Analysis of Merge Sort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 16, 18 Nov 2021

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`

Merging 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

Analysing merge

- Merge `A` of length m, `B` of length 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 $m + n$

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

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

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

```python
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 $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$

```
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)
```

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`

Can also be done with dictionaries

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

Quicksort

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 16, 18 Nov 2021

Shortcomings of merge sort

- 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

Shortcomings of merge sort

- 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
- Merging happens because elements in the left half need to move to the right half and vice versa
  - Consider an input of the form `[0,2,4,6,1,3,5,9]`

Shortcomings of merge sort

- 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
- Merging happens because elements in the left half need to move to the right half and vice versa
  - Consider an input of the form `[0,2,4,6,1,3,5,9]`
- Can we divide the list so that everything on the left is smaller than everything on the right?
  - No need to merge!

Divide and conquer without merging

- Suppose the median of `L` is m

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- Recurslvely sort left and right halves
  - `L` is now sorted, no merge!

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- Recurslvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- RecursIvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- Recurslvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

- How do we find the median?

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- RecursIvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

- How do we find the median?
  - Sort and pick up the middle element

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- Recurslvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

- How do we find the median?
  - Sort and pick up the middle element
  - But our aim is to sort the list!

Divide and conquer without merging

- Suppose the median of `L` is m
- Move all values $\leq m$ to left half of `L`
  - Right half has values $> m$
- RecursIvely sort left and right halves
  - `L` is now sorted, no merge!
- Recurrence: $T(n) = 2T(n/2) + n$
  - Rearrange in a single pass, time $O(n)$
- So $T(n)$ is $O(n \log n)$

- How do we find the median?
  - Sort and pick up the middle element
  - But our aim is to sort the list!
- Instead pick some value in `L` — pivot
  - Split `L` with respect to the pivot element

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13

  - Identify pivot

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13

  - Identify pivot
  - Mark lower elements and upper elements

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition L into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13

  - Identify pivot
  - Mark lower elements and upper elements
- Rearrange the elements as lower–pivot–upper

32	22	13	43	78	63	57	91

Quicksort [C.A.R. Hoare]

- Choose a pivot element
  - Typically the first element in the array
- Partition `L` into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions

Input list is l

lower + [l[0]] + upper

High level view of quicksort

- Input list

43	32	22	78	63	57	91	13

  - Identify pivot
  - Mark lower elements and upper elements
- Rearrange the elements as lower–pivot–upper

32	22	13	43	78	63	57	91

- Recursively sort the lower and upper partitions

lower = [x for x in l[1:] if x<=l[0]]

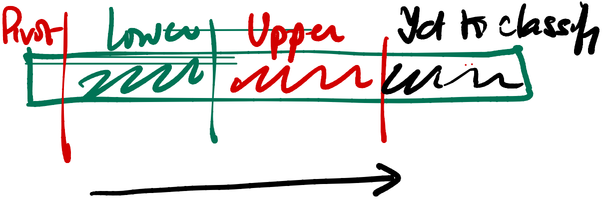
upper = [x for x in l[1:] if x>l[0]]

Partitioning

- Scan the list from left to right

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified

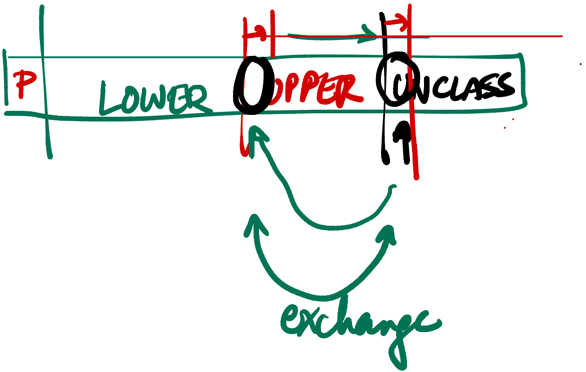


Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
    - If it is larger than the pivot, extend Upper to include this element



Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

Unclassified

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13
			↑			↑	

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	78	63	57	91	13

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

Unclassfi

43 $\boxed{32\ 22}$ $\boxed{78\ 63}$ 13 57 91

$\boxed{32\ 22\ 13}$ $\boxed{63\ 78}$ 57 91

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

43	32	22	13	78	63	57	91

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments

LOWER - PIVOT - UPPER

Partitioning

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Examine the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

13	32	22	43	78	63	57	91

- Pivot is always the first element
- Maintain two indices to mark the end of the Lower and Upper segments
- After partitioning, exchange the pivot with the last element of the Lower segment

Quicksort code

- Scan the list from left to right
- Four segments: Pivot, Lower, Upper, Unclassified
- Classify the first unclassified element
  - If it is larger than the pivot, extend Upper to include this element
  - If it is less than or equal to the pivot, exchange with the first element in Upper. This extends Lower and shifts Upper by one position.

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$

pivot = L[l]

L : L[l+1 : lower]

R = L[lower : upper]

L[lower] is the pivot & is excluded

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

not 1 but l

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
  - $T(n) = 2T(n/2) + n$
  - $T(n)$ is $O(n \log n)$

O(n)

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
  - $T(n) = 2T(n/2) + n$
  - $T(n)$ is $O(n \log n)$
- Worst case? Pivot is maximum or minimum
  - Partitions are of size 0, $n - 1$
  - $T(n) = T(n-1) + n$
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n)$ is $O(n^2)$

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis

- Partitioning with respect to the pivot takes time $O(n)$
- If the pivot is the median
  - $T(n) = 2T(n/2) + n$
  - $T(n)$ is $O(n \log n)$
- Worst case? Pivot is maximum or minimum
  - Partitions are of size 0, $n-1$
  - $T(n) = T(n-1) + n$
  - $T(n) = n + (n-1) + \cdots + 1$
  - $T(n)$ is $O(n^2)$
- Already sorted array: worst case!

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
  - Values don't matter, only relative order is important
  - Analyze behaviour over permutations of $\{1, 2, \ldots, n\}$
  - Each input permutation equally likely

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Analysis . . .

- However, average case is $O(n \log n)$
- Sorting is a rare situation where we can compute this
  - Values don't matter, only relative order is important
  - Analyze behaviour over permutations of $\{1, 2, \ldots, n\}$
  - Each input permutation equally likely
- Expected running time is $O(n \log n)$

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position randomly at each step

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Randomization

- Any fixed choice of pivot allows us to construct worst case input
- Instead, choose pivot position randomly at each step
- Expected running time is again $O(n \log n)$

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Quicksort in practice

- Can be implemented iteratively
  - Recursive calls — disjoint segments, no recombination of results required
  - Explicitly track endpoints of each segment to be sorted

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Quicksort in practice

- Can be implemented iteratively
  - Recursive calls — disjoint segments, no recombination of results required
  - Explicitly track endpoints of each segment to be sorted
- In practice, quicksort is very fast

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Quicksort in practice

- Can be implemented iteratively
  - Recursive calls — disjoint segments, no recombination of results required
  - Explicitly track endpoints of each segment to be sorted
- In practice, quicksort is very fast
- Very often the default algorithm used for in-built sort functions
  - Sorting a column in a spreadsheet
  - Library sort function in a programming language

