Merge Sort

Alexandra Stefan

Last updated: 11/29/2022

Mergesort

- Idea
- Code/pseudocode
- Properties:
  - Stable - Yes
  - Adaptive – No
  - Better cache usage than Quicksort (local data moves)
- Time complexity: O(NlgN)
  - Recurrence formula
- Space complexity: O(N)
    - O(N) for merge + O(lgN) for recursion stack
- Implementation tricks
  - Use ‘infinity’ – CLRS
  - Use Bitonic sequence
- Variations
  - Use insertion sort for small N
  - Bottom-up (iterative)
  - Works for linked lists
  - External sorting – see wikipedia section “Use with tape drives”

Merge Sort – Divide and Conquer Technique

Divide and conquer	Merge sort
Divide the problem in smaller problems	Split the problem in 2 halves.
Solve these problems	Sort each half.
Combine the answers	Merge the sorted halves.

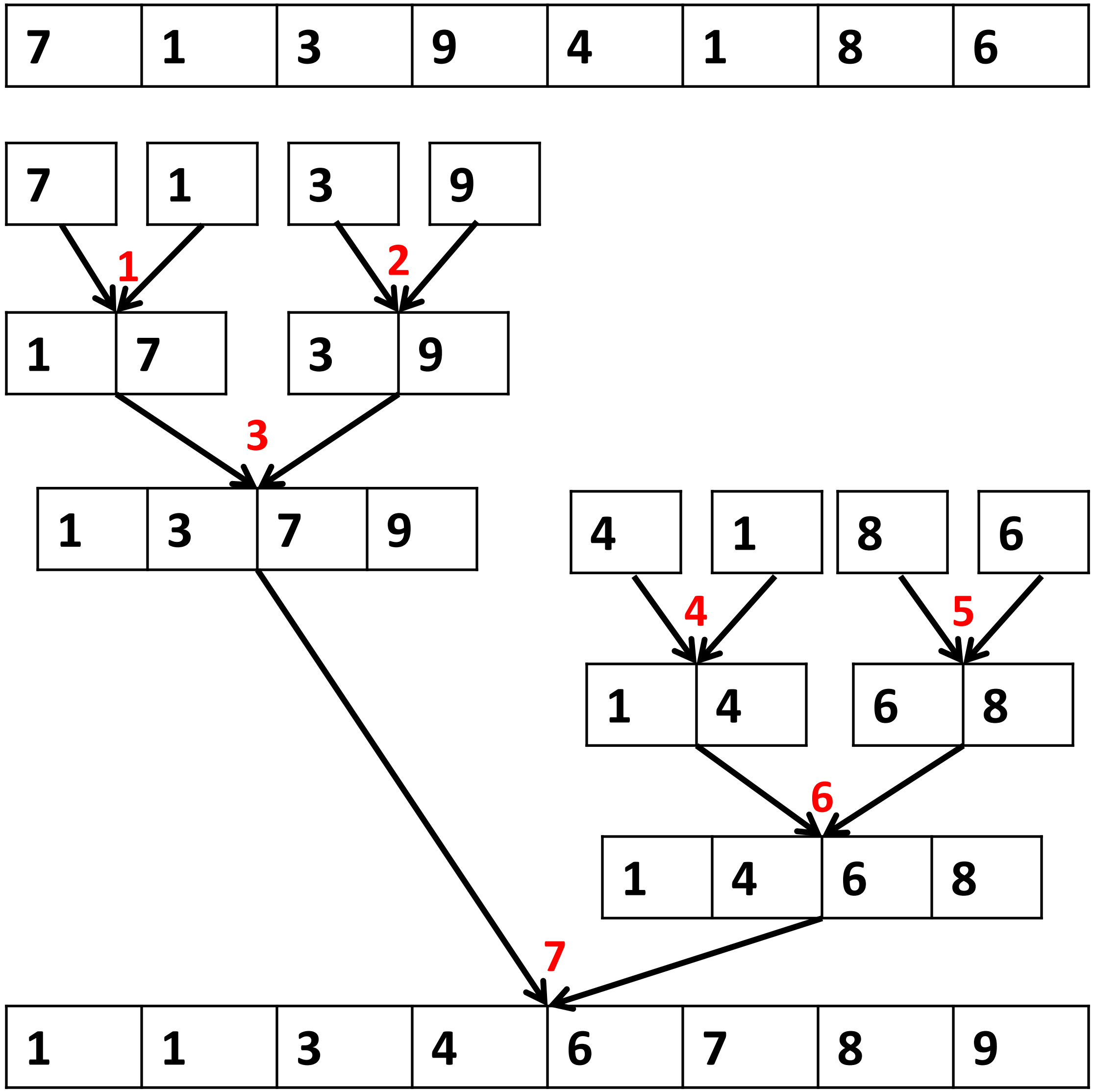
Each of the three steps will bring a contribution to the time complexity of the method.

