FACULTY OF ENGINEERING & TECHNOLOGY

ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

DATA STRUCTURES AND ALGORITHMS - COMP2421

Procject4

Prepared by: Lana Hamayel

Student ID: 1200209

Instructor: Dr. Saadeddin Hroub

Section: 3

29th.January.2024

Abstract

In this project, two main goals are mentioned. Firstly, the exploration of three new ways of sorting algorithms involves showcasing how they work with examples and code, and considering the speed, space, and time complexity required in different scenarios. Secondly, Dynamic Programming is explored, with an illustration of what it is and providing three problems it can solve. Examples of problem-solving, both with and without Dynamic Programming, are included.

Contents

Abstract I

Sorting Algorithms 1

1. Pancake Sort 1

I. Algorithm 1

II. solved example 2

III. Code 4

IV. Algorithm properties 5

2. Time Sort 6

I. Algorithm 6

II. Solved example 7

III. Code 9

IV. Algorithm properties 10

3. Odd Even Sort 10

I. Algorithm 11

II. Solved example 11

III. Code 13

IV. Algorithm properties 14

Dynamic Programing 15

I. Definition 15

II. Problem solved using DP 15

III. Comparing between Dynamic and non-dynamic Programming 16

References 18

Sorting Algorithms

1. Pancake Sort

This algorithm is inspired from flipping the pancakes. The reason why it's called Pancake sort is because it looks like sorting pancakes on a plate using a spatula. You can only use the spatula to flip a few of the top pancakes on the plate.[1]

I. Algorithm

When the biggest number is found, the array is flipped in a specific way. After the largest number is identified, the section of the array from the beginning to where the biggest number is found is flipped. Therefore, the entire section is flipped.[1]

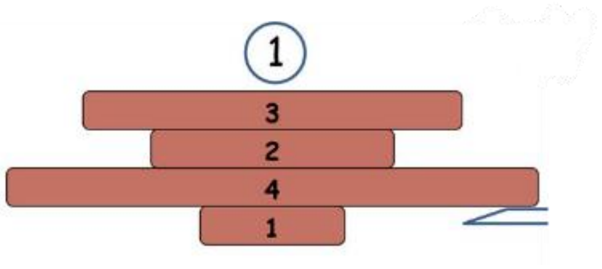
In other words:

1. Input: Stack of pancakes, each of different sizes.
2. Algorithm: Slip a flipper under one of the pancakes and over the whole stack above the flipper.
3. Output: Arrange in order of size (smallest on top)

II. solved example

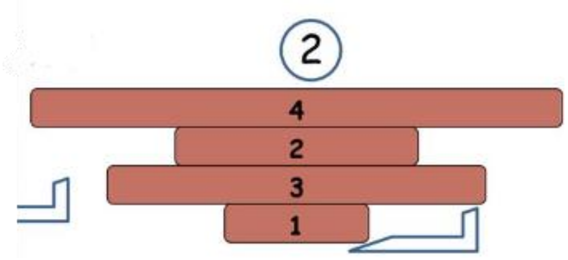
suppose we have this initial pancake order: [3, 2, 4, 1]

First flip: [3, 2, 4, 1]



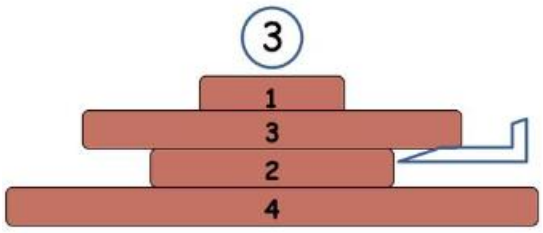
After first flip: [4, 2, 3, 1]

Second flip: [4, 2, 3, 1]



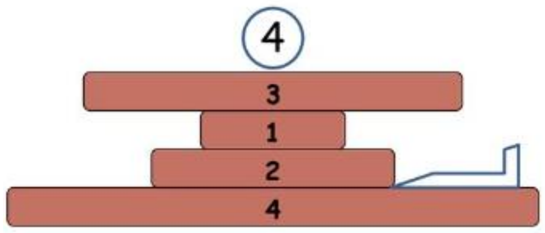
After second flip: [1, 3, 2, 4]

