Lecture 20, 27 October 2026

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 20, 27 Oct 2026

Asymptotic worst-case complexity

Time complexity

$T(n)$ — input of size n

Average? Worst Case

Adversarial

Order of magnitude

$112n^2 + 360n \rightsquigarrow O(n^2)$

$f(n)$ is $O(g(n))$ if $\exists c$ s.t. $f(n) \leq c \cdot g(n)$

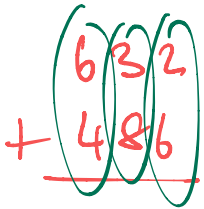
Asymptotic worst-case complexity

Input size ?

- Lists -- "Size" - length

- Checking primality?
  - Check all factors $2 \ldots n-1$ $O(n)$ checks

Asymptotic worst-case complexity



6632
4486

Effort ≈ digits

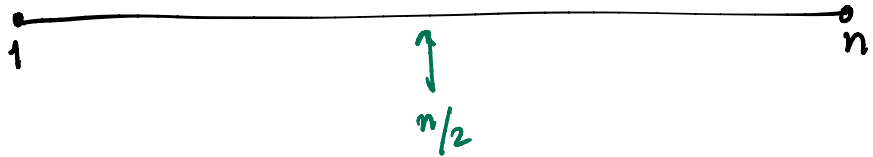
digits in base b → $\log_b(n) + 1$

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}				

Searching

Binary search - "Monotonicity"



Prune the search interval by 1/2

$n \to n/2 \to n/2^2 \to \cdots \to n/2^k \rightsquigarrow$ Stop at $k = \log_2 n$

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:]))
```

Check for v=3 in [4]

ln = 1

m = 0

left l[0:m]

inclusive

excludes l[m]

[4]

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:]))
```

n

constant

n/2

Complexity ?

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

$T(0) = T(1) = 1$ — Base Case

Recurrence

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

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

$1 + T(n/2^3)$

$\sim 1 + T(1)$

$1 + 1 + \cdots + 1$

$\log_2(n)$

Searching a sorted list — binary search, with position

Assume distinct

```
def binarysearchpos(v,l):
  if l == []:
    return(-1)

  m = len(l)//2

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

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

m+1 n-1
0 n-m

Sorting

How to sort?

How long does it take?

n^2 - bad

$n \log n$ - good target

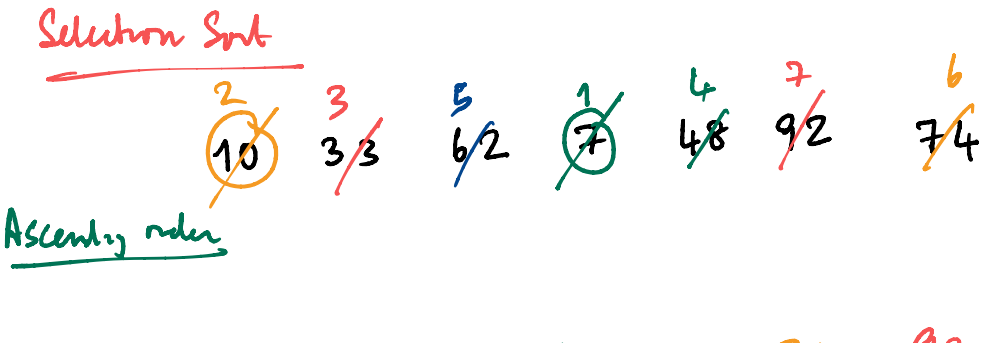
Naive strategies



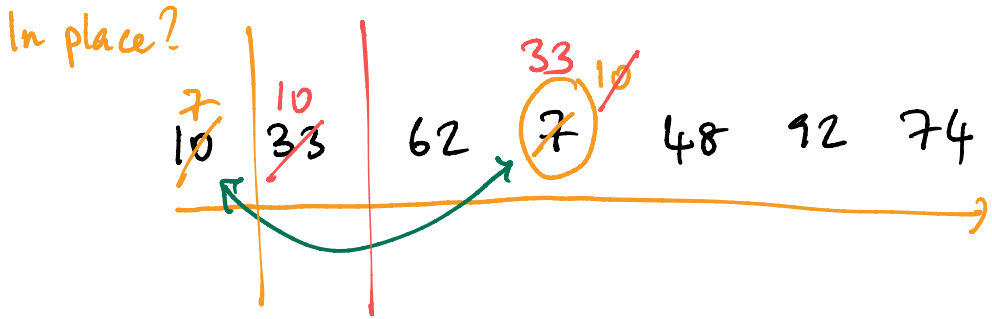


"Selection" Sort

Sorting

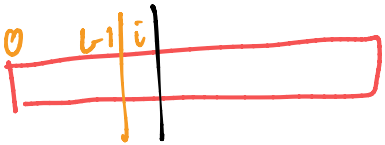

Selection Sort
2
3
5
1
4
7
6
10
3 3
6 2
7
4 8
9 2
7 4
Ascending order
7
10
33
48
62
74
92
In place?

Sorting



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)
```