Resources:

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- CLRS, chapter 2.3

Merging order

The actual sorting is done when merging in this order:




```
Merge_sort(A,le,ri)
   if (le>=ri) return
   else
        m = floor(le+(ri-le)/2)
        Merge_sort(A,le,m);
        Merge_sort(A,m+1,ri);
        Merge(A,le,m,ri);
```

Merge-Sort Execution

```
Merge_sort(A,le,ri) // n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

Each row shows the array after each call to the Merge function finished.

– Red items were moved by Merge.

	0	1	2	3	4	5	6	7
Original	7	1	3	9	4	1	8	6
After 1st call to Merge	**1**	**7**	3	9	4	1	8	6
After 2nd call to Merge	1	7	**3**	**9**	4	1	8	6
After 3rd call to Merge	**1**	**3**	**7**	**9**	4	1	8	6
	1	3	7	9	**1**	**4**	8	6
	1	3	7	9	1	4	**6**	**8**
	1	3	7	9	**1**	**4**	**6**	**8**
After last call to Merge	**1**	**1**	**3**	**4**	**6**	**7**	**8**	**9**

```
MS(0,7)   // m = 3
 |MS(0,3)   // m = 1
 |   |MS(0,1)   // m = 0
 |   |   MS(0,0)   //le==r, basecase
 |   |   MS(1,1)   //le==r
 |   |   Merge(0,0,1)
 |   |MS(2,3)    // m = 2
 |   |   MS(2,2)
 |   |   MS(3,3)
 |   |   Merge(2,2,3)
 |   |Merge(0,1,3)
 |MS(4,7)   // m = 5
 |   |MS(4,5)   // m = 4
 |   |   MS(4,4)
 |   |   MS(5,5)
 |   |   Merge(4,4,5)
 |   |MS(6,7)  // m = 6
 |   |   MS(6,6)
 |   |   MS(7,7)
 |   |   Merge(6,6,7)
 |   |Merge(4,5,7)
 |Merge(0,3,7)
```

Notation: *MS(le,ri)* for *Merge-Sort(A,le,ri)*

Merge (CLRS)

```
Merge(A,le,m,r)
```

```
Merge_sort(A,le,ri) //n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

7	**1**	**3**	**9**	**4**	**1**	**6**	**8**

Merge sort (CLRS)

- What part of the algorithm does the actual sorting (moves the data around)?

7	**1**	**3**	**9**	**4**	**1**	**6**	**8**

```
Merge_sort(A,le,ri)
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

Merge Sort

- Is it stable?
  - Variation that would not be stable?

- How much extra memory does it need?

- Is it adaptive?
  - Best, worst, average cases?

Merge Sort

- Is it stable? - YES
  - Variation that would not be stable?
- Is it adaptive? - NO
  - Best, worst, average cases?
- How much extra memory does it need?
  - Pay attention to the implementation!
  - Linear: **Θ(n)**
    - **Extra memory needed for the copy array in the worst case is n.**
      - **Note that the extra memory used in merge is freed up, therefore we do not have to repeatedly add it and we will NOT get Θ(nlgn) extra memory.**
    - **extra memory due to recursion: c*lg n (i.e. Θ(lg n))**
      - **There will be at most lg n open recursive calls. Each one of those needs constant memory (one stack frame)**
    - **Total extra memory: n + c*lg n = Θ(n).**

Recurrences

Given: $S(N) = S(N-1) + 3$
Fill in:

$S(?) =$

$S(N/2) =$

$S(N-10) =$

Given: $R(N) = 1R(N/5) + R(N-4) + N^2 \lg N$
Fill in:

$R(?) =$

$R(N/2) =$

$R(N-10) =$

Recurrences

$$S(N) = S(N-1) + 3 \quad \ldots\ldots = \underline{\quad\ldots N\ldots\quad}$$

$$S\left(\frac{N}{2}\right) = S\left(\frac{N}{2}-1\right) + 3 = S\left(\frac{N}{2}-1\right) + 3$$

$$S(N-10) = S(N-10-1) + 3$$

$$R(N) = 2R\left(\frac{N}{5}\right) + R(N-4) + N^2 \lg N$$

$$R(\square) = 2R\left(\frac{\square}{5}\right) + R(\square - 4) + \square^2 \lg \square$$

$$R\left(\frac{N}{2}\right) = 2R\left(\frac{\frac{N}{2}}{5}\right) + R\left(\frac{N}{2} - 4\right) + \left(\frac{N}{2}\right)^2 \lg \frac{N}{2}$$

Time complexity – Write the recurrence formula

- Let T(n) be the time complexity to sort (with merge sort) an array of n elements.
  - Assume n is a power of 2 (i.e. $n = 2^k$).
- What is the time complexity to:
  - Split the array in 2: ______
  - Sort each half (with MERGESORT): ______
  - Merge the answers together: ______

