Lecture 2

Analyzing and Designing Algorithms

Slides are based on the textbook and its notes

Overview

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of “divide and conquer” in the context of merge sort.

The sorting problem

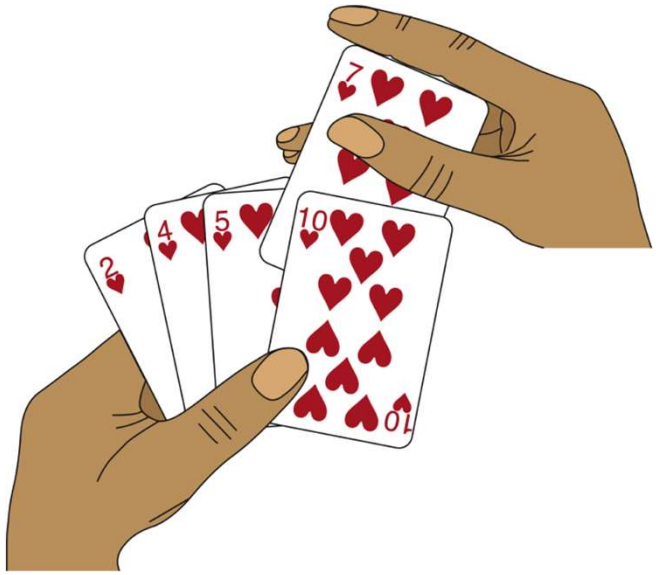
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- The sequences are typically stored in arrays.
- We also refer to the numbers as ***keys***. Along with each key may be additional information, known as ***satellite data***.
- We will see several ways to solve the sorting problem. Each way will be expressed as an ***algorithm***: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Insertion sort

- A good algorithm for sorting a small number of elements. It works the way you might sort a hand of playing cards:
  - Start with an empty left hand and the cards face down on the table.
  - Then remove one card at a time from the table, and insert it into the correct position in the left hand.
  - To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.



  - At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

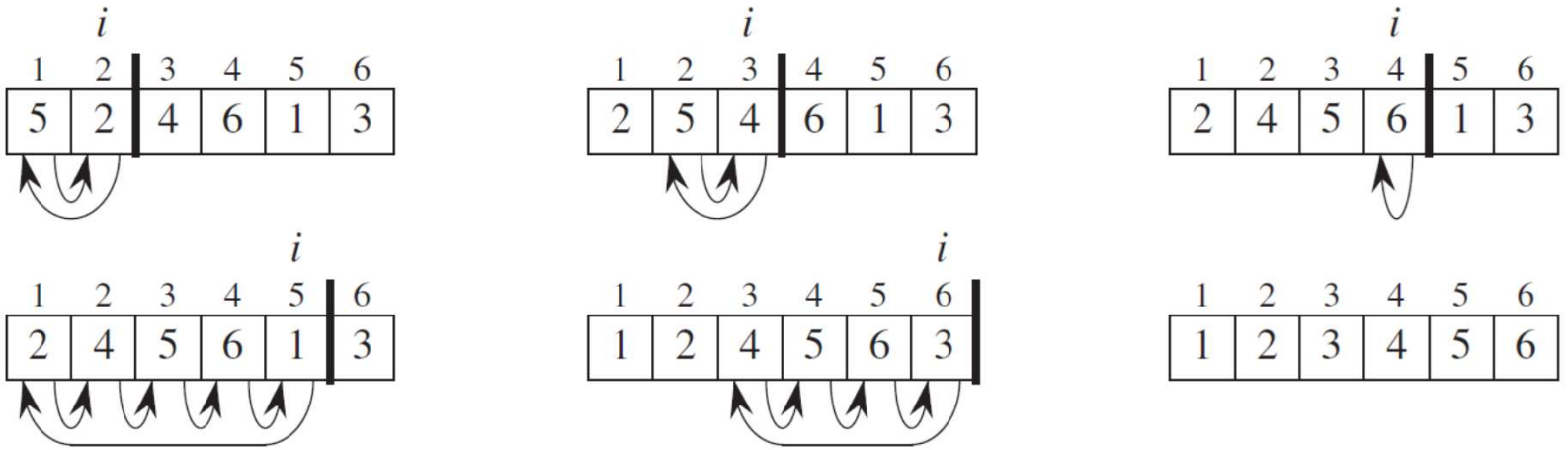
Insertion sort pseudocode

- We use a procedure INSERTION-SORT.
  - Takes as parameters an array $A[1 : n]$ and the length n of the array.
  - We use ":" to denote a range or subarray within an array. The notation $A[i : j]$ denotes the $j - i + 1$ array elements $A[i]$ through and including $A[j]$.
  - The array A is sorted ***in place***: the numbers are rearranged within the array, with at most a constant number outside the array at any time.
- Remarks:
  - Note that the meaning of our subarray notation differs from its meaning in Python. In Python, $A[i : j]$ denotes the $j - i$ array elements $A[i]$ through $A[j - 1]$, but does not include $A[j]$.
  - We usually use 1-origin indexing, as we do here. If you use 0-origin indexing, you need to be careful to get the indices right. You can either always subtract 1 from each index or allocate each array with one extra position and just ignore position 0.

Insertion sort pseudocode

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Example



The correctness of insertion sort

- We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer.

