Recall subroutine MERGE:

```
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
```

<u>Note</u>: Lines 12-17 in <u>for</u> loop have the following <u>loop invariant</u>:

At the start of each iteration of the <u>for</u> loop in lines 12-17, the subarray $A[p..k-1]$ contains $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements in their arrays that have not been copied back to A.

<u>Initialization</u>: At the start of <u>for</u> loop, $k=p \Rightarrow A[p..k-1]$ has no elements, $k-p=p-p=0$ smallest elements from L and R are in $A[p..k-1]$. At the start, $i=j=1$ and $L[1]$ and $R[1]$ are the smallest elements in L and R that have not been copied into A.

<u>Maitenance</u>: need to show that iteration preserves loop invariant.

Let $L[i] \leq R[j]$. We assume that $A[p..k-1]$ has $k-p$ smallest elements of L and R, and $L[i]$ and $R[j]$ are smallest in L and R that have not been copied to A. Then $L[i]$ goes back to A (line 14). Now we have $A[p..k]$ has $k-p+1$ smallest elements of L and R. Then both counters i and k are increased by one. Now, if instead $L[i] > R[j]$, the same argument applies but for element $R[j]$, counters j and k.

<u>Termination</u>: At termination, $k=r+1$. $A[p..k-1]$ is $A[p..r]$, and it contains $k-p = r+1-p = r-p+1$ elements from L and R, in sorted order. But arrays L and R have $n_1+1+n_2+1 = n_1+n_2+2$ elements.

$$n_1+n_2+2 = \cancel{q}-p+1+r-\cancel{q}+2 = r-p+1+2$$

Now, let's take a look at MERGE_SORT procedure that calls MERGE as subroutine.

```
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)
```

Here $\lfloor x \rfloor$ is the floor of x, i.e. the greatest integer $\leq$ than x.

$\lceil x \rceil$: the ceiling of x, i.e. the least integer $\geq$ to x.

$$x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1 \quad \text{for any real } x$$

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n \quad \text{for any integer } n$$

<u>Note</u> If $p \geq r$, then array $A[p..r]$ has only one element and the array is automatically sorted. Otherwise, one divides

A into two subarrays: $A(p, q)$ containing containing $\lfloor n/2 \rfloor$ elements and $A(q+1, r)$ containing $\lceil n/2 \rceil$ elements.

To start, we call

MERGE_SORT(A, 1, A.length)