```
Merge_sort(A,le,ri) //n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

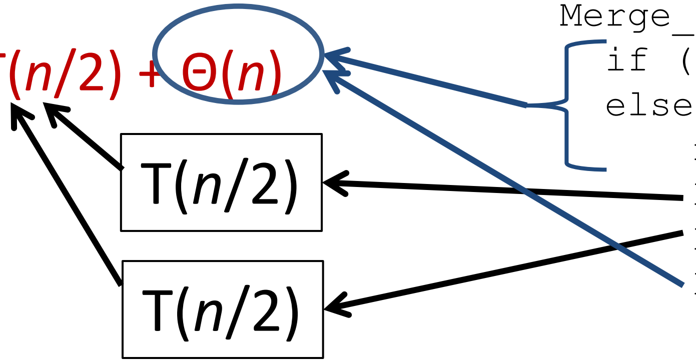
Time complexity

- Let T(n) be the time complexity to sort (with merge sort) an array of n elements.
  - Assume n is a power of 2 (i.e. $n = 2^k$).
- What is the time complexity to:
  - Split the array in 2: **c**
  - Sort each half (with MERGESORT): **T(n/2)**
  - Merge the answers together: **cn (or Θ(n))**

```
Merge_sort(A,le,ri) //n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

Merge sort (CLRS)

- Recurrence formula
  - Here *n* is the number of items being processed
  - Base case:
    - T(0) = O(1), T(1) = O(1)
    - (In the code, see for what value of n there is NO recursive call. Here when *le≥r* => *n ≤ 1*)
  - Recursive case:
    - $T(n) = 2T(n/2) + cn$
    - also ok:
    - $T(n) = 2T(n/2) + \Theta(n)$

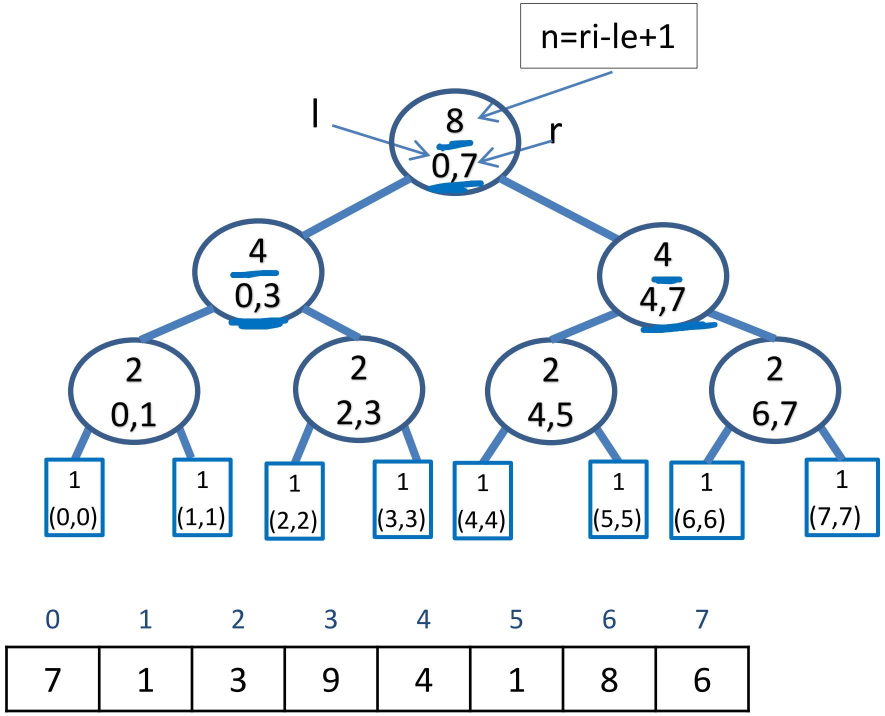



