More on Merge sort.

Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p..r]$. Initially, $p=1$, $r=n$, but these values change as we recurse through subproblems.

To sort $A[p..r]$:

Divide by splitting into two arrays $A[p..q]$ and $A[q+1..r]$, where q is the halfway point of $A[p..r]$.

Conquer by recursively sorting the two subarrays $A[p..q]$ and $A[q+1..r]$.

Combine by merging the two sorted subarrays $A[p..q]$ and $A[q+1..r]$ to

produce a single sorted subarray $A[p..r]$. To accomplish this step, we use a procedure MERGE(A, p, q, r).

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

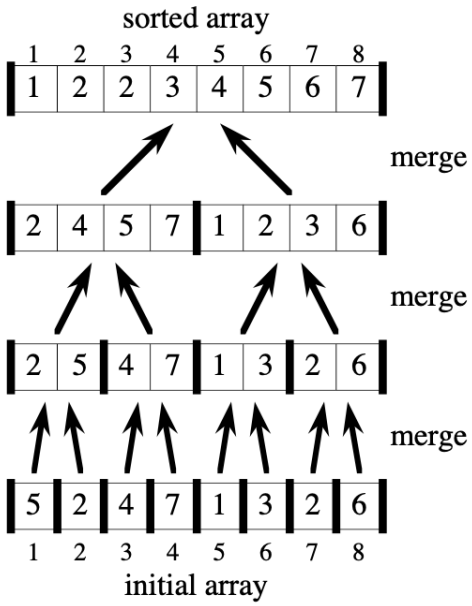
```
MERGE-SORT(A, p, r)
  if p < r                          // check for base case
      q = ⌊(p + r)/2⌋               // divide
      MERGE-SORT(A, p, q)           // conquer
      MERGE-SORT(A, q + 1, r)       // conquer
      MERGE(A, p, q, r)             // combine
```

$q = \lfloor (p+r)/2 \rfloor$: floor of $\frac{p+r}{2}$

Ex $p=1$, $r=8$ $\Rightarrow q = \lfloor \frac{1+8}{2} \rfloor = \lfloor \frac{9}{2} \rfloor = 4$

$A = [5, 2, 4, 7, 1, 3, 2, 6]$: original array

$A = [1, 2, 2, 3, 4, 5, 6, 7]$: sorted array



$q = \lfloor \frac{1+8}{2} \rfloor = \lfloor \frac{9}{2} \rfloor = 4$

$q = \lfloor \frac{1+4}{2} \rfloor = 2, \quad q = \lfloor \frac{5+8}{2} \rfloor = 6$

$q = \lfloor \frac{1+2}{2} \rfloor = 1, \quad q = \lfloor \frac{3+4}{2} \rfloor = 3$

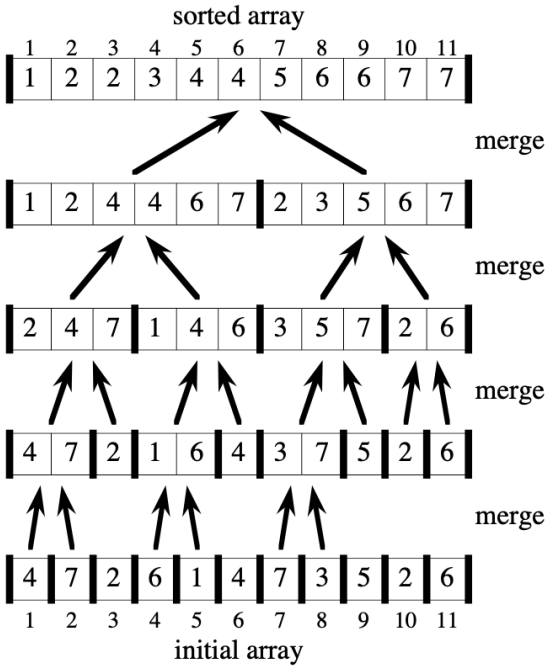
etc.