- Here is the loop invariant for INSERTION-SORT:
- **Loop invariant**: At the start of each iteration of the "outer" **for** loop – the loop indexed by i – the subarray $A[1 : i - 1]$ consists of the elements originally in $A[1 : i - 1]$ but in sorted order.

Three things of a loop invariant

- To use a loop invariant to prove correctness, we must show three things about it:
  - **Initialization**: It is true prior to the first iteration of the loop.
  - **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
  - **Termination**: The loop terminates, and when it does, the invariant – usually along with the reason that the loop terminated – gives us a useful property that helps show that the algorithms is correct.

Steps for using loop invariants

- A loop-invariant proof is a form of mathematical induction:
  - To prove that a property holds, you prove a base case and an inductive step.
  - Showing that the invariant holds before the first iteration is like the base case (Prove $P(1)$ is true).
  - Showing that the invariant holds from iteration to iteration is like the inductive step (Assume $P(k)$ is true for some $n = k$).
  - The termination part differs from the usual use of mathematical induction. Typically, you use the loop invariant along with the condition that caused the loop to terminate. Mathematical induction applies the inductive step infinitely (Prove $P(k + 1)$ is true), but in a loop invariant the "induction" stops when the loop terminates.

The proof of the correctness of insertion sort

- **Initialization**: Just before the first iteration, $i = 2$. The subarray $A[1 : i - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.
- **Maintenance**: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[i - 1]$, $A[i - 2]$, $A[i - 3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[i]$) is found. At that point, the value of *key* is placed into this position. The subarray $A[1 : i]$ then consists of the elements originally in $A[1 : i]$, but in sorted order.
- **Termination**: The outer **for** loop starts with $i = 2$. Each iteration increases i by 1. The loop ends when $i > n$, which occurs when $i = n + 1$. Therefore, the loop terminates and $i - 1 = n$ at that time. Pluggin n in for $i - 1$ in the loop invariant, the subarray $A[1 : n]$ consists of the elements originally in but in sorted order. In other words, the entire array is sorted.

Analyzing algorithms

- We want to predict the resources that the algorithm requires. Usually, running time.
- Why not just code up the algorithm, run the code, and time it?
- Because that would tell you how long the code takes to run
  - on your particular computer,
  - on that particular input,
  - with your particular compiler or interpreter,
  - using your particular libraries linked in,
  - with the particular background tasks running at the time.
- You would not be able to predict how long the code would take on a different computer, with a different input, if implemented in a different programming language, etc.. Instead, devise a formula that characterizes the running time.

Random-access machine (RAM) model

- In order to predict recourse requirements, we need a computational model.
  - Instructions are executed one after another. No concurrent operations.
  - It is too tedious to define each of the instructions and their associated time costs.
  - Instead, we recognize that we will use instructions commonly found in real computers:
    - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling. Also, shift left/shift right (good for multiplying/dividing by 2^k).
    - Data movement: load, store, copy.
    - Control: conditional/unconditional branch, subroutine call and return.
- Each of these instructions takes a constant amount of time. Ignore memory hierarchy (cache and virtual memory).

Random-access machine (RAM) model

- The RAM (random-access machine) model uses integer and floating-point types.
  - We do not worry about precision, although it is crucial in certain numerical applications.
  - There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$).
    - $c \geq 1$ => we can hold the value of n => we can index the individual elements.
    - c is a constant => the word size cannot grow arbitrarily.

How do we analyze an algorithm's running time?

- The time taken by an algorithm depends on the input.
  - Sorting 1,000 numbers takes longer than sorting 3 numbers.
  - A given sorting algorithm may even take differing amounts of time on two inputs of the same size. For example, we will see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.
- **Input size**: Depends on the problem being studied.
  - Usually, the number of items in the input. Like the size n of the array being sorted.
  - But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
  - Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Running time

- On a particular input, it is the number of primitive operations (steps) executed.
  - Want to define steps to be machine-independent.
  - Figure that each line of pseudocode requires a constant amount of time.
  - One line may take a different amount of time than another, but each execution of line k takes the same amount of time c_k.
  - This is assuming that the line consists only of primitive operations.
    - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
    - If the line specifies operations other than primitive ones, then it might take more than constant time.

Analysis of insertion sort

	INSERTION-SORT(A, n)	*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8	$A[j+1] = key$	c_8	$n-1$

- Assume that the kth line takes time c_k, which is a constant. (Since the third line is a comment, it takes no time.)
- For $i = 2, 3, \ldots, n$, let t_i be the number of times that the **while** loop test is executed for that value of i.
- Note that when a **for** or **while** loop exits in the usual way – due to the test in the loop header – the test is executed one time more than the loop body.

Analysis of insertion sort

- The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

- Let $T(n)$ = running time of INSERTION-SORT, then

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1)$$
$$+ c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

- The running time depends on the values of t_i. These vary according to the input.

Analysis of insertion sort: Best case

- The array is already sorted.
  - Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = i - 1$).
  - All t_i are 1.
  - The running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

  - Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_k) => $T(n)$ is a *linear function* of n.

Analysis of insertion sort: Worst case

- The array is in reverse sorted order.
  - Always find that $A[i] > key$ in the **while** loop test.
  - Have to compare *key* with all elements to the left of the ith position => compare with $i - 1$ elements.
  - Since the **while** loop exits because i reaches 0, there is one additional test after the $i - 1$ tests => $t_i = i$.

$$\sum_{i=2}^{n} t_i = \sum_{i=2}^{n} i \text{ and } \sum_{i=2}^{n} (t_i - 1) = \sum_{i=2}^{n} (i - 1).$$

$\sum_{i=1}^{n} i$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\frac{n(n+1)}{2}$.

