integers but not negative since it doesn't handle it. As a result, adding all of the array elements into the priority queue takes O (N log N) time. Furthermore, the output is only retrieved when all threads have been processed, i.e., after all, elements have 'woken up'. Because it takes O(arr[i]) time to wake the i^{th} array element's thread. So, it will take a maximum of O(max(input)) for the largest element in the array to wake up, so the time complexity is O (NlogN + max(input)) [1].
- At sorted Ascending and descending, the Time complexity stays the same.

About its state space:

- The operating system's internal priority queue handles everything. As a result, auxiliary space is unnecessary. So, we don't since each thread has its own stack space [1].

1.2: Quick Sort Algorithm:

Quick Sort's core operation is partition (). The goal of partitioning is to organize the pivot which is the central point in the correct location in the sorted array, with all smaller elements to the left of the pivot and all greater elements to the right, after the pivot is correctly positioned, the partition is done recursively on either side of it, and the array is eventually sorted [3].

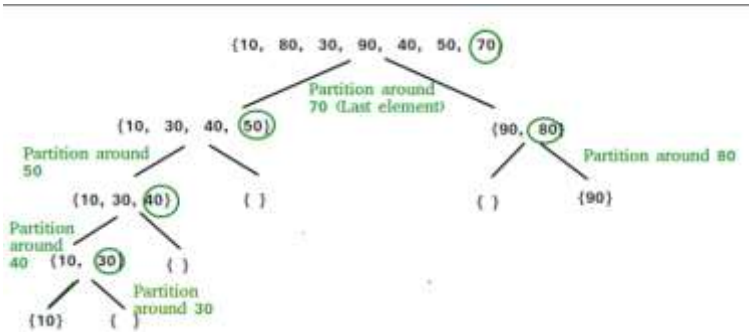

Figure 8: How it works [3]

Let's consider the array as an example, array = [10,80,30,90,40], and let 40 be the pivot.

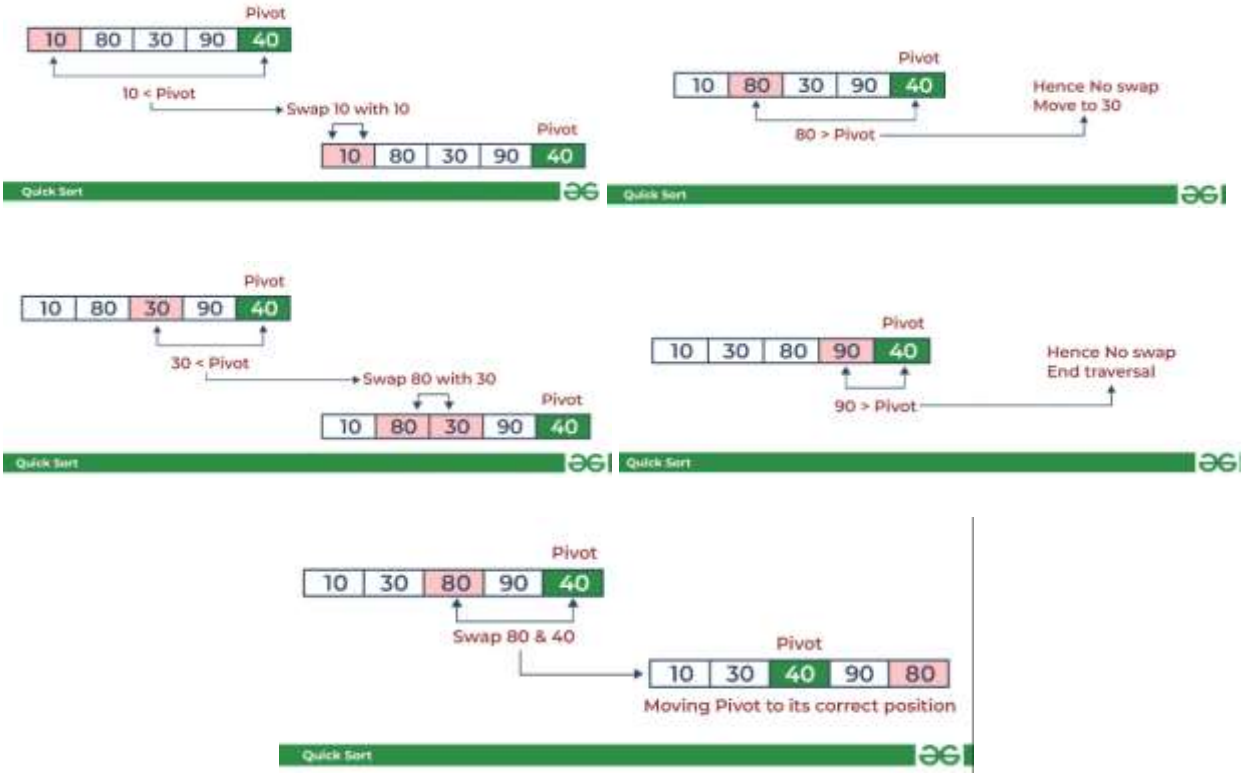

Figure 9: Example of quick Sort [3]

