Insertion Sort

Algorithm InsertionSort(A):
Input: An array A of n comparable elements
Output: The array A with elements rearranged in nondecreasing order
for k from 1 to n − 1 **do**
Insert A[k] at its proper location within A[0], A[1], . . ., A[k].

Code Fragment 5.9: High-level description of the insertion-sort algorithm.

Sequence → unsorted ← 3 2 5 7 4

- We will divided it into two sub-list

3 | 2 5 7 4
"Sorted" " unsorted List"

Visually

As the Algorithm starts, we will take the item in the very Left position of the unsorted sequence

3 | (2) 5 7 4
Sorted Unsorted List

- and move it into the sorted sub list

3 2 | 5 7 4
Sorted Unsorted List

- once it's in the sorted sublist we will Compare that Value to the Value of it's Left

(3)(2) | 5 7 4
Sorted Unsorted List

If the Value is higher, then we change position of these two items

Compare

(3)(2) | 5 7 4
Sorted Unsorted List

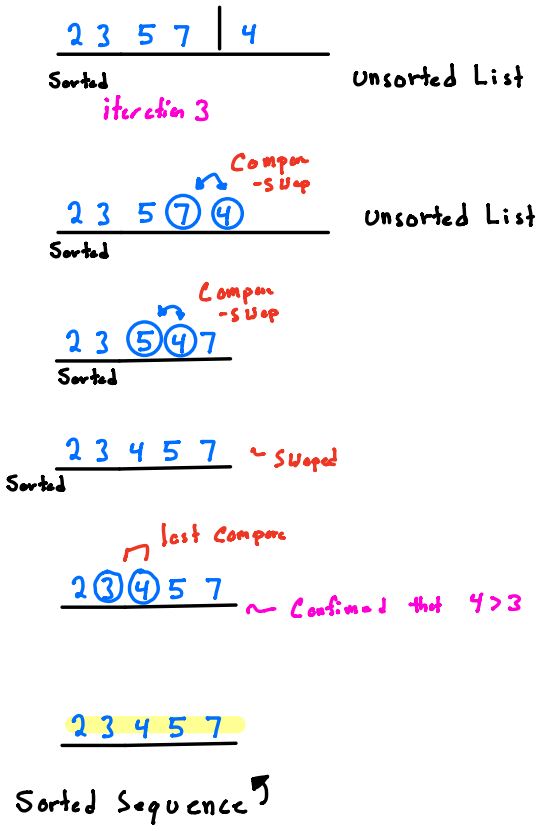
Changed position of two items

2 3 | 5 7 4
Sorted Unsorted List

iteration 1

- We will Continue down the sorted sublist doing this over and over until we find an item that is not higher than the item were trying to sort

2 3 5 | 7 4 ✓
Sorted Unsorted List
iteration 2



Python Code:

list to be sorted

```
def insertion_sort(list_a):

    indexing_length = range(1, len(list_a))

    for i in indexing_length:

        value_to_sort =list_a[i]

        while list_a[i-1] > value_to_sort and i>0: # python allows negative indexing

            list_a[i], list_a[i-1] = list_a[i-1], list_a[i]

            i = i -1

    return list_a

print(insertion_sort([2,12,123,122,42,312,42,124,9,10,1,2,3,45,9,0]))
```

length of List passed thru

Basically how many iterations we need to go thru the unsorted list

i will start @ 1, since we started our indexing from 1 to end of the list_a.

Check each Value in our Un sorted list

Value will switch position

Output:

```
[0, 1, 2, 2, 3, 9, 9, 10, 12, 42, 42, 45, 122, 123, 124, 312]
```

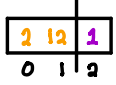
How the Algorithm works.

```
def insertion_sort(list_to_sort):
    indexing_length = range(1, len(list_to_sort))
    for i in indexing_length:
        value_to_sort = list_to_sort[i]

        while list_to_sort[i-1] > value_to_sort and i>0: # python allows negative indexing
            list_to_sort[i], list_to_sort[i-1] = list_to_sort[i-1], list_to_sort[i]
            i = i -1  # move down the sorted list to check if number is less than the previous number
    return list_to_sort


print(insertion_sort([2,12,1,0,42,312,42,124,9,10,1,2,3,45,9,0]))
'''output'''
[0, 0, 1, 1, 2, 2, 3, 9, 9, 10, 12, 42, 42, 45, 124, 312]
```

Index 0 1 2 3 4

iteration 2

Sorted list RN



i in indexing_length = 2

i = 2

```
value_to_sort = list_to_sort[i]
```

value_to_sort = 1

```
while list_to_sort[i-1] > value_to_sort and i>0:
```

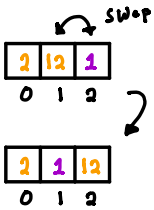
i-1 = 2

12 > 1 and i > 0 TRUE

```
list_to_sort[i], list_to_sort[i-1] = list_to_sort[i-1], list_to_sort[i]
```

i = 2, i-1 = 1

swap




```
i = i -1
```

i = 2-1

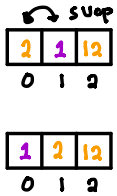
i = 1

```
while list_to_sort[i-1] > value_to_sort and i>0: # python
```

