Computer Science Fundamentals 107 Lecture 23 Contents

Shell Sort – another n^2 sorting algorithm

Merge Sort – an **n log(n)** sorting algorithm

Textbook: Chapter 5

Note: we do not study quicksort in CompSci 107

Shell Sort or diminishing increment sort

Remember:

- insertion sort does fewer comparisons than selection sort but it does more moves.
- insertion sort is very efficient if the elements to be sorted have runs of sorted elements.

Shell sort

On average shell sort minimises moves and comparisons.

Shell sort is based on the insertion sort algorithm,

BUT: it makes **comparisons and moves between elements which are not contiguous** (instead of many small one step comparisons and moves)

Shell Sort or diminishing increment sort

Shell sort divides the list into lots of small lists, and does an insertion sort on those elements e.g., for the following list, say the gap (increment) used is **3**

0	1	2	3	4	5	6
[19,	53,	22,	47,	38,	21,	3]

Insertion Sort	19,			47,			3	ONE PASS
Insertion Sort		53,			38			
Insertion Sort			22,			21		

0	1	2	3	4	5	6
[3,	38,	21,	19,	53,	22,	47]

Then keep reducing the gap (increment) until the gap is 1.

The normal insertion sort algorithm uses a gap of 1.

Shell Sort or diminishing increment sort

Example from book (page 182). Start with a **gap of 3**.
Below shows the first pass (just one pass):

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

ONE PASS

Shell Sort or diminishing increment sort

Now using a **gap of 1** (i.e., an ordinary insertion sort).
Below is the second (and last) pass:

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

ONE PASS

Notice: for this last pass, there were only four swaps required (one move inserting 20, two moves inserting 31 and one move inserting 54).

Customarily the gap size is halved (floor division - //) after each pass.

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.

Generally, we choose a starting gap size of half the length of the list and halve this gap size (floor division - //) after each pass.

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

Shell Sort - Exercise

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

```
 0   1   2   3   4   5   6   7   8
54  31  93  55  77  26  44  17  20
```

List to sort

PASS 1 – gap size 4

PASS 2 – gap size 2

PASS 3 – gap size 1

Shell sort – uses next slide code

```
def shell_sort(a_list):



def gap_insertion_sort(a_list, start, gap): #see next slide

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

Shell sort code continued

```
def gap_insertion_sort(a_list, start, gap):




def shell_sort(a_list):    #see previous slide

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

Shell Sort – Big O

This is an improvement on all the previous n^2 sorting algorithms.

The Big O for shell sort can be shown to be between **O(n)** and **$O(n^2)$**

Choosing good values for the gaps can make the shell sort performance noticeably better than **$O(n^2)$**.

Remember that insertion sort is quite good for sorting small lists. Therefore making the sublists small and using insertion sort for the small sublists makes this sort more efficient.

Merge Sort

This is a divide and conquer algorithm.

Cut the list in half.

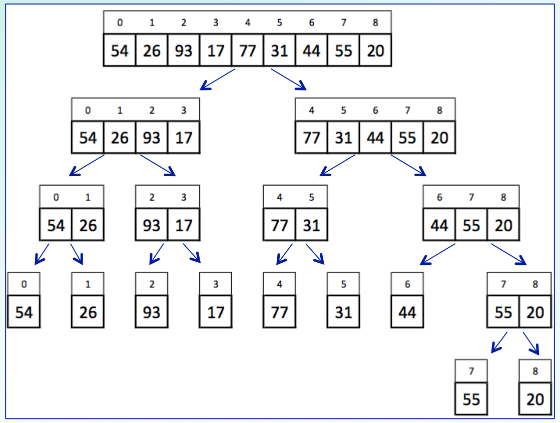
Sort each half.