About its time complexity:

- Average case: O (log(N)) Quick sort's average-case performance is typically excellent in reality, making it one of the fastest sorting algorithms [3].
- For the sorted ascending and descending arrays: For sorted ascending data, a regularly poor pivot choice, such as choosing the smallest element, leads to a worst-case temporal complexity of O(n^2). To avoid this, common strategies include random or median-of-three pivot selection. In sorted descending data, useless pivot selection results in a worst-case time complexity of O(n^2), whereas improved pivot selection keeps the average case at O (n log n).

And its space complexity:

- If we ignore the recursive stack space, the answer is O (1). If we think about the recursive stack space, quick sort could result in O (N) [3].

1.3: Gnome Sort Algorithm:

Gnome Sort, sometimes known as Stupid Sort, is based on the idea of a garden gnome sorting his flower pots. A garden gnome organizes flower pots using the following approach: He looks at the flower pot next to him and looks at the previous one; if they are in the correct order, he moves one pot forward; otherwise, he swaps them and moves one pot backward. If there is no before pot (he is at the beginning of the pot line), he moves forward; if there is no pot next to him (he is at the end of the pot line), he stops [4].

As an example, to show how it works:

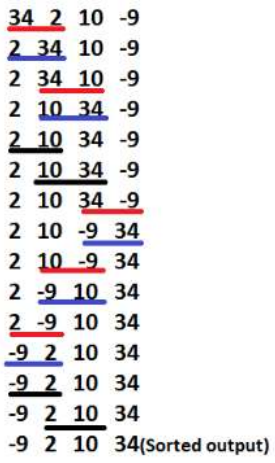

Figure 10: Example of Gnome Sort algorithm [4]

If you're at the beginning of the array, proceed to the correct element (arr [0] to arr [1]). If the current array element is bigger or equal to the previous array element, move one step right. If the current array element is less than the preceding array element, swap these two elements and proceed one step backward. Repeat steps 2 and 3 until reaches the end of the array. If you reach the end of the array, pause and sort it. [4].

The Time complexity:

- Because there is no nested loop (just one while), it appears that this is a linear O(N) time approach. However, the time complexity is O(n^2) [4].
- Sorted Ascending Gnome Sort works best with previously or nearly sorted data. In the best-case situation, the time complexity approaches O(n). Meanwhile, Sorted Descending outperforms the worst scenario but falls short of the best.

The space complexity:

- Gnome Sort is an in-place sorting algorithm, which means it needs a fixed amount of additional RAM. The space complexity equals O (1) [4].

2) Dynamic Programming:

is a problem-solving strategy that divides larger issues into smaller subproblems and solves each subproblem only once. The solutions to the subproblems are saved in a table and used to solve bigger problems. This strategy is used to handle optimization problems that require us to select the optimal answer from a large number of options [5].

As An examples:

2.1: Fibonacci numbers:

The sequence known as Fibonacci consists of the following numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, and so on [5]. In terms of math, the sequence Fn of Fibonacci numbers is described by the recursive relation: F{n} = F{n-1} + F{n-2}, with seed values and F0 = 0 and F1 = 1.

The Dynamic Programming technique keeps solutions to subproblems in an array to reduce duplicate computations. The non-Dynamic Programming approach employs recursion, which can result in exponential temporal complexity due to repeating analyses of the same subproblems. In this situation, dynamic programming considerably enhances the computational efficiency of Fibonacci numbers.

2.2: Min Cost Path:

a function that calculates the minimal cost path from (0, 0) to (M, N). Each cell in the matrix indicates the cost of traversing that cell. The total cost of a path to (M, N) is the sum of all payments (source and destination). You can only move down, right, and diagonally lower cells from a given cell, i.e., cells (i+1, j), (i, j+1), and (i+1, j+1) can be moved [6].

