CompSci 105

Sorting Algorithms – Part 2

Shell Sort – another n^2 (or better!) sorting algorithm
Merge Sort – an $n \log(n)$ sorting algorithm

Textbook: Chapter 5

Note: we do not study quicksort in CompSci 105

Shell Sort or diminishing increment sort

Remember:

Insertion sort has fewer comparisons than selection sort
Selection sort has fewer moves-swaps than insertion sort

=> IDEA: compare/shift non-neighbouring elements

Shell sort
On average shell sort has fewer comparisons than selection sort and bubble sort and fewer moves than insertion sort

Shell sort is based on the insertion sort algorithm,
BUT: It instead of shifting elements many times by one step, it makes larger moves

Shell Sort or diminishing increment sort

Divide the list into lots of small lists, e.g., for the following list, say the gap (increment) used is **3**

Sublist 1
Sublist 2
Sublist 3

0	1	2	3	4	5	6
3	58	22	19	58	22	47

Insertion Sort 3, 19, 47
Insertion Sort 58, 58
Insertion Sort 22, 22

ONE ITERATION (with 3 sublists)

Then repeat sorting with reduced gap (=> fewer, but larger sublists) until gap is 1.

NOTE: The normal insertion sort algorithm uses a gap of 1, but in this algorithm sorting with gap 1 very efficient because list almost sorted due to previous steps.

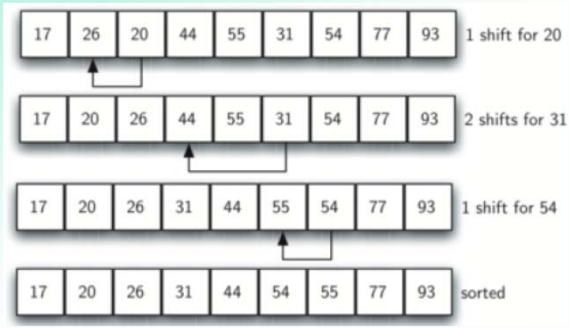
Shell Sort or diminishing increment sort

Example from book (page 182). Start with a gap of 3.
The results after the first pass are:

			0	1	2	3	4	5	6	7	8	
Initial	List		54	26	93	17	77	31	44	55	20	
Start 0	by 3											2 shift
Start 1	by 3											1 shift
Start 2	by 3											3 shift
End of pass 1												

Shell Sort or diminishing increment sort

Now use a gap of 1 (i.e., ordinary insertion sort):



Note: After previous step list is almost sorted => only four moves required for this final step

Shell Sort algorithm

Choose a gap size, do an insertion sort on all the sublists using this chosen gap size (this is a total of **one pass** of the collection), repeat using smaller gap sizes until finally the gap size is one.

In practice, it turns out that only occasionally there are small values on the right hand side. Therefore the final insertion sort needs to move few elements.

A default option for gap sizes is 2^k-1, i.e. [..., 31, 15, 7, 3,1]
Research in the optimal gap sequence is ongoing
A often quoted empirical derived gap sequence is [701, 301, 132, 57, 23, 10, 4, 1]

Shell Sort - Exercise

Start with a gap size of half the length of the list, halve the gap size after each pass (text book implementation). Show the elements at the end of each pass.

0	1	2	3	4	5	6	7	8	
54	26	93	17	77	31	44	55	20	List to sort
									PASS 1
									PASS 2
									PASS 3

Shell Sort - Code


```
def shell_sort(a_list):
    gap = len(a_list) // 2
    while gap > 0:
        for start_position in range(gap):
            gap_insertion_sort(a_list, start_position, gap)
        #print("for gap: ", gap, " - ", a_list)
        gap = gap // 2

def gap_insertion_sort(a_list, start, gap):
    for i in range(start + gap, len(a_list), gap):
        current_value = a_list[i]
        position = i
        while position >= gap and a_list[position - gap] > current_value:
            a_list[position] = a_list[position - gap]
            position = position - gap
        a_list[position] = current_value
```

NOTE: We use this gap sequence because it is simple and used in the text book. However, it is a poor choice (see slide 10)

Shell Sort – Code (continued)


```
def main():
    a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print("before: ", a_list)
    shell_sort(a_list)
    print("after:  ", a_list)

main()
```

```
before:  [54, 26, 93, 17, 77, 31, 44, 55, 20]
for gap:  4  -  [20, 26, 44, 17, 54, 31, 93, 55, 77]
for gap:  2  -  [20, 17, 44, 26, 54, 31, 77, 55, 93]
for gap:  1  -  [17, 20, 26, 31, 44, 54, 55, 77, 93]
after:   [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Shell Sort – Big O

This is an improvement on all the previous sorting algorithms.

The Big O for Shell Sort depends on the gap sequence and input values – in general between O(n) and O(n^2)

Gap sequence $n/2$, $n/4$, … , 1 => worst case O(n^2)

Gap sequence 2^k-1 (…, 31, 15, 7, 3, 1) => worst case O($n^{1.5}$)

Gap sequence …, 109, 41, 19, 5, 1 => worst case O($n^{1.333}$)

Merge Sort

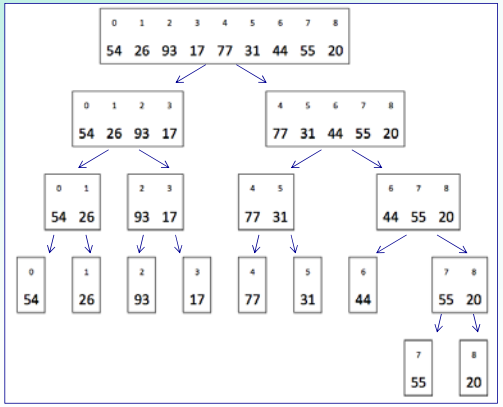
This is a divide and conquer algorithm.

- Cut the list in half
- Sort each half
- Merge the two sorted halves

You have already seen the divide and conquer algorithm using binary search on a sorted collection of items.

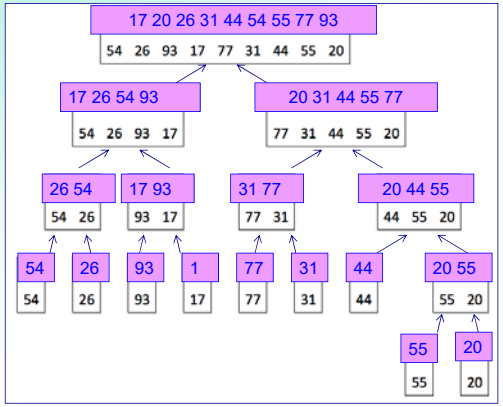
Merge Sort

Below is the call tree for the merge sort algorithm:



Merge Sort

Below is the tree of the merged parts returned (the pink parts) by the merge sort algorithm:



Splitting lists


