

# Searching and Sorting

## Lecture-11

D.S. Hwang

Department of Computer Science  
Dankook University

# Outline

Linear Search

Binary Search

Sorting

More Efficient Sorting Algorithm

Mergesort

Summary

# Types in Computer Science

- ▶ Involves studying how to organize, store, and retrieve data.
- ▶ Searching and sorting are fundamental parts of programming.
- ▶ These algorithms can show a good way to explain how computer scientists compare the efficiency of different algorithms and to explain what they mean by efficiency.
- ▶ We will cover several algorithms for searching and sorting lists and then use them to explore what it means for one algorithm to be faster than another.

# Linear Search

Python lists have a method index that searches for a particular item in linear.

```
1 import random
2 lst = [];
3 for i in range(10):
4     lst.append( random.randint(1,100) );
5 print(lst)
6 print ('The index of the minimum value is %d.'
7       % lst.index(min(lst)));
8 print('The samllest value is %d.'
9       % min(lst));
10
11 print('The index of the minimum value is %d.'
12       % lst.index(min(lst[1:5]),1,5));
13 print('The samllest value is %d.'
14       % min(lst[1:5]));
```

92

48	88	3	81	92	64	44	96
----	----	---	----	----	----	----	----

# Linear Search

# Basic Linear Search



- ▶ uses variable  $i$  as the current index and marches through the values in  $L$
- ▶ The first check in the loop condition,  $i \neq \text{len}(L)$ , makes sure we look at only valid indices.
- ▶ The second check,  $L[i] \neq v$ , causes the loop to exit when we find  $v$ .
- ▶ return  $n$  if  $v$  is not in  $L$ .

```
1 def linear_search(v, L):  
2     i = 0  
3     while  $i \neq \text{len}(L)$  and  $L[i] \neq v$ :  
4         i = i + 1  
5     return i
```

# Basic Linear Search

- ▶ The first version requires two checks each time through the loop.
- ▶ The first check `i != len(L)` is almost unnecessary until `i` becomes `len(L)`.
- ▶ We can fix this trouble.

```
1 def linear_search(v, L):  
2     i = 0  
3     for value in L:  
4         if value == v:  
5             return i  
6         i += 1  
7     return len(L)
```

# Sentinel Search

- ▶ The problem of the basic linear search is that we check  $i \neq \text{len}(L)$  every time through the loop even though it can never be false except when  $v$  is not in  $L$ .
- ▶ We will add  $v$  to the end of  $L$ .

```
1 def linear_search(v, L):  
2     ✓ L.append(v) # Append v to L  
3     i = 0  
4     while L[i] != v:  
5         i = i + 1  
6     ✓ L.pop() # Remove the sentinel.  
7     return i
```



# Timing the Searches

We used to time the three mentioned searches on a list.

- ▶ use module time
- ▶ time() returns the current time in seconds.
- ▶ time\_it() returns how long the search takes.

```
1 import time
2 import linear_search_1
3 import linear_search_2
4 import linear_search_3
5
6 def time_it(search, v, L):
7     ✓ t1 = time.time()
8     search(v, L)
9     ✓ t2 = time.time()
10    return (t2 - t1) * 1000.0
```

# Timing the Searches

```
1 def print_times(v, L):  
2     # Get list.index's running time.  
3     t1 = time.time()  
4     L.index(v)  
5     t2 = time.time()  
6     index_time = (t2 - t1) * 1000.  
7  
8     # Get the other three running times.  
9     basic_time = time_it(linear_search_1.linear_search, v, L)  
10    for_time = time_it(linear_search_2.linear_search, v, L)  
11    sentinel_time = time_it(linear_search_3.linear_search, v, L)  
12    print("%d\t%.02f\t%.02f\t%.02f\t%.02f" %  
13          (v, basic_time, for_time, sentinel_time, index_time))
```

# Timing the Searches

Running times for linear search(milliseconds)

Case	✓ basic	✓ for	✓ sentinel	✓ list.index
First	0.01	0.01	0.03	0.01
Middle	138	69	62	17
Last	273	139	124	35