Since $\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$.

Analysis of insertion sort: Worst case

Letting $l = i - 1$, we see that $\sum_{i=2}^{n}(i-1) = \sum_{l=1}^{n-1} l = \frac{n(n-1)}{2}$.

Running time is

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on the statement costs c_k) => $T(n)$ is a *quadratic function* of n.

Worst-case and average-case analysis

- We usually concentrate on finding the worst-case running time: the longest running time for *any* input of size n.
- Reasons
  - The worst-case running time gives a guaranteed upper bound on the running time for any input.
  - For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present and searches for absent items may be frequent.
  - Why not analyze the average case? Because it is often about as bad as the worst case.
  - ***Example***: Suppose that we randomly choose n numbers as the input to insertion sort. On average, the key in $A[i]$ is less than half the elements in $A[1 : i - 1]$ and it is greater than the other half. => On average, the while loop has to look halfway through the sorted subarray $A[1 : i - 1]$ to decide where to drop key. => $t_i \approx i/2$. The average-case running time is still a quadratic function of n.

Order of growth

- Another abstraction to ease analysis and focus on the important features. Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.

Example: For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.

- Drop lower-order terms => an^2.
- Ignore constant coefficient => n^2.
- But we cannot say that the worst-case running time $T(n)$ *equals* n^2.
- It *grows like* n^2. But it does not *equal* n^2.
- We say that the running time is $\Theta(n^2)$ to capture the notion that the *order of growth* is n^2.
- We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Designing algorithms

- There are many ways to design algorithms.
- For example, insertion sort is ***incremental***: having sorted $A[1 : i-1]$, place $A[i]$ correctly, so that $A[1 : i]$ is sorted.

- **Divide and conquer**: Another common approach.
- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
  - **Base case**: If the subproblems are small enough, just solve them by brute force.
- **Combine** the subproblem solutions to give a solution to the original problem.

Merge sort

- A sorting algorithm based on divide and conquer.

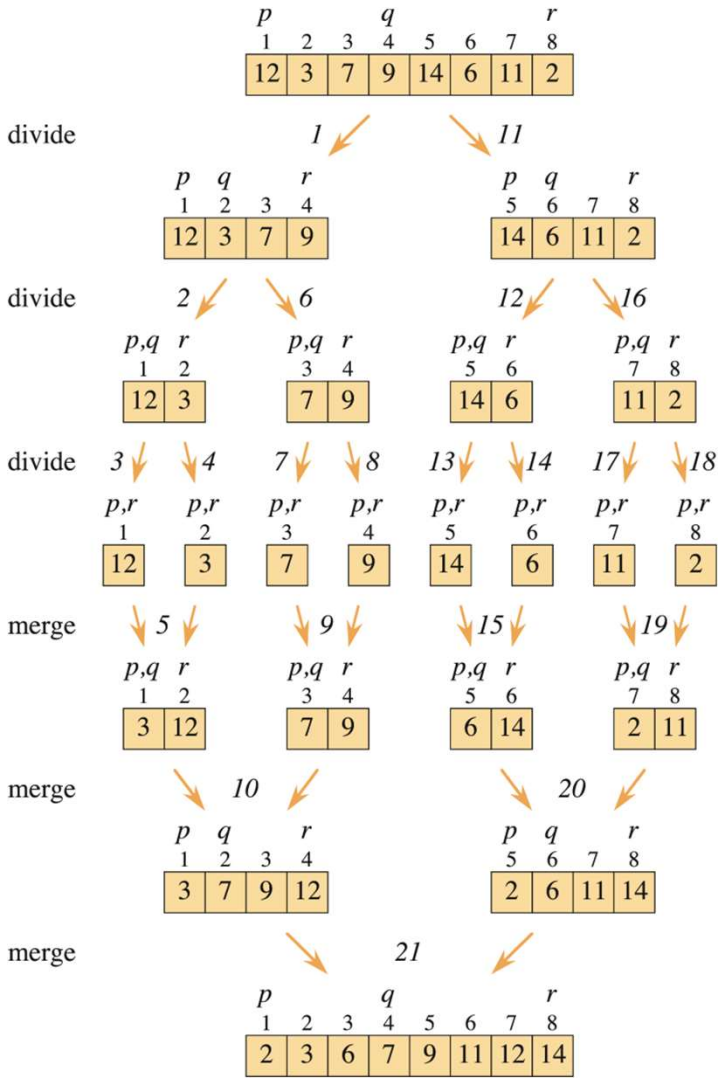
- Because we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p : r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

- To sort $A[p : r]$:
  - **Divide** by splitting into two subarrays $A[p : q]$ 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 will define a procedure MERGE(A, p, q, r).
  - The recursion bottoms out when the subarray has just 1 element, so that it is trivially sorted.

