Recursive Insertion Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 16, 19 Oct 2023

Insertion sort

- You are the TA for a course
  - Instructor has a pile of evaluated exam papers
  - Papers in random order of marks
  - Your task is to arrange the papers in descending order of marks

Strategy 2

- Move the first paper to a new pile
- Second paper
  - Lower marks than first paper? Place below first paper in new pile
  - Higher marks than first paper? Place above first paper in new pile
- Third paper
  - Insert into correct position with respect to first two
- Do this for the remaining papers
  - Insert each one into correct position in the second pile

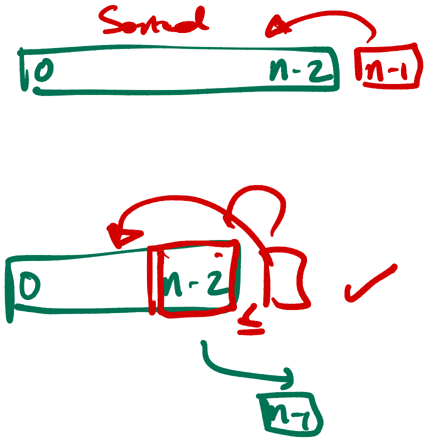
Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`

```
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
    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)
```

Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`
- A recursive formulation
  - Inductively sort `L[:i]`
  - Insert `L[i]` in `L[:i]`



Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list
- An iterative formulation
  - Assume `L[:i]` is sorted
  - Insert `L[i]` in `L[:i]`
- A recursive formulation
  - Inductively sort `L[:i]`
  - Insert `L[i]` in `L[:i]`

```
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 of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`

```
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 of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$

```
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 of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$

```
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 of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$ ✓ n-1

```
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 of recursive insertion sort

- For input of size n, let
  - $TI(n)$ be the time taken by `Insert`
  - $TS(n)$ be the time taken by `ISort`
- First calculate $TI(n)$ for `Insert`
  - $TI(0) = 1$
  - $TI(n) = TI(n-1) + 1$
  - Unwind to get $TI(n) = n$
- Set up a recurrence for $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$
- Unwind to get $1 + 2 + \cdots + n - 1$