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 recursive call tree for the merge sort algorithm:



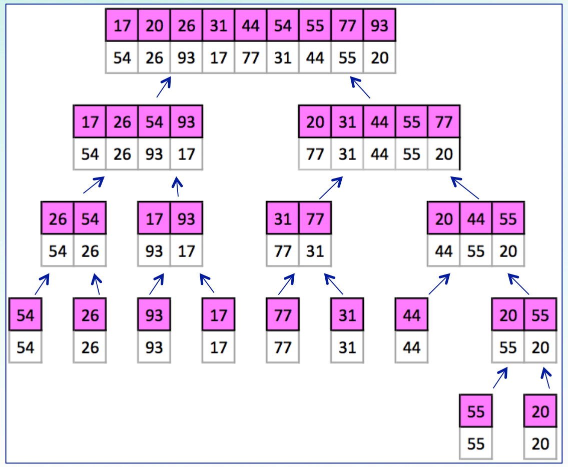
1. Cut the list in half

2. Call mergesort with each half

3. Merge the two sorted halves

Merge Sort

Below is the call tree showing the merged parts (the pink parts) 'returned' by the mergesort algorithm:



1. Cut the list in half
2. Call mergesort with each half
3. **Merge the two sorted halves**

Code used - slicing lists

```
def slicing_lists_example():
  list1 = [54, 26, 93, 17, 20]
  list2 = list1[:2]
  list3 = list1[2:]
  print(list1, list2, list3)
  print(list1==list2, list2==list3, list1==list3)

def main():
  slicing_lists_example()

main()
```

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

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

Code used - initialising variables

```
def main():
  i = j = k = 0

main()
```

Same code.

```
def main():
  i = 0
  j = 0
  k = 0

main()
```

Merging the two halves of the list

```
def merge_two_halves(a_list, left_half, right_half):



def main():
  a = [0, 0, 0, 0, 0, 0]
  merge_two_halves(a, [54, 76, 93], [24, 98])
  print(a)

main()
```

```
[24, 54, 76, 93, 98]
```

Merging the two halves of the list

```
def merge_two_halves(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]:
```

Merge sort Code

```
def merge_sort(a_list):





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

Use the merge_two_halves() function on slide 17 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

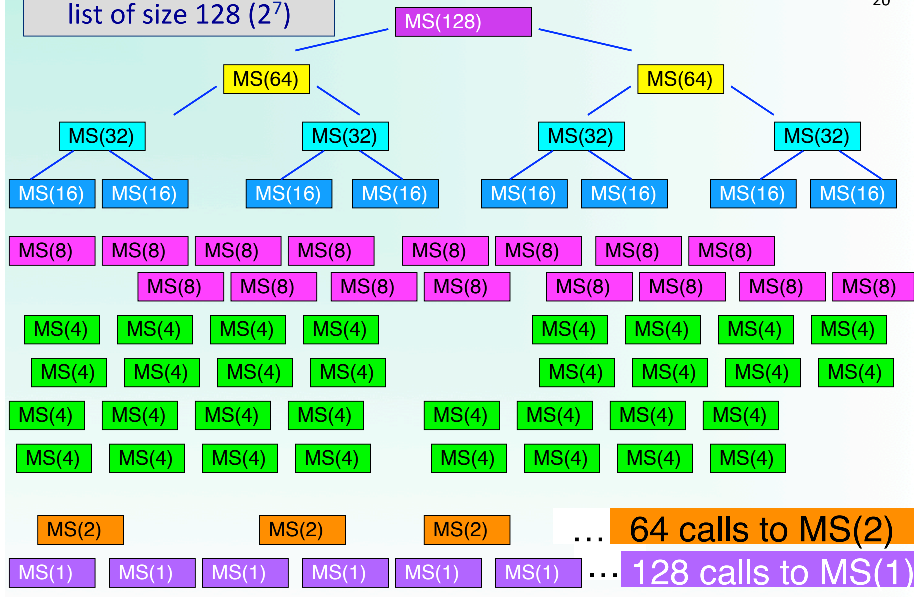
In a similar way to the binary search algorithm we halve the list every time the merge sort method is called (halve the problem size). This gives us a recursive call tree with height O(log n), i.e., for 2^n elements there are n levels of calls.

At each level of the recursion, we have to merge the values from the two sorted halves back together. This moves every element in the list - O(n).

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

But there is a the disadvantage of having using extra space when halving the list to pass to the recursive calls (if we use splicing).

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


MS(128)
MS(64)
MS(64)
MS(32)
MS(32)
MS(32)
MS(32)
MS(16)
MS(8)
MS(4)
MS(2)
MS(1)
…
64 calls to MS(2)
…
128 calls to MS(1)