Computing Science 1P

COMPSCI 1001

Lecture 4: Searching

February 7th, 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

Searching in unstructured list

```
def find(key,data,default):
  for i in range(len(data)):
    if data[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. $O(n^2)$

2

Searching in 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, sometimes we talk about the average case and sometimes the worst case
  - In this situation they are both the same: order n (n length of the list)

Searching in 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
  - But quantum computers of useful size have not yet been built
  - 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

- The only alternative is to change the data structure…
  - Don't use an unstructured list!

- **<u>Simple idea</u>**: 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

- **<u>Example</u>**: 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)

5

Binary search

- Search for the key: **`cat`**

- It could be anywhere in the list

- The list has length 12
- Divide it by 2 and look at position 6

`cat < garage`

`android`	0
`badger`	1
`cat`	2
`door`	3
`ending`	4
`fireman`	5
`garage`	6
`handle`	7
`iguana`	8
`jumper`	9
`kestrel`	10
`lemon`	11

6

Binary search

- Search for the key: **cat**
- Because the list is ordered, we now know that **cat** must be before **garage**, i.e. it is in the first half of the list
- Now repeat, searching in a list of length 6

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**
- Because the list is ordered, we now know that **cat** must be before **garage**, i.e. it is in the first half of the list
- Now repeat, searching in a list of length 6
- Divide by 2 and look at position 3 **cat < door**

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**
- Divide by 2 and look at position 3 **cat < door**
- We now know that **cat** must be before **door**

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**
- Divide by 2 and look at position 3 **cat < door**
- We now know that **cat** must be before **door**
- Now repeat, searching in a list of length 3

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**
- Divide by 2 and look at position 3 **cat < door**
- We now know that **cat** must be before **door**
- Now repeat, searching in a list of length 3

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**
- Now repeat, searching in a list of length 3
- Divide by 2 and look at position 1
- **cat > badger**
- We now know that **cat** must be after **badger**

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

Binary search

- Search for the key: **cat**

- We now know that **cat** must be after **badger**

- We have narrowed down the possible position of **cat** to just one place and in fact **cat** is there, so we have found it

- If a different word is there, then **cat** is not in the list

android	0
badger	1
cat	2
door	3
ending	4
fireman	5
garage	6
handle	7
iguana	8
jumper	9
kestrel	10
lemon	11

13

Binary search

- Idea is simple, but implementing it correctly requires care
  - Many possibilities for "off by one" errors
    - How we include/exclude boundaries when halving the list?
    - What is the midpoint (odd/even-numbered lists)?
