MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2004
Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

Referencing materials by
Nathan Companez, Aamir Cheema, Arun Konagurthu and Lloyd Allison

GROUP OF EIGHT AUSTRALIA

Faculty of Information Technology, Monash University

Ready?

Agenda

- Sorting Algorithms
  - Comparison based
    - Selection
    - Insertion
  - Non-comparison based (the IMBA ones)
    - Counting
    - Radix

MONASH University

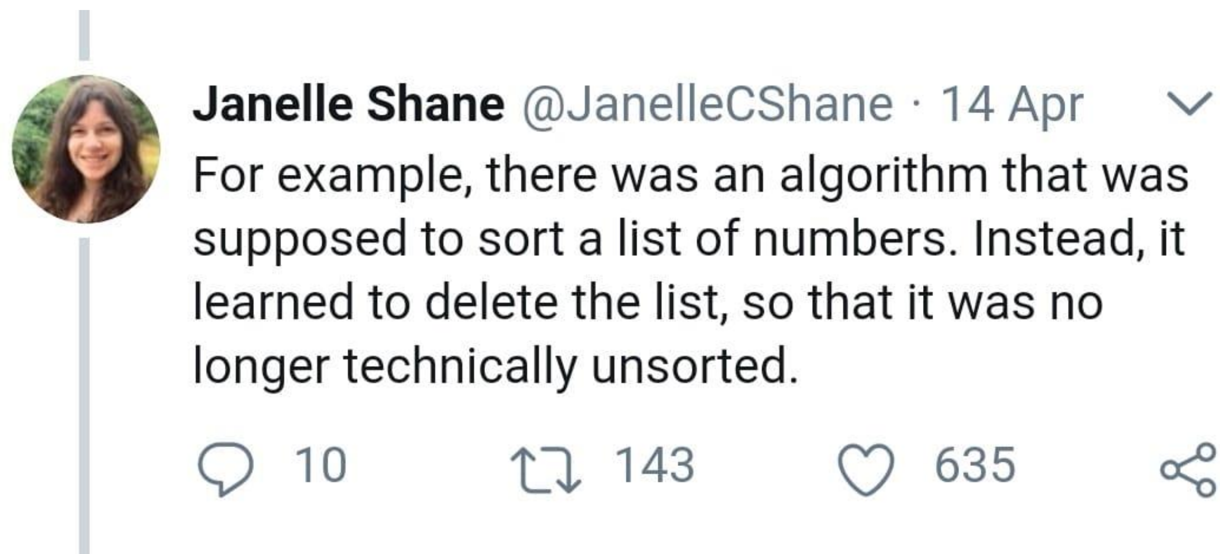
Let us begin…

Sorting

- We are back to sorting!
  - Bubble
  - Insertion
  - Selection
  - Merge
  - Quick

Sorting

- We are back to sorting!
  - Bubble
  - Insertion
  - Selection
  - Merge
  - Quick



Sorting

- We are back to sorting!
  - Bubble
  - Insertion
  - Selection
  - Merge
  - Quick

- All of these are known as comparison based sorting. Why?

Sorting

- We are back to sorting!
  - Bubble
  - Insertion
  - Selection
  - Merge
  - Quick

- All of these are known as comparison based sorting. Why? Because we compare between items to know if $a < b$ or $b > a$

Sorting

- We are back to sorting!
  - Bubble
  - Insertion
  - Selection
  - Merge
  - Quick

- All of these are known as comparison based sorting. Why? Because we compare between items to know if $a < b$ or $b > a$
- Now let us analyze them based on what we have learnt!

Questions?

- Correctness
- Complexity

- Correctness
  - Loop invariant
  - Termination
- Complexity
  - Time
  - Space

Sorting

Selection Sort

- Correctness
  - Loop invariant something don't to make the list is sorted
  - Termination
- Complexity
  - Time
  - Space

loop through list to find the minimum and swap to first position, from then on, find the remaining minimum and swap to position after minimum value position

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

Sorting
Selection Sort

- Correctness
  - Loop invariant
  - Termination

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
  - Termination

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
    i - 1, end of array = length - 1
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N] elements in list[i..N] >= list[0..i-1]
  - Termination
    for j loop

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N]
  - Termination

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N]
  - Termination
    - i and j always increment and both reach the end of the list

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N]
  - Termination
    - i and j always increment and both reach the end of the list
  - So why is it working then?

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
  - Loop invariant
    - my_list[0…i-1] is sorted
    - my_list[0…i-1] <= my_list[i…N]
  - Termination
    - i and j always increment and both reach the end of the list
  - So why is it working then?
    - i keep increment till n and we know from invariant 0…i-1 is sorted, thus we will sort the entire list!

```python
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

Sorting
Selection Sort

- Correctness
- Complexity
  - Time
  - Space

```python
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
- Complexity
  - Time
  - Space

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

swaping in-place (in input list)

Sorting

Selection Sort

- Correctness
- Complexity
  - Time
    - Best = O(N^2)
    - Worst = O(N^2)
  - Space

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]: Comparison O(n)
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

Sorting
Selection Sort

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
  - Space

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
  - Space

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

MONASH University

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
  - Space
    - O(N) for the input list
    - Auxiliary?

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
  - Space
    - O(N) for the input list
    - Auxiliary? O(1)

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

swaping in-place

since just swap in input list and move on

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
  - Space
    - O(N) for the input list
    - Auxiliary? O(1) in place

constant space of array
since swap in array, in place

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

- Correctness
- Complexity
  - Time

    i loop through whole list to be pilot to compare in j loop with rest of the list
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
    - But what if I tell you comparing the items have a cost of O(k)
      - Like comparing between words, you need to compare the alphabets

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

MONASH University

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
    - But what if I tell you comparing the items have a cost of O(k)
      - Like comparing between words, you need to compare the alphabets
      - We know complexity is based on comparison O(N^2) comparisons…

```
def selection_sort(my_list):
    for i in range(len(my_list)):
        minimum = i
        # find the minimum
        for j in range(i+1, len(my_list)):
            if my_list[minimum] > my_list[j]:
                minimum = j
        # swap
        my_list[i],my_list[minimum] = my_list[minimum],my_list[i]
```

swaping in-place (in input list)

loop through list to find the minimum and swap to first position, from then on, find the remaining minimum and swap to position after minimum value position

MONASH University

- Correctness
- Complexity
  - Time
    - Best = O(N^2) because no matter what we have to find the minimum and cant terminate earlier!
    - Worst = O(N^2)
    - But what if I tell you comparing the items have a cost of O(k)
      - Like comparing between words, you need to compare the alphabets
      - We know complexity is based on comparison O(N^2) comparisons…
      - So our final complexity is O(kN^2)

- Correctness
- Complexity
- Stable?

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable?

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable? No! but why?

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable? No! but why?
  - [4a, 2, 3, 4b, 1]

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable? No! but why?
  - [4a, 2, 3, 4b, 1]
  - Minimum is 1, so we swap

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable? No! but why?
  - [4a, 2, 3, 4b, 1] 4a used to be in front of 4a, now 4a after 4b
  - Minimum is 1, so we swap
  - [1, 2, 3, 4b, 4a]

- Correctness
- Complexity
- Stable?
  - Relative ordering doesn't change
  - Is it stable? No! but why?
  - [4a, 2, 3, 4b, 1] during swaping, relative order does not maintain
  - Minimum is 1, so we swap
  - [1, 2, 3, 4b, 4a]
  - Now we see that 4a is behind 4b!

Questions?

- Correctness
- Complexity

- Correctness
- Complexity

Problem 1. Write psuedocode for insertion sort, except instead of sorting the elements into non-decreasing order, sort them into non-increasing order. Identify a useful invariant of this algorithm.