```
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)
```

Merge Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 16, 19 Oct 2023

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half

Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$
- How can we bring the complexity below $O(n^2)$?

Strategy 3

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted halves to get a fully sorted list

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

32	74	89
21	55	64

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

32	74	89
~~21~~	55	64

21

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ 74 89

~~21~~ 55 64

21 32

Combining two sorted lists

- Combine two sorted lists A and B into a single sorted list C
  - Compare first elements of A and B
  - Move the smaller of the two to C
  - Repeat till you exhaust A and B

~~32~~	74	89
~~21~~	~~55~~	64

21	32	55

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ 74 89

~~21~~ ~~55~~ ~~64~~ ?

21 32 55 64

Combining two sorted lists

- Combine two sorted lists A and B into a single sorted list C
  - Compare first elements of A and B
  - Move the smaller of the two to C
  - Repeat till you exhaust A and B

~~32~~ ~~74~~ 89

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`

~~32~~ ~~74~~ ~~89~~

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74 89

Combining two sorted lists

- Combine two sorted lists `A` and `B` into a single sorted list `C`
  - Compare first elements of `A` and `B`
  - Move the smaller of the two to `C`
  - Repeat till you exhaust `A` and `B`
- Merging `A` and `B`

~~32~~ ~~74~~ ~~89~~

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74 89

Merge sort

- Let `n` be the length of L

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

43	32

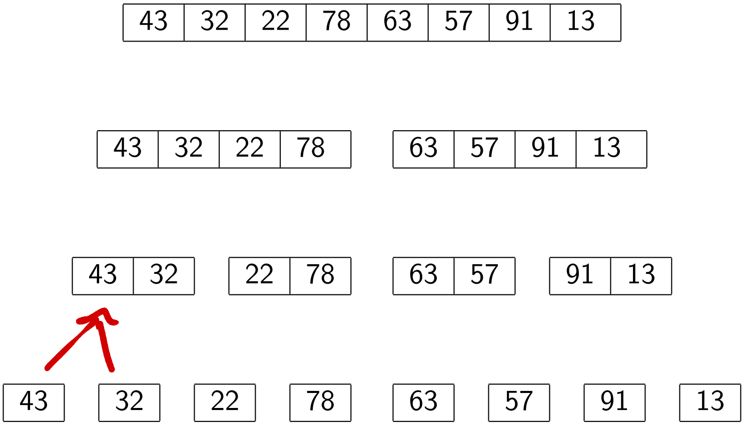
22	78

63	57

91	13

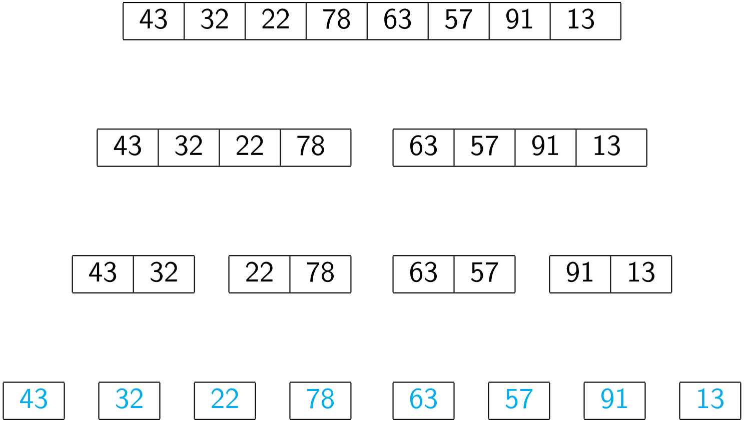
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



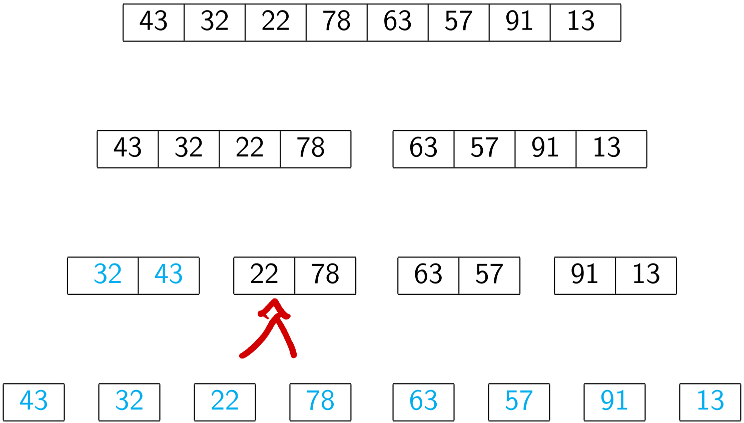
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