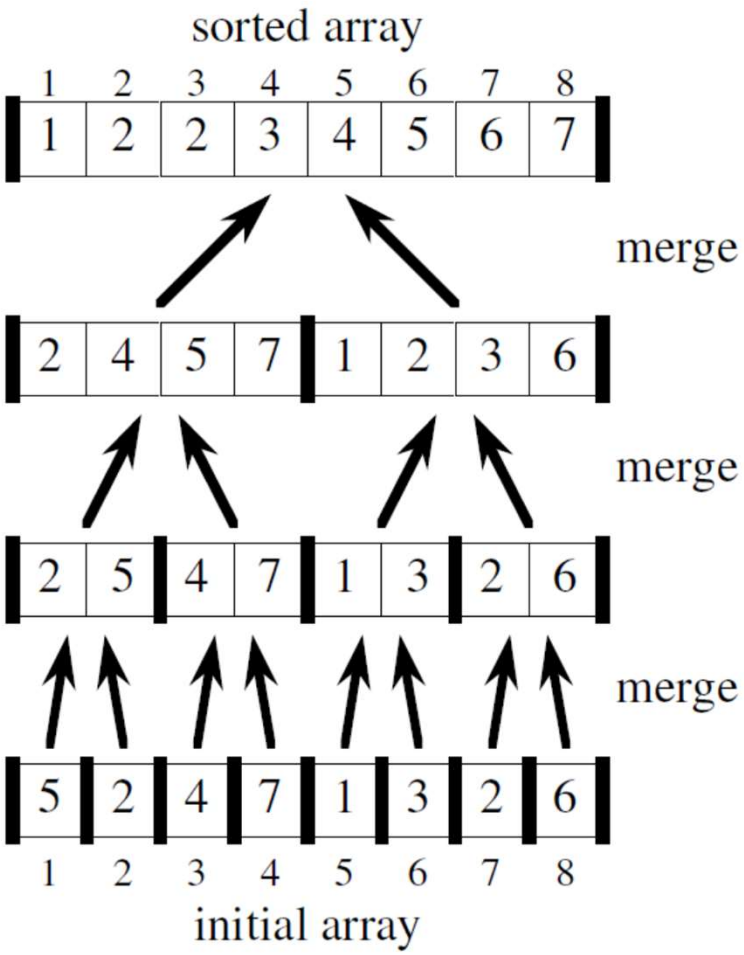
Merge sort

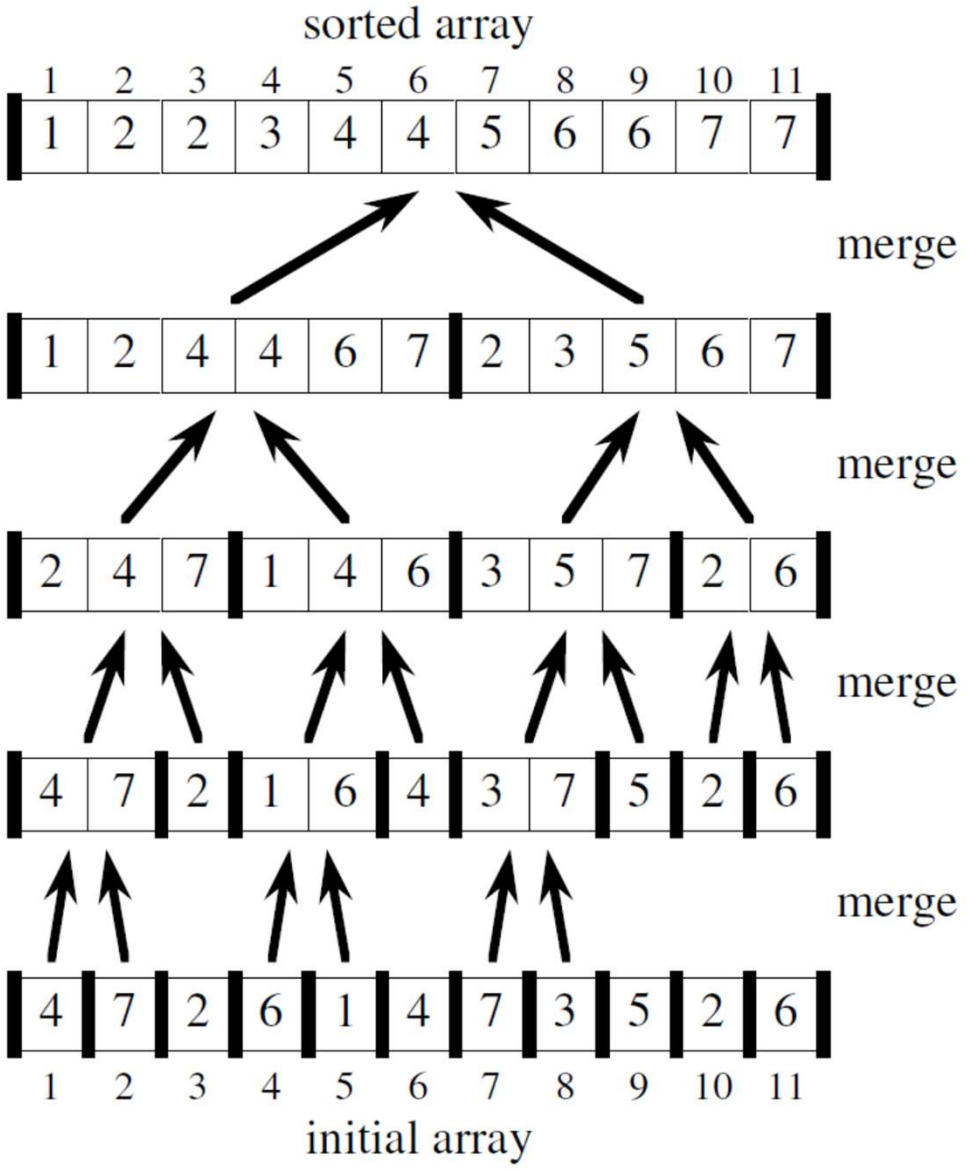
```
MERGE-SORT(A, p, r)
  if p ≥ r                          // zero or one element?
      return
  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
  MERGE(A, p, q, r)
```

