

# Lab 2 - Sorting Algorithms

- Bubble sort
- Insert sort
- Selection sort
- Merge sort
- Quick sort
- Heap sort
- Shell sort
- Counting sort

## Bubble sort

Go through the whole array and sort it by swapping two elements if they are in the wrong order, and keeping them if they are in the right order. Repeat this  $n-1$  times, every time go to index one less than from the previous iteration.

```
def bubble_sort(a):
    n = len(a)
    for i in range(0, n-1):
        for j in range(0, n-1-i):
            if a[j] > a[j+1]:
                a[j], a[j+1] = a[j+1], a[j]
    return a
```

## Insertion sort

Go through the whole array, element by element, and look on the left to check at what position does this element belong in, and then insert it there.

```
def insertion(a):
    for i in range(1, len(a)):
        temp = a[i]
        j = i-1
        while a[j] > temp and j >= 0:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = temp
    return a
```

## Selection sort

Separate the array into two parts, unsorted and sorted. Go through the unsorted array and find the smallest element, and then append it to the sorted part.

```
def selection(a):
    for i in range(len(a)-1):
        x = i
        for j in range(i, len(a)):
            if a[j] < a[x]:
                x = j
        a[x], a[i] = a[i], a[x]
    return a
```

## Merge sort

Split array into two and sort them recursively, then merge. In the simplest case, when the array is one element, just return it.

```
def merge_sort(a):
    if len(a) < 2:
        return a
    m = len(a) // 2
```

```

    left = merge_sort(a[:m])
    right = merge_sort(a[m:])
    return merge(left, right)

def merge(a,b):
    result = []
    while len(a) > 0 and len(b) > 0:
        if a[0] < b[0]:
            result.append(a[0])
            a.pop(0)
        else:
            result.append(b[0])
            b.pop(0)
    result.extend(a)
    result.extend(b)
    return result

```

## Quick sort

This is a recursive algorithm. If the array is one element long, return as it is. Else pick a pivot element (Juan picks the last one) then partition the array so that all the elements that are smaller or equal than the pivot (with the pivot itself) are on one side of the array, and the rest are on the other side. Then just recursively sort two partitions, and leave the pivot as it is, because it is in the right position already.

```

def quick_sort(a):
    quick_sort_helper(a, 0, len(a)-1)
    return

def quick_sort_helper(a, low, high):
    if low >= high or low < 0: return

    p = partition(a, low, high)
    quick_sort_helper(a, low, p-1)

```

```

    quick_sort_helper(a, p+1, high)

def partition(a, low, high):
    pivot = a[high]
    # The index one before the correct position of the pivot
    i = low - 1

    for j in range(low, high):
        if a[j] <= pivot:
            i += 1
            a[j], a[i] = a[i], a[j]
    i += 1
    a[i], a[high] = a[high], a[i]
    return i

```

## Heap sort

The heapsort algorithm begins by rearranging the array into a binary max-heap. A heap is an array/binary tree data structure where each array element is a tree node and its children are at indices  $2i+1$  and  $2i+2$  where  $i$  is the index of that node. A max heap is a heap where each node is greater or equal to its children. The algorithm then repeatedly swaps the root of the heap (the greatest element remaining in the heap) with its last element, which is then declared to be part of the sorted suffix. Then the heap, which was damaged by replacing the root, is repaired so that the greatest element is again at the root. This repeats until only one value remains in the heap.

```

def heap_sort(a):
    build_max_heap(a)
    n = len(a)
    for i in range(n-1, 0, -1):
        a[0], a[i] = a[i], a[0]
        max_heapify(a, 0, i)

def build_max_heap(a):

```

```

n = len(a)
for i in range(n//2-1, -1, -1):
    max_heapify(a, i, n)

def max_heapify(a,i,n):
    largest = i
    left = 2*i+1
    right = 2*i+2
    if left < n and a[left] > a[largest]:
        largest = left
    if right < n and a[right] > a[largest]:
        largest = right
    if largest != i:
        a[i], a[largest] = a[largest], a[i]
        max_heapify(a, largest, n)

```

## Counting sort

Counting sort goes through the array and records the count of every element. Then it adds the count of every value to the count of this value plus one, to get the correct position of each element. Then it creates another array and fills it up with the elements from the original array according to the order given by the count array.

```

def counting_sort(a):
    n = len(a)
    m = max(a)
    count = [0 for _ in range(m+1)]
    for e in a:
        count[e] += 1
    for i in range(1, m+1):
        count[i] += count[i-1]

    sorted = [0 for _ in range(n)]
    for i in range(n-1, -1, -1):

```

```
sorted[count[a[i]]-1] = a[i]
count[a[i]] -= 1
return sorted
```

## Shell sort

It is a variation of insertion sort, but this time it sorts the array in gaps (intervals).

```
def shell_sort(a):
    gap = len(a)//2
    while gap > 0:
        for i in range(gap, len(a)):
            temp = a[i]
            j = i - gap
            while j >= 0 and a[j] > temp:
                a[j+gap] = a[j]
                j -= gap
            a[j+gap] = temp
        gap //= 2
    return a
```