# Searching in a List

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 15, 15 Nov 2021

# 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 $n > 0$, $T(n) = T(n\,//\,2) + 1$
- Recurrence for $T(n)$
  - $T(0) = 1$
  - $T(n) = T(n\,//\,2) + 1$, $n > 0$

```
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"
- $T(n) \quad = T(n\,//\,2) + 1$

```
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"
- $$\begin{aligned} T(n) &= T(n\,//\,2) + 1 \\ &= (T(n\,//\,4) + 1) + 1 \end{aligned}$$

```
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"
- $$\begin{aligned} T(n) &= T(n /\!/ 2) + 1 \\ &= (T(n /\!/ 4) + 1) + 1 = T(n /\!/ 2^2) + \underbrace{1 + 1}_{2} \end{aligned}$$

```
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"
- $$\begin{aligned} T(n) &= T(n\,//\,2) + 1 \\ &= (T(n\,//\,4) + 1) + 1 = T(n\,//\,2^2) + \underbrace{1 + 1}_{2} \\ &= \cdots \\ &= T(n\,//\,2^k) + \underbrace{1 + \cdots + 1}_{k} \end{aligned}$$

```
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!

# Selection Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 15, 15 Nov 2021

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- 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

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- 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 1

- Scan the entire pile and find the paper with minimum marks

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- 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 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- 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 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time

# Sorting a list

- Sorting a list makes many other computations easier
  - Binary search
  - Finding the median
  - Checking for duplicates
  - Building a frequency table of values
- How do we sort a list?
- 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 1

- Scan the entire pile and find the paper with minimum marks
- Move this paper to a new pile
- Repeat with the remaining papers
  - Add the paper with next minimum marks to the second pile each time
- Eventually, the new pile is sorted in descending order

# Sorting a list

74 32 89 55 21 64

# Sorting a list

74 32 89 55 ~~21~~ 64

21

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| 74 | ~~32~~ | 89 | 55 | ~~21~~ | 64 |

| | |
|---|---|
| 21 | 32 |

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| 74 | ~~32~~ | 89 | ~~55~~ | ~~21~~ | 64 |
| 21 | 32 | 55 | | | |

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| 74 | ~~32~~ | 89 | ~~55~~ | ~~21~~ | ~~64~~ |

| | | | |
|---|---|---|---|
| 21 | 32 | 55 | 64 |

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| ~~74~~ | ~~32~~ | 89 | ~~55~~ | ~~21~~ | ~~64~~ |
| 21 | 32 | 55 | 64 | 74 | |

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| ~~74~~ | ~~32~~ | ~~89~~ | ~~55~~ | ~~21~~ | ~~64~~ |
| 21 | 32 | 55 | 64 | 74 | 89 |

# Selection sort

- Select the next element in sorted order

# Selection sort

- Select the next element in sorted order
- Append it to the final sorted list

# Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...

# Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order

# Selection sort

- Select the next element in sorted order
- Append it to the final sorted list
- Avoid using a second list
  - Swap the minimum element into the first position
  - Swap the second minimum element into the second position
  - ...
- Eventually the list is rearranged in place in ascending order

```
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 of selection sort

- Correctness follows from the invariant

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$

```
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 of selection sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $n - i$ steps to find minimum in `L[i:]`
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n) = n(n+1)/2$
- $T(n)$ is $O(n^2)$