Initial call: MERGE-SORT$(A, 1, n)$



Merge sort
(Bottom-up views for $n = 8$ and $n = 11$)





Merging

- **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 restriction on p, q, r, neither subarray is empty.
- **Output**: The two subarrays are merged into a single sorted subarray in $A[p : r]$.
- We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$ = number of elements being merged.
- **What is *n*?** Until now, n has stood for the size of the original problem. But now we are 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 y n, in general n will be the size of a give subproblem.

The MERGE procedure

- The MERGE procedure copies the two subarrays $A[p : q]$ and $A[q + 1 : r]$ into temporary arrays L and R ("left" and "right"), and then merges the values in L and R back into $A[p : r]$.
  - First compute the lengths n_L and n_R of the subarrays $A[p : q]$ and $A[q + 1 : r]$, respectively.
  - Then create arrays $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$ with respective length n_L and n_R.
  - The two **for** loops copy the subarrays $A[p : q]$ into L and $A[q + 1 : r]$ into R.
  - The first **while** loop repeatedly identifies the smallest value in L and R that has yet to be copied back into $A[p : r]$ and copies it back in.

The MERGE procedure

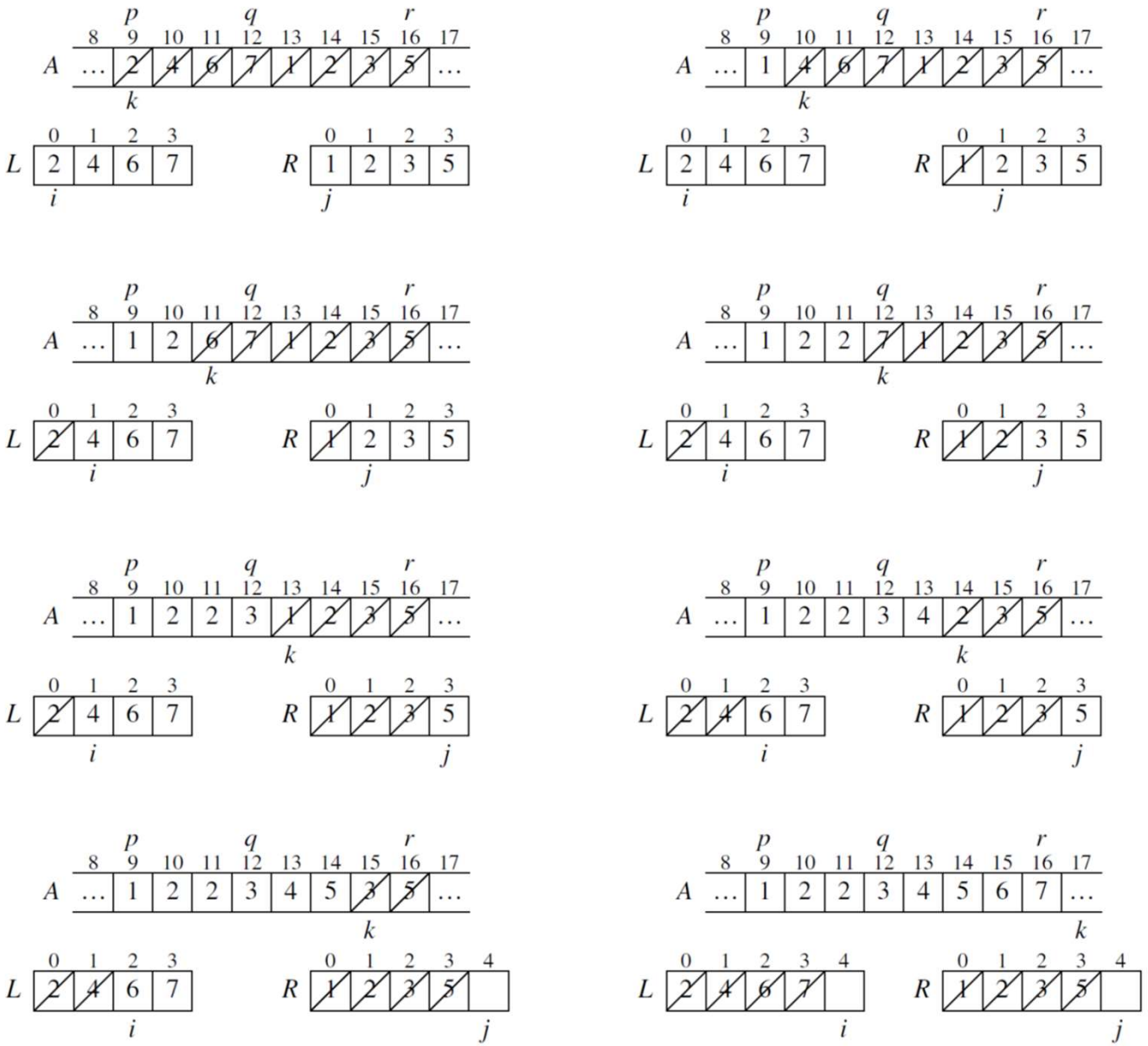
- As the comments indicate, the index k gives the position of A that is being filled in, and the indices i and j give the positions in L and R, respectively, of the smallest remaining values.
- Eventually, either all of L or all of R will be copied back into $A[p : r]$, and this loop terminates.
- If the loop terminated because all of R was copied back, that is, because $j = n_R$, then i is still less than n_L, so that some of L has yet to be copied back, and these values are the greatest in both L and R.
- In this case, the second **while** loop copies these remaining values of L into the last few positions of $A[p : r]$.
- Because $j = n_R$, the third **while** loop iterates zero times.
- If instead the first while loop terminated because $i = n_L$, then all of L has already been copied back into $A[p : r]$, and the third **while** loop copies the remaining values of R back into the end of $A[p : r]$.