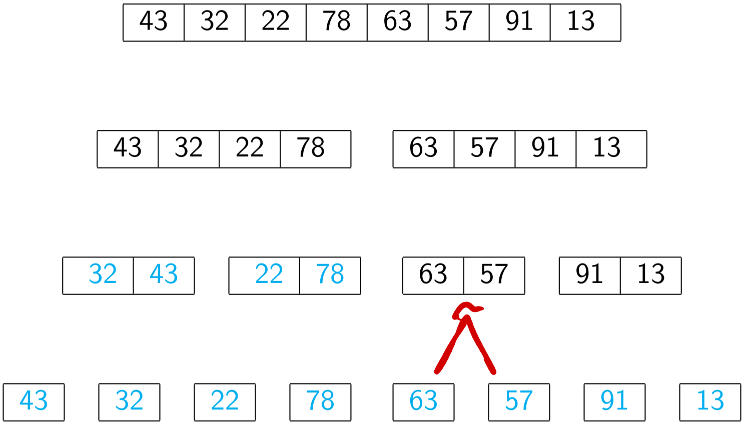
Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

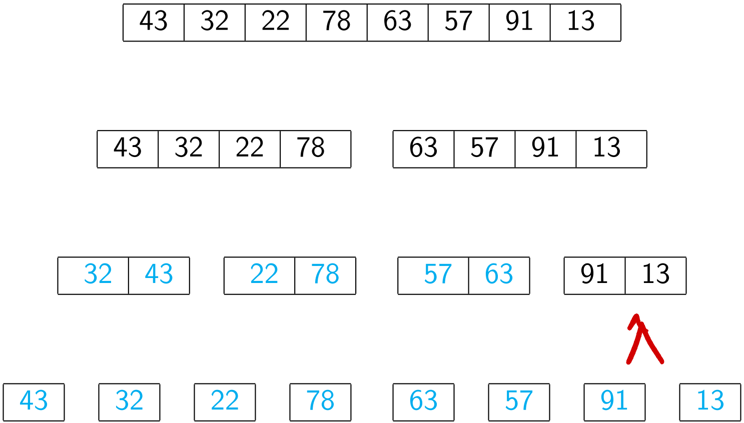


Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

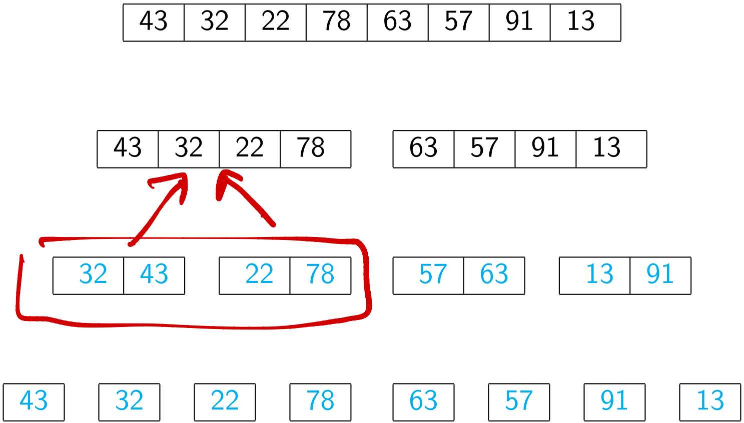


- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

32	43

22	78

57	63

13	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

22	32	43	78

63	57	91	13

32	43

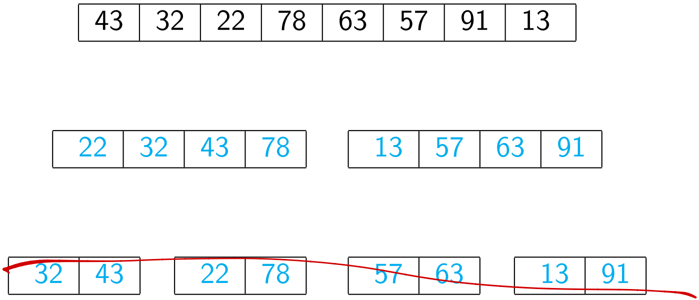
22	78

57	63

13	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!



Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

43	32	22	78	63	57	91	13

22	32	43	78

13	57	63	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

13	22	32	43	57	63	78	91

22	32	43	78

13	57	63	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

13	22	32	43	57	63	78	91

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Merge sort

- Let `n` be the length of L
- Sort `A[:n//2]`
- Sort `A[n//2:]`
- Merge the sorted halves into `B`
- How do we sort `A[:n//2]` and `A[n//2:]`?
  - Recursively, same strategy!

Divide and Conquer

- Break up the problem into disjoint parts
- Solve each part separately
- Combine the solutions efficiently

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`

Merging sorted lists

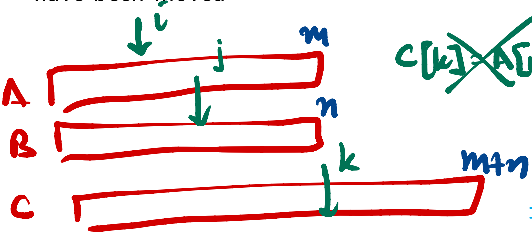
- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`
  - Otherwise, compare first elements of `A` and `B`
    - Move the smaller of the two to `C`

Merging sorted lists

- Combine two sorted lists `A` and `B` into `C`
  - If `A` is empty, copy `B` into `C`
  - If `B` is empty, copy `A` into `C`
  - Otherwise, compare first elements of `A` and `B`
    - Move the smaller of the two to `C`
  - Repeat till all elements of `A` and `B` have been moved

Merging sorted lists

- Combine two sorted lists A and B into C
  - If A is empty, copy B into C
  - If B is empty, copy A into C
  - Otherwise, compare first elements of A and B
    - Move the smaller of the two to C
  - Repeat till all elements of A and B have been moved



~~C[k] = A[i]~~