Ex $p=1,\ r=11 \Rightarrow q=\left\lfloor \frac{1+11}{2} \right\rfloor = \left\lfloor \frac{12}{2} \right\rfloor = 6$

$A=[4,7,2,6,1,4,7,3,5,2,6]$

$\Downarrow$

$A=[1,2,2,3,4,4,5,6,6,7,7]$



$q=\left\lfloor \frac{1+11}{2} \right\rfloor = 6$

$q=\left\lfloor \frac{1+6}{2} \right\rfloor = 3, \quad q=\left\lfloor \frac{7+11}{2} \right\rfloor = 9$

$q=\left\lfloor \frac{1+3}{2} \right\rfloor = 2, \quad q=\left\lfloor \frac{7+10}{2} \right\rfloor = 8$

More on Merging

Let's discuss the MERGE procedure.

Input: Array A and indices p, q, r such that

- $p \le q < r$
- subarray $A[p..q]$ is sorted and subarray $A[q+1..r]$ is sorted. By the restrictions on p, q, r, neither subarray is empty.

Output: The two subarrays are merged into a single subarray in $A[p..r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$ = the number of elements being merged.

What is n? Originally, we think of n as of the size of the problem. But now we're using it as the size of a

subproblem. We will use this technique when we analyze recursive algorithms. Although we may denote the original problem size by n, in general n will be the size of a given subproblem.

Idea behind linear-time merging

Think of two piles of cards.

- Each pile is sorted and placed face-up on a table with the smallest cards on top.
- We will merge these cards into a single sorted pile, face-down on the table.
- A basic step:
  - Choose the smaller of the two cards
  - Remove it from its pile, thereby exposing a new top card.
  - Place the chosen card face-down onto the output pile.

- Repeatedly perform basic steps until one input pile is empty.
- Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.
- Each basic step should take constant time, since we just check the two top cards.
- There are $\leq n$ basic steps, since each basic step removes one card from the input piles, and we start with n cards in the input piles.

We don't actually need to check whether a pile is empty before each basic step.

- Put on the bottom of each input file a special **sentinel** card.
- It contains a special value that we use to simplify the code.

- We use ∞, since that's guaranteed to "lose" to any other value.
- The only way that ∞ cannot lose is when both piles have ∞ exposed as their top cards.
- But when that happens, all the nonsentinel cards have already been placed into the output pile.
- We know in advance that there are exactly $r-p+1$ nonsentinel cards $\Rightarrow$ stop once we performed $r-p+1$ basic steps. Never a need to check for sentinels, since they'll always lose.
- Rather than even counting basic steps, just fill up the output array from index p up through and including index r.