Psudocode of merging

```
MERGE(A, p, q, r)
1  n_L = q − p + 1          // length of A[p : q]
2  n_R = r − q              // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
8  i = 0                    // i indexes the smallest remaining element in L
9  j = 0                    // j indexes the smallest remaining element in R
10 k = p                    // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

- ***Running time***
  - The first two **for** loops take $\Theta(n_L + n_R) = \Theta(n)$ time. Each of the three lines before and after the **for** loops takes constant time.
  - Each iteration of the three **while** loops copies exactly one value from L or R into A, and every value is copied back into A exactly one time. Therefore, these three loops make a total of n iterations, each taking constant time, for $\Theta(n)$ time.
  - Total running time: $\Theta(n)$.

Example of a call of MERGE(*A*, 9, 12, 16)



Analyzing divide-and-conquer algorithms

- Use a **recurrence equation** (more commonly, a **recurrence**) to describe the running time of divide-and-conquer algorithm.
- Let $T(n)$ = running time on a problem of size n.
  - If the problem size is small enough (say, $n \le n_0$ for some constant n_0, have a base case. The brute-force solution takes constant time: $\Theta(1)$).
  - Otherwise, divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
  - Let the time to divide a size-n problem be $D(n)$.
  - Have a subproblems to solve, each of size n/b => each subproblem takes $T(n/b)$ time to solve => spend $aT(n/b)$ time solving subproblems.
  - Let the time to combine solutions be $C(n)$.
  - Get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_0 , \\ aT(n/b) + D(n) + C(n) & \text{otherwise} . \end{cases}$$

Analyzing merge sort

- For simplicity, assume that n is a power of 2 => each divide step yields two subproblems, both of size exactly $n/2$.
- The base case occurs when $n = 1$. When $n \geq 2$, time for merge sort steps:
  - **Divide**: Just compute q as the average of p and r => $D(n) = \Theta(1)$.
  - **Conquer**: Recursively solve two subproblems, each of size $n/2$ => $2T(n/2)$.
  - **Combine**: MERGE on an n-element subarray takes $\Theta(n)$ time => $C(n) = \Theta(n)$.
- Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n: $\Theta(n)$ => 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 Lecture 4, we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$.
- 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.

Recursion tree for analyzing merge sort

- Let c_1 be constant that describes the running time for the base case and c_2 be a constant for the time per array element for the divide and conquer steps.
- Assume that the base case occurs for $n = 1$, so than $n_0 = 1$.
- Rewrite the recurrence as

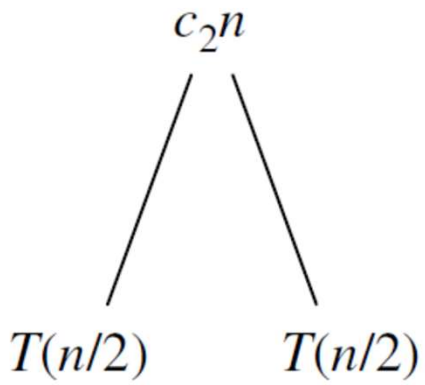
$$T(n) = \begin{cases} c_1 & \text{if } n = 1 , \\ 2T(n/2) + c_2 n & \text{if } n > 1 . \end{cases}$$

where $c_1 > 0$ represents the time required to solve a problem of size 1, and $c_2 > 0$ is the time per array element of the divide and combine steps.

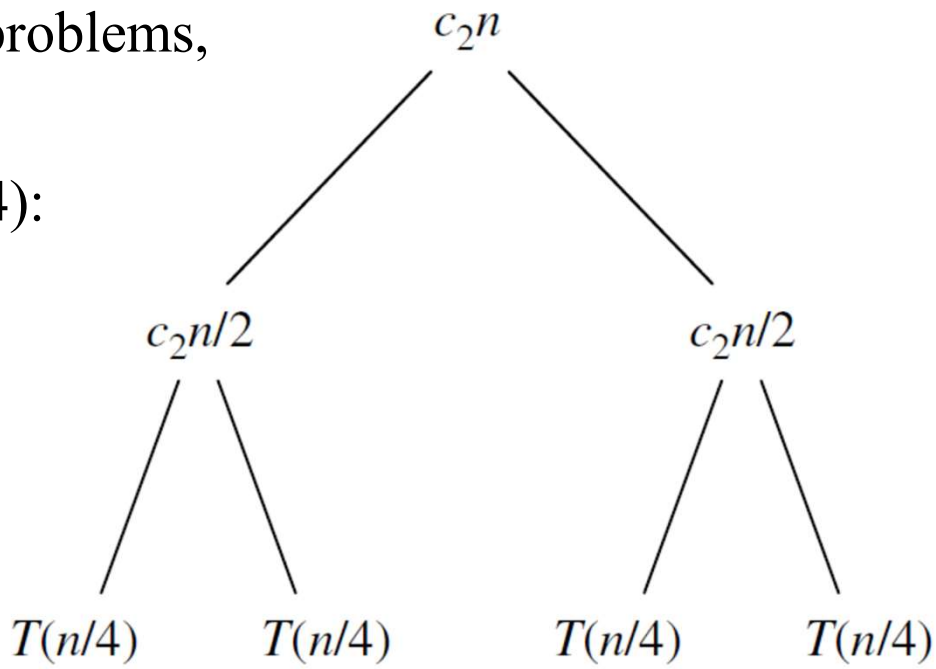
- Draw a ***recursion tree***, which shows successive expansions of the recurrence. In that tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