```
Merge_sort(A,le,ri) //n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

Tree of recursive calls to Merge-Sort for n = 8

```
Merge_sort(A,le,r) //N = ri-le+1
  if (le>=ri) return  // base case
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

MergeSort(A,0,7) processes the 8 elements between indexes 0 and 7 (inclusive).
The tree below shows all the recursive calls made.




```
MS(0,7)   // m = 3
 |MS(0,3)   // m = 1
 |   |MS(0,1)   // m = 0
 |   |   MS(0,0)   //le==r, basecase
 |   |   MS(1,1)   //le==r
 |   |   Merge(0,0,1)
 |   |MS(2,3)    // m = 2
 |   |   MS(2,2)
 |   |   MS(3,3)
 |   |   Merge(2,2,3)
 |   |Merge(0,1,3)
 |MS(4,7)   // m = 5
 |   |MS(4,5)   // m = 4
 |   |   MS(4,4)
 |   |   MS(5,5)
 |   |   Merge(4,4,5)
 |   |MS(6,7)  // m = 6
 |   |   MS(6,6)
 |   |   MS(7,7)
 |   |   Merge(6,6,7)
 |   |Merge(4,5,7)
 |Merge(0,3,7)
```

Recursion Tree - brief

Fill in the recursion tree for merge sort. If TC is Θ(n) use cn (show the constant).

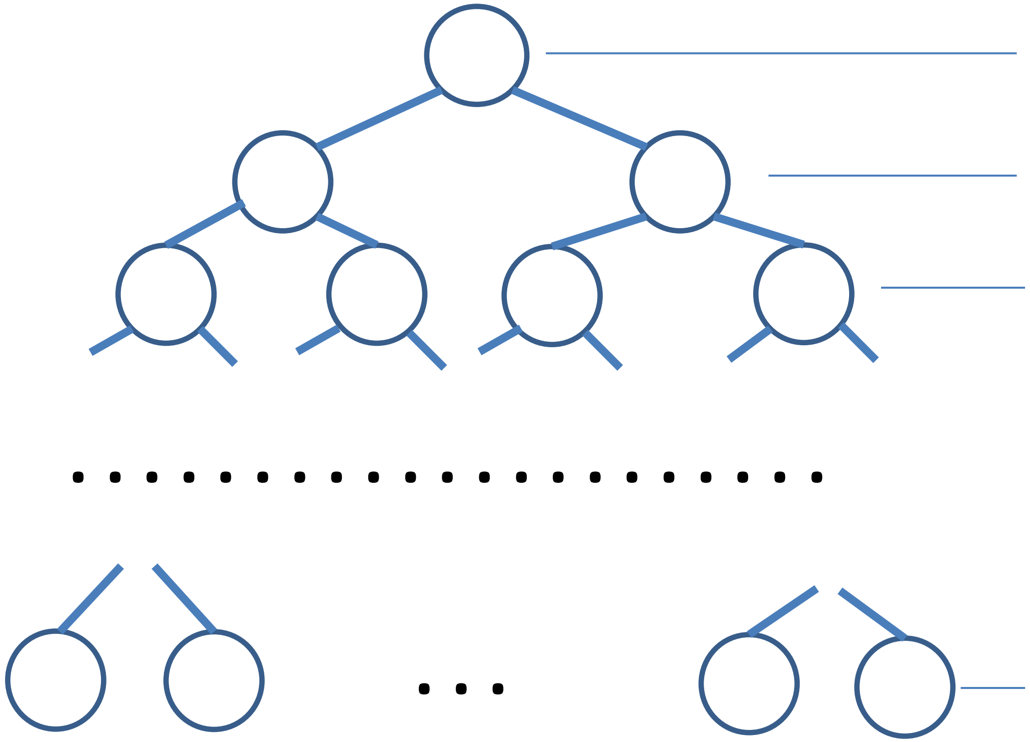
For each node, put the problem size outside, and the TC of that call, inside: $pb\ size$ TC

Assume that n is a power of 2: $n = 2^k$.