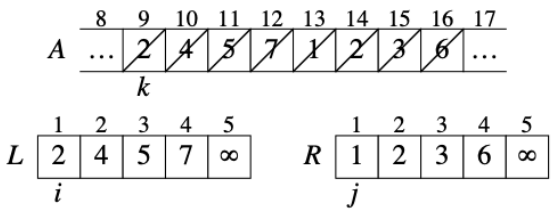
Pseudocode

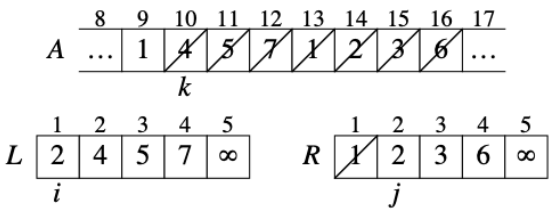
```
MERGE(A, p, q, r)
  n_1 = q − p + 1
  n_2 = r − q
  let L[1 .. n_1 + 1] and R[1 .. n_2 + 1] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i − 1]
  for j = 1 to n_2
      R[j] = A[q + j]
  L[n_1 + 1] = ∞
  R[n_2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

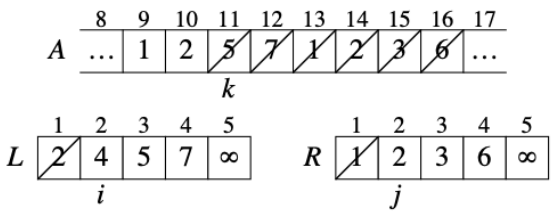
We used a loop invariant to show that MERGE works correctly. Now let's also look at an example to demonstrate that the procedure works correctly.

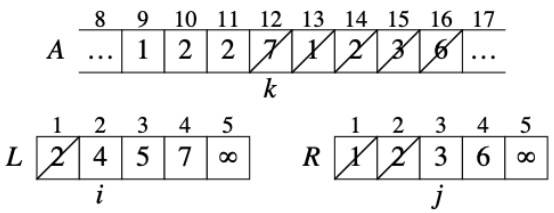
Ex

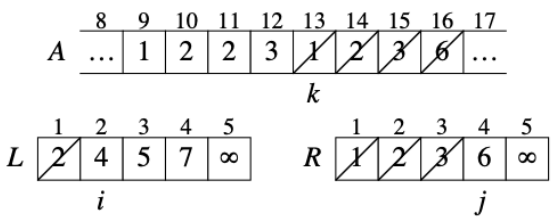
a call of MERGE(9, 12, 16)

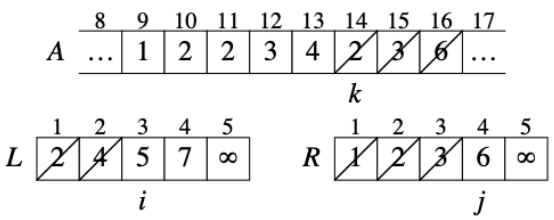

8 9 10 11 12 13 14 15 16 17
A … 2 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

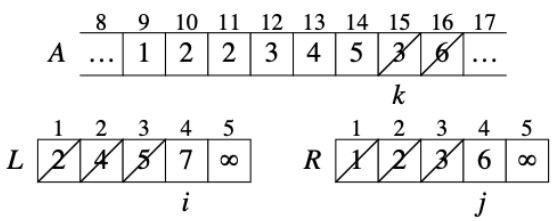

8 9 10 11 12 13 14 15 16 17
A … 1 4 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

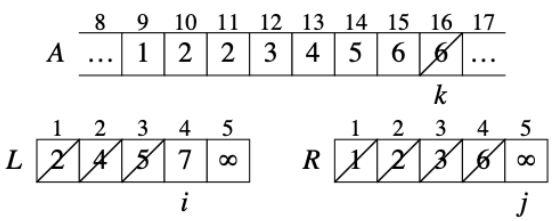

8 9 10 11 12 13 14 15 16 17
A … 1 2 5 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

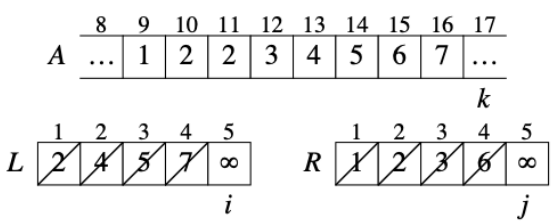

8 9 10 11 12 13 14 15 16 17
A … 1 2 2 7 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

Analyzing divide-and-conquer algorithms

Use a *recurrence equation* (more commonly, a *recurrence*) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n.

- If problem size is small enough (say, $n \le c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let time to divide a size-n problem be $D(n)$.
- Have a subproblems to solve, each of size n/b $\Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time to solve a subproblems.

- Let the time to combine solutions be $C(n)$.
- We get the recurrence

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

Analyzing merge sort

For simplicity, assume that n is a power of 2
$\Rightarrow$ each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \ge 2$, time for merge sort steps:

Divide: Just compute q as the average of p and r $\Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2$ $\Rightarrow 2\,T(n/2)$.

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

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

