Analysis of Merge Sort

Madhavan Mukund

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

Programming, Data Structures and Algorithms using Python

Week 2

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
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $T(n) = 2T(n/2) + n$
  $= 2\,[2T(n/4) + n/2] + n$

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

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

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

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

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

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