Merge Sort

Alexandra Stefan

Last updated: 3/27/2018

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:

- http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Merge sort

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

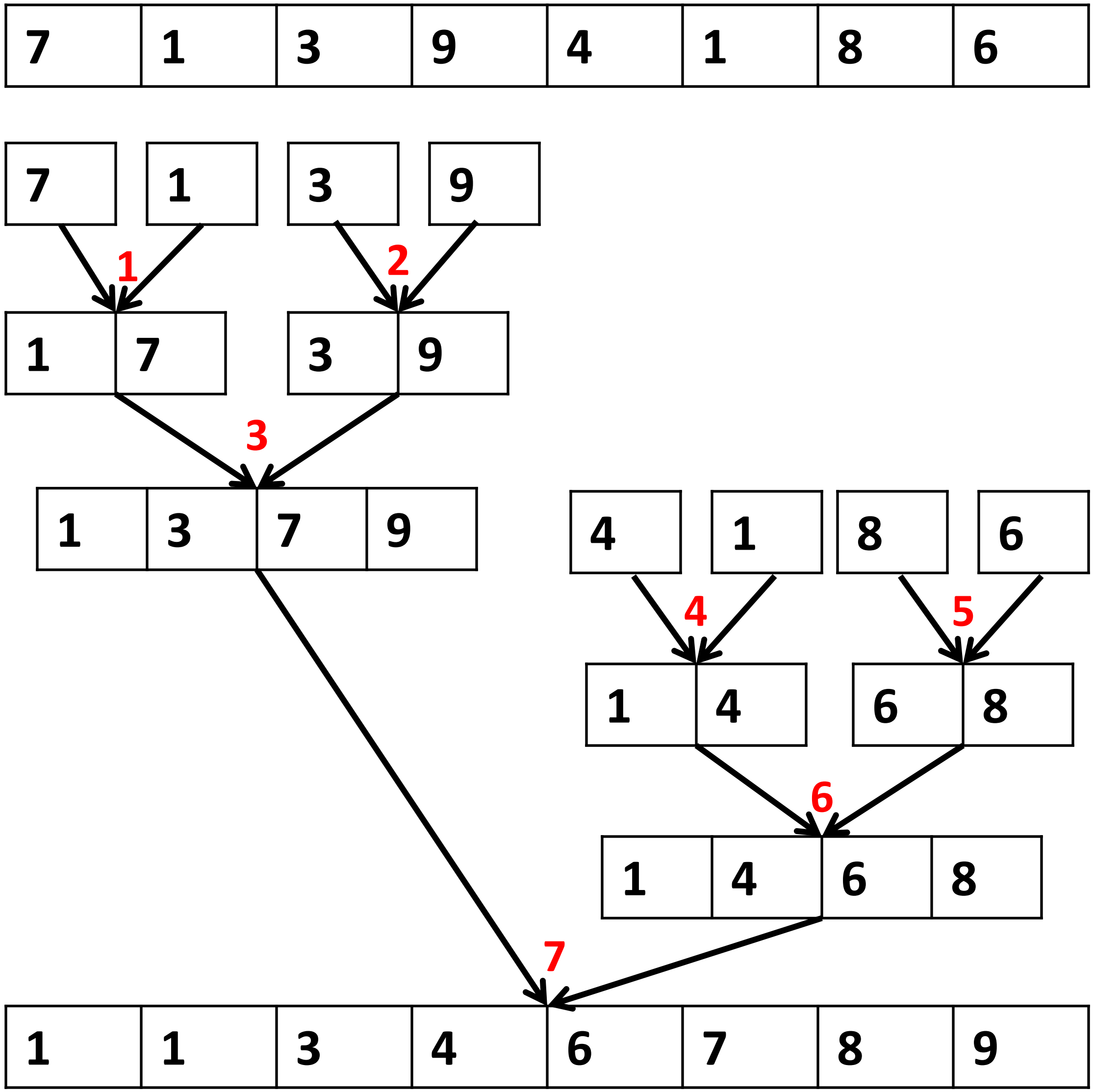
MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