```
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)
```

# Summary

- Selection sort is an intuitive algorithm to sort a list

# Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list

# Summary

- Selection sort is an intuitive algorithm to sort a list
- Repeatedly find the minimum (or maximum) and append to sorted list
- Worst case complexity is $O(n^2)$
  - Every input takes this much time
  - No advantage even if list is arranged carefully before sorting

# Insertion Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 15, 15 Nov 2021

# Sorting a list

- 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

# Sorting a list

- 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

# Sorting a list

- 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

# Sorting a list

- 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

# Sorting a list

- 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

# Sorting a list

- 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

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| 74 | 32 | 89 | 55 | 21 | 64 |

# Sorting a list

~~74~~ 32 89 55 21 64

74

# Sorting a list

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

32 74

# Sorting a list

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

32 74 89

# Sorting a list

| | | | | | |
|---|---|---|---|---|---|
| ~~74~~ | ~~32~~ | ~~89~~ | ~~55~~ | 21 | 64 |
| 32 | 55 | 74 | 89 | | |

# Sorting a list

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

21 32 55 74 89

# Sorting a list

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

21 32 55 64 74 89

# Insertion sort

- Start building a new sorted list

# Insertion sort

- Start building a new sorted list
- Pick next element and insert it into the sorted list

# 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]`

# 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]

```
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]`

```
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 iterative insertion sort

- Correctness follows from the invariant

```
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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency

```
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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times

```
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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $i$ steps to 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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $i$ steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$

```
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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $i$ steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$

```
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(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)
```

# Analysis of iterative insertion sort

- Correctness follows from the invariant
- Efficiency
  - Outer loop iterates $n$ times
  - Inner loop: $i$ steps to insert `L[i]` in `L[:i]`
  - $T(n) = 0 + 1 + \cdots + (n-1)$
  - $T(n) = n(n-1)/2$
- $T(n)$ is $O(n^2)$

```
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(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)
```

# 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 $TS(n)$
  - $TS(0) = 1$
  - $TS(n) = TS(n-1) + TI(n-1)$