Recursion tree for analyzing merge sort

- For the original problem, have a cost of $c_2 n$, plus the two subproblems, each costing $T(n/2)$:

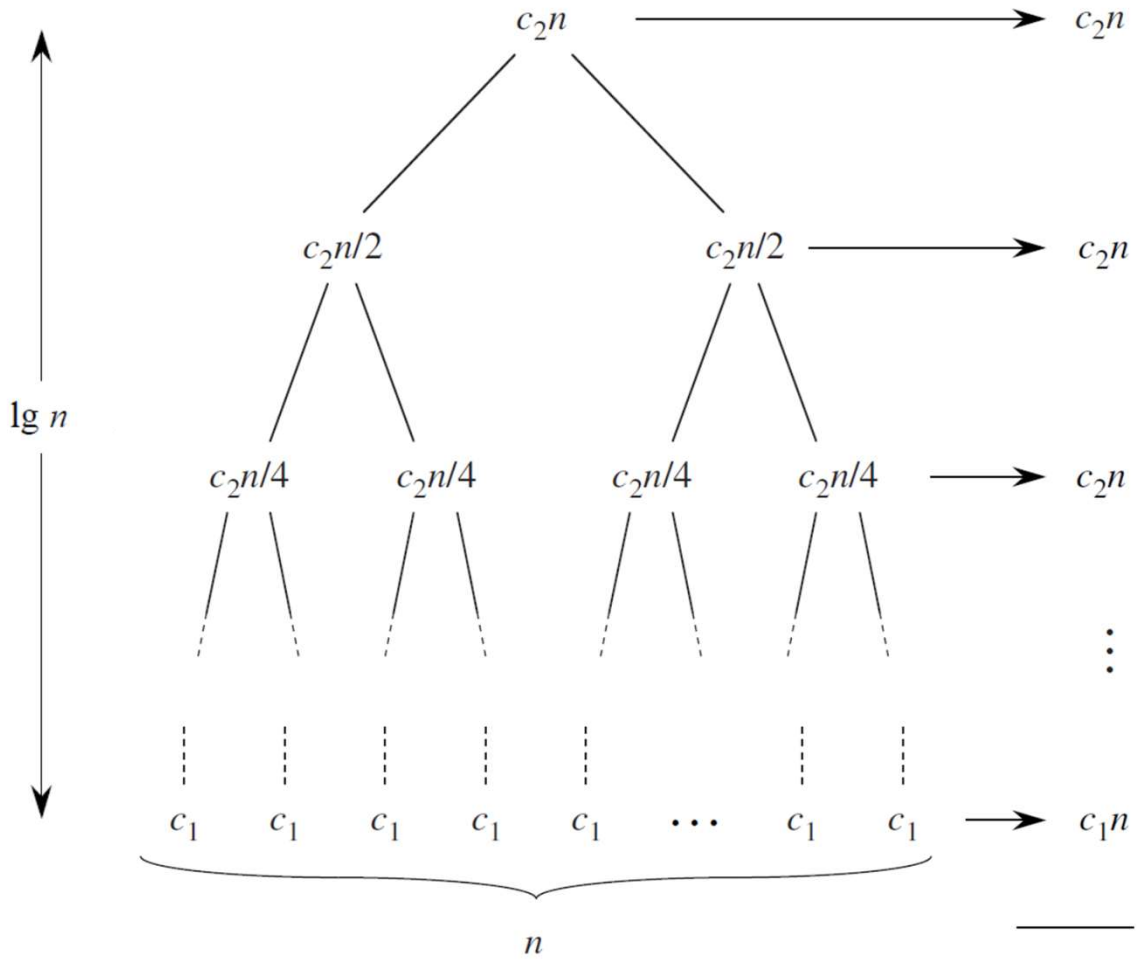


- For each of the size-$n/2$ subproblems, have a cost of $c_2 n/2$, plus two subproblems, each costing $T(n/4)$:



Recursion tree for analyzing merge sort

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



Recursion tree for analyzing merge sort

- Each level except the bottom has cost $c_2 n$.
  - The top level has cost $c_2 n$.
  - The next level down has 2 subproblems, each contributing cost $c_2 n/2$.
  - The next level has 4 subproblems, each contributing cost $c_2 n/4$.
  - Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves => cost per level stays the same.

Recursion tree for analyzing merge sort

- There are $\lg n + 1$ levels (height is $\lg n$, due to the subproblem size hits $n = 1$ when $n/2^i = 1$ or, equivalently, when $i = \lg n$).
  - 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 of 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 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 the inductive argument.
- Total cost is sum of costs at each level. There are $\lg n + 1$ levels, each level except the bottom costs $c_2 n \Rightarrow c_2 n \lg n$. The bottom level has n leaves in the recursion tree, each costing $c_1 \Rightarrow c_1 n$.
- Total cost is $c_2 n \lg n + c_1 n$. Ignore low-order term of $c_1 n$ and constant coefficient $c_2 \Rightarrow \Theta(n \lg n)$.

The correctness of the MERGE procedure

- We first state a loop invariant for the **while** loop of lines 12-18 of the MERGE procedure and then use it, along with the **while** loops of lines 20-23 and 24-27, to prove that the MERGE procedure is correct.

- **Loop invariant**: At the start of each iteration of the **while** loop of lines 12-18, the subarray $A[p : k-1]$ contains $i+j$ smallest elements of $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- We must show that this loop invariant holds prior to the first iteration of the **while** loop of lines 12-18, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates. In fact, we will consider as well the **while** loops of lines 20-23 and lines 24-27 to show that the MERGE procedure works correctly.

The correctness of the MERGE procedure

- **Initialization**: Prior to the first iteration of the loop, we have $k = p$ so that the subarray $A[p : k - 1]$ is empty. Since $i = j = 0$, this empty subarray contains the $i + j = 0$ smallest elements of L and R, and both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- **Maintenance**: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p : k - 1]$ contains the $i + j$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p : k]$ will contain the $i + j + 1$ smallest elements. Incrementing i in line 15 and k in line 18 reestablishes the loop invariant for the next iteration. If it was instead the case that $L[i] > R[j]$, then lines 16-18 perform the appropriate action to maintain the loop invariant.

The correctness of the MERGE procedure