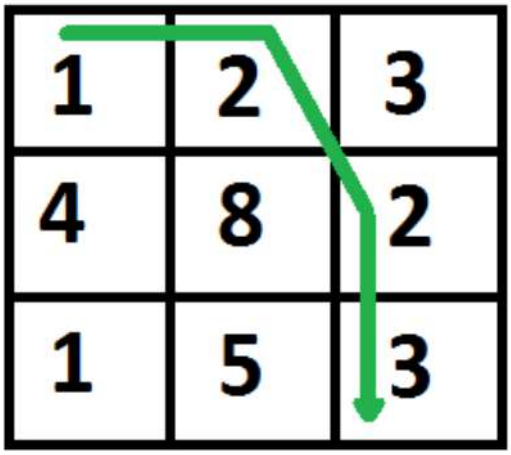

Figure 11: Example on the Min Cost Path

2.3: Count Number of Coins:

certain an integer array of coins [] of size N indicating different amounts and an integer total, the job is to determine the number of coins needed to make a certain value sum [7].

- Input: sum = 10, coins [] = {2, 5, 3, 6}
- Output: 5
- Explanation: There are five solutions:
- {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.

3) Appendixes:

3.1: Sleep Sort:

```
void sleep_sort(int arr[], int n) {
    pthread_t threads[n];
    for (int i = 0; i < n; i++) {
        int* num = (int*)malloc(sizeof(int));
        *num = arr[i];
        pthread_create(&threads[i], NULL, (void*(*)(void*))sleep, (void*)num);
   }
    for (int i = 0; i < n; i++) {
        pthread_join(threads[i], NULL);
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

3.2: Quick Sort:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
```

```
    return i + 1;
}
void quick_sort_helper(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort_helper(arr, low, pi - 1);
        quick_sort_helper(arr, pi + 1, high);
    }
}
void quick_sort(int arr[], int n) {
    quick_sort_helper(arr, 0, n - 1);
}
```

3.3: Gnome Sort:

```
void gnome_sort(int arr[], int n) {
    int index = 0;
    while (index < n) {
        if (index == 0 || arr[index] >= arr[index - 1]) {
            index++;
        } else {
            int temp = arr[index];
            arr[index] = arr[index - 1];
            arr[index - 1] = temp;
            index--;
        }
    }
}
```

3.4: Fibonacci numbers:

```
// Fibonacci Series using Recursion

#include <stdio.h>

int fib(int n)

{  if (n <= 1)

       return n;

   return fib(n - 1) + fib(n - 2);

}

int main()

{

   int n = 9;

   printf("%dth Fibonacci Number: %d", n, fib(n));

   return 0;

}

// ------------------------------------------------------------------------------------------------------------

// Fibonacci Series using Space Optimized Method

#include <stdio.h>

int fib(int n)

{

    int a = 0, b = 1, c, i;

    if (n == 0)

        return a;

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

        c = a + b;

        a = b;
```

```
        b = c;
    }
    return b;
}
int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
//------------------------------------------------------------------------------
```

References

[1] R. Belwariar, "geeksforgeeks," geeksforgeeks, [Online]. Available: https://www.geeksforgeeks.org/sleep-sort-king-laziness-sorting-sleeping/. [Accessed 31 1 2024].

[2] G. Code, "Sleep Sort (Craziest Sorting Algorithm) in 1 Minute with Visualisation & Code," Go Code , [Online]. Available: https://www.youtube.com/watch?v=Cp9mdJmVtvo. [Accessed 31 1 2024].

[3] GeeksforGeeks, "GeeksforGeeks," GeeksforGeeks, 16 10 2023. [Online]. Available: https://www.geeksforgeeks.org/quick-sort/. [Accessed 31 1 2024].

[4] R. Belwariar, "geeksforgeeks," geeksforgeeks, 10 1 2023. [Online]. Available: https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/. [Accessed 31 1 2024].

[5] geeksforgeeks, "geeksforgeeks," geeksforgeeks, 3 10 2023. [Online]. Available: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/. [Accessed 1 2 2024].

[6] geeksforgeeks, "geeksforgeeks," geeksforgeeks, 9 10 2023. [Online]. Available: https://www.geeksforgeeks.org/min-cost-path-dp-6/. [Accessed 1 2 2024].

[7] geeksforgeeks, "geeksforgeeks," geeksforgeeks, 20 10 2023. [Online]. Available: https://www.geeksforgeeks.org/coin-change-dp-7/. [Accessed 1 2 2024].