```
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)
```

# Summary

- Insertion sort is another intuitive algorithm to sort a list

# Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list

# Summary

- Insertion sort is another intuitive algorithm to sort a list
- Create a new sorted list
- Repeatedly insert elements into the sorted list
- Worst case complexity is $O(n^2)$
  - Unlike selection sort, not all cases take time $n^2$
  - If list is already sorted, `Insert` stops in 1 step
  - Overall time can be close to $O(n)$

# Merge Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 15, 15 Nov 2021

# Beating the $O(n^2)$ barrier

- Both selection sort and insertion sort take time $O(n^2)$
- This is infeasible for $n > 10000$

# 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 $\texttt{n}$ 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!

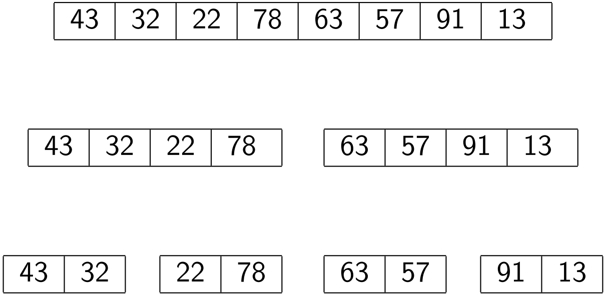
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 43 | 32 | 22 | 78 | 63 | 57 | 91 | 13 |

| | | | |
|---|---|---|---|
| 43 | 32 | 22 | 78 |

| | | | |
|---|---|---|---|
| 63 | 57 | 91 | 13 |

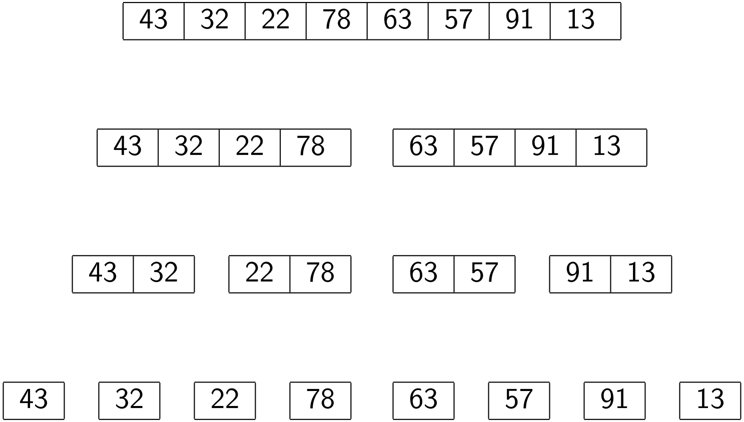
# Merge sort

- 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!



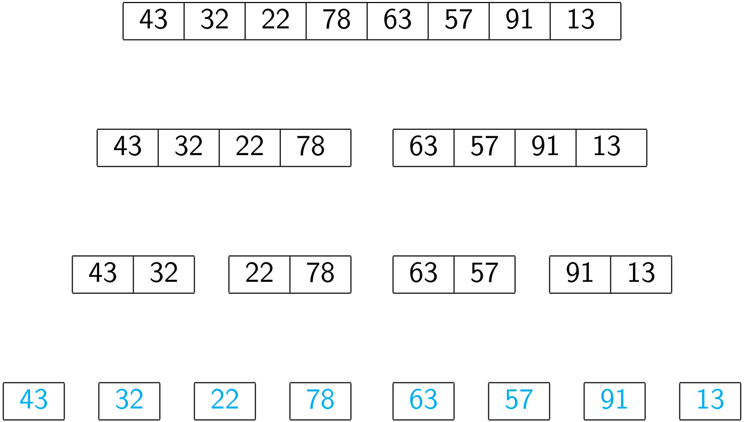
# Merge sort

- 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!



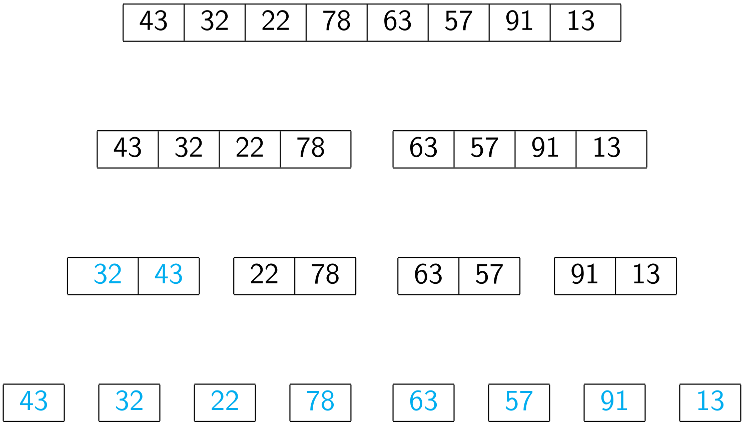
# Merge sort

- 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!



