Lecture 19, 24 October 2024

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 19, 24 Oct 2024

Asymptotic worst-case complexity

Input size n $\longrightarrow$ running time $f(n)$ as $n \to \infty$

Asymptotic

For a given input size, many different inputs

Worst case

$f(n) = O(g(n))$

Upper bound

$\exists c.\ \forall n \quad f(n) \leq c\,g(n)$

Asymptotic worst-case complexity

$n^2 + 7n + 22$ is $O(n^2)$

Focus on largest term

Ignore constants

Takes constant time $\rightarrow$ $O(1)$

$$k \leq c \cdot 1$$

Orders of magnitude

Input size	Values of $t(n)$						
	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1000	10^6
100	6.6	100	66	10^4	10^6	10^{30}	10^{157}
1000	10	1000	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}			
10^6	20	10^6	10^7	10^{12}			
10^7	23	10^7	10^8				
10^8	27	10^8	10^9				
10^9	30	10^9	10^{10}				
10^{10}	33	10^{10}	10^{11}				

feasibility boundary

Searching

Given $[x_0, x_1, \ldots, x_{n-1}]$, check if v belongs to sequence

General case – scan each x_i

Worst case – v is not present

$O(n)$

Sorted sequence (assume distinct) $x_0 < x_1 < x_2 \ldots < x_{n-1}$

Check $x_{n/2}$ $(x_{n//2})$

Halve search interval till you reach an interval of size 1

Searching a sorted list — 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:]))
```

[7, 12]

Search 6

$(0+1)//2 = 0$

exclude l[m]

Searching a sorted list — 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:]))
```

Analysis

Informal

$$\underbrace{n \rightarrow n/2 \rightarrow n/4 \rightarrow \cdots \rightarrow 1}_{\log_2 n \text{ steps}}$$

Searching a sorted list — 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:]))
```

More formally

$T(n)$ - time for input n

$T(n) = 1 + T(n/2)$

$T(0) = T(1) = 1$ - think $O(1)$

Searching a sorted list — 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:]))
```

$T(n) = 1 + \underbrace{T(n/2)}_{?}$

$+ 1 + T(n/4)$

$1 + T(n/8)$

$\vdots$

$T(1)$

k steps

$T(n) = \underbrace{1+1+1 \ldots +1}_{k} + T\left(\frac{n}{2^k}\right)$

$k = \log n$

$= (\log n) + 1$

$\therefore T(n) = O(\log n)$

```
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:]))
```

← $O(m)$ steps

What if l is a (real) list, not an array?

Accessing $l[i]$ takes i steps
– "walk" to $l[i]$

$$T(n) = \underbrace{n/2}_{\text{walk to } l[m]} + T(n/2)$$

$$T(n) = n/2 + T(n/2)$$
$$= n/2 + n/4 + T(n/4) \ldots$$

Searching a sorted list — 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:]))
```

$T(n) = \underbrace{n/2 + n/4 + n/8 + \cdots 1}_{n}$

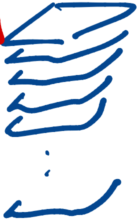
Not an array — may as well do sequential search

Convert an input to ascending order

Stack of cards / papers

"Selection Sort"

l[0]



l[n-1]

Find max

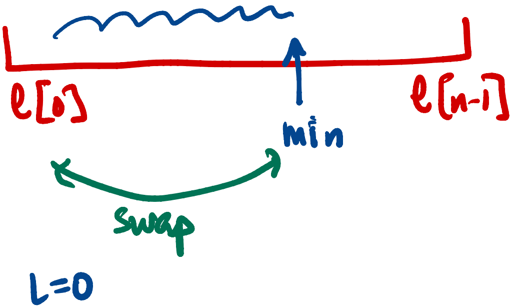
mark paper

Move to new pile

Repeat

Selection sort

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)     // Base case
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos: position of minimum in L[i:]
        for j in range(i+1,n):
          if L[j] < L[mpos]:
              mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i],L[mpos]) = (L[mpos],L[i])
        # Now L[:i+1] is sorted
    return(L)
```



Selection sort

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos: position of minimum in L[i:]
        for j in range(i+1,n):
            if L[j] < L[mpos]:
                mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i],L[mpos]) = (L[mpos],L[i])
        # Now L[:i+1] is sorted
    return(L)
```

Analysis

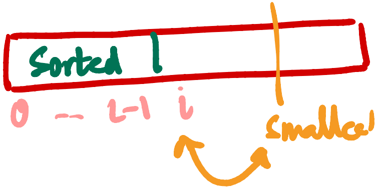
i	inner loop
0	$n-1$
1	$n-2$
2	$n-3$
⋮	⋮
$n-2$	1
$n-1$	0

$$\sum_{j=1}^{n-1} j \Rightarrow O(n^2)$$

Selection sort

