Contents

Lecture Notes for Chapter 2: Getting Started

Chapter 2 overview

Goals

- 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.

Insertion 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***. *[You might want to clarify that "satellite data" does not necessarily come from a satellite.]*

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.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise.

English is sometimes the best way.

When issues of control need to be made perfectly clear, we often use ***pseudocode***.

- Pseudocode is similar to C, C++, Pascal, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for *expressing algorithms to humans*. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode. Therefore, unlike for "real" programming languages, we cannot create a compiler that translates pseudocode to machine code.

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.

Pseudocode

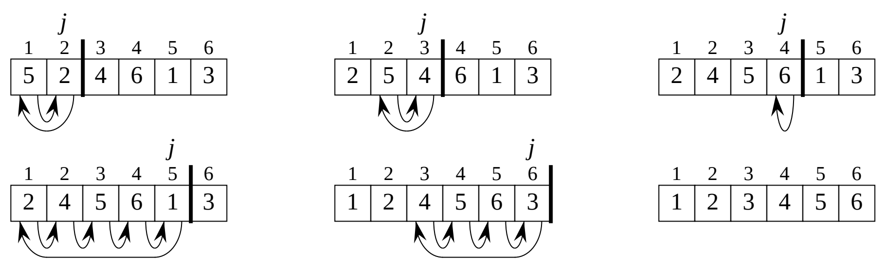
We use a procedure INSERTION-SORT.

- Takes as parameters an array $A[1 \,..\, n]$ and the length n of the array.
- As in Pascal, we use ". ." to denote a range within an array.
- *[We usually use 1-origin indexing, as we do here. There are a few places in later chapters where we use 0-origin indexing instead. If you are translating pseudocode to C, C++, or Java, which use 0-origin indexing, you need to be careful to get the indices right. One option is to adjust all index calculations in the C, C++, or Java code to compensate. An easier option is, when using an array* $A[1 \,..\, n]$*, to allocate the array to be one entry longer—*$A[0 \,..\, n]$*—and just don't use the entry at index* 0*.]*
- *[In the lecture notes, we indicate array lengths by parameters rather than by using the length attribute that is used in the book. That saves us a line of pseudocode each time. The solutions continue to use the length attribute.]*
- 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.

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n - 1$
// Insert $A[j]$ into the sorted sequence $A[1 \mathinner{.\,.} j-1]$.	0	$n - 1$
$i = j - 1$	c_4	$n - 1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n - 1$

[Leave this on the board, but show only the pseudocode for now. We'll put in the "cost" and "times" columns later.]

Example



[Read this figure row by row. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1 \mathinner{.\,.} j]$—from the part of the array that is unaffected by this iteration—$A[j+1 \mathinner{.\,.} n]$. The last part of the figure shows the final sorted array.]

Correctness

We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

> **Loop invariant:** At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1 \mathinner{.\,.} j-1]$ consists of the elements originally in $A[1 \mathinner{.\,.} j-1]$ but in sorted order.

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: When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like 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.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

For insertion sort

Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 \,..\, j-1]$ 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 $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop ends when $j > n$, which occurs when $j = n+1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$ but in sorted order. In other words, the entire array is sorted.

Pseudocode conventions

[Covering most, but not all, here. See book pages 20–22 for all conventions.]

- Indentation indicates block structure. Saves space and writing time.
- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a **for** loop is still defined when the loop exits (unlike in Pascal).
- **//** indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use ***objects***, which have ***attributes***. For an attribute *attr* of object x, we write $x.attr$. (This notation matches $x.attr$ in Java and is equivalent to `x->attr` in C++.) Attributes can cascade, so that if $x.y$ is an object and this object has attribute *attr*, then $x.y.attr$ indicates this object's attribute. That is, $x.y.attr$ is implicitly parenthesