Computing Science 1P

COMPSCI 1001

Tutorial 3

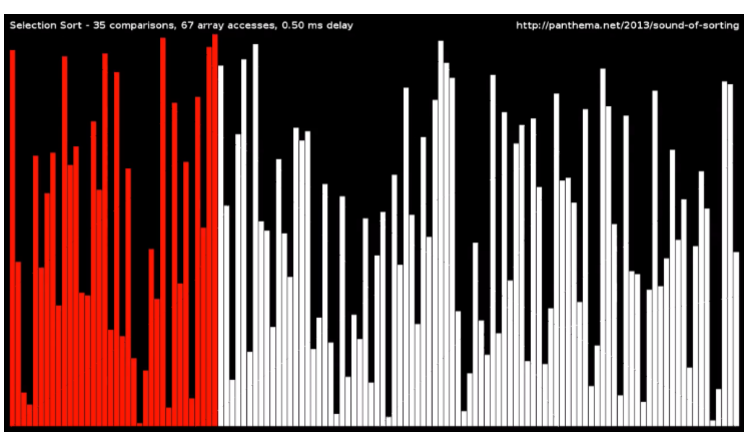
February 5th, 2020

Lecturer: Dr Mohamed Khamis
Mohamed.Khamis@Glasgow.ac.uk
https://www.gla.ac.uk/schools/computing/staff/mohamedkhamis/
http://mkhamis.com/

University of Glasgow

1

15 Sorting Algorithms in 6 Minutes



Warning: Contains bright rapidly flashing colours

https://www.youtube.com/watch?v=kPRA0W1kECg

2

Questions from slido

- Python uses TimSort
  - We'll discuss it in next week's tutorial, but if you are curious check this video: https://www.youtube.com/watch?v=_dlzWEJoU7I
- When choosing a sorting algorithm, should we prioritize time complexity or space complexity?
  - Time complexity is usually more important / more expensive
    - Space can be reused
    - If you solve a problem faster, you can reuse the space faster

3

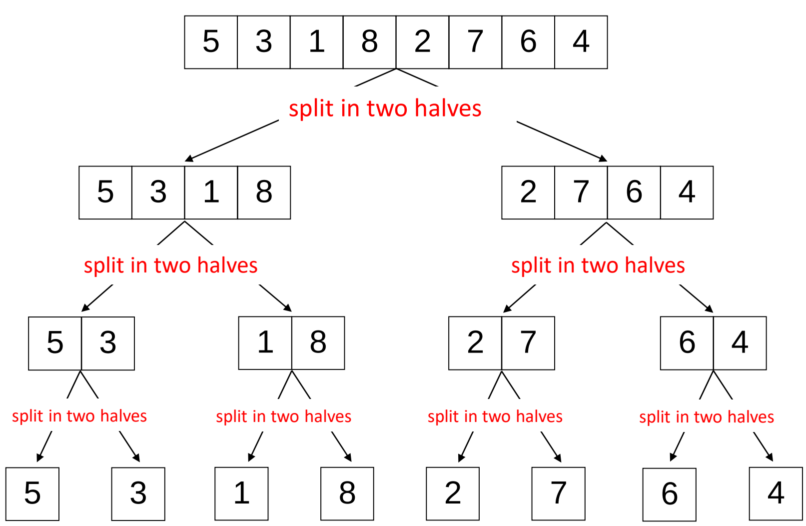
Questions from slido

- Would the best case scenario for merge sort time complexity not be when there is a list of length one? That is the base case of the recursion
  - No (details below)
- Why is best case of merge sort not len(x)=1, hence O(1)?
  - A list of length one would always be O(1) in all the sorts we discussed so far. E.g., In merge sort, n = 1 → n log n = 0.
  - But when analysing the time complexity of an algorithm, the part of your code that grows the fastest with respect to input is the most relevant.
  - Best case scenario refers (by definition) to the best case scenario of an input of size n.
- Would you take a recursive approach to the quicksort?
  - Yes, in fact, that's one of your tasks for next week's lab ☺
- Can I get into the boyd orr at the weekends ?
  - AFAIK, it is not possible for Level 1-2 students.