Number of levels: ___________

Cost at level ℓ: ____________

Total cost of the tree: _________________________ = Θ(_________)



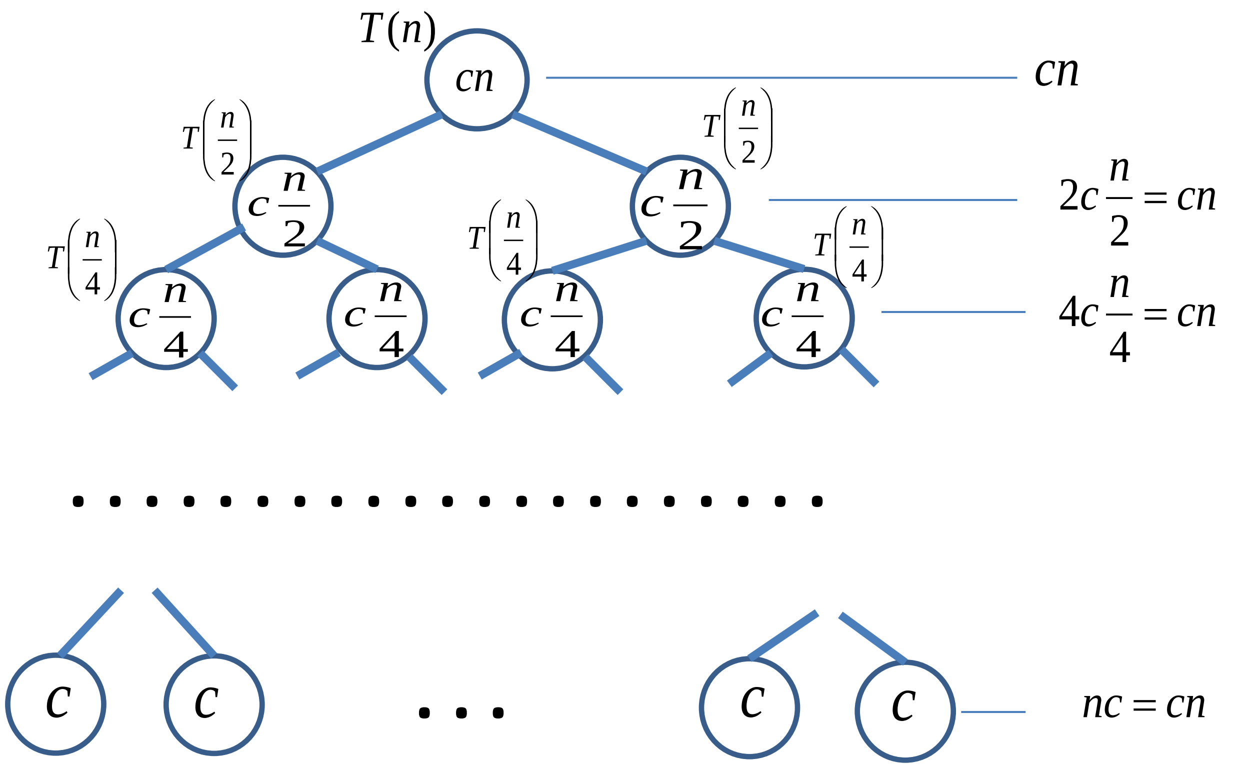
Recursion Tree - brief

Assume that n is a power of 2: $n = 2^k$.

Number of levels: lg n + 1

Each level has the same cost: cn

Total cost of the tree: (lg n + 1)(cn) = cn lg n + cn = Θ(nlg n)



Recursion Tree: T(n) = 2T(n/2) + cn

Assume that n is a power of 2: $n = 2^k$.

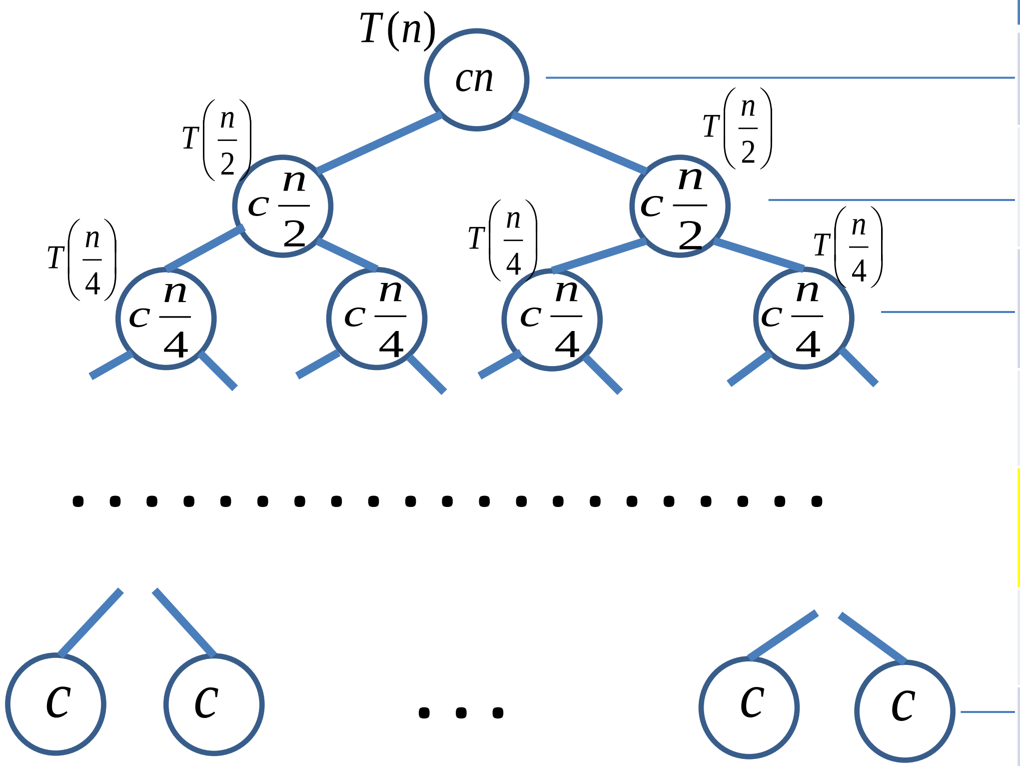
Number of levels: lg n + 1

Each level has the same TC: cn

Total TC of the tree: (1+lg n)(cn) = cn lg n + cn = Θ(nlg n)

CLRS book: see page 38

$n/2^k = 1 \Rightarrow$
$n = 2^k$
$k = \log_2 N$



Level	Arg/ pb size	Nodes per level	1 node TC	Level TC
0	n	1	cn	cn
1	n/2	2	c(n/2)	2cn/2 =cn
2	n/4	4	c(n/4)	4cn/4 =cn
…				
i	$n/2^i$	2^i	$c(n/2^i)$	$2^i cn/2^i$ =cn
…				
k=lgn	1 ($=n/2^k$)	2^k (=n)	$c=c*1=cn/2^k$	$2^k cn/2^k$ =cn

Merge sort Variations
(Sedgewick, wiki)

- *Mergesort with insertion sort for small problem sizes* (when N is smaller than a cut-off size).
  - The base case will be at say n≤10 and it will run insertion sort.
  - *replace*