Third flip: [1, 3, 2, 4]



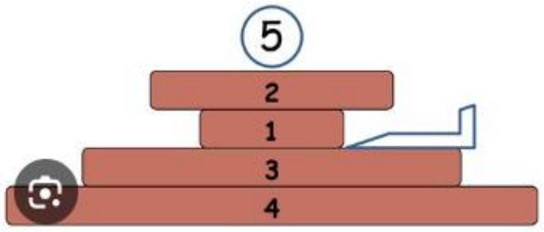
After third flip: [3, 1, 2, 4]

Fourth flip: [3, 1, 2, 4]



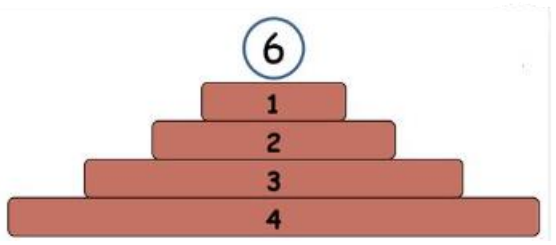
After fourth flip: [2, 1, 3, 4]

Fifth flip: [2, 1, 3, 4]



After fifth flip: [1, 2, 3, 4]

Sixth flip: [1, 2, 3, 4]



After sixth flip: [1,2, 3, 4], DONE!

III. Code

```
/* Reverses arr[0..i] */
void flip(int arr[], int i)
{
   int temp, start = 0;
   while (start < i) {
      temp = arr[start];
      arr[start] = arr[i];
      arr[i] = temp;
      start++;
      i--;
   }
}

// Returns index of the
// maximum element in
// arr[0..n-1]
int findMax(int arr[], int n)
{
   int mi, i;
   for (mi = 0, i = 0; i < n; ++i)
      if (arr[i] > arr[mi])
         mi = i;
   return mi;
}

// The main function that
// sorts given array using
// flip operations
void pancakeSort(int* arr, int n)
{
   // Start from the complete
   // array and one by one
   // reduce current size
   // by one
   for (int curr_size = n; curr_size > 1;
        --curr_size)
   {
      // Find index of the
      // maximum element in
      // arr[0..curr_size-1]
      int mi = findMax(arr, curr_size);

      // Move the maximum
      // element to end of
      // current array if
      // it's not already
      // at the end
      if (mi != curr_size - 1) {
         // To move at the end,
         // first move maximum
         // number to beginning
         flip(arr, mi);

         // Now move the maximum
         // number to end by
         // reversing current array
         flip(arr, curr_size - 1);
      }
   }
}
[2]
```

IV. Algorithm properties

- Time complexity: O(n*n) where n is the number of elements to be sorted.
- Space complexity: the space complexity is O (1) since this sorting type doesn't need to allocate more memory.
- Stability: it's not stable since there is no guarantee that the equal elements weren't swept
- In place: since there is no need for another memory in order this sorting is done.
- Running time

  ♣ Sorted (ascending): O(n) this is the best case.

  ♣ Sorted (descending) :O(n^2) this is the worst case.

  ♣ Not sorted: O(n^2) this will be the average case.[2]

2. Time Sort

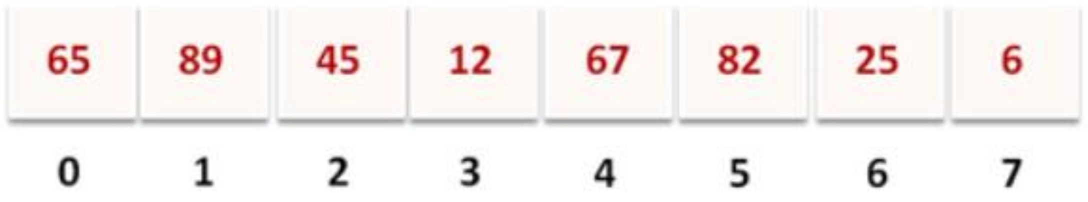
Tim Sort is a really cool sorting algorithm that mixes merge sort and insertion sort together. It's super awesome because it works really well with real-world data. That's why it's one of the most popular sorting algorithms out there. In fact, even the sort function in Python is based on Tim Sort.[4]