0 .. n-1

$T(n)$?

n

$n-1, n-2, \ldots, 1$

$(n-1)+(n-2)+\cdots+1$

$= \frac{n(n-1)}{2} \sim O(n^2)$

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)
```

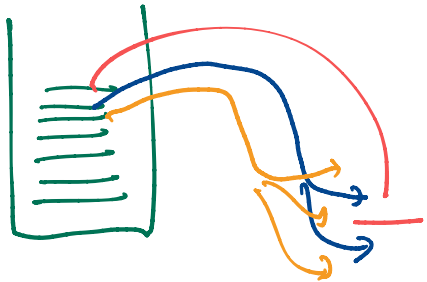
$O(n^2)$ is always the case

Does not care if L is sorted --

Trivial fix

— Check already sorted

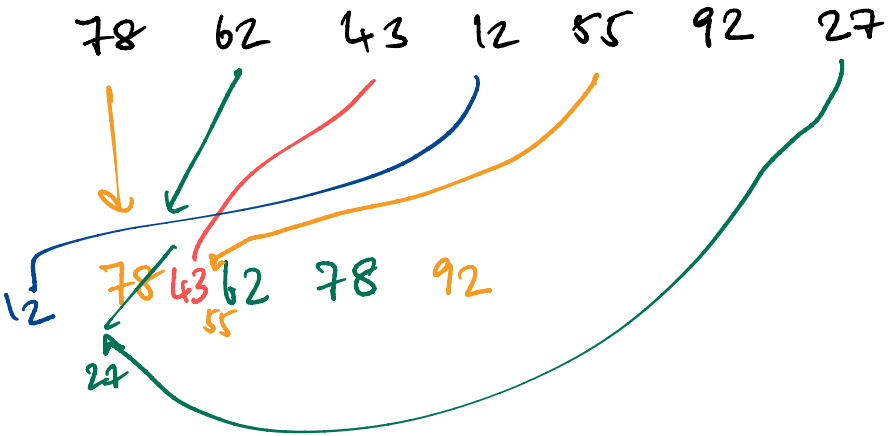
Other "natural" sort



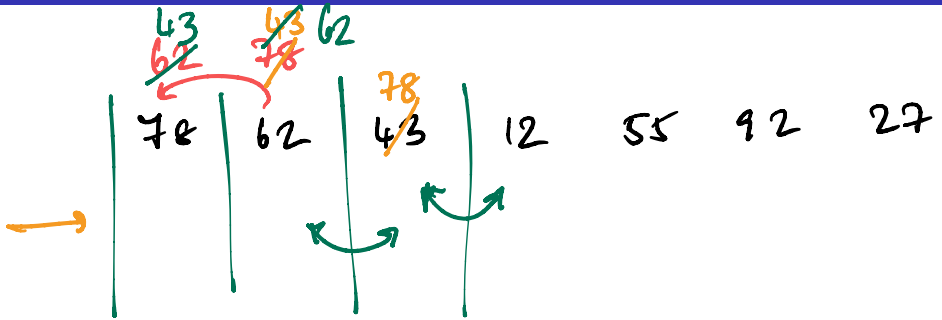
Each new value is "inserted" in the correct position

Insertion sort

Sorting



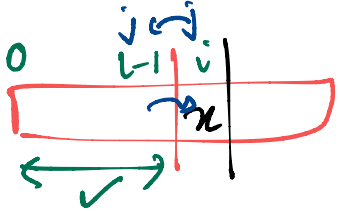
Sorting



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)
```

n iterations



Iteration	0	1	--	n-1
Work	1	2	---	n

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

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 order

Ascending order?