Solving the merge-sort recurrence

By the master Theorem in Chapter 4 (Thm 4.1 more-later), we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$.

Note: $\lg n = \log_2 n$.

Compared to insertion sort ($\Theta(n^2)$ worst-case time), merge sort is faster. Trading a factor of n for a factor of $\lg n$ is a good deal.

On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time

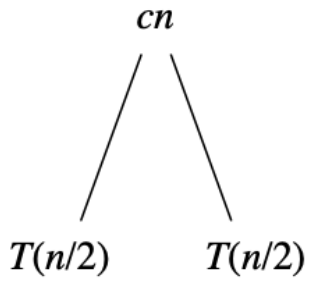
grows more slowly than insertion sort's.

We can understand how to solve the merge-sort recurrence without the master theorem.

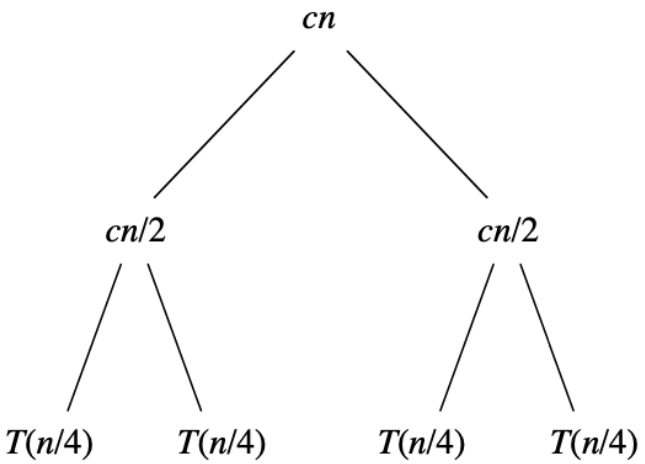
- Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps.
- We rewrite the recurrence as

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

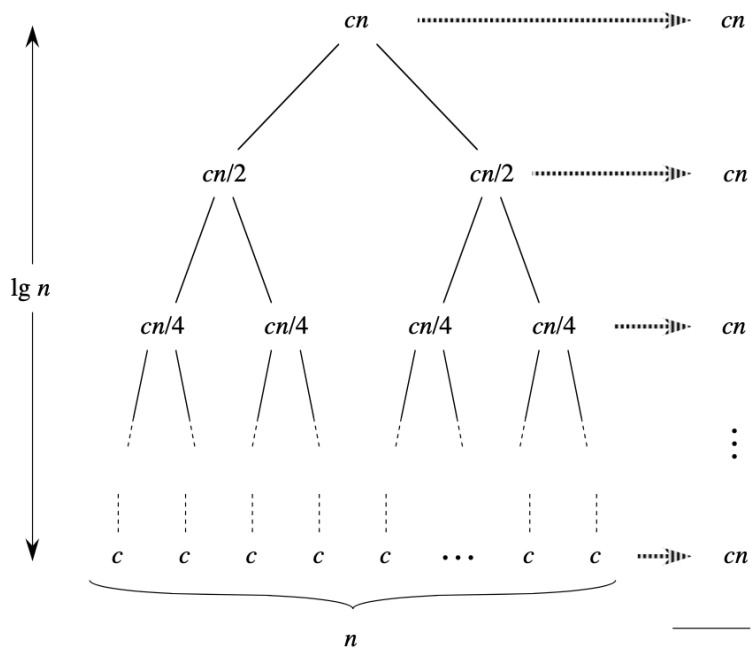
- Draw a recursion tree, which shows successive expansion of the recurrence.
- For the original problem, we have a cost of cn, plus two subproblems, each costing $T(n/2)$:



- For each of the size $n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



- Continue expanding until the problem sizes get down to 1:



Total: $cn \lg n + cn$

- Each level has cost cn.
  - * The top level has cost cn.
  - * The next level down has 2 subproblems, each contributing cost $cn/2$.
  - * The next level has 4 subproblems, each contributing cost $cn/4$.
  - * Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves
    $\Rightarrow$ cost per level stays the same.
- There are $\lg n + 1$ levels (height is $\lg n$).
  - * Use induction.
  - * Base case: $n = 1 \Rightarrow$ 1 level, and $\lg 1 + 1 = 0 + 1 = 1$.

* Inductive hypothesis is that a tree for a problem size 2^i has $\lg 2^i + 1 = i + 1$ levels.

* Because we assume that the problem size is a power of 2, the next problem size up is after 2^i is 2^{i+1}.

* A tree for a problem size of 2^{i+1} has one more level than the size-2^i tree $\Rightarrow$ $i+2$ levels

* Since $\lg 2^{i+1} + 1 = i + 2$, we are done with inductive argument.

- Total cost is sum of costs at each level. Have $\lg n + 1$ levels, each costing cn $\Rightarrow$ total cost is $cn \lg n + cn$.

- Ignore low-order term of cn and count coefficient $c \Rightarrow \Theta(n \lg n)$.