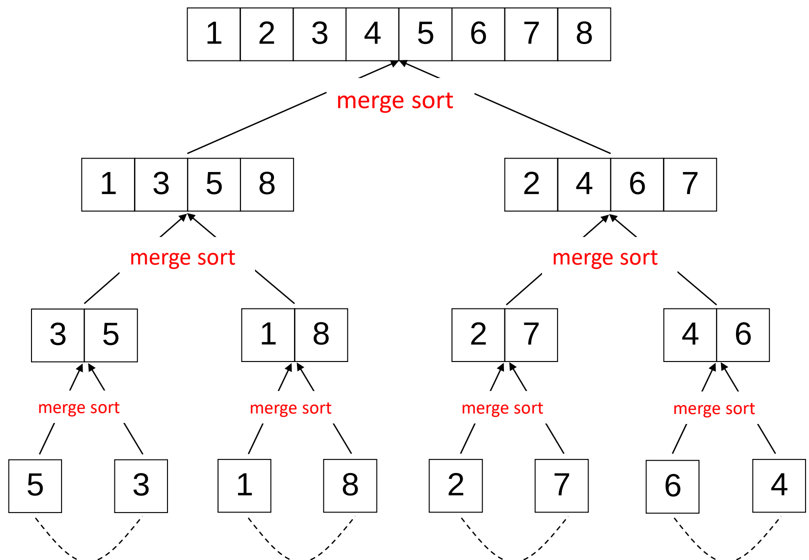
4

RECAP ON MERGESORT AND QUICKSORT

Merge sort



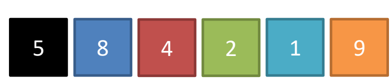
Merge sort



7

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  – Pick an element: we'll call it the pivot
  – Move all elements smaller than the pivot to its left
  – Move all elements larger than the pivot to its right



8

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

1. Swap your pivot with the element at the end of the list

9

Quick sort

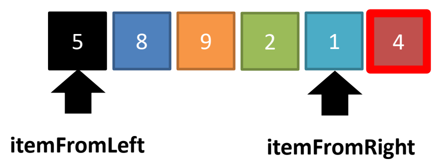
- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

2. Look for 2 items:
   - **itemFromLeft** that is larger than the pivot
   - **itemFromRight** that is smaller than the pivot

10

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right



2. Look for 2 items:
   - **itemFromLeft** that is larger than the pivot
   - **itemFromRight** that is smaller than the pivot
3. Swap them!

11

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

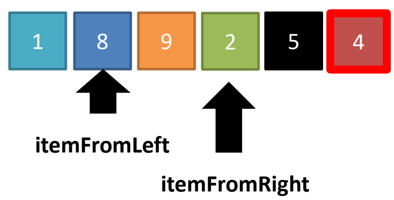
2. Look for 2 items:
   - **itemFromLeft** that is larger than the pivot
   - **itemFromRight** that is smaller than the pivot
3. Swap them!

12

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

2. Look for 2 items:
- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot



It is a Swap!

13

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

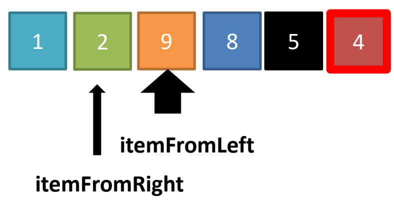
2. Look for 2 items:
- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot

14

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

2. Look for 2 items:
- **itemFromLeft** that is larger than the pivot
- **itemFromRight** that is smaller than the pivot

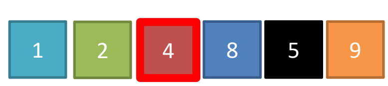


4. When index of **itemFromLeft** > index of **itemFromRight**, then we swap pivot with **itemFromLeft**.

15

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

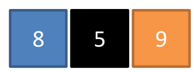


Now the **pivot** is in the correct position!