2 > 1 and i > 0 ✓

```
list_to_sort[i], list_to_sort[i-1] = list_to_sort[i-1], list_to_sort[i]
```

swap




```
i = i -1
```

i = 1-1 = 0

Fails

```
while list_to_sort[i-1] > value_to_sort and i>0: # python
```

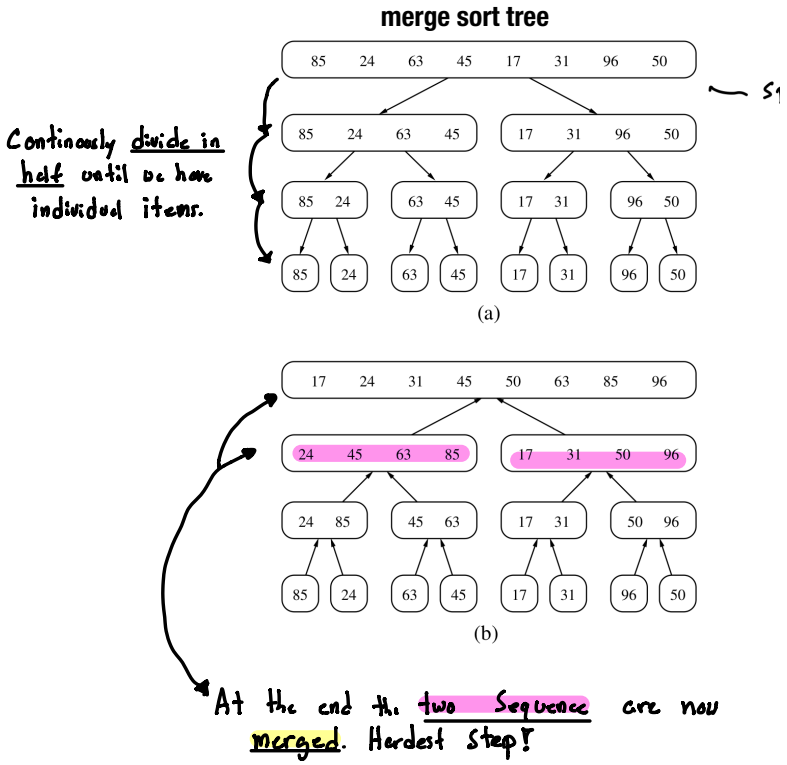
..... Next iteration

~ Chapter 12

Merge Sort

recurrence or repetition

- Each node on the tree (T) represents a recursive invocation (or call) of the merge sort algorithm.
- Divide and conquer Algorithm
- Split object(list) into two sequences and sort each one, then merge them at the end.
- Dividing the sequence by $\lceil \frac{n}{2} \rceil$
- Very efficient for large data sets.



Continously divide in half until we have individual items.

At the end the two sequence are now merged. Hardest step!

Proposition 12.1: *The merge-sort tree associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.*

Array Based Merge

- Python list

```
1  def merge(S1, S2, S):
2    """Merge two sorted Python lists S1 and S2 into properly sized list S."""
3    i = j = 0
4    while i + j < len(S):
5      if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6        S[i+j] = S1[i]              # copy ith element of S1 as next item of S
7        i += 1
8      else:
9        S[i+j] = S2[j]              # copy jth element of S2 as next item of S
10       j += 1
```

```
1  def merge_sort(S):
2    """Sort the elements of Python list S using the merge-sort algorithm."""
3    n = len(S)
4    if n < 2:
5      return                      # list is already sorted ✓
6    # divide
7    mid = n // 2
8    S1 = S[0:mid]                 # copy of first half
9    S2 = S[mid:n]                 # copy of second half
10   # conquer (with recursion)
11   merge_sort(S1)                # sort copy of first half
12   merge_sort(S2)                # sort copy of second half
13   # merge results
14   merge(S1, S2, S)              # merge sorted halves back into S
```

Example of the code ran:

D = [85, 24, 63, 45, 17, 21, 96, 50]

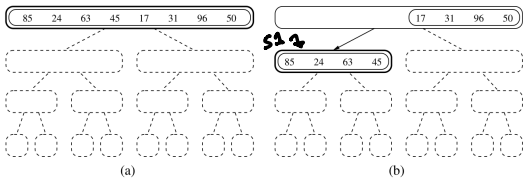
Merge_sort(D)

print(D)

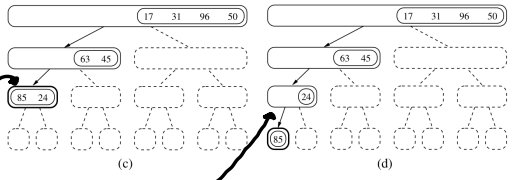
Returned: [17, 24, 31, 45, 50, 63, 85, 96]

How the Code works:

1) The list is passed in the merge_sort function

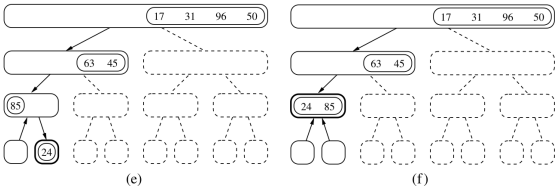


@ each node it splits the sequence in half.



Merge_sort(S2)
→ S1 = [85, 24]
S2 = [63, 95]

Merge_Sort([85, 24])
S1 = [85]
S2 = [24]



Since n < 2
return
S2 = [24]
Merge_sort(S2)

Since n is < 2 for S2,
we go to merge(S1, S2, S)

S = [24, 85]

S1 = [85]
S2 = [24]
S = [85, 24]

Confused here