where A.length $= n$

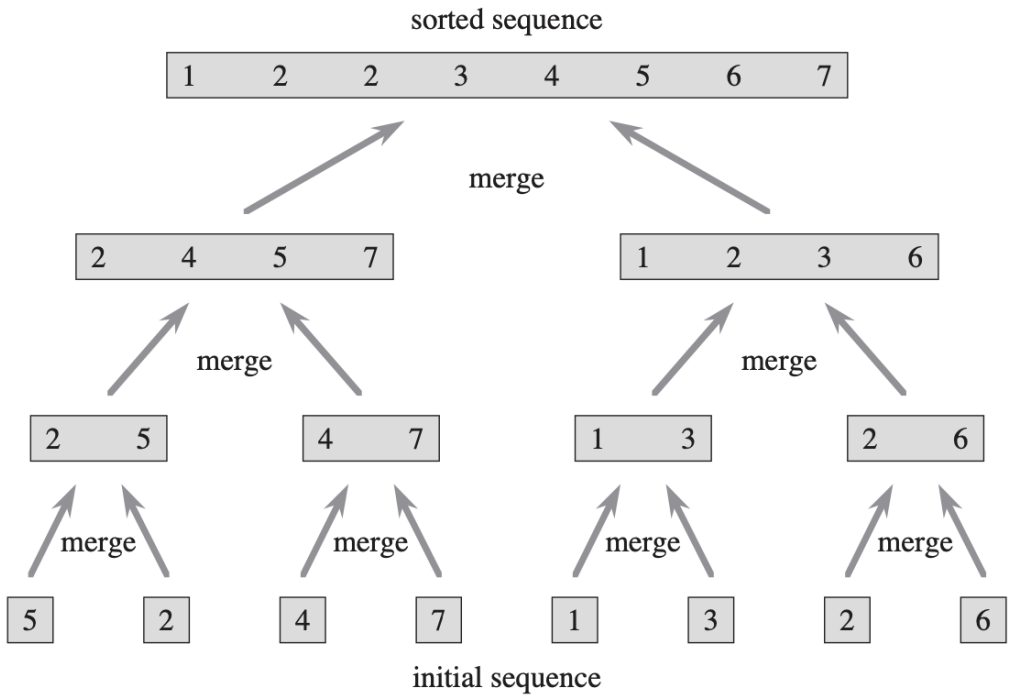


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Running time for MERGE subroutine is $\Theta(n)$, where $n = r - p + 1$. Lines 1, 2, 3, 8, 9, 10, 11 have const cost $\Rightarrow \Theta(1)$. For loops 4, 5 and 6, 7 have costs $\Theta(n_1)$ and $\Theta(n_2)$, respectively. $\Rightarrow$ loops 4, 5, 6, 7 have total cost $\Theta(n_1 + n_2) = \Theta(n)$, since $n_1 + n_2 = n$. There are n iterations in <u>for</u> loop 12–17 $\Rightarrow$ its cost is also $\Theta(n)$. Hence, total run time for MERGE subroutine is $\Theta(n)$.

Analyzing divide-and-conquer algorithms

We use recurrence equations (recurrence) when algorithms call themselves recursively. These recurrence equations relate running time of the whole problem to running time to solve subproblems. Divide-and-conquer algorithms will have three components in their recurrence equations.

Let $T(n)$: running time to solve problem of size n.

- If problem size is small, $n \leq c$, $c = \text{const}$, then we solve the problem using brute-force approach. The running time is constant time $\Rightarrow \Theta(1)$.

- Otherwise, we divide the problem into subproblems. Assume we have a subproblems of size $1/b$ ~~ata~~ of the original problem, i.e. if original ~~b~~ problem is of size n, the a subproblems will be of size n/b.

For merge sort, $a=b=2$, but in general, $a \neq b$.

- Let $D(n)$ the running time to divide into subproblems.
- If we have a ~~b~~ subproblems of size n/b, then it takes $T(n/b)$ to solve each subproblem. Hence, to solve a subproblems of size n/b, ~~the running~~ the time is $aT(n/b)$.
- Let $C(n)$ be time to combine solutions of subproblems.

We get <u>recurrence</u>:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \le c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

Analyzing merge sort

For simplicity, we assume that n is exact power of 2, i.e. $n = 2^k$.

For merge sort, the base case is when $n=1$.

For $n \geq 2$, we divide array of size n into $a=2$ subarrays of size $n/2$ and solve them recursively.

<u>Divide</u>: just compute index q that is the middle of array $A[p..r]$, i.e. q is average of p and r. $\Rightarrow D(n) = \Theta(1)$.

<u>Conquer</u>: recursively solve two problems of size $n/2 \Rightarrow 2T(n/2)$

<u>Combine</u>: the subroutine MERGE takes $\Theta(n)$ time ~~to com~~ for n elements

$\Rightarrow C(n) = \Theta(n)$

Hence,

$$T(n) = \begin{cases} \Theta(1), & \text{if } n=1 \\ 2T(n/2) + \underbrace{\Theta(1) + \Theta(n)}_{\Theta(n)}, & \text{otherwise} \end{cases}$$

$$\Rightarrow T(n) = \begin{cases} \Theta(1), & \text{if } n=1 \\ 2T(n/2) + \Theta(n), & \text{otherwise} \end{cases}$$

<u>Solving a recurrence for merge-sort</u>

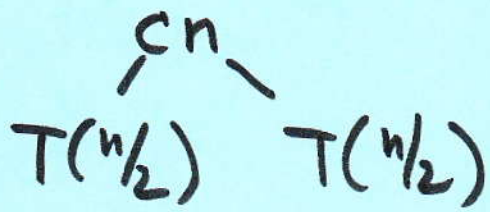
We will show that $T(n) = \Theta(n \lg n)$ where $\lg n = \log_2 n$. The insertion sort has $T(n) = \Theta(n^2)$. Then merge sort would work faster than insertion for large n.