16

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  – Pick an element: we'll call it the pivot
  – Move all elements smaller than the pivot to its left
  – Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

17

Quick sort

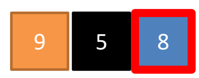
- Another popular algorithm that is also order of n(log n)
- Core idea:
  – Pick an element: we'll call it the pivot
  – Move all elements smaller than the pivot to its left
  – Move all elements larger than the pivot to its right

QuickSort is a recursive function – let's apply it on the right part of the list.

18

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right



QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  - Pick an element: we'll call it the pivot
  - Move all elements smaller than the pivot to its left
  - Move all elements larger than the pivot to its right

QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

- Another popular algorithm that is also order of n(log n)
- Core idea:
  – Pick an element: we'll call it the pivot
  – Move all elements smaller than the pivot to its left
  – Move all elements larger than the pivot to its right

How would you implement QuickSort?

QuickSort is a recursive function – let's apply it on the right part of the list.

Quick sort

```
def quickSort(mylist):
    less = []
    more = []
    if len(mylist) <= 1:
        return mylist
    else:
        pivot = mylist[0]
        for i in mylist:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + [pivot] + more

a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
a = quickSort(a)
```

There is an issue in this implementation. Can you spot it?

Quick sort

```
def quickSort(mylist):
    less = []
    pivotList = [] # a list of elements equal to the pivot
    more = []
    if len(mylist) <= 1:
        return mylist
    else:
        pivot = mylist[0]
        for i in mylist:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more

a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
a = quickSort(a)
```

In next week's lab: implement QuickSort using list comprehension.

SEARCHING

Searching in an unstructured list

```
def find(key, mylist,default):
  for i in range(mylist):
    if mylist[i] == key:
      return i
  return default
```

What do you think is the complexity of this algorithm:

A. $O(\log_2 n)$
B. $O(n\log_2 n)$
C. $O(n)$
D. $O(n^2)$

Searching in an unstructured list

- What can we say about the time taken by **find** ?
  - Like sorting, the relevant measure is the number of comparisons
- It is possible that the key is at the end of the list…
  - So we have to compare the given key with every key in the list
- Imagine testing **find** with a large number of random lists
  - On average it will have to search half way along the list
- When analysing algorithms, we talk about the average case, best case and the worst case
  - Best case: O(1)
  - Worst case: O(n)
  - Average case: O(n)

Searching in an unstructured list

- We can't do better than order n for searching in an unstructured list… why?
  - It is possible that the desired key is at the end
- An algorithm for quantum computers takes square root of n operations to search in an unstructured list
  - To find out more, look up Grover's algorithm
    - https://en.wikipedia.org/wiki/Grover%27s_algorithm
- But let's stick to conventional algorithms…

More efficient search

- **Simple idea**: use an ordered list instead
  - Put the data in the list in such a way that the keys are in order
  - Often this means alphabetical order, numerical order, etc
- **Example**: in a dictionary, words are in alphabetical order
  - We can take advantage of this to find words quickly
  - For simplicity, we assume there are no duplicates (dictionary keys are all unique anyway)

Think about it for Friday!

Summary

- Merge sort
  - A divide and conquer algorithm
  - Splits the list into two halves
  - Sorts each half using MergeSort (recursive)
  - Merges the two resulting halves.
  - Time-Complexity:
    - Worst case: O(n log n)
    - Best case: O(n log n)
  - Space complexity: O(n)
- Quick sort
  - Another divide and conquer algorithm
  - Very fast (e.g., compared to merge sort)
  - Selects a pivot, puts all smaller elements on its left, and all larger elements on its right.
  - Applies quick sort on both sublists on the left and the right of the pivot.
  - Time Complexity
    - Worst case: $O(n^2)$
    - Best case: O(n log n)
  - Space complexity:
    - The described previous implementation: O(n)
    - Best case: O(log n)

Ask questions on sli.do #CS1P