Algorithms

What is an algorithm?

An algorithm is a procedure, recipe, process to accopmplish a task. It takes value as input and delivers a value as output.

How to develop an algortihm?

The development of an algorithm has four crucial steps:

1. **Specifiaction**: Be clear what the problem is
2. **Design**: Specify structure of the solution, usually in pseudcode
3. **Development**: Convert pseudocode in chosen language (C, Python, Java etc.)
4. **Testing**: if all inputs deliver all necessary outputs

Example: Linear search in Pseudocode

```
# Linear search last occurence
    p = NIL;
    for i = 1 to n do
        if A[i]==v then p=i;
    return p;
```

Here we are defining our variable p as 0. For the range of 1 to n in our array A, we are checking the values (v). If we find a value which matches our input, we set it equal to p and return it.

```
# Linear search first occurence
    i = 1;
    while i <= n and A[i] != v do i++;
    if i <= n then return i;
    else return NIL;
```

Here we are setting i to 1. While i is less or equal number of array elements and there is no match in value with the array, keeping adding 1. If i is less or equal n, return i else return 0.

```c
#include <stdio.h>
#define n 5
int i, v;
int a[] = { 11, 1, 4, -3, 22 };
int main() {
    i = 0; v = -2;
    while (i < n && a[i] != v) { i++; }
    if (i < n) { printf("%d\n", i); }
    else { printf("NIL\n"); }
}
```

Here we see how we would do the linear search task with C. As visible, the pseudocode comes very close to what we write in C. Pseudcode allows us to roughly structure out our code, before we write it any language. The Java code for this algorithm is also very close to pseudocode, making it a invaluable tool for construction of algorithms.

```
import java.io.*;
class search {
    static int n = 5;
    static int i, v;
    static int a[] = { 11, 1, 4, -3, 22 };
public static void main(String args[]) {
    i = 0; v = 22;
    while (i < n && a[i] != v) { i++; }
    if (i < n) { System.out.println(i); }
    else { System.out.println("NIL"); }
    }
}
```

Prime number filter in pseudocode

```
A[0] = False; A[1] = False;
for i = 2 to n do A[i] = True;
for i = 2 to floor(n/2) do
    for j = 2 to floor(n/i) do
        A[i*j] = False;
```

This is called the *Sieve of Eratosthenes*. Here we are given an array with all numbers from 0 to i. We set the first two elements, being `A[0] = 0` and `A[1] = 1`, to False. We know that these 2 numbers are not primes. Starting from `i = 2`, we set all the other elements to True. We then divide our number of array elements by 2 using `(floor(n/2))` - the floor function rounds the float to an integer. What we then do is check which of the remaining elements we have in our array are possible to reached as a multiple `A[i*j]` of our elements i and j. Here we can see how this happens.

Sorting algotihms

Sorting is one of the most fundamental parts of algorithmic design. It is often used as a subalgorithm in bigger algorithms. We measure the efficiency and time complexity of an algorithm in 2 different methods: number of comparisons and number of exchange operations.

Bubble sort

```
for i = n to 2 do
    for j = 2 to i do
        if A[j] < A[j-1] then
            t = A[j];
            A[j] = A[j-1];
            A[j-1] = t;
```

In bubble sort, what we do is compare a number and it's neighbour to check if the number has already reached is right place or not. In the pseudcode algorithm we have 3 loops. The first loop `i=n` tells us how many elements our array has and iterates over the whole array. The second loop tells us to start the comparison between the first and second element and keep it going until the last element of the array, because of (i=n). The third loop is where the sorting actually happens. The three-step "formula" we see here is called the swap. In future algorithms, it will be reffered to as swap. As an example fo this algortihm: `swap(A[j], A[j-1])`

```
void bubblesort(int *array, int length)
{
   int i, j, tmp;

   for (i = 1; i < length; i++)
   {
      for (j = 0; j < length - 1 ; j++)
      {
          if (array[j] > array[j + 1])
          {
              tmp = array[j];
              array[j] = array[j + 1];
              array[j + 1] = tmp;
          }
      }
   }
}
```

Selection sort

```
for i = 1 to n-1 do
    k=1;
    for j=i+1 to n do
        if A[j] < A[k] then k = j;
    swap(A[j], A[k])
```

Selection sort finds the minimum element of the array and places it at the first place. It then condtinues through the array searching for the smallest elements and places it at the respective spot. In pseudocode, we ask our algorithm to loop from 1 to n-1 (the last element is already sorted once we sort the rest of the array) and set k at first place. This k is generally the smallest element in our array. We then compare the next element with our smallest element k and if the new element j is smaller than k, we set it as our new k.

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Insertion sort

```
for i = 2 to n do
    j=i-1;
    t=A[i];
    while j >= 1 and t < A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

We start with an array of A[n] elements. What then happens is that we define 3 variables: i, key and j. i is our iterator through the array. It keeps track if or if not we are in bounds (i < n). The variable key lets us keep track of which element we are sorting at the moment. The variable j is `i-1` letting us keep track of the variable right before i. So here we will be looking at 2 elements together. Our condition `while (j >= 0 && arr[j] > key)` lets us iterate over the array while j is not 0, which would make it out of bounds of our array and j is greater than our variable key. We then set our array element `a[j+1]` equal to `a[j]` and move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position.

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Recursion algotihms

- recursive object: it contains itself and is defined in terms of itself
- reursive procedure: recalling the procedure multiple times and having a termination condition
- multiple recursion: keep recalling the function inside itself
- mutual recursion: 2 functions (e.g. odd(n) and even(n) who define themself be calling each other)

Fibonacci numbers The way the fibonacci numbers work is quite tricky. Lets see the first steps of the recursion to better understand the pattern:

1. Calculate fib(0) = 0
2. Calculate fib(1) = 1
3. Calculate fib(2) = fib(0) + fib (1) = 1
4. Calculate fib(3) = fib(2) + fib(1)
5. Calculate fib(4) = fib(3) + fib(2)

The next number is always a caluclation derived from the previous 2 numbers allowing us to recursively call the function fib(n) again and again to calculate the next step.

Coded in C

Odd and even numbers This is an example of mutual recursion where the functions call each other to get the necessary value

```
odd(n) = false if n = 0, even(n-1) if n > 0  even(n) = true if n = 0, odd(n-1) if n > 0
```

Drawing figures with recursion

- Sierperinski triangle
- Hilbert curve
- Sieperinski curve