For small size arrays, the insertion sort will work faster, so insertion sort can be combined with merge sort when subproblem is of some size k (can be estimated). When $n=k$, instead of using merge recursively, one can apply insertion instead to achive faster algorithm

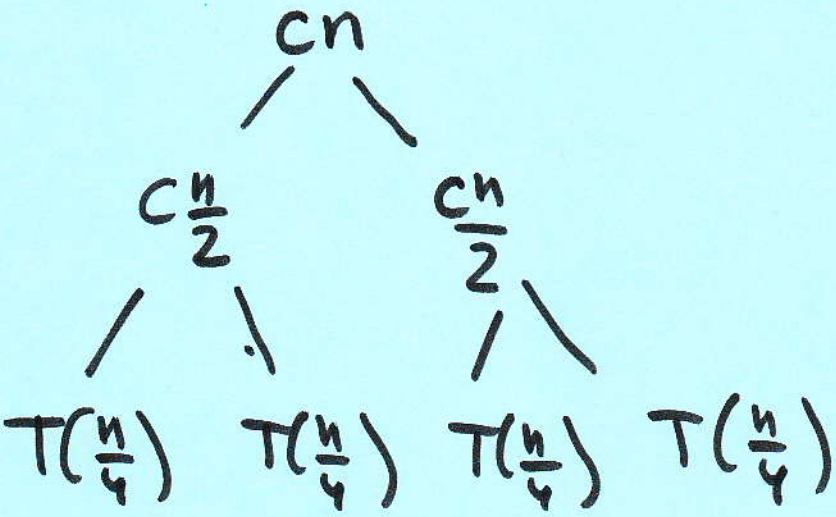
Other simplifications.

$$T(n) = \begin{cases} c, & \text{if } n=1 \\ 2T(\frac{n}{2}) + cn, & \text{otherwise} \end{cases}$$