p	r	q

Merge sort

The actual sorting is done when merging in this order:




```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge-Sort Execution

MERGE-SORT(A, p, r)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

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

MS(0,7) // q = 3
  MS(0,3) // q = 1
    MS(0,1) // q = 0
      MS(0,0) //p==r, basecase
      MS(1,1) //p==r
      Merge(0,0,1)
    MS(2,3) // q = 2
      MS(2,2)
      MS(3,3)
      Merge(2,2,3)
    Merge(0,1,3)
  MS(4,7) // q = 5
    MS(4,5) // q = 4
      MS(4,4)
      MS(5,5)
      Merge(4,4,5)
    MS(6,7) // q = 6
      MS(6,6)
      MS(7,7)
      Merge(6,6,7)
    Merge(4,5,7)
  Merge(0,3,7)

Notation: *MS(p,r)* for *Merge-Sort(A,p,r)*

Merge sort (CLRS)

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

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, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge sort (CLRS)

MERGE(A, p, q, r)

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$

What is the SPACE complexity for this line?
How would you implement this line?
(What C code would you write?)
Look at your code.
What is your space complexity? (keep the constant)

MERGE-SORT(A, p, r)

if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT$(A, q + 1, r)$
    MERGE(A, p, q, r)

7	1	3	9	4	1	6	8

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?
- How much extra memory does it need?
  - Pay attention to the implementation!
  - Linear: **Θ(n)**
    - **Extra memory needed for arrays L and R 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.**
    - **There will be at most lg n <u>open</u> recursive calls. Each one of those needs constant memory (one stack frame) => extra memory due to recursion: c*lg n (i.e. Θ(lg n))**
    - **Total extra memory: n + c*lg n = Θ(n).**
- Is it adaptive? - NO
  - Best, worst, average cases?