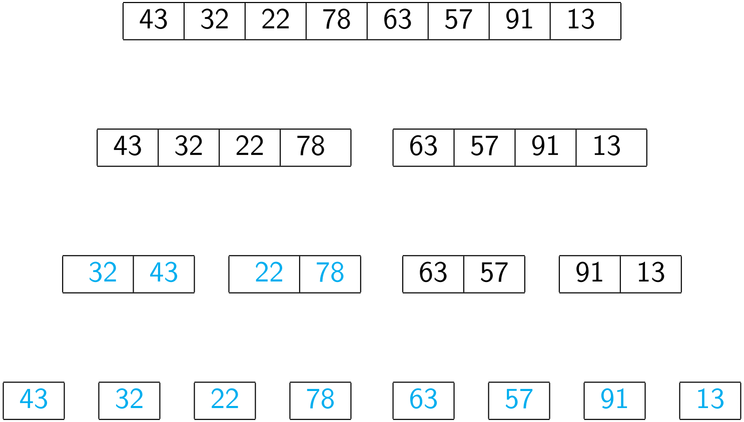
# Merge sort

- 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!



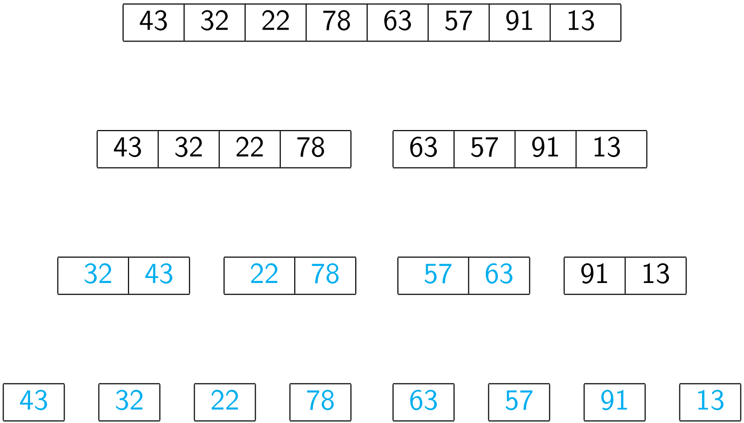
# Merge sort

- 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!



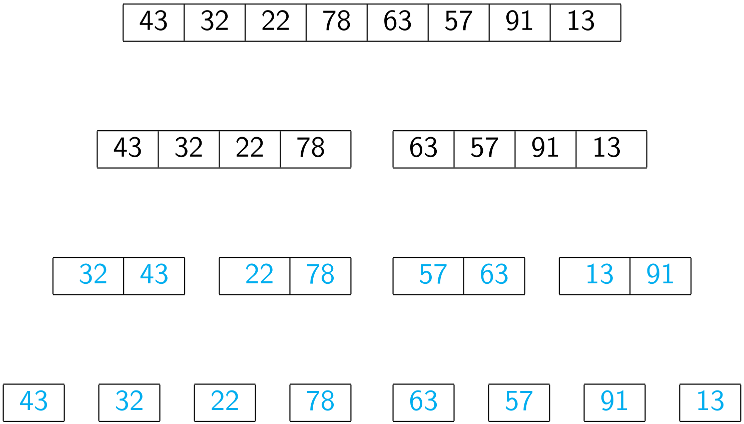
# Merge sort

- 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!



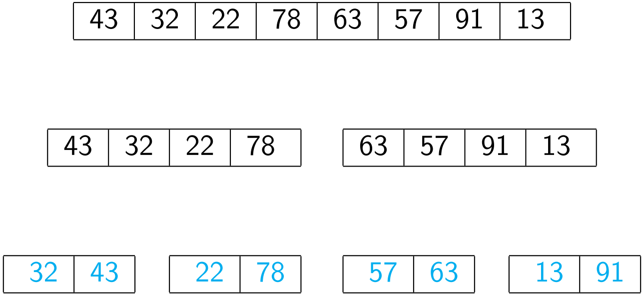
# Merge sort

- 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!

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 43 | 32 | 22 | 78 | 63 | 57 | 91 | 13 |

| | | | |
|---|---|---|---|
| 22 | 32 | 43 | 78 |

| | | | |
|---|---|---|---|
| 63 | 57 | 91 | 13 |

| | |
|---|---|
| 32 | 43 |

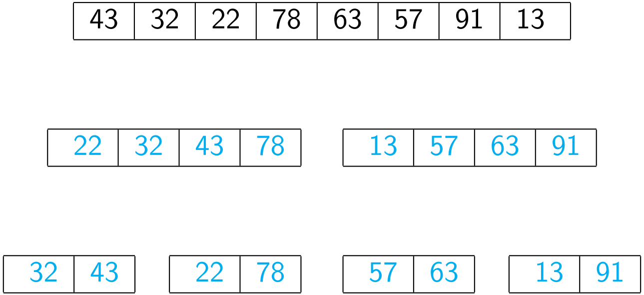
| | |
|---|---|
| 22 | 78 |

| | |
|---|---|
| 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

```
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)
```

# 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

# 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)
```

# Summary

- Merge sort using divide and conquer to sort a list

# Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves

# Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half

# Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves

# Summary

- Merge sort using divide and conquer to sort a list
- Divide the list into two halves
- Sort each half
- Merge the sorted halves
- Next, we have to check that the complexity is less than $O(n^2)$