COSC 2306
Data Programming

Sort

Insertion sort coding

```
def insertion_sort(a_list):
    for index in range(1, len(a_list)): #index of pass
        current_value = a_list[index]
        position = index
        while position > 0 and a_list[position - 1] > current_value:
            #index of comparison during insertion
            a_list[position] = a_list[position - 1]
            position = position - 1
        a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertion_sort(a_list)
print(a_list)
```

Insertion Sort

- Analysis
  - Worst case: $O(n^2)$
  - Insertion sort is better than selection sort when the array is partially sort or almost sort
  - Memory friendly (adjacent elements)
  - For small arrays
    - Insertion sort is appropriate due to its simplicity
  - For large arrays
    - Insertion sort is still slow

Shell Sort

Shell sort description

- **Shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position
  - shell sort allows swap in a bigger but fewer steps

Shell sort

- How to reduce the number of item movements in insertion sort?
- List elements viewed as sublists at a particular distance
  - Each sublist sorted
  - Elements far apart move closer to their final position

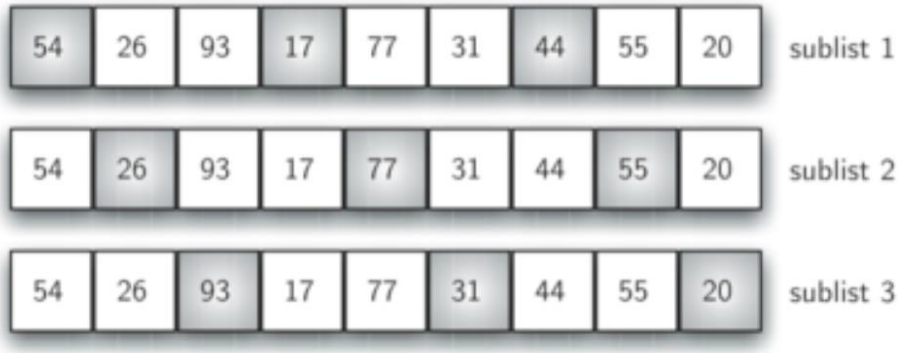


Figure 6: A Shell Sort with Increments of Three

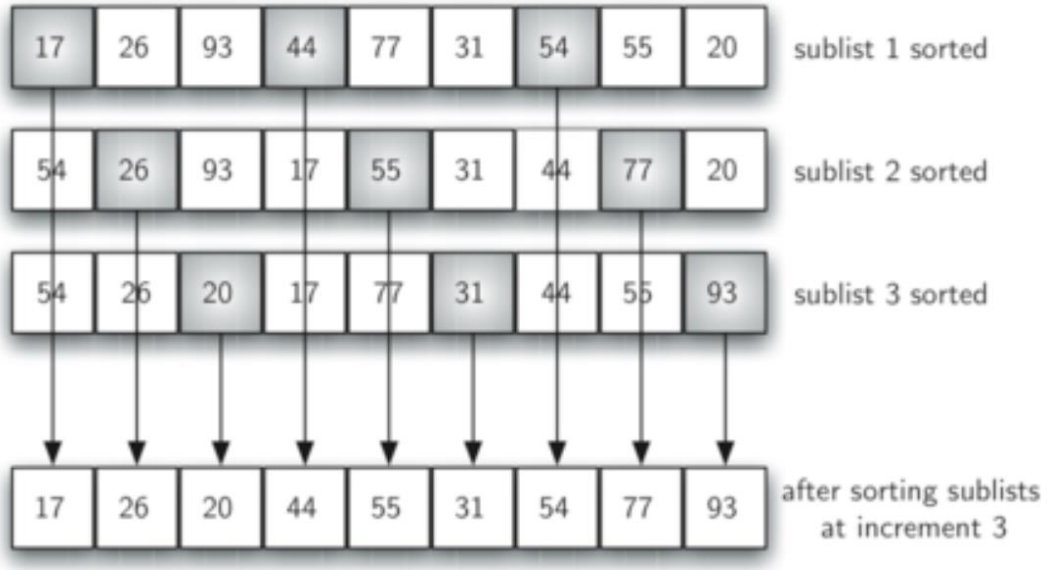


Figure 7: A Shell Sort after Sorting Each Sublist

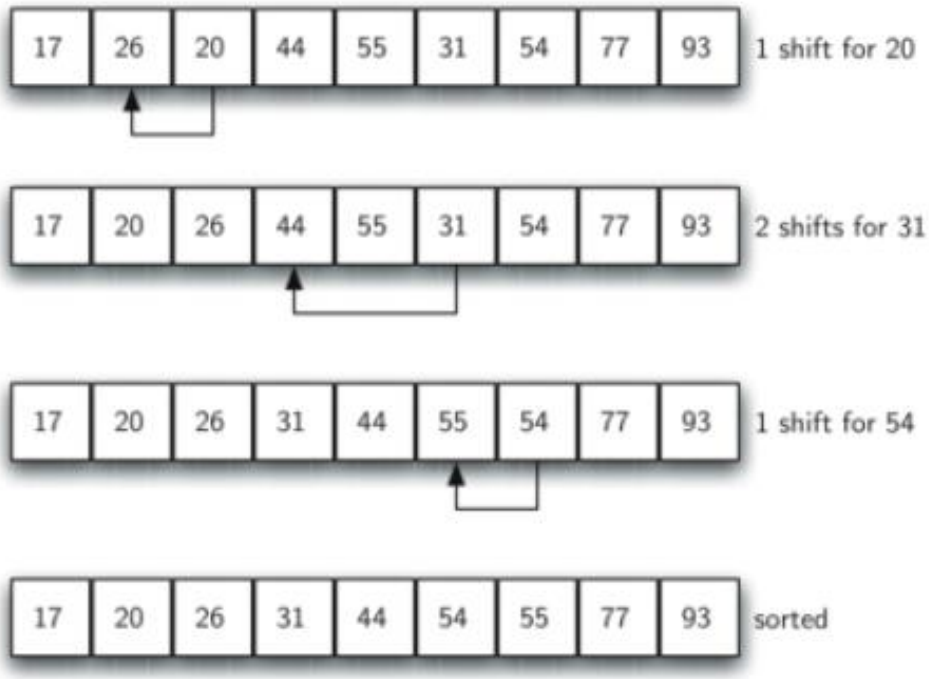


Figure 5.20: Shell Sort: A Final Insertion Sort with Increment of 1

Shell sort coding

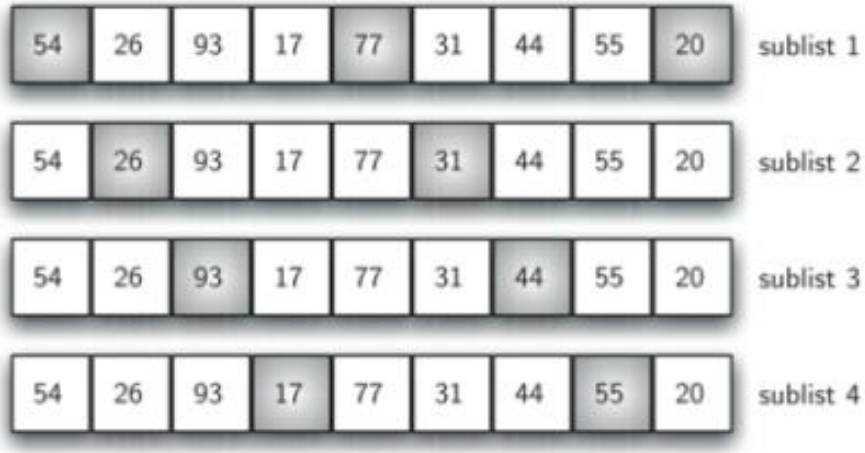


Figure 5.21: Initial Sublists for a Shell Sort

- Begin with $n/2$ sublists
- On the next pass, $n/4$ sublists are sorted
- Eventually, a single list is sorted with the basic insertion sort