```
if (le >= ri) return;
```

with

```
if (le+9 >= ri) {insertionsort(A, le, ri); return;}
```

- *Bottom-up* mergesort,
  - Iterative.
- Mergesort *using lists* and not arrays.
  - Both top-down and bottom-up implementations

1	3	7	9	8	4	1

- *Improve constant in TC (remove some operations)*
  - *Bitonic sequence* (first increasing, then decreasing) – see Sedgewick
    - Eliminates the index boundary check for the subarrays.
  - One copy/move operation instead of two – see Sedgewick
    - Alternate between regular array and the auxiliary one
    - Will copy data only once (not twice) per recursive call.
- External sorting – see wikipedia page, section “Use with tape drives”
  - Uses the bottom-up version

Merge sort bottom-up

- Notice that after each pass, subarrays of certain sizes (2, 4, 8, 16) or less are sorted.
  - Colors show the subarrays of specific sizes: 1, 2, 4, 8, 11.

7	1	***3***	9	***4***	1	***6***	8	***5***	15	***2***	Size 1
1	***7***	3	9	***1***	***4***	6	8	***5***	***15***	2	Size 2
1	***3***	***7***	***9***	1	4	6	8	***2***	***5***	***15***	Size 4
1	***1***	***3***	***4***	***6***	***7***	***8***	***9***	2	5	15	Size 8
1	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***15***	Size 16 (11)

Merge (CLRS, Chapter 2.3)