I. Algorithm

The Array is split into Run chunks. These runs are sorted one by one using insertion sort, and then they are combined using the combine function from merge sort. If the size of the Array is smaller than the run, it is sorted only using Insertion Sort. The run's size can vary from 32 to 64 depending on the size of the array. It is important to note that the merge method works well when the size of subarrays is a power of 2. The success of the insertion sort algorithm with small arrays is the basis for this concept.[4]

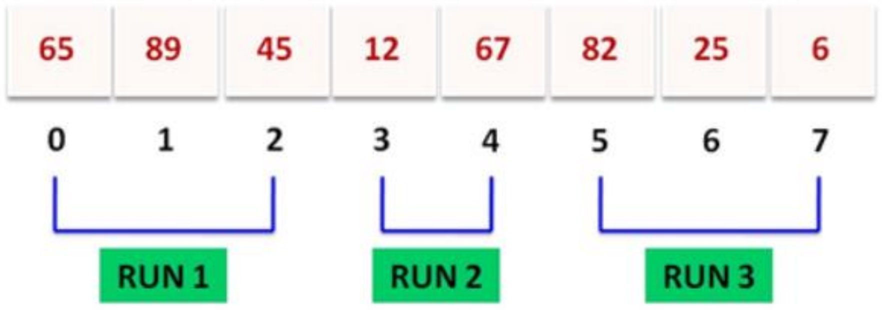
II. Solved example

Suppose there is array of 8 elements as follow:



Step 1:

 the array will be divided into number of runs.in this example supposes the run is taken as follow:



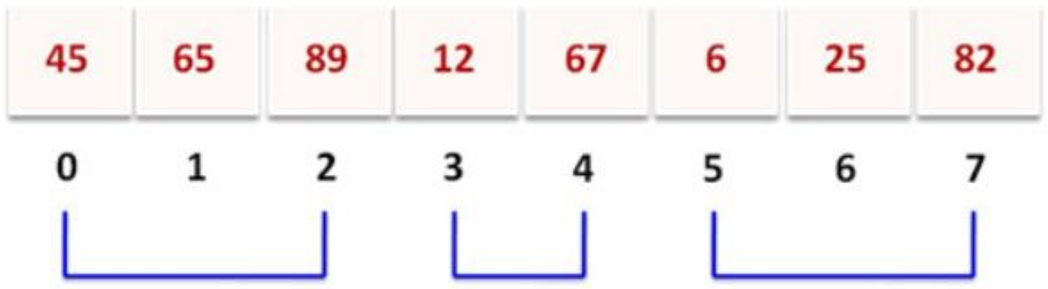
Step 2:

Now the whole array is divided into three parts called run. First part is of three element, second part is of two element and third part is of three elements.

Now for each the part of array, the insertion sort function will be called and each part will be sorted using the insertion sort technique.

Step 3:

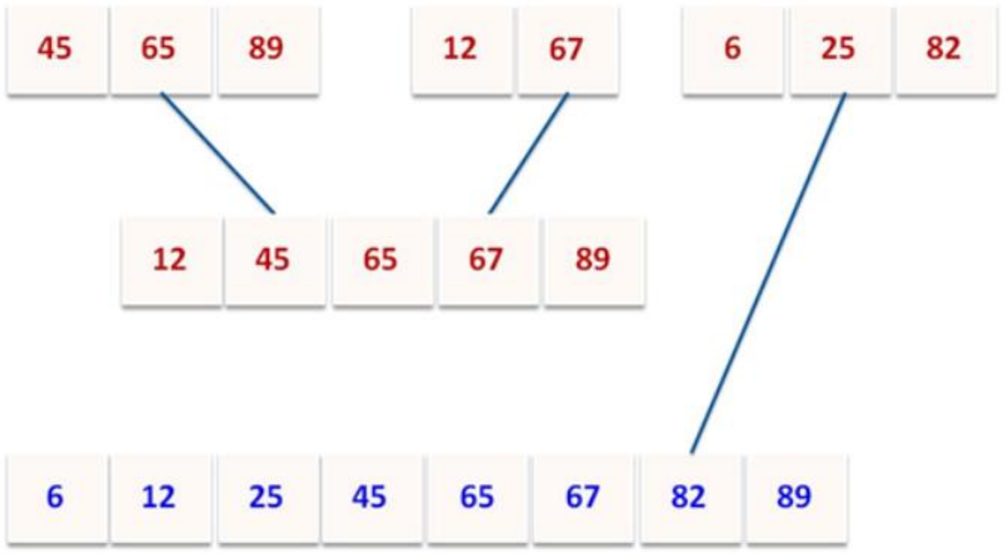
After sorting each run, we will get the following array:



Step 4:

Then, these three parts of the array will be merged using the merger sort technigue in sorted order.

The recursion tree of merger sort will be:



DONE! We got the sorted array.

III. Code

```
void insertionSort(int arr[], int left, int right) {
   for (int i = left + 1; i <= right; i++) {
      int temp = arr[i];
      int j = i - 1;
      while (j >= left && arr[j] > temp) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = temp;
   }
}

void merge(int arr[], int l, int m, int r) {
   int len1 = m - l + 1, len2 = r - m;
   int left[len1], right[len2];

   for (int i = 0; i < len1; i++)
      left[i] = arr[l + i];
   for (int i = 0; i < len2; i++)
      right[i] = arr[m + 1 + i];

   int i = 0, j = 0, k = l;

   while (i < len1 && j < len2) {
      if (left[i] <= right[j]) {
         arr[k] = left[i];
         i++;
      } else {
         arr[k] = right[j];
         j++;
      }
      k++;
   }

   while (i < len1) {
      arr[k] = left[i];
      k++;
      i++;
   }

   while (j < len2) {
      arr[k] = right[j];
      k++;
      j++;
   }
}

void timSort(int arr[], int n) {
  for (int i = 0; i < n; i += RUN)
    insertionSort(arr, i, (i + RUN - 1) < (n - 1) ? (i + RUN - 1) : (n - 1));

  for (int size = RUN; size < n; size = 2 * size) {
    for (int left = 0; left < n; left += 2 * size) {
      int mid = left + size - 1;
      int right = (left + 2 * size - 1) < (n - 1) ? (left + 2 * size - 1) : (n - 1);

      if (mid < right)
        merge(arr, left, mid, right);
    }
  }
}
[5]
```

IV. Algorithm properties

- Time complexity: the time sort takes O (n Log(n)) in its average and worst case, and since its adaptive the best case is when the input is sorted and then it takes only O (n).
- space complexity: O (n).
- Stability: Stable.
- Place or not: out of place, as it takes up more space.
- Running time

  ♣ Sorted (ascending): O (n).

  ♣ Sorted (descending): O (n Log(n)).

  ♣ Not sorted: O (n Log(n)).[4]

3. Odd Even Sort

An odd-even sort, also known as brick sort, is a simple way to organize a list of numbers. It was made for computers that work together and talk to each other. It's like the bubble sort in some ways. First, it looks at pairs of numbers that are next to each other. If one number is bigger than the other, it switches them. Then, it does the same thing but with different pairs of numbers. It keeps doing this until the list is all sorted. It goes back and forth between different pairs of numbers until everything is in order.[3]

I. Algorithm

It works by dividing the algorithm into two parts: even and odd. In the even phase, it sorts the elements with odd indexes, and in the odd phase, it sorts the elements with even indexes. It keeps repeating this process until there are no more changes happening.[3]

II. Solved example

Assume we have unsorted array = {19,2,72,3,18,57,603,490,45,101}.