Naturally, a loop here to reduce the number of sublists

Use insertion sort for each sublist
Naturally, another loop here to loop over sublists

Shell sort coding

```
def shell_sort(a_list):
    sublist_count = len(a_list) // 2
    while sublist_count > 0:
        for pos_start in range(sublist_count):
            gap_insertion_sort(a_list, pos_start, sublist_count)
        sublist_count = sublist_count // 2

def gap_insertion_sort(a_list, start, gap):
    for index in range(start + gap, len(a_list), gap):
        current_value = a_list[index]
        position = index
        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
```

```
a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
shell_sort(a_list)
print(a_list)
```

Shell sort analysis

The complexity falls somewhere between $O(n)$ and $O(n^2)$

Shell sort summary

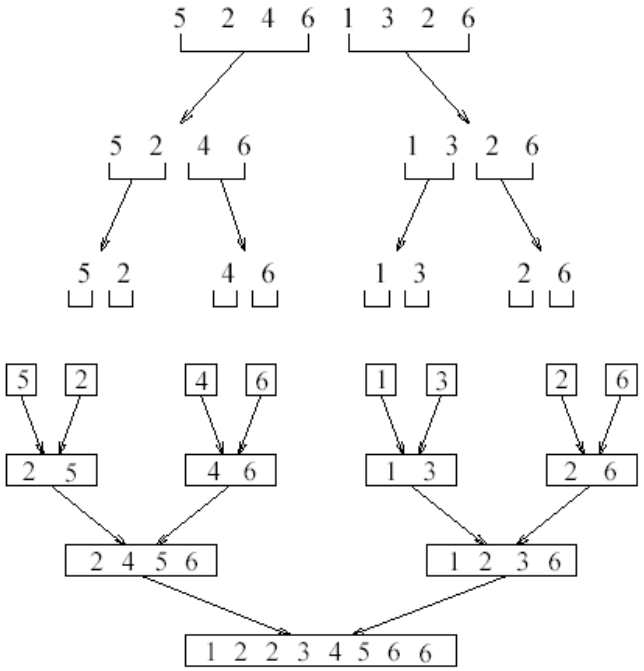
- Shell sort algorithm is a generalization of the insertion algorithm
- Two key advantages of shell sort:
  - Makes the list more ordered
  - Allows elements to be moved longer distance to reduce the swapping cost
- To determine the gap, we can use n/2, n/4, … n/i till n/i = 1 sublists
- Complexity is between O(n) and $O(n^2)$

Merge Sort

Merge Sort: Main Idea

Based on divide-and-conquer strategy

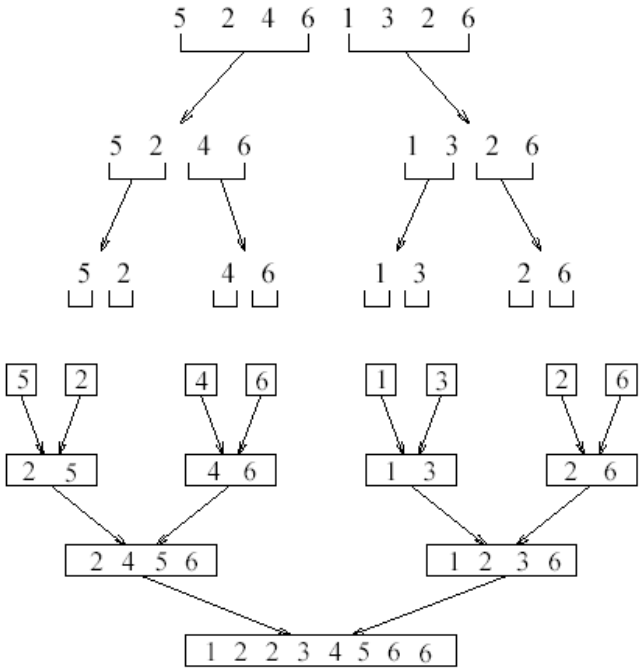
- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list