```
Merge(A,le,m,ri)
n1=m-le+1+1 // +1 for inf
n2=ri-m+1 // +1 for inf
let L[n1], R[n2] be arrays
for j=0 to n1-2 //j++
    L[j]=A[le+j]
for j=0 to n2-2 // j++
    R[j]=A[m+1+j]
L[n1] = inf
R[n2] = inf
j=0,
i=0
for k=le to ri // k++
    if L[i] ≤ R[i]
        A[k]=L[i]
        i++
    else
        A[k] = R[j]
        j++
```

```
Merge_sort(A,le,r) //N = ri-le+1
if (le>=ri) return // base case
else
   m = floor(le+(ri-le)/2)
   Merge_sort(A,le,m);
   Merge_sort(A,m+1,ri);
   Merge(A,le,m,ri);
```

7	1	3	9	4	1	5

0	1	2	3	4	5	6
7	1	3	9	4	1	5

abc0

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

Stack of fct frames

0	1	2	3	4	5	6
7	1	3	9	4	1	5

abc0

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
Merge_sort(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
Merge_sort(___,___,___);
Merge_sort(__,__,__);
Merge(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

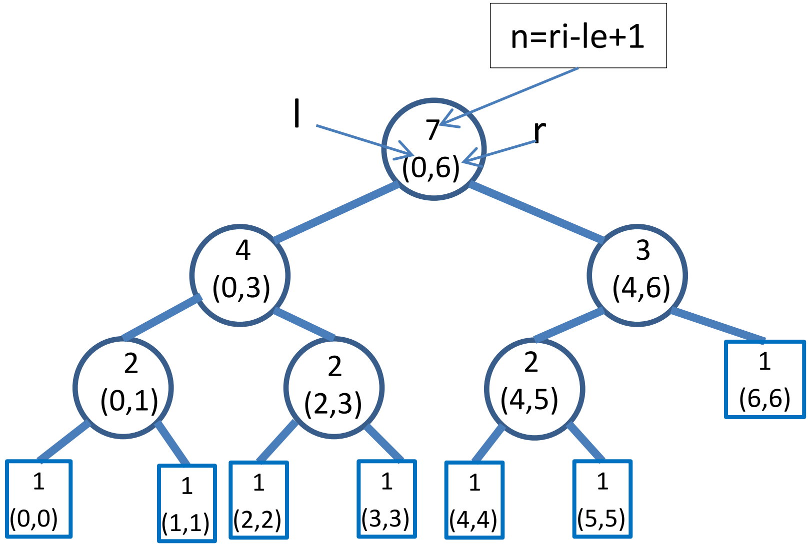
```
MS(___,___,___)//N =
if (le>=ri) return
m = floor(__+__/2)
MS(___,___,___);
MS(__,__,__);
Mrg(__,__,__,__);
```

Stack of fct frames

Extra, more examples

Tree of recursive calls to Merge-Sort for N = 7

MergeSort(A,0,6) processes the 7 elements between indexes 0 and 6 (inclusive).
The tree below shows all the recursive calls made.



0	1	2	3	4	5	6
7	1	3	9	4	1	8

```
Merge_sort(A,le,ri) //n = ri-le+1
  if (le>=ri) return
  else
      m = floor(le+(ri-le)/2)
      Merge_sort(A,le,m);
      Merge_sort(A,m+1,ri);
      Merge(A,le,m,ri);
```

```
MS(0,6)    // m = 3
 |MS(0,3)    // m = 1
 |   |MS(0,1)    // m = 0
 |   |   MS(0,0)    //le==ri, basecase
 |   |   MS(1,1)    //le==ri
 |   |   Merge(0,0,1)
 |   |MS(2,3)     // m = 2
 |   |   MS(2,2)
 |   |   MS(3,3)
 |   |   Merge(2,2,3)
 |   |Merge(0,1,3)
 |MS(4,6)    // m = 5
 |   |MS(4,5)    // m = 4
 |   |   MS(4,4)
 |   |   MS(5,5)
 |   |   Merge(4,4,5)
 |   |MS(6,6)   // q = 6
 |   |Merge(4,5,6)
 |Merge(0,3,6)
```