```
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)
```

len(C) < m+n

Boundary

Real merge

Merge sort

- To sort `A` into `B`, both of length n

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise

not n=0
else you will keep
splitting singleton list

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`
  - Merge `L` and `R` into `B`

Merge sort

- To sort `A` into `B`, both of length n
- If $n \leq 1$, nothing to be done
- Otherwise
  - Sort `A[:n//2]` into `L`
  - Sort `A[n//2:]` into `R`
  - Merge `L` and `R` into `B`

```
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)
```

Analysing merge

- Merge A of length m, B of length n

merge (A, B)

if A == [] :
    return B
elif B == [] :
    return A
elif A[0] < B[0] :
    return ([A[0]] + merge(A[1:], B))
--

```
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 + (n-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)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$

```
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 + (n-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)
```

Analysing merge

- Merge A of length m, B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C

```
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 + (n-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)
```

Analysing merge

- Merge A of length m, B of length n
- Output list C has length $m + n$
- In each iteration we add (at least) one element to C
- Hence merge takes time $O(m + n)$

```
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 + (n-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)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m + n$
- In each iteration we add (at least) one element to `C`
- Hence `merge` takes time $O(m + n)$
- Recall that $m + n \leq 2(\max(m, n))$

```
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 + (n-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)
```

Analysing merge

- Merge `A` of length m, `B` of length n
- Output list `C` has length $m+n$
- In each iteration we add (at least) one element to `C`
- Hence `merge` takes time $O(m+n)$
- Recall that $m+n \leq 2(\max(m,n))$
- If $m \approx n$, `merge` take time $O(n)$

```
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 + (n-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)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k

```
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)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k
- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
    - Solve two subproblems of size $n/2$
    - Merge the solutions in time $n/2 + n/2 = n$

```
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)
```

Analysing mergesort

- Let $T(n)$ be the time taken for input of size n
  - For simplicity, assume $n = 2^k$ for some k
- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
    - Solve two subproblems of size $n/2$
    - Merge the solutions in time $n/2 + n/2 = n$
- Unwind the recurrence to solve

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $T(n) \quad = 2T(n/2) + n$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n \end{aligned}$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $T(n) = 2T(n/2) + n$
  $= 2[2T(n/4) + n/2] + n = 2^2 T(n/2^2) + 2n$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\,[2T(n/2^3) + n/2^2] + 2n = 2^3\,T(n/2^3) + 3n \end{aligned}$$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$

- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\;\;\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$

- When $k = \log n$, $T(n/2^k) = T(1) = 1$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3T(n/2^3) + 3n \\ &\vdots \\ &= 2^kT(n/2^k) + kn \end{aligned}$$
- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n}T(1) + (\log n)n = n + n\log n$

$n \cdot 1$

```
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)
```

Analysing mergesort

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n = 2^2\,T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3\,T(n/2^3) + 3n \\ &\;\;\vdots \\ &= 2^k\,T(n/2^k) + kn \end{aligned}$$
- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n}\,T(1) + (\log n)n = n + n\log n$
- Hence $T(n)$ is $O(n \log n)$

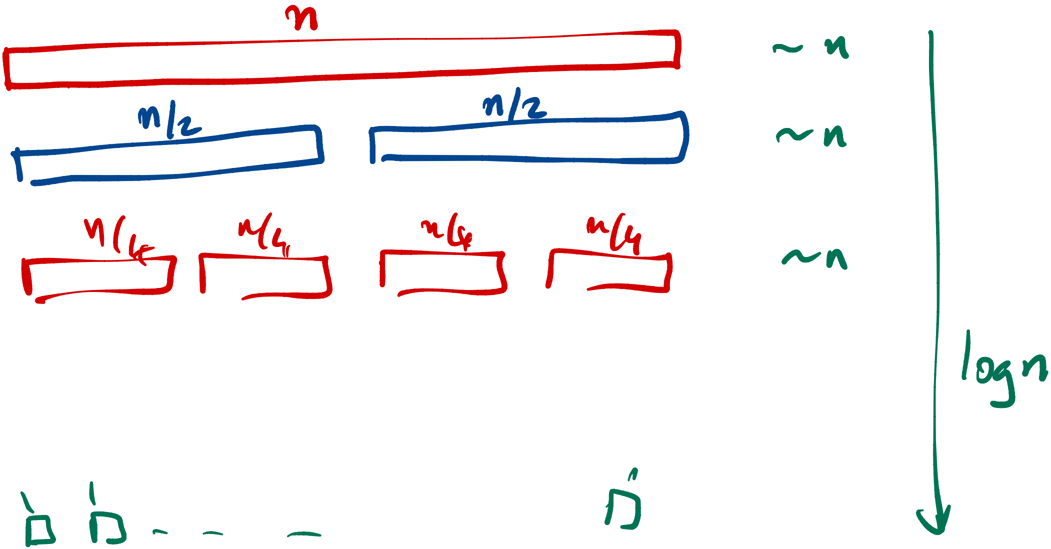
```
def mergesort(A):
  n = len(A)

  if n <= 1:
     return(A)

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

  B = merge(L,R)

  return(B)
```



n
n/2
n/2
n/4
n/4
n/4
n/4
~n
~n
~n
log n

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`
- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly

Summary

- Merge sort takes time $O(n \log n)$ so can be used effectively on large inputs
- Variations on merge are possible
  - Union of two sorted lists — discard duplicates, if `A[i] == B[j]` move just one copy to `C` and increment both `i` and `j`
  - Intersection of two sorted lists — when `A[i] == B[j]`, move one copy to `C`, otherwise discard the smaller of `A[i]`, `B[j]`
  - List difference — elements in `A` but not in `B`
- Merge needs to create a new list to hold the merged elements
  - No obvious way to efficiently merge two lists in place
  - Extra storage can be costly
- Inherently recursive
  - Recursive calls and returns are expensive

l1.extend(l2)
↓
l1 + l2

l1.append(v)
l1 + [v]