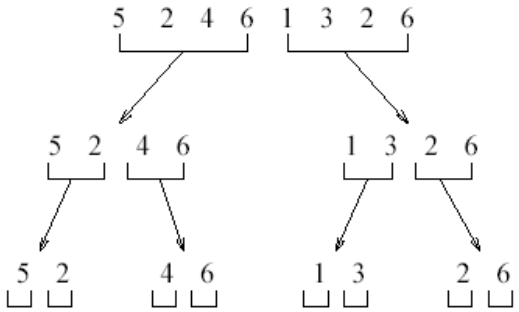
Merge Sort: Main Idea

Questions:

- How do we split the list?
- How do we merge the two sorted lists?



Split

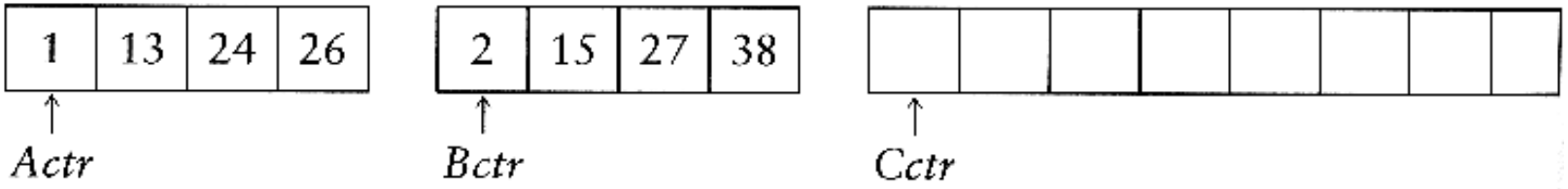


If the input list is an array A[*0..N-1*]:

- Represent a sub-array by two integers *left* and *right*.
- To divide A[*left* .. *right*], compute *center*=(*left*+*right*)/2
- Obtain A[*left* .. *center*] and A[*center*+1 .. *right*]