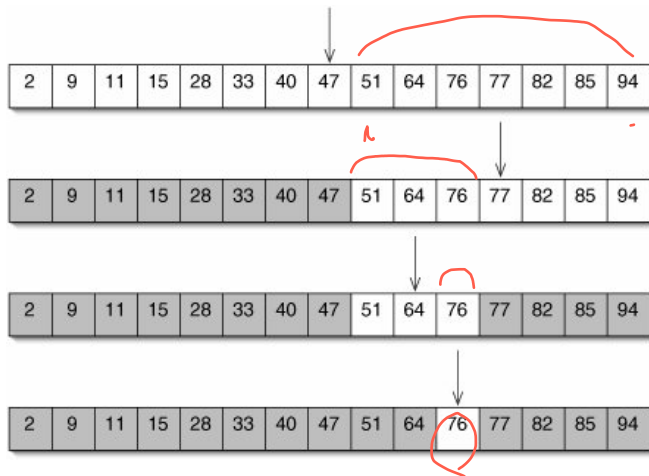
# Binary Search

Binary search assumes that the items of a list is sorted.

- ▶ Compare  $v$  to the middle value.
- ▶ Narrow the sublist having  $v$ .
- ▶ Repeat the above steps until  $v$  is found or not.

```
1 def binary_search(v, L):  
2     i = 0; j = len(L) - 1  
3     while i != j + 1:  
4         m = (i + j) / 2  
5         if L[m] < v:  
6             i = m + 1  
7         else:  
8             j = m - 1  
9     if 0 <= i < len(L) and L[i] == v:  
10        return i  
11    else:  
12        return -1
```

# Binary Search



# Binary Search

$N$  values can be searched in roughly  $\log_2 N$  steps.

```
1 >>> x
2 [10, 50, 100, 1000, 10000, 100000, 1000000]
3 >>> print(math.log(x))
4 [ 2.30258509  3.91202301  4.60517019  6.90775528  9.21034037
5 11.51292546 13.81551056]
6 >>>
```

Running times for binary search(milliseconds)

Case	list.index	binary_search	Ratio
✓ First	0.03	0.05	0.66
✓ Middle	107	0.04	2643
✓ Last	15.73	0.04 (Wow!)	4304

# Sorting

The following data shows the number of acres burned in forest fires in Canada from 1918 to 1987.

563	7590	1708	2142	3323	6197	1985	1316	1824	472
1346	6029	2670	2094	2464	1009	1475	856	3027	4271
3126	1115	2691	4253	1838	828	2403	742	1017	613
3185	2599	2227	896	975	1358	264	1375	2016	452
3292	538	1471	9313	864	470	2993	521	1144	2212
2212	2331	2616	2445	1927	808	1963	898	2764	2073
500	1740	8592	10856	2818	2284	1419	1328	1329	1479

Let us find out how much forest was destroyed in the N worst years.

# Sorting

Let us find out how much forest was destroyed in the  $N$  worst years.

- ▶ Sort the data in ascending order.
- ▶ Take the last  $N$  values of the sorted list.

```
1 def find_largest(N, L):  
2     """Return the N largest values in L  
3     in order from smallest to largest."""  
4     copy = L[:]  
5     copy.sort()  
6     return copy[:N]
```

$N$ :