- **Termination**: Each iteration of the loop increments either i or j. Eventually, either $i \geq n_L$, so that all elements in L have been copied back into A, or $j \geq n_R$, so that all elements in R have been copied back into A. By the loop invariant, when the loop terminates, the subarray $A[p : k-1]$ contains the $i+j$ smallest elements of $L[0 : n_L - 1]$ and $R[0 : n_R - 1]$, in sorted order. The subarray $A[p : k-1]$ consists of $r-p+1$ positions, the last $r-p+1-(i+j)$ which have yet to be copied back into A.

- Suppose that the loop terminated because $i \geq n_L$. Then the **while** loop of lines 20-23 iterates 0 times, and the **while** loop of lines 24-27 copies the remaining $n_R - j$ elements of R into the rightmost $n_R - j$ position of $A[p : r]$. These elements of R must be the $n_R - j$ greatest values in L and R. Thus, we just need to show that the correct number of elements in R are copied back into $A[p : r]$, that is, $r-p+1-(i+j) = n_R - j$.

The correctness of the MERGE procedure

- We use two facts to do so.
- First, because the number of positions in $A[p : r]$ equals the combined sizes of the L and R arrays, we have $r - p + 1 = n_L + n_R$, or $n_L = r - p + 1 - n_R$.
- Second, because $i \geq n_L$ and the **while** loop of lines 12-18 increases i by at most 1 in each iteration, we must have that $i = n_L$ when this loop terminated. Thus we have

$$\begin{aligned} r - p + 1 - (i + j) &= r - p + 1 - n_L - j \\ &= r - p + 1 - (r - p + 1 - n_R) - j \\ &= n_R - j \end{aligned}$$

- If instead the loop terminated because $j \geq n_R$, then you can show that the remaining $n_L - i$ elements of L are the greatest values in L and R, and that the **while** loop of lines 20-23 copies them into the last $r - p + 1 - (i + j)$ positions of $A[p : r]$. In either case, we have shown that the MERGE procedure merges the two sorted subarrays $A[p : q]$ and $A[q + 1 : r]$ correctly.

Reading

- Sections 2.1 ~ 2.3
- Appendix A Summations

Python notes

- In this lecture, there are two Python programs as follows, you can find them in "Chapter 2" folder after you unzip `clrsPython.zip`.
  - `insertion_sort.py`
  - `merge_sort.py`

Written exercise 2.1

- Consider the following procedure SUM-ARRAY(A, n). It computes the sum of the n numbers in array $A[1 : n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY(A, n) procedure returns the sum of the numbers in $A[1: n]$.

```
SUM-ARRAY(A, n)
1   sum = 0
2   for i = 1 to n
3       sum = sum + A[i]
4   return sum
```

Written exercise 2.2

- Consider sorting n numbers stored in array $A[1 : n]$ by first finding the smallest element of $A[1 : n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2 : n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3 : n]$, and exchange it with $A[3]$. Continue in this manner for the first $n - 1$ elements of A. Write procedure for this algorithm, which is known as ***selection sort***. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

Written exercise 2.3

- Use mathematical induction to show that when $n \geq 2$ is an exact power of 2, the solution of the recurrence

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

is $T(n) = n \lg n$.

Written exercise 2.4

- You can also think of insertion sort as a recursive algorithm. In order to sort $A[1 : n]$, recursively sort the subarray $A[1 : n - 1]$ and then insert $A[n]$ into the sorted subarray $A[1: n - 1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.