↓ Even index

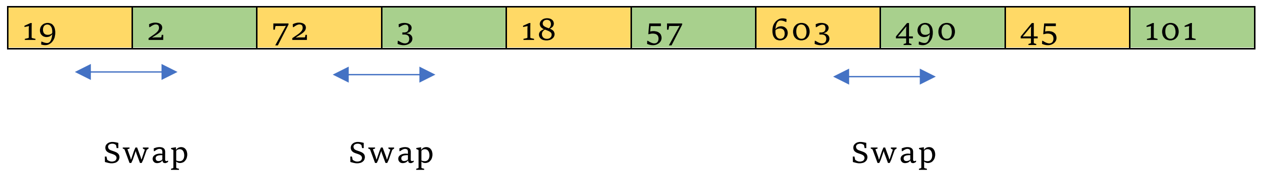
19	2	72	3	18	57	603	490	45	101

↓

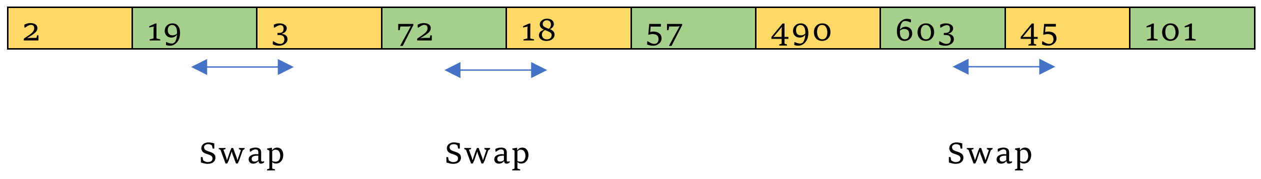
Odd index

<<unsorted array>>

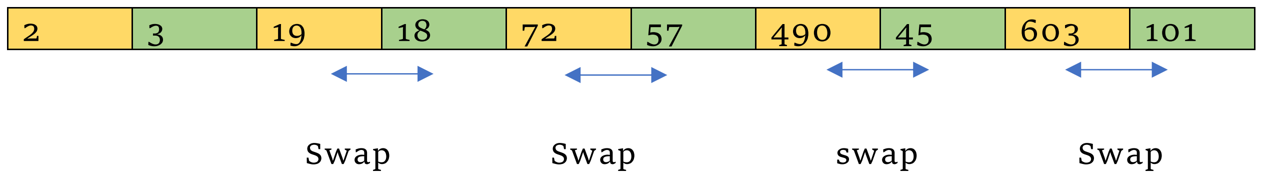
Step 1:



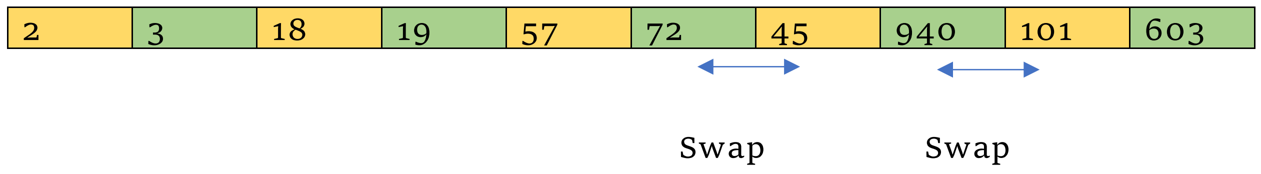
Step 2:



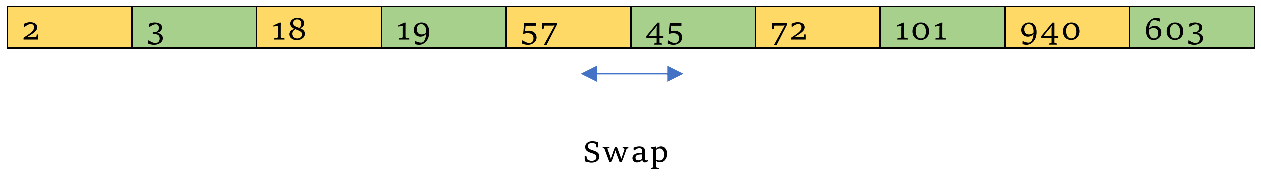
Step 3:



Step 4:



Step 5:



Sorted array:

2	3	18	19	45	57	72	101	940	603

DONE!