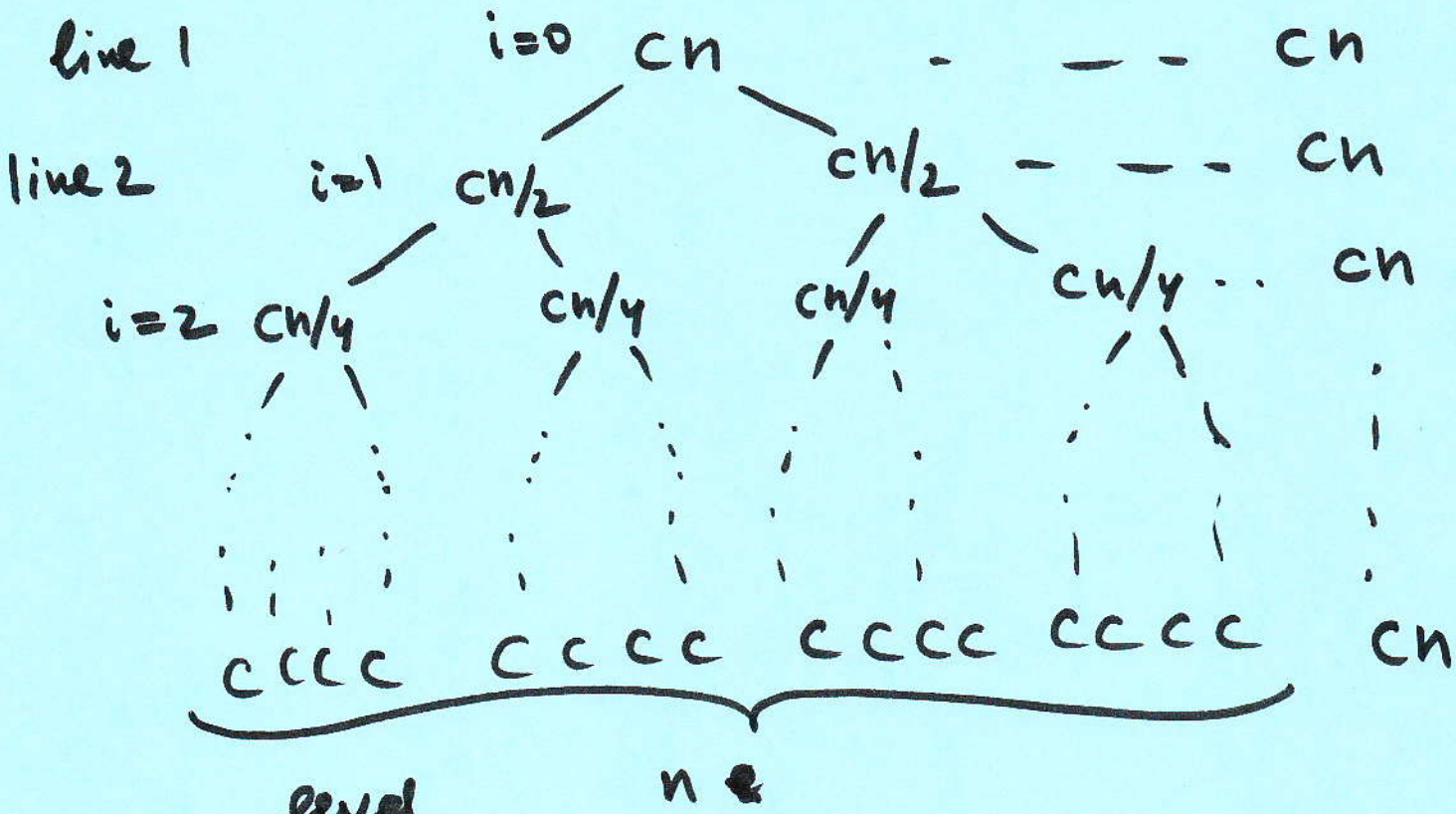
Let's draw the recurrence tree.



For each subproblem of size $n/2$, we will have the cost in combine step and cost to solve two subproblems of size $n/4$



Continue until we reach array of length 1



Every ~~line~~ level has total cost: cn.

For example, in line 2: $cn/2 + cn/2 = cn$

etc.

In the last line we have n subarrays of length 1 with cost c per subarray $\Rightarrow$

$\Rightarrow$ total $= cn$

For array of size n, there are $\lg n + 1$ levels. We will show this by induction. Here $\lg n = \log_2 n$.

Base: $n = 1$, $\lg n = 0 \Rightarrow \cancel{\lg 1}^{0} + 1 = 1$ level

Assume we have $n = 2^i$.

with value i, level has 2^i leaves (elements in the level) with cost $c/2^i$ per each.

Consider level with $n = 2^i$ leaves.
By induction assumption, it has $\lg 2^i + 1$ levels, but $\lg 2^i + 1 = i + 1$ levels

~~Nex~~ induction step, $n = 2^{i+1} = 2 \cdot 2^i$ since n is exact power of 2. Therefore, there will one extra level compared to 2^i, i.e.

$$(i+1) + 1 = \lg \underbrace{2^{i+1}} + 1 = \lg n + 1 \text{ with } n = 2^{i+1}$$

$\Rightarrow$ we have $\lg n + 1$ levels. Each level has cost $cn \Rightarrow$ Total time is $T(n) = cn(\lg n + 1)$
$= \Theta(n \lg n)$