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))**
- We will see other ways to answer this question later.

Merge sort (CLRS)

- Recurrence formula
  - Here *n* is the number of items being processed
  - Base case:
    - T(1) = c
    - (In the code, see for what value of n there is NO recursive call. Here when *p<r* is false => *p≤r* => *n ≤ 1*)
  - Recursive case:
    - T(*n*) = 2T(*n*/2) + c*n*
    - also ok:
    - T(*n*) = 2T(*n*/2) + Θ(*n*)

T(*n*/2)

T(*n*/2)

MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Recursion Tree

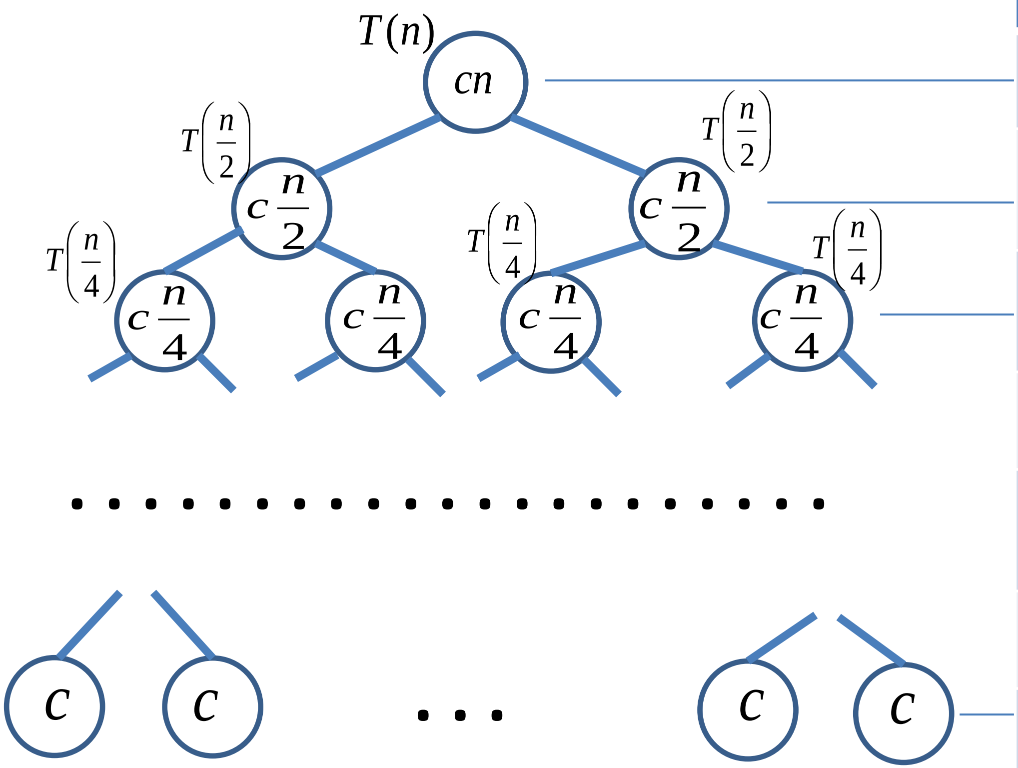
Assume that n is a power of 2: $n = 2^k$.

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 = $\Theta(n\lg n)$

CLRS book: see page 38



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

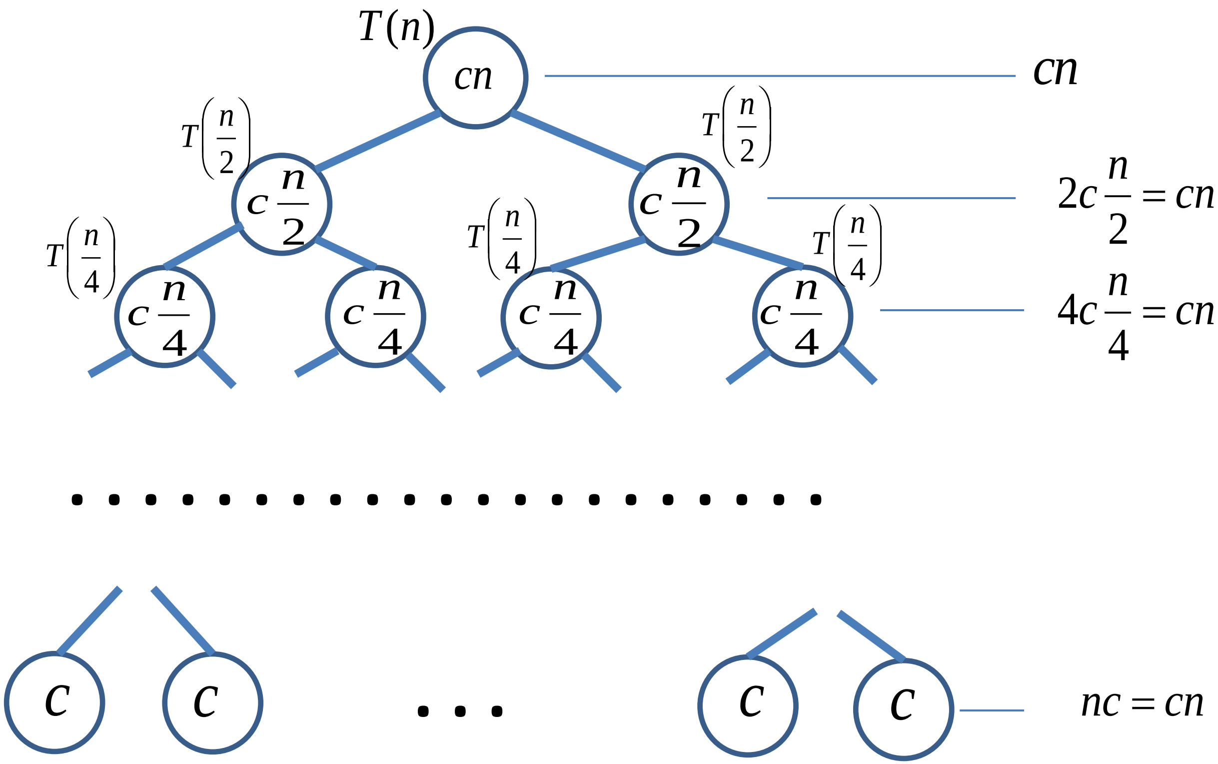
Recursion Tree - brief

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

Number of levels: lg n + 1

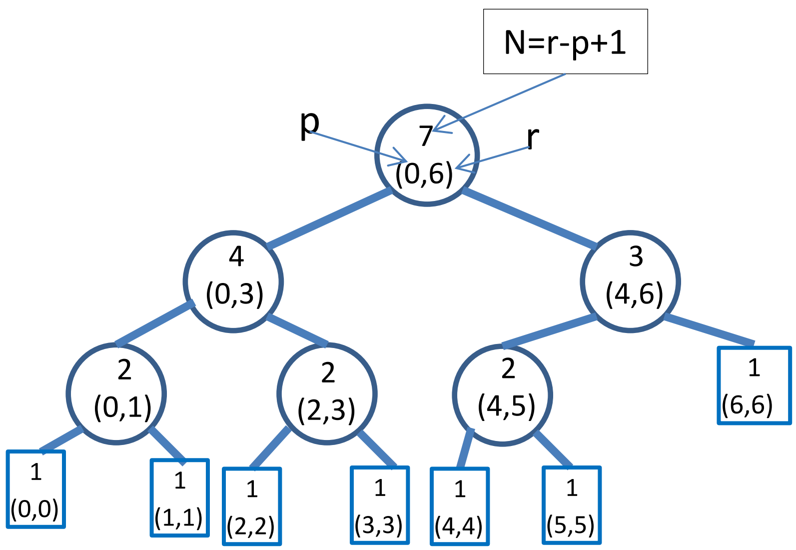
Each level has the same cost: cn

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



Tree of recursive calls to Merge-Sort

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



Mergesort Variations (Sedgewick)

- 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.
- Bottom-up mergesort,
  - Iterative.
- Mergesort using lists and not arrays.
  - Both top-down and bottom-up implementations
- Sedgewick – mergesort uses one auxiliary array (not two)
  - Alternate between regular array and the auxiliary one
    - Will copy data only once (not twice) per recursive call.
  - Constant extra space (instead of linear extra space).
    - More complicated, somewhat slower.
  - Bitonic sequence (first increasing, then decreasing)
    - Eliminates the index boundary check for the subarrays.

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)