```
def SelectionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        mpos = i
        # mpos: position of minimum in L[i:]
        for j in range(i+1,n):
            if L[j] < L[mpos]:
                mpos = j
        # L[mpos] : smallest value in L[i:]
        # Exchange L[mpos] and L[i]
        (L[i],L[mpos]) = (L[mpos],L[i])
        # Now L[:i+1] is sorted
    return(L)
```

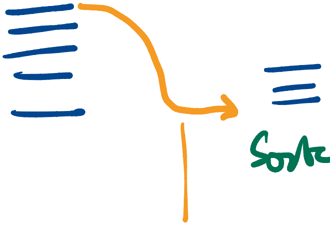
Correctness



Sorting

Another intuitive sort

Unsorted stack



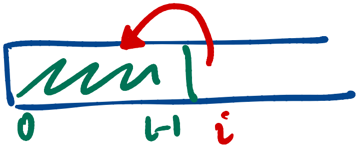
Sorted Stack

Insert into correct position

Insertion Sort

Insertion sort

```
def InsertionSort(L):
   n = len(L)
   if n < 1:
      return(L)          // Base
   for i in range(n):
      # Assume L[:i] is sorted   ← Invariant
      # Move L[i] to correct position in L[:i]
      j = i
      while(j > 0 and L[j] < L[j-1]):
        (L[j],L[j-1]) = (L[j-1],L[j])
        j = j-1
      # Now L[:i+1] is sorted
   return(L)
```



5 4 3 2 1
4 5 3 2 1
3 4 5 2 1

where L[i]
is smaller than all of L[0:i]

Insertion sort

```
def InsertionSort(L):
    n = len(L)
    if n < 1:
        return(L)
    for i in range(n):
        # Assume L[:i] is sorted
        # Move L[i] to correct position in L[:i]
        j = i
        while(j > 0 and L[j] < L[j-1]):
            (L[j],L[j-1]) = (L[j-1],L[j])
            j = j-1
        # Now L[:i+1] is sorted
    return(L)
```

Worst case - descending

Best case - ascending

Selection Sort - no worst/best case

i	Inner loop
0	0
1	1
2	
⋮	
n-1	n-1

$$\sum_{i=1}^{n-1} j$$

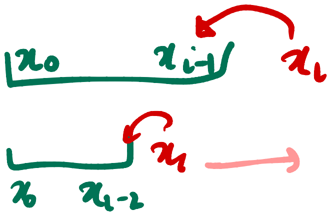
$O(n^2)$

Insertion sort

```python
def Insert(L,v):
    n = len(L)
    if n == 0:
      return([v])
    if v >= L[-1]:
      return(L+[v])
    else:
      return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
    n = len(L)
    if n < 1:
       return(L)
    L = Insert(ISort(L[:-1]),L[-1])
    return(L)
```

last elem



Insert last element into sorted n-2 prefix

Insertion sort

```
def Insert(L,v):
    n = len(L)
    if n == 0:
      return([v])
    if v >= L[-1]:
      return(L+[v])
    else:
      return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
    n = len(L)
    if n < 1:
       return(L)
    L = Insert(ISort(L[:-1]),L[-1])
    return(L)
```

Analysis?

ISort

$$T(n) = T(n-1) + \underbrace{T'(n-1)}_{\text{Insert}} \quad n$$

Insert

$$T'(n) = T'(n-1) + 1$$

$$\Rightarrow O(n) \qquad \underbrace{1+1+1 \cdots +1}_{n}$$

Insertion sort

```
def Insert(L,v):
    n = len(L)
    if n == 0:
      return([v])
    if v >= L[-1]:
      return(L+[v])
    else:
      return(Insert(L[:-1],v)+L[-1:])

def ISort(L):
    n = len(L)
    if n < 1:
       return(L)
    L = Insert(ISort(L[:-1]),L[-1])
    return(L)
```

$T(n) = T(n-1) + n$

Insert

$T'(n-1)$

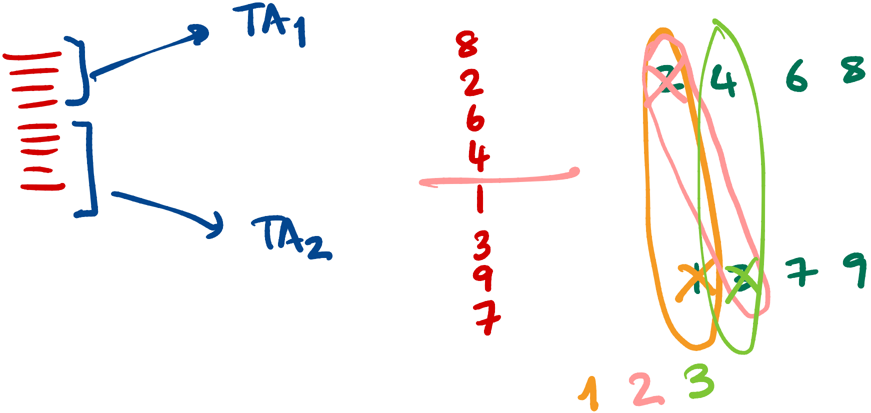
$= T(n-2) + (n-1) + n$

$\vdots$

$= 1 + 2 + 3 \ldots + n$

$O(n^2)$

How to go below $O(n^2)$ for sorting?



Sorting

Merging two sorted lists into one list

$n/2$, $n/2$

$\Downarrow$

n

Each comparison adds 1 element to output

$O(n)$

Sorting

What did TA_1 & TA_2 do?

Sub-TAs

Merge Sort