    - What happens when we have a hit?

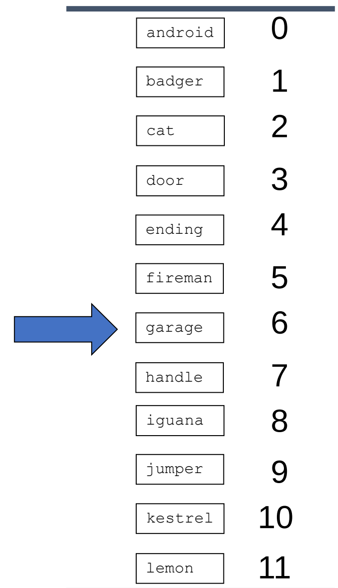
- Searching in dictionary is used as example of binary search
  - But we don't really use dictionaries in exactly this way

- Usually we flick through the pages quickly to find the right letter, then do something similar to binary search
  - A typical dictionary has extra structure to support this process (e.g. words in the page headers, thumbholes for indexing, etc)

14

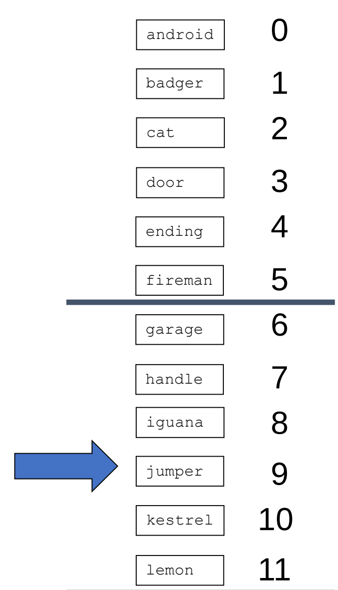
Another example

- Search for the key: **handle**



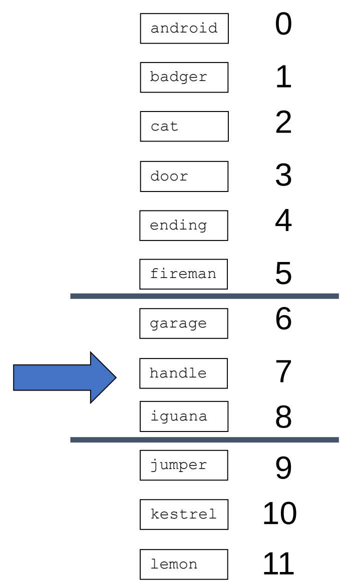
Another example

- Search for the key: **handle**



Another example

- Search for the key: **`handle`**



Analysing binary search

- Remember that we are interested in the number of comparisons
- Suppose that we are searching in a list of length n
- We compare the middle item with the search key
  - The result might tell us we have found the key, but in general it narrows down the region of the list in which we are searching
- The possible region of the list is now half the size it was

Analysing binary search

- We keep halving the size of the region, until we narrow it down to a single position in which the key should be found

- How many times do we have to halve the size?

  n = 16: 8, 4, 2, 1 — 4 comparisons
  n = 64: 32, 16, 8, 4, 2, 1 — 6 comparisons

- It is the logarithm of n to base 2 (power of 2 which gives n)

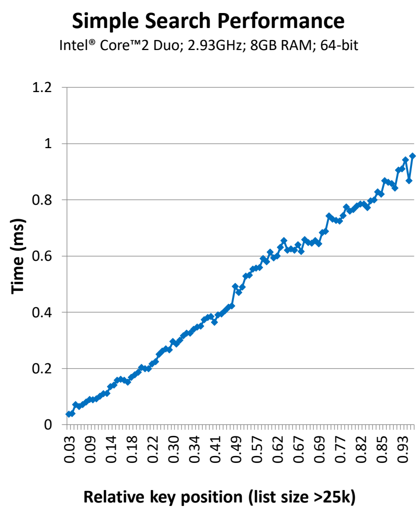
- Binary search is an order log n algorithm…

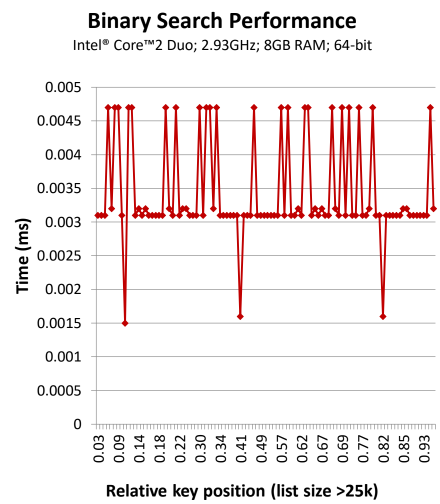
Analysing binary search

- We can compare the efficiency of an order n algorithm (simple search) with that of an order log n algorithm:

n	log n	time	n	time
10	3		10	
100	6		100	
1 000	9	9 microsec	1 000	1 millisec
10 000	12	12 microsec	10 000	10 millisec
100 000	15	15 microsec	100 000	100 millisec
1 000 000	18	18 microsec	1 000 000	1 sec
10 000 000	21	21 microsec	10 000 000	10 sec

Performance comparison





Implementing binary search

```
def find(key,data,default):
    lower = 0
    upper = len(data)-1
    length = upper - lower + 1
    while length > 1:
        midpoint = lower + length//2 # Floor division
        if key < data[midpoint]:
            upper = midpoint - 1  # look at lower half
        else:
            lower = midpoint    # look at upper half
        length = upper - lower + 1
    if key == data[lower]:
        return lower
    else:
        return default   # the error value we pass in
```

Refining binary search

- It might turn out that when we look at the midpoint of the list, the key we want happens to be there
  - We might as well take advantage of that case…

```
while length > 1:
    midpoint = lower + length/2
    if key < data[midpoint]:
        upper = midpoint - 1
    elif key > data[midpoint]:
        lower = midpoint
    else:
        return midpoint
```


25

Summary

- **Search algorithms**
  - Hard for unstructured list: need to search all items
  - Can sort the data (e.g. using merge sort)
  - Then we can use binary search: much more efficient
- **Binary search**
  - Look at data entry half way through the list
  - Compare with key and then narrow search to top/bottom half
  - Repeat until only one item is in the buffer
  - $\text{Log}_2 n$ complexity

27