```
def quicksort(L,l,r):  # Sort L[l:r]
  if (r - l <= 1):
    return(L)
  (pivot,lower,upper) = (L[l],l+1,l+1)
  for i in range(l+1,r):
    if L[i] > pivot:  # Extend upper segment
      upper = upper+1
    else: # Exchange L[i] with start of upper segment
      (L[i], L[lower]) = (L[lower], L[i])
      # Shift both segments
      (lower,upper) = (lower+1,upper+1)
  # Move pivot between lower and upper
  (L[l],L[lower-1]) = (L[lower-1],L[l])
  lower = lower-1
  # Recursive calls
  quicksort(L,l,lower)
  quicksort(L,lower+1,upper)
  return(L)
```

Summary

- The worst case complexity of quicksort is $O(n^2)$

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs
- Quicksort works in-place and can be implemented iteratively

Summary

- The worst case complexity of quicksort is $O(n^2)$
- However, the average case is $O(n \log n)$
- Randomly choosing the pivot is a good strategy to beat worst case inputs
- Quicksort works in-place and can be implemented iteratively
- Very fast in practice, and often used for built-in sorting functions
  - Good example of a situation when the worst case upper bound is pessimistic

Sorting: Concluding Remarks

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 16, 18 Nov 2021

Stable sorting

- Often list values are tuples
  - Rows from a table, with multiple columns / attributes
  - A list of students, each student entry has a roll number, name, marks, ...

Stable sorting

- Often list values are tuples
  - Rows from a table, with multiple columns / attributes
  - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number

Stable sorting

- Often list values are tuples
  - Rows from a table, with multiple columns / attributes
  - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?

Stable sorting

- Often list values are tuples
  - Rows from a table, with multiple columns / attributes
  - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?
- Stability of sorting is crucial in many applications

Stable sorting

- Often list values are tuples
  - Rows from a table, with multiple columns / attributes
  - A list of students, each student entry has a roll number, name, marks, ...
- Suppose students have already been sorted by roll number
- If we now sort by name, will all students with the same name remain in sorted order with respect to roll number?
- Stability of sorting is crucial in many applications
- Sorting on column B should not disturb sorting on column A

Stable sorting

- The quicksort implementation we described is not stable
  - Swapping values while partitioning can disturb existing sorted order

Stable sorting

- The quicksort implementation we described is not stable
  - Swapping values while partitioning can disturb existing sorted order
- Merge sort is stable if we merge carefully
  - Do not allow elements from the right to overtake elements on the left
  - While merging, prefer the left list while breaking ties

Other criteria

- Minimizing data movement
  - Imagine each element is a heavy carton
  - Reduce the effort of moving values around

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for "external" sorting
  - Database tables that are too large to store in memory all at once
  - Retrieve in parts from the disk and write back

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for "external" sorting
  - Database tables that are too large to store in memory all at once
  - Retrieve in parts from the disk and write back
- Other $O(n \log n)$ algorithms exist — heapsort

Best sorting algorithm?

- Quicksort is often the algorithm of choice, despite $O(n^2)$ worst case
- Merge sort is typically used for "external" sorting
  - Database tables that are too large to store in memory all at once
  - Retrieve in parts from the disk and write back
- Other $O(n \log n)$ algorithms exist — heapsort
- Sometimes hybrid strategies are used
  - Use divide and conquer for large n
  - Switch to insertion sort when n becomes small (e.g., $n < 16$)