```
def splitting_list_example():
    list = [54, 26, 93, 17, 20]
    listL = list[:2]        # elements 0 to 1
    listR = list[2:]        # elements 2 to end of list
    print(list, listL, listR)

def main():
    splitting_list_example()

main()
```

```
[54, 26, 93, 17, 20] [54, 26] [93, 17, 20]
```

Slicing will be useful when halving the list in the merge sort code.

Merging the two halves of the list


```
def merge(a_list, left_half, right_half):
    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            a_list[k] = left_half[i]
            i = i + 1
        else:
            a_list[k] = right_half[j]
            j = j + 1
        k = k + 1
    # CONTINUED ON RIGHT
```

same as

```
i = 0
j = 0
k = 0
```

```
# CONTINUED
while i < len(left_half):
    a_list[k] = left_half[i]
    i = i + 1
    k = k + 1

while j < len(right_half):
    a_list[k] = right_half[j]
    j = j + 1
    k = k + 1
```

Merging the two halves of the list


```
a = [0, 0, 0, 0, 0, 0, 0, 0]
merge (a, [1, 2, 5], [3, 4, 6, 8, 10])
print(a)
```

```
[1, 2, 3, 4, 5, 6, 8, 10]
```

```
[1, 2, 5]    [3, 4, 6, 8, 10]
[1, 2, 3, 4, 5, 6, 8, 10]
```

Merge sort Code


```
def merge_sort(a_list):
    if len(a_list) > 1:
        mid = len(a_list) // 2
        left_half = a_list[:mid]
        right_half = a_list[mid:]

        merge_sort(left_half)
        merge_sort(right_half)
        merge(a_list, left_half, right_half)

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
print("before: ", a_list)
merge_sort(a_list)
print("after: ", a_list)
```

Uses the function on slide 15 to merge the two halves.

```
before:  [54, 26, 93, 17, 77, 31, 44, 55, 20]
after:   [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Merge Sort – Big O

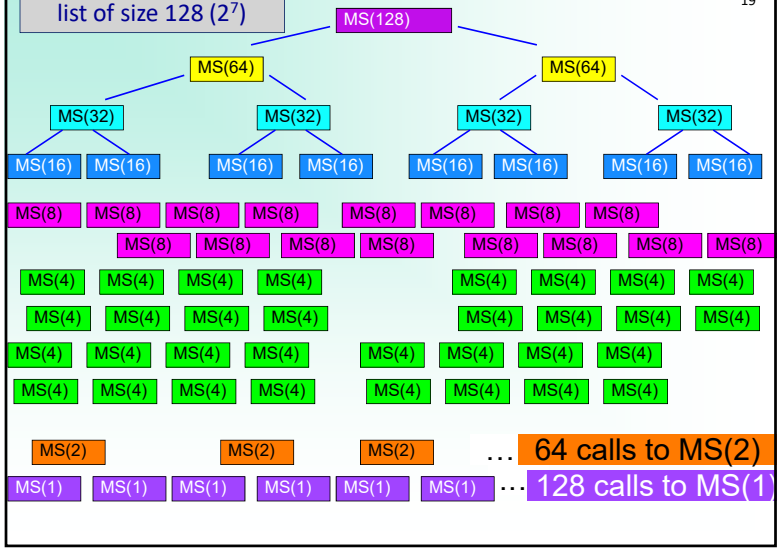
The time for sorting a list of size 1 is constant, i.e. T(1)=1
The time for sorting a list of size n is the time of sorting the two halves plus the time for merging, i.e.
T(n)=2*T(n/2)+n

Can proof: T(n) = n + n log n

=> Big O is **O(n log(n))**

But there is a penalty of having to use extra space for the two halves of a split list

Recursive call tree for a list of size 128 (2^7)



Summary

	Best	Worst	Average	Extra Memory
Bubble Sort (lecture)	O(n^2)	O(n^2)	O(n^2)	O(1)
Bubble Sort (optimised)	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Shell Sort (best gap sequence)	O(n)	O(n (log n)^2)	O(n (log n)^2)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Tim Sort (used in Python, hybrid of Merge Sort and Insertion Sort)	O(n)	O(n log n)	O(n log n)	O(n)

Note: A comparison based sorting algorithm can NOT be better than O(n log n) in the average and worst case