https://www.studytonight.com/operating-system/cpu-scheduling

Python - Sorting Algorithms

https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Bubble Sort

Bubble sort is the one usually taught in introductory CS classes since it clearly demonstrates how sort works while being simple and easy to understand. Bubble sort steps through the list and compares adjacent pairs of elements. The elements are swapped if they are in the wrong order. The pass through the unsorted portion of the list is repeated until the list is sorted. Because Bubble sort repeatedly passes through the unsorted part of the list, it has a worst case complexity of $O(n^2)$.

6 5 3 1 8 7 2 4

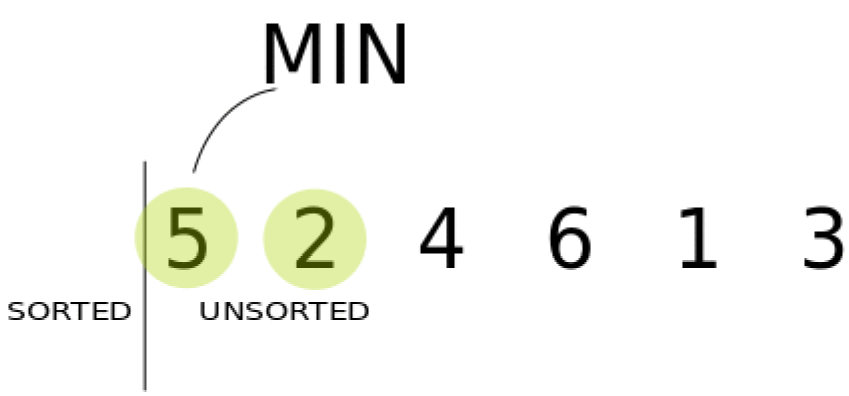
```
Defbubble_sort(arr):
        def swap(i, j):
            arr[i], arr[j] = arr[j], arr[i]
        n = len(arr)
        swapped = True

        x = -1
        while swapped:
            swapped = False
            x = x + 1
            for i in range(1, n-x):
                if arr[i - 1] > arr[i]:
                    swap(i - 1, i)
                    swapped = True

        return arr
```

Selection Sort

Selection sort is also quite simple but frequently outperforms bubble sort. If you are choosing between the two, it's best to just default right to selection sort. With Selection sort, we divide our input list / array into two parts: the sublist of items already sorted and the sublist of items remaining to be sorted that make up the rest of the list. We first find the smallest element in the *unsorted sublist* and place it at the end of the *sorted sublist.* Thus, we are continuously grabbing the smallest unsorted element and placing it in sorted order in the *sorted sublist.* This process continues iteratively until the list is fully sorted.



```
def
selection_sort(arr):
        for i in range(len(arr)):
            minimum = i

            for j in range(i + 1, len(arr)):
                # Select the smallest value
                if arr[j] < arr[minimum]:
                    minimum = j
            # Place it at the front of the
            # sorted end of the array
            arr[minimum], arr[i] = arr[i],
        arr[minimum]

        return arr
```

Insertion Sort

Insertion sort is both faster and well-arguably more simplistic than both bubble sort and selection sort. Funny enough, it's how many people sort their cards when playing a card game! On each loop iteration, insertion sort removes one element from the array. It then finds the location where that element belongs within another *sorted* array and inserts it there. It repeats this process until no input elements remain.

6 5 3 1 8 7 2 4

```
def
insertion_sort(arr):

        for i in range(len(arr)):
            cursor = arr[i]
            pos = i

            while pos > 0 and arr[pos - 1] >
      cursor:
                # Swap the number down the
      list
                arr[pos] = arr[pos - 1]
```

```
            pos = pos - 1
        # Break and do the final swap
        arr[pos] = cursor
    return arr
```

Merge Sort

Merge sort is a perfectly elegant example of a Divide and Conquer algorithm. It simple uses the 2 main steps of such an algorithm:

(1) Continuously *divide* the unsorted list until you have *N* sublists, where each sublist has 1 element that is “unsorted” and *N* is the number of elements in the original array.

(2) Repeatedly merge i.e *conquer* the sublists together 2 at a time to produce new sorted sublists until all elements have been fully merged into a single sorted array.

6 5 3 1 8 7 2 4

```
def
merge_sort(ar
```

r):

```
    # The last array split
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    # Perform merge_sort recursively on
both halves
    left, right = merge_sort(arr[:mid]),
merge_sort(arr[mid:])
    # Merge each side together
    return merge(left, right, arr.copy())
def merge(left, right, merged):
    left_cursor, right_cursor = 0, 0
    while left_cursor < len(left) and
right_cursor < len(right):

        # Sort each one and place into the
result
        if left[left_cursor] <=
right[right_cursor]:

merged[left_cursor+right_cursor]=left[left
```

```
_cursor]
            left_cursor += 1
        else:
            merged[left_cursor + right_cursor] = right[right_cursor]
            right_cursor += 1

    for left_cursor in range(left_cursor, len(left)):
        merged[left_cursor + right_cursor] = left[left_cursor]

    for right_cursor in range(right_cursor, len(right)):
        merged[left_cursor + right_cursor] = right[right_cursor]
    return merged
```

Quick Sort

Quick sort is also a divide and conquer algorithm like merge sort. Although it’s a bit more complicated, in most standard implementations it performs significantly faster than merge sort and rarely reaches its worst case complexity of O(n²). It has 3 main steps:

(1) We first select an element which we will call the *pivot* from the array.

(2) Move all elements that are smaller than the pivot to the left of the pivot; move all elements that are larger than the pivot to the right of the pivot. This is called the partition operation.

(3) Recursively apply the above 2 steps separately to each of the sub-arrays of elements with smaller and bigger values than the last pivot.

```
def
partition(array,
begin, end):
    pivot_idx = begin
    for i in xrange(begin+1, end+1):
        if array[i] <= array[begin]:
            pivot_idx += 1
            array[i],     array[pivot_idx]     =
        array[pivot_idx], array[i]
    array[pivot_idx],     array[begin]     =
```

```
array[begin], array[pivot_idx]
    return pivot_idx
def quick_sort_recursion(array, begin,
end):
    if begin >= end:
        return
    pivot_idx = partition(array, begin, end)
    quick_sort_recursion(array, begin,
pivot_idx-1)
    quick_sort_recursion(array,
pivot_idx+1, end)
def quick_sort(array, begin=0,
end=None):
    if end is None:
        end = len(array) - 1

    return quick_sort_recursion(array,
begin, end)
```