$[N:]$



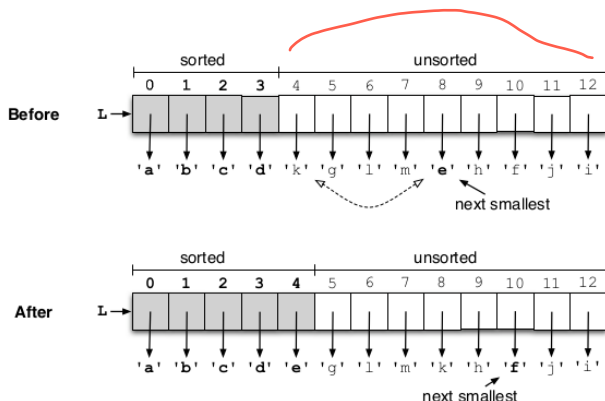
# Selection Sort

- ▶ Partition the data into the sorted and unsorted subarrays.
- ▶ Find the  $i$ th smallest value and move it to its position.

```
1 def selection_sort(L):  
2     """Reorder the items in L from smallest to largest."""  
3     i = 0  
4     while i != len(L):  
5         # Find the index of the smallest item in L[i:].  
6         # Swap that smallest item with L[i].  
7         i = i + 1
```

# Selection Sort

- ▶ Partition the data into the sorted and unsorted subarrays.
- ▶ Find the  $i$ th smallest value and move it to its position.



# Selection Sort


```
1 def selection_sort(L):
2     i = 0
3     while i != len(L):
4         # Find the index of the smallest item in L[i:].
5         L[i], L[smallest] = L[smallest], L[i]
6         i = i + 1
7
8     ✓ def find_min(L, b):
9         smallest = b
10        i = b + 1;
11        while i != len(L):
12            if L[i] < L[smallest]:
13                smallest = i;
14            i = i + 1
15        return smallest
```

# Selection Sort

```
1 def selection_sort(L):
2     i = 0
3     while i != len(L):
4         smallest = find_min(L, i)
5         L[i], L[smallest] = L[smallest], L[i]
6         i = i + 1
7
8 def find_min(L, b):
9     smallest = b
10    i = b + 1;
11    while i != len(L):
12        if L[i] < L[smallest]:
13            smallest = i;
14        i = i + 1
15    return smallest
```

# Insertion Sort

- ▶ Keep a sorted section at the beginning of the list
- ▶ Take the next item from the unsorted section and insert it where it belongs in the sorted section



```
1 def insertion_sort(L):  
2     """Reorder the values in L from smallest  
3     to largest."""  
4     i = 0  
5     while i != len(L):  
6         # Insert L[i] where it belongs in L[0:i+1].  
7         i = i + 1
```

# Insertion Sort

```
1 def insertion_sort(L):
2     """Reorder the values in L from smallest
3     to largest."""
4     i = 0
5     while i != len(L):
6         insert(L, i)
7         i = i + 1
8
9 def insert(L, b):
10     """Insert L[b] where it belongs in L[0:b + 1];
11     L[0:b - 1] must already be sorted."""
12     i = b
13     while i != 0 and L[i - 1] >= L[b]:
14         i = i - 1
15         value = L[b]
16     del L[b]
17     L.insert(i, value)
```

# Selection vs. Insertion Sort

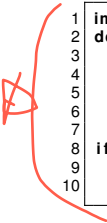
List Length	Selection Sort	Insertion Sort
10	0.1	0.1
1000	481	300
2000	1640	1223
3000	3612	2772
4000	6536	4957
5000	10112	7736
10000	40763	32382

Why is insertion sort slightly faster?

- ▶ On average, only half of the values need to be scanned in order to find the location in which to insert the new value.
- ▶ In selection sort, every value in the unsorted section needs to be examined in order to select the smallest one.

# Binary Sort

- ▶ Use module bisect
- ▶ Since  $N$  values have to be inserted, the overall running time ought to be  $N \log_2 N$ .
- ▶ To create an empty slot in the list, we have to move all the values above that slot up one place.
- ▶ The total time is  $N(N + \log_2 N) \approx N^2$



```
1 import bisect
2 def bin_sort(values):
3     '''Sort values, creating a new list.'''
4     result = []
5     for v in values:
6         bisect.insort_left(result, v)
7     return result
8 if __name__ == "__main__":
9     # Create a list L
10    sorted = bin_sort( L )
```

