Searching in a List

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 2

Search problem

- Is value `v` present in list `l`?

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Search problem

- Is value `v` present in list `l`?
- Naive solution scans the list
- Input size n, the length of the list
- Worst case is when `v` is not present in `l`
- Worst case complexity is $O(n)$

```
def naivesearch(v,l):
  for x in l:
    if v == x:
      return(True)
  return(False)
```

Searching a sorted list

- What if `l` is sorted in ascending order?

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
  - If midpoint is `v`, the value is found
  - If `v` less than midpoint, search the first half
  - If `v` greater than midpoint, search the second half
  - Stop when the interval to search becomes empty

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Searching a sorted list

- What if `l` is sorted in ascending order?
- Compare `v` with the midpoint of `l`
    - If midpoint is `v`, the value is found
    - If `v` less than midpoint, search the first half
    - If `v` greater than midpoint, search the second half
    - Stop when the interval to search becomes empty
- Binary search

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 // 2 = 0$, so next call reaches empty interval

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Binary search

- How long does this take?
  - Each call halves the interval to search
  - Stop when the interval become empty
- $\log n$ — number of times to divide n by 2 to reach 1
  - $1 \,//\, 2 = 0$, so next call reaches empty interval
- $O(\log n)$ steps

```
def binarysearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(binarysearch(v,l[:m]))
  else:
    return(binarysearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \mathbin{//} 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \mathbin{//} 2) + 1$, $n > 0$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n\,//\,2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n\,//\,2) + 1$, $n > 0$
- Solve by "unwinding"

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n\,//\,2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n\,//\,2) + 1$, $n > 0$
- Solve by "unwinding"
- $T(n) \;\; = T(n\,//\,2) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n\,//\,2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n\,//\,2) + 1$, $n > 0$
- Solve by “unwinding”
- $T(n) \;= T(n\,//\,2) + 1$
  $\quad\quad\;\; = (T(n\,//\,4) + 1) + 1$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n \,//\, 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n \,//\, 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n \,//\, 2) + 1 \\ &= (T(n \,//\, 4) + 1) + 1 = T(n \,//\, 2^2) + \underbrace{1 + 1}_{2} \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n /\!/ 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n /\!/ 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n /\!/ 2) + 1 \\ &= (T(n /\!/ 4) + 1) + 1 = T(n /\!/ 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n /\!/ 2^k) + \underbrace{1 + \cdots + 1}_{k} \end{aligned}$$

```python
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n\,//\,2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n\,//\,2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n\,//\,2) + 1 \\ &= (T(n\,//\,4) + 1) + 1 = T(n\,//\,2^2) + \underbrace{1+1}_{2} \\ &= \cdots \\ &= T(n\,//\,2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Alternative calculation

- $T(n)$: the time to search a list of length n
  - If $n = 0$, we exit, so $T(n) = 1$
  - If $n > 0$, $T(n) = T(n // 2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n // 2) + 1$, $n > 0$
- Solve by "unwinding"
- $$\begin{aligned} T(n) &= T(n // 2) + 1 \\ &= (T(n // 4) + 1) + 1 = T(n // 2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n // 2^k) + \underbrace{1 + \cdots + 1}_{k} \\ &= T(1) + k, \text{ for } k = \log n \\ &= (T(0) + 1) + \log n = 2 + \log n \end{aligned}$$

```
def bsearch(v,l):
  if l == []:
    return(False)

  m = len(l)//2

  if v == l[m]:
    return(True)

  if v < l[m]:
    return(bsearch(v,l[:m]))
  else:
    return(bsearch(v,l[m+1:]))
```

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time

Summary

- Search in an unsorted list takes time $O(n)$
  - Need to scan the entire list
  - Worst case is when the value is not present in the list
- For a sorted list, binary search takes time $O(\log n)$
  - Halve the interval to search each time
- In a sorted list, we can determine that v is absent by examining just $\log n$ values!