Sorting

Insertion Sort

- Correctness
- Complexity

every time new i comes
would compare all the value
of the sorted list infront

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

my_list[1] = my_list[0]

original value of my_list[0] = key

Sorting

Insertion Sort

- Correctness
  - Loop invariant
  - Termination
- Complexity

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

- Correctness
  - Loop invariant
  - Termination
    - Simple, I skip this
- Complexity

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Insertion Sort

- Correctness
  - Loop invariant
    - my_list[0…i-1] sorted
  - Termination
    - Simple, I skip this
- Complexity

```python
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

MONASH University

- Correctness
- Complexity
  - Best
  - Worst

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

MONASH University

- Correctness
- Complexity
  - Best O(N) comparison
    - Each loop only look and compare with left item once
  - Worst

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

no this step

if the list is in order

- Correctness
- Complexity
  - Best O(N) comparison
    - Each loop only look and compare with left item once
  - Worst O(N^2)
    - Each loop keep look left, compare and swap till beginning of list

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

MONASH University

- Correctness
- Complexity
  - Best O(N) comparison
    - Each loop only look and compare with left item once
  - Worst O(N^2)
    - Each loop keep look left, compare and swap till beginning of list
  - So if O(k) is the comparison cost, when we have O(kN^2) worst case!

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

- Correctness
- Complexity
  - Best O(N) comparison
    - Each loop only look and compare with left item once
  - Worst O(N^2)
    - Each loop keep look left, compare and swap till beginning of list
  - So if O(k) is the comparison cost, when we have O(kN^2) worst case!
  - What about space?

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

- Correctness
- Complexity
  - Best O(N) comparison
    - Each loop only look and compare with left item once
  - Worst O(N^2)
    - Each loop keep look left, compare and swap till beginning of list
  - So if O(k) is the comparison cost, when we have O(kN^2) worst case!
  - What about space?
    - O(N) for the input list — no newly created temporary list
    - O(1) auxiliary cause it is in-place

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

compare key with my_list[j] backward
until a my_list[j] <= key, then insert key my_list[j+1]
which maintain array is sorted for array[:i]

Sorting
Insertion Sort

- Correctness
- Complexity
- Stability

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Sorting

Insertion Sort

- Correctness
- Complexity
- Stability
  - Yes

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Sorting

Insertion Sort

- Correctness
- Complexity
- Stability
  - Yes
  - Don’t swap if value is the same

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Sorting

Insertion Sort

- Correctness
- Complexity
- Stability
  - Yes
  - Don't swap if value is the same
  - Most shifting will ensure stability

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        key = my_list[i]
        j = i - 1
        # keep shifting to left if left is greater
        while j >= 0 and key < my_list[j]:
            my_list[j+1] = my_list[j]
            j = j - 1
        my_list[j+1] = key
```

Questions?

Summary

Sorting

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$ – can be made $O(N \log N)$	$O(N \log N)$	Depends	No

Summary

Sorting

	Best	Worst	Average		
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$		
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$		
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$		
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$ – can be made $O(N \log N)$	$O(N \log N)$	Depends	No

- The recursion stack takes up memory!!!

Summary
Auxiliary for Recursion

- The recursion stack takes up memory!!!
  - So that is why it isn't in place!

Summary

Auxiliary for Recursion

- The recursion stack takes up memory!!!
  - So that is why it isn’t in place!
  - If I have recursion log N times, then I take O(log N) space for the recursion alone!
  - If each recursion is k, then my total space is O(k log N)!!!

Summary
Auxiliary for Recursion

- The recursion stack takes up memory!!!
  – So that is why it isn't in-place!
    - Iterative is easier to get in-place
  – If I have recursion log N times, then I take O(log N) space for the recursion alone!
  – If each recursion is k, then my total space is O(k log N)!!!

Summary
Sorting

	Best	Worst	Average	Stable?	In-place?
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	No	Yes
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	No	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	Yes	No
Quick Sort	$O(N \log N)$	$O(N^2)$ – can be made $O(N \log N)$	$O(N \log N)$	Depends	No

MONASH University

- So… what is the lower bound for the sorting algorithms that we have learnt?
  - Bubble
  - Insertion
  - Selection
  - Quick
  - Merge

- These are all comparison based
- $\Omega(N \log N)$
- We will see more of this later

Questions?

MONASH University

Thank you