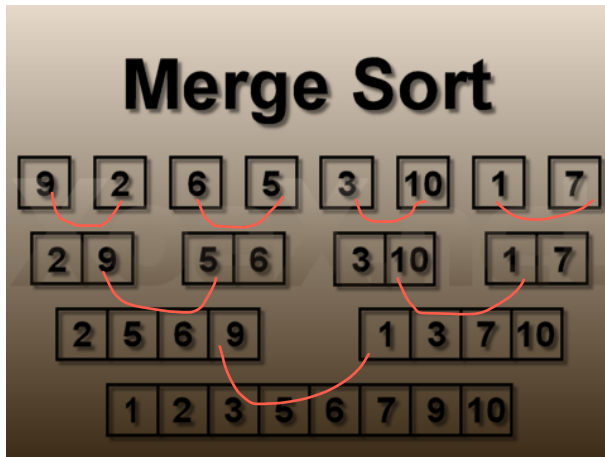


# Merge Sort

- ▶ Take two sorted lists
- ▶ Merge them into the sorted list

```
1 def merge(L1, L2):
2     newL = []
3     i1 = 0; i2 = 0
4     while i1 != len(L1) and i2 != len(L2):
5         if L1[i1] <= L2[i2]:
6             newL.append(L1[i1])
7             i1 += 1
8         else:
9             newL.append(L2[i2])
10            i2 += 1
11     # Gather any leftover items from the two sections.
12     newL.extend(L1[i1:])
13     newL.extend(L2[i2:])
14     return newL
```

# Merge Sort



# Merge Sort

- ▶ Take list  $L$  and make a list of one-item lists from it.
- ▶ As long as there are two lists left to merge, merge them, and append the new list to the list of lists.

```
1  # Make a list of 1-item lists so that we can start merging.  
2  workspace = []  
3  for i in range(len(L)):  
4      workspace.append([L[i]])
```

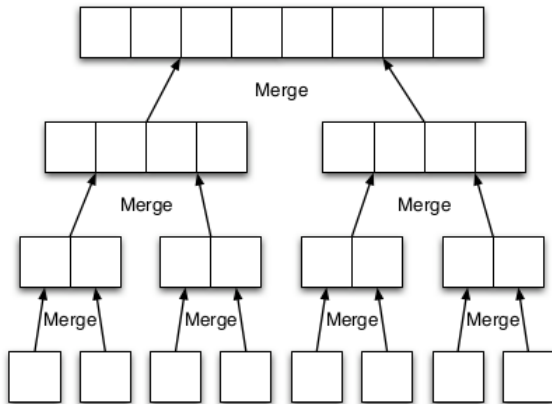
# Merge Sort

- ▶ Take list  $L$ , and make a list of one-item lists from it.
- ▶ Start index  $i$  off at 0.
- ▶ As long as there are two lists, at indices  $i$  and  $i + 1$ , merge them, append the new list to the list of lists, and increment  $i$  by 2.

```
1 def mergesort(L):  
2     workspace = []  
3     for i in range(len(L)):  
4         workspace.append([L[i]])  
5     i = 0  
6     while i < len(workspace) - 1:  
7         L1 = workspace[i]; L2 = workspace[i + 1]  
8         newL = merge(L1, L2)  
9         workspace.append(newL)  
10        i += 2  
11    if len(workspace) != 0:  
12        L[:] = workspace[-1][:]
```

Making new lists, we need to copy the last of the merged lists back into parameter  $L$ .

# Merge Sort



# Merge Sort Analysis

- ▶ The first part of the function, creating the list of one-item lists, takes  $N$  iterations, one for each item.
- ▶ The second part continually merge lists, taking roughly  $N$  steps.
- ▶ Each level has a total of  $N$  items to be merged, each of these  $\log_2 N$  levels takes roughly  $N$  steps.
- ▶ Mergesort needs  $N \log_2 N$ .

# Summary

- ▶ Introduce searching algorithm such as linear and binary.
- ▶ Learn some sorting methods such as binary, insertion, selection, merge sorts.
- ✓ ▶ Compare the time complexity of searching and sorting methods.
- ▶ Look at how the running time of an algorithm grows as a function of the size of its inputs