Outer loop

L[i] > L[i-1] always

Only 1 comparison

O(n)

Insertion sort

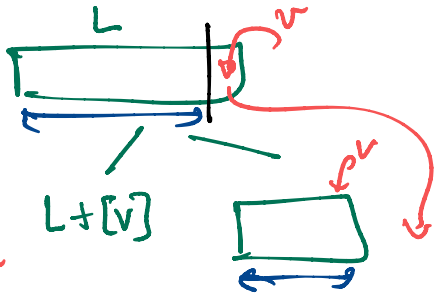
```
def Insert(L,v):
    n = len(L)
    if n == 0:
      return([v])
    if v >= L[-1]:
      return(L+[v])
    else:
      return(Insert(L[:-1],v)+L[-1:])

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

sorted

check last value in L

Insert in prefix, append last value



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 ?

IT(n) — Insert

ST(n) — Sort

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

$1 + T(n-2)$

$T(0) = T(1) = 1$

$\Downarrow$

n

$ST(0) = ST(1) = 1$

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

$IT(n-1)$

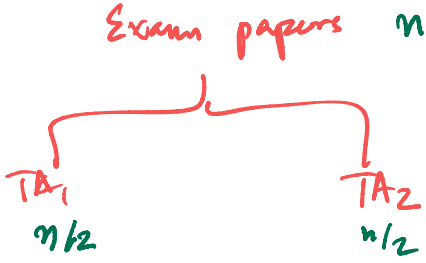
$n + (n-1) + (n-2) \ldots$

Sorting

Selection Sort
Insertion Sort
] Worst Case $O(n^2)$

Bubble Sort []

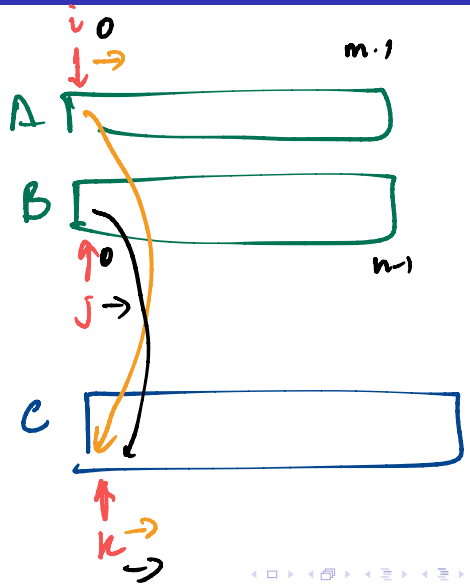
Better strategy?





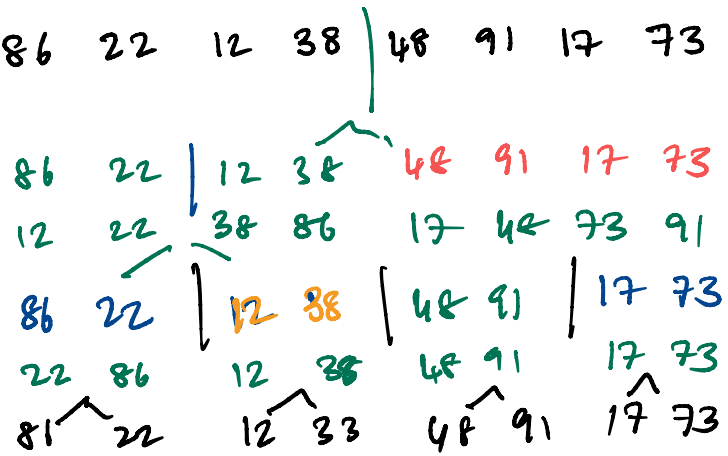
Merging sorted lists

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```



Merging sorted lists

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```



Merge sort

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (m-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

"Divide & conquer"

What is the complexity?