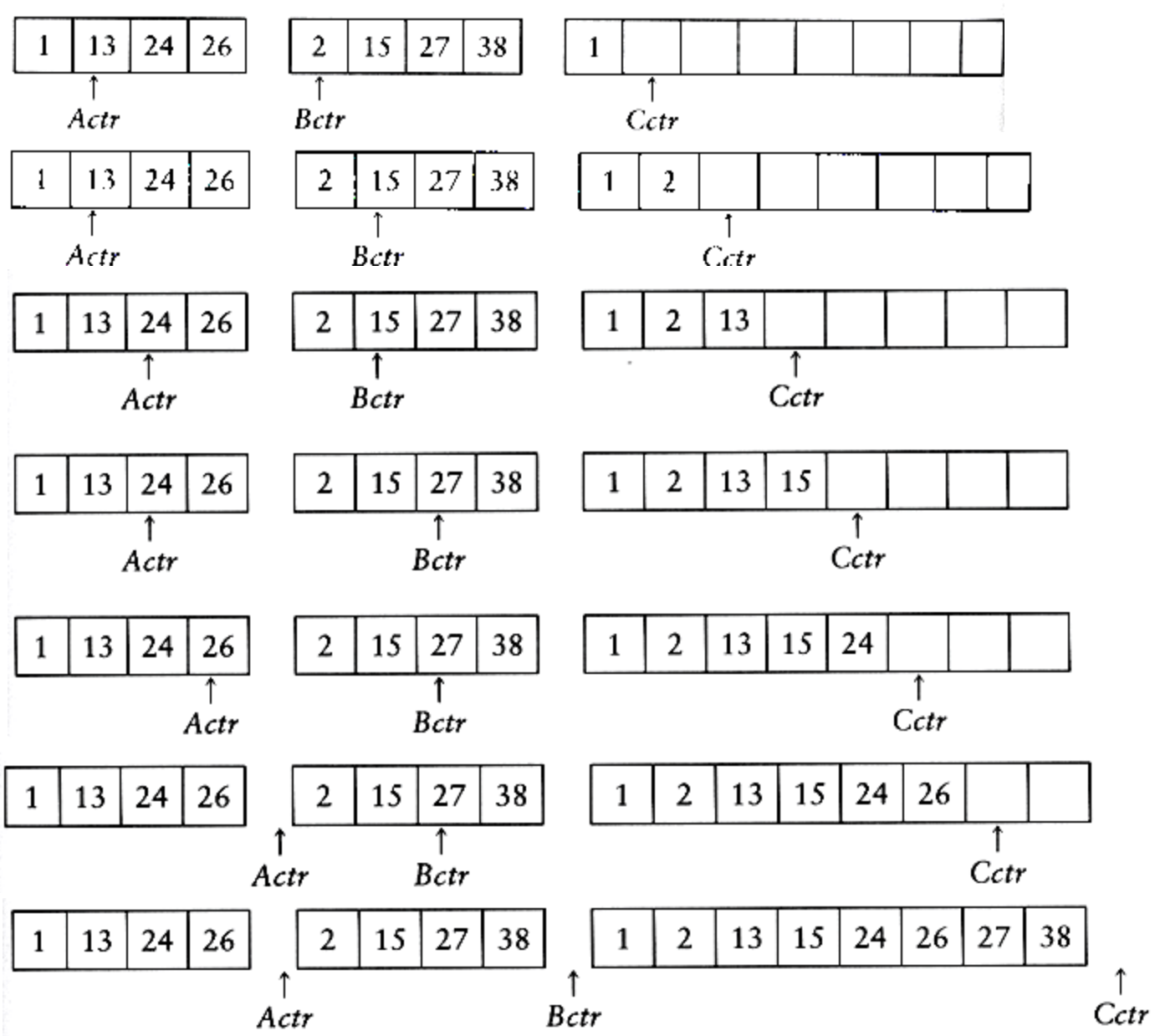
Merge

- Input: two sorted array A and B
- Output: an output sorted array C
- Three references: *Actr*, *Bctr*, and *Cctr*
  - initially set to the beginning of their respective arrays



  - The smaller of A[*Actr*] and B[*Bctr*] is copied to the next entry in C[*Cctr*], and the appropriate counters are advanced
  - When either input list is exhausted, the remainder of the other list is copied to C directly

Merge: Example



Merge Sort Coding

- Write a recursive function mergeSort(array) to do merge sort.

```
Base case: if the array has only a single
element, it is already sorted.

General case: length of array > 1; you split it
into two halves, recursively sort each half
(two recursive calls), then merge the two
sorted halves into one sorted array.
```

Merge Sort Coding

```
def mergeSort(array):
    if len(array) <= 1: #base case
        return array
    else: #general case
        mid = len(array)//2
        L = array[:mid]
        R = array[mid:]
        #Sort the two halves
        mergeSort(L)
        mergeSort(R)
        #Until reach end of L/R, pick larger
        #between L and R and place in merged
        a = b = c = 0 #a for L, b for R, c for merged
        while a < len(L) and b < len(R):
            if L[a] < R[b]:
                array[c] = L[a]
                a += 1
            else:
                array[c] = R[b]
                b += 1
            c += 1
        #When we run out of L or R,
        #put the remaining directly
        #into merged
        while a < len(L):
            array[c] = L[a]
            a += 1
            c += 1
        while b < len(R):
            array[c] = R[b]
            b += 1
            c += 1
        return array
```

Analysis of Merge Sort

- Divide step: O(1)
- Conquer step: 2 x T(N/2) time
- Combine step: O(N) time

- Recursive equation: $T(N) = 2T(N/2) + N$

- Recursion: $T(N) = 2[2*T(N/4) + N/2] + N = \ldots = 2^k*T(N/2^k) + k*N$

- Solve it by let $N/2^k = 1$, then $k = \log(n)$ (binary search)

- We have $T(N) = N + N*\log N$

- Overall complexity: O(nlogn)