III. Code

```
function oddEvenSort(list) {
  function swap(list, i, j) {
    var temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  }

  var sorted = false;
  while (!sorted) {
    sorted = true;
    for (var i = 1; i < list.length - 1; i += 2) {
      if (list[i] > list[i + 1]) {
        swap(list, i, i + 1);
        sorted = false;
      }
    }
    for (var i = 0; i < list.length - 1; i += 2) {
      if (list[i] > list[i + 1]) {
        swap(list, i, i + 1);
        sorted = false;
      }
    }
  }
}
[3]
```

IV. Algorithm properties

- Space complexity: the space complexity is O (1) since we don't need to allocate extra memory for the matrix.
- Time complexity: worst case O(n*n), best case O(n) and average case O(n).
- Stability: it's not stable since there is no guarantee that the equal elements weren't swept.
- In place: yes, it's. since it doesn't need another memory to implement it.
- Running time

  ♣ Sorted (ascending): O(n) since the algorithm will perform the minimum number of sweepings. thus, it will be the best case.

  ♣ Sorted (descending) :O(n*n) since the algorithm will perform the maximum number of sweepings. thus, it will be the worst case.

  ♣ Not sorted: O (n* n) this will be the average case.[3]

Dynamic Programing

I. Definition

Dynamic Programing (DP): is a great way of programming that can explore all the different ways to solve a problem. [7]

The DP has two main approaches that makes it so powerful. The first one is that it can break down a big problem into smaller parts called "overlapping subproblems". The Second approach is that it can find the best solution by using the best solutions to the overlapping subproblems. [7]

To make it clearer, assume that we have smaller parts of puzzle pieces which we can use it again and again in order to find the best way to solve the problem and got the right solution.

II. Problem solved using DP

a. Boolean Parenthesis Problem

counting how many different ways can put parentheses in a Boolean expression to get different answers. The expression has symbols like T for true and F for false, and operators like & for AND, | for OR, and ^ for XOR.[6]

b. Fibonacci sequence Problem

The dynamic programing is used to find the nth Fibonacci number in Fibonacci sequence which representing a series of numbers. Each number is the sum of the two preceding ones.[6]

c. Longest Path in Matrix

In order to calculate the length of the increasing path in matrix in smarter way the DP is used.[6]

III. Comparing between Dynamic and non-dynamic Programming

a. Recursion without DP

```
int fib(int x)
{
    if (x < 2)
        return 1;

    return fib(x-1) + fib(x-2);
}
[8]
```

b. Recursion with DP

```
int fib(int x)
{
    static vector<int> cache(N, -1);

    int& result = cache[x];

    if (result == -1)
    {
        if (x < 2)
            result = 1;
        else
            result = fib(x-1) + fib(x-2);
    }

    return result;
}[8]
```

References

[1] "https://en.wikipedia.org/wiki/Pancake_sorting," [Online]. [Accessed 31st Jan 2024].

[2] "https://www.geeksforgeeks.org/pancake-sorting/," [Online]. [Accessed 31st Jan 2024].

[3] "https://en.wikipedia.org/wiki/Odd%E2%80%93even_sort," [Online]. [Accessed 31st Jan 2024].

[4] "https://en.wikipedia.org/wiki/Timsort," [Online]. [Accessed 31st Jan 2024].

[5] "https://www.geeksforgeeks.org/timsort/," [Online]. [Accessed 31st Jan 2024].

[6] "https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/," [Online]. [Accessed 31st Jan 2024].

[7] "https://leetcode.com/explore/featured/card/dynamic-programming/630/an-introduction-to-dynamic-programming/4034/," [Online]. [Accessed 31st Jan 2024].

[8] "https://stackoverflow.com/questions/12133754/whats-the-difference-between-recursion-memoization-dynamic-programming," [Online]. [Accessed 31st Jan 2024].