CompSci 105

Simple Sorting Algorithms

Motivation
Bubble Sort
Selection Sort
Insertion Sort

Textbook: Chapter 5

Sorting: One of the Most Common Activities on Computers

Example 1:

- Alphabetically sorted names, e.g. names in telephone book, street names in map

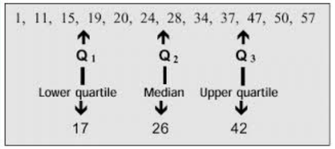
Advantages:

- Can use efficient search algorithms:
- Binary search finds item in O(log n) time
- Interpolation search finds item in O(log log n) time if uniformly distributed

Sorting: One of the Most Common Activities on Computers (cont'd)

Example 2:

- Sorted numbers, e.g. house prices, student IDs, grades, rankings



Advantages:

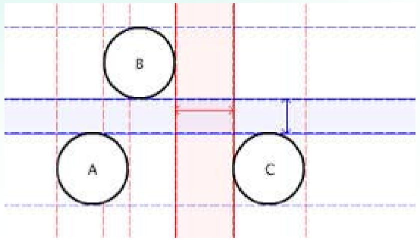
- Can use efficient search algorithms (see example 1)
- Easy to find position or range of values in sorted list, e.g. minimum value, median value, quartile values, all students with A grades, all houses within a certain price range etc.

Sorting: One of the Most Common Activities on Computers (cont'd)

Example 3:

- Sort objects in space

Advantages: Can use efficient search algorithms, e.g. for collision detection



Sorting: Important Properties to Investigate

How efficient is the sorting algorithm?
(Note: can depend on order of input data set, e.g. is it almost sorted or completely unsorted?)

- **How much memory** does sorting algorithm require?

- **How easy** is algorithm to implement?
(for simple problems and small data sets, simple sorting algorithm usually sufficient)

Sorting: Need a comparison operator

Any information which needs to be kept in sorted order will involve the comparison of items (<,=,>), e.g. strings and numbers:

ints/floats

-34 < -1 < 0 < 1 < 245

Characters

A < B < C ... < X < Y < Z

A < ... < Z < a < b < c ... < y < z

Strings

'Hungry' < 'Money' < ‘More’ < 'money' < 'work'

Sorting: Need a comparison operator

Any information which needs to be kept in sorted order will have a **key**, the sort key (e.g., id, name, code number, …). The key determines the position of the individual object in the collection.

Commonly the key is a number.

When comparing keys which are strings, the Unicode (ASCII) values of the string are used (e.g., 'a' is Ox00061, 'A' is Ox00041 and ' ' is Ox00020).

Python sorted() function - 1

Python has an inbuilt sort function: **sorted()**

The sorted() function takes any iterable and returns a list containing the sorted elements. (Note that all sequences are iterable.)

```
a = [5, 2, 3, 1, 4]
b = sorted(a)
print("a -", a)
print("b -", b)
print(b == a)
```

```
a - [5, 2, 3, 1, 4]
b - [1, 2, 3, 4, 5]
False
```

```
a = (5, 2, 3, 1, 4)
b = sorted(a)
print("a -", a)
print("b -", b)
print(b == a)
```

```
a - (5, 2, 3, 1, 4)
b - [1, 2, 3, 4, 5]
False
```

Python sorted() function - 2

```
a = "bewonderful"
b = sorted(a)   # sorted always returns a list
print("a -", a)
print("b -", b)
print(b == a)
```

```
a - bewonderful
b - ['b', 'd', 'e', 'e', 'f', 'l', 'n', 'o', 'r', 'u', 'w']
False
```

```
a = {4:5, 2:9, 1:6, 3:7}
b = sorted(a)     # for dictionary sorted() returns sorted list of keys
print("a -", a)   # print sorts output by keys
print("b -", b)
print(b == a)
```

```
a - {1: 6, 2: 9, 3: 7, 4: 5}
b - [1, 2, 3, 4]
False
```


Python list method, sort()

- As well as the Python inbuild sorted() function, the **sort()** method can be used to sort the elements of a list **in place**.

```
a = [5, 2, 3, 1, 4]
print("a -", a)
a.sort()
print("a -", a)
```

```
a - [5, 2, 3, 1, 4]
a - [1, 2, 3, 4, 5]
```


Python sorted() function, list sort()

We already have the Python sorting functions. Why bother looking at sorting algorithms?

- It gives us a greater understanding of how our programs work.
- Best sorting function depends on application
- Useful for developing sorting algorithms for specific applications

In particular, we are interested in how much processing it takes to sort a collection of items (i.e., the Big O).

Also as Wikipedia says: "*useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.*"

In Python, Timsort is used (for both sorted() and sort()).

Sorting: The Expensive Bits

- In order to sort items we will need to compare items and swap them if they are out of order.

Number of comparisons and the number of swaps are the costly operations in the sorting process and these affect the efficiency of a sorting algorithm (Big O).

Sorting Considerations

An **internal sort** requires that the collection of data fit entirely in the computer's main memory.

An **external sort**: the collection of data will not fit in the computer's main memory all at once but must reside in secondary storage.

For very large collections of data it is costly to create a new structure (list) and fill it with the sorted elements so we will look at sorting **in place**.

Sorting Considerations

One **pass** is defined as one trip through the data structure (or part of the structure) comparing and, if necessary, swapping elements along the way. (In these examples the data structure is a list of ints.)

In these discussions we sort from smallest (on the left of the list) to largest (on the right of the list).

Bubble Sort

IDEA:

Given is a list L of n value {L[0], … , L[n-1]}

Divide list into unsorted (left) and sorted part (right – initially empty): Unsorted: {L[0], … , L[n-1]} Sorted: {}

In each pass compare adjacent elements and swap elements not in correct order => largest element is "bubbled" to the right of the unsorted part

Reduce size of unsorted part by one and increase size of sorted part by one. After i-th pass: Unsorted: {L[0], … , L[n-1-i]}
Sorted: {L[n-i],…,L[n-1]}

Repeat until unsorted part has a size of 1 – then all elements are sorted

Bubble Sort - Example

29	10	14	37	13	List to sort
10	14	29	13	37	PASS 1 (4 Comp, 3 Swap)
10	14	13	29	37	PASS 2 (3 Comp, 1 Swap)
10	13	14	29	37	PASS 3 (2 Comp, 1 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 0 Swap)

Bubble Sort - Exercise

54 26 93 17 77 31 44 55 20 — List to sort

PASS 1

PASS 2

PASS 3

PASS 4

PASS 5

PASS 6

PASS 7

PASS 8

Some Useful Python Features

```
def print_section(a_list, i, j):
    print(i, j, a_list[i:j])

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
for x in range(0,len(a_list),3):
    print(a_list[x],end=" ") # Output: 54 17 44
print(a_list)               # Output: [54, 26, 93, 17, 77, 31, 44, 55, 20]
i,j = 2,5
print_section(a_list,i,j)    # Output: 2 5 [93, 17, 77]
print_section(a_list,0,9)  # Output: 0 9 [54, 26, 93, 17, 77, 31, 44, 55, 20]
```

list[i:j] // gives the subsection of a list from index i to index j-1

range(i,j,d) // creates range of values from i to j with step size d

i,j=2,5 // parallel assignment of multiple values

Swapping elements

```
def swap1(a_list, i, j):
    temp = a_list[i]
    a_list[i] = a_list[j]
    a_list[j] = temp

def swap2(a_list, i, j):
    a_list[i], a_list[j] = a_list[j], a_list[i]

a_list = [54, 26, 93, 17, 77]
print("before: ", a_list)          # Output: [54, 26, 93, 17, 77]
swap1(a_list, 0, 4)
print("after: ", a_list)           # Output: [77, 26, 93, 17, 54]
swap2(a_list, 1, 2)
print("after: ", a_list)           # Output: [77, 93, 26, 17, 54]
```


Bubble Sort Code

```
def my_bubble_sort(a_list):
    for pass_num in range(len(a_list)-1, 0, -1):
        for i in range(0, pass_num):
            if a_list[i] > a_list[i+1]:
                a_list[i], a_list[i+1] = a_list[i+1], a_list[i]
        #print(pass_num, "-", a_list) # enable to see each pass

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

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

Bubble Sort – Big O

•For a list with *n* elements:

The number of comparisons?

pass 1	pass 2	pass 3	… last pass
n-1	n-2	n-3	… 1

$1 + 2 + \ldots + (n\text{-}3) + (n\text{-}2) + (n\text{-}1) = \frac{1}{2}(n^2 - n)$

Big O is n^2 – $O(n^2)$

10 times bigger means it takes a 100 times longer)

On average, the number of swaps is half the number of comparisons.

Bubble Sort – Big O

• What if the data is already sorted?

5	10	14	32	35	List to sort
5	10	14	32	35	PASS 1
5	10	14	32	35	PASS 2
5	10	14	32	35	PASS 3
5	10	14	32	35	PASS 4

Swaps?

Comparisons?

Bubble Sort – Big O

• What if the data is in reverse order?

35	32	14	10	5	List to sort
32	14	10	5	35	PASS 1
14	10	5	32	35	PASS 2
10	5	14	32	35	PASS 3
5	10	14	32	35	PASS 4

Swaps?

Comparisons?

Bubble Sort – Summary

Simple to understand.

Lots of comparisons (O(*n*^2)) and lots of swaps each pass (O(*n*^2) on average).

We can improve bubble sort. How?

Note what happens with bubble sort if it contains elements in reverse order, e.g. [3,2,1] -> [2,3,1] -> [2,1,3] -> [1,2,3]

Can we reduce the number of swaps (assignments of values)?

Selection Sort

IDEA:

Given is a list L of n value {L[0], … , L[n-1]}

Divide list into unsorted (left) and sorted part (right – initially empty): Unsorted: {L[0], … , L[n-1]} Sorted: {}

In each pass find largest value and place it to the right of the unsorted part using **a single swap**

Reduce size of unsorted part by one and increase size of sorted part by one. After i-th pass: Unsorted: {L[0], … , L[n-1-i]}
Sorted: {L[n-i],…,L[n-1]}

Repeat until unsorted part has a size of 1 – then all elements are sorted

Selection Sort - Example

29	10	14	37	13	List to sort
29	10	14	13	37	PASS 1 (4 Comp, 1 Swap)
13	10	14	29	37	PASS 2 (3 Comp, 1 Swap)
13	10	14	29	37	PASS 3 (2 Comp, 0 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 1 Swap)

Selection Sort - Exercise

54 26 93 17 77 31 44 55 20 List to sort

PASS 1

PASS 2

PASS 3

PASS 4

PASS 5

PASS 6

PASS 7

PASS 8

Selection Sort - Exercise

11 34 26 90 37 58 10 47 36 List to sort

PASS 1

PASS 2

PASS 3

PASS 4

PASS 5

PASS 6

PASS 7

PASS 8

Selection Sort – swap elements

```
def swap_elements(a_list, i, j):
    a_list[i], a_list[j] = a_list[j], a_list[i]
```

11	34	26	90	37	58	10	47	36
			↑					↑

Each pass we need to swap two elements of the list. For example, at the end of the first pass we want to swap the element at position 3 with the element at position 8.

After the first pass:

11	34	26	36	37	58	10	47	**90**
			↑					↑

Selection Sort Code

```
def swap_elements(a_list, i, j):
    a_list[i], a_list[j] = a_list[j], a_list[i]

def my_selection_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        position_largest = 0
        for i in range(1, pass_num+1):
            if a_list[i] > a_list[position_largest]:
                position_largest = i
        swap_elements(a_list, position_largest, pass_num)
        #print(pass_num, "-", a_list) # enable to see each pass

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

NOTE: No check whether swap necessary

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


Selection Sort – Big O

For a list with *n* elements

The number of comparisons?

pass 1	pass 2	pass 3	...	last pass
n-1	n-2	n-3	...	1

$$1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = \tfrac{1}{2}(n^2 - n)$$

Big O is n^2 – $O(n^2)$

Note: one swap each pass (NOTE: implementation swaps elements even if indices are the same, i.e. no swap necessary)

Selection Sort – Big O

- What if the data is already sorted?

5	10	14	32	35	List to sort
5	10	14	32	**35**	PASS 1
5	10	14	**32**	**35**	PASS 2
5	10	**14**	**32**	**35**	PASS 3
5	**10**	**14**	**32**	**35**	PASS 4

Swaps?

Comparisons?

Selection Sort – Big O

- What if the data is in reverse order?

35	32	14	10	5	List to sort
5	32	14	10	35	PASS 1
5	10	14	32	35	PASS 2
5	10	14	32	35	PASS 3
5	10	14	32	35	PASS 4

Swaps?

Comparisons?

Selection Sort - Summary

Simple to understand – divide array into unsorted (left) and sorted part (right, initially empty

Find largest value in unsorted part and place at end – after each pass sorted part increases by one and unsorted part reduces by one.

Unsorted part				**sorted part**				
11	34	26	10	36	37	47	58	90

Lots of comparisons O(n^2), one swap per pass (O(n))

Comparison Bubble Sort vs. Selection Sort

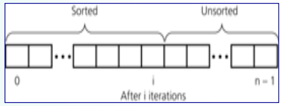
Bubble and Selection sort use the same number of comparisons

Bubble sort does O(n) swaps per pass on average, but Selection sort only 1 swap per pass (i.e. O(n^2) vs. O(n) in total)

Selection sort typically executes faster than bubble sort.

How can we do better? IDEA: Reduce number of comparisons by inserting into sorted array

Insertion Sort



IDEA:

Given is a list L of n value {L[0], … , L[n-1]}

Divide list into sorted (left – initially only one element) and sorted part (right): Sorted: {L[0]} Unsorted: {L[1], … , L[n-1]}

In each pass take left most element from unsorted part and place it into correct position of sorted part

Reduce size of unsorted part by one and increase size of sorted part by one. After i-th pass: Sorted: {L[0],…,L[i]}
Unsorted: {L[i+1], … , L[n-1-i]}

Repeat until unsorted part is an empty list – then all elements are sorted

Insertion Sort - Example

29	10	14	13	18	List to sort
10	29	14	13	18	PASS 1 (1 Comp, 1 Shift)
10	14	29	13	18	PASS 2 (2 Comp, 1 Shift)
10	13	14	29	18	PASS 3 (3 Comp, 2 Shift)
10	13	14	18	29	PASS 4 (2 Comp, 1 Shift)

37

Insertion Sort - Exercise

54 26 93 17 77 31 44 55 20	List to sort

PASS 1

PASS 2

PASS 3

PASS 4

PASS 5

PASS 6

PASS 7

PASS 8

38

Insertion Sort - Exercise

35	34	26	90	37	28	10	27	36	List to sort

PASS 1

PASS 2

PASS 3

PASS 4

PASS 5

PASS 6

PASS 7

PASS 8

39

Insertion Sort – making room for the element to be inserted

6	28	34	35	37	90	10	27	36

For example, to insert 10 into the sorted part of the list we need to store 10 into a temporary variable and move all the elements which are bigger than 10 up one position, then insert 10 into the empty slot.

temp=10

6	28	34	35	37	90	_	27	36

Shift 5 list values

6	_	28	34	35	37	90	27	36

6	10	28	34	35	37	90	27	36

40

Insertion Sort Code

```
def my_insertion_sort(a_list):
  for index_number in range(1, len(a_list)):
      item_to_insert = a_list[index_number]
      index = index_number - 1
      while index >= 0 and a_list[index] > item_to_insert:
          a_list[index + 1] = a_list[index]
          index -= 1
      a_list[index + 1] = item_to_insert
      #print(index_number, "-", a_list) # enable to see each pass

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

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


Insertion Sort – Big O

For a list with n elements

The number of comparisons in the **WORST CASE**?

pass 1	pass 2	pass 3	...			last pass
1	2	3	...	n-3	n-2	n-1

$$1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = \tfrac{1}{2}(n^2 - n)$$

In the average case about half of that: Big O is n^2 – **$O(n^2)$**

NOTE 1: Best case **$O(n)$** **… when does this occur?** ☺

Note 2: The number of shifts is equal or one smaller than the number of comparisons, so same order of magnitude.

Insertion Sort – Big O

- What if the data is already sorted?

5	10	14	32	35	List to sort
5	10	14	32	35	PASS 1
5	10	14	32	35	PASS 2
5	10	14	32	35	PASS 3
5	10	14	32	35	PASS 4

Move elements?

Comparisons?

Insertion Sort – Big O

- What if the data is in reverse order?

35	32	14	10	5	List to sort
32	35	14	10	5	PASS 1
14	32	35	10	5	PASS 2
10	14	32	35	5	PASS 3
5	10	14	32	35	PASS 4

Move elements?

Comparisons?

Insertion Sort – Summary

Insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less.

Insertion sort is almost 40% faster than Selection sort
– on average, it does half as many comparisons
but it does more moves.

For small lists, Insertion sort is appropriate due to its simplicity.

For almost sorted lists Insertion Sort is a

For large lists, all O(*n*^2) algorithms, including Insertion Sort, are prohibitively inefficient.

45

Simple Sorting Algorithms – Summary

All sorting algorithms discussed so far had an O(*n*^2) average and worst case complexity
=> In practice for large lists usually to slow

The Timsort algorithm (written in C – not using the Python interpreter) used by Python combines elements from MergeSort and Insertion Sort
- Worst case and average case complexity O(*n* log *n*)
- Very fast for almost sorted lists

NOTE 1: All comparison based sorting algorithms require at least O(*n* log *n*) time in the worst and average case

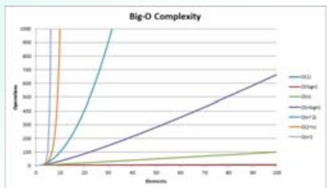
NOTE 2: In applications where writing data is expensive Selection sort may be better.

46

Running Time Matters ☺

The usefulness of an algorithm in practice depends on the data size *n* and the complexity (Big O) of the algorithm (time and memory).

In general algorithms with linear, logarithmic or low polynomial running time are acceptable
- **O (log *n*)**
- **O (*n*)**
- **O (n^K) where K is a small constant, (in many cases K <= 2 is ok)**



Algorithms with exponential or high polynomial running time are often of limited use.
- **O (n^K) where K is a large constant, say >3**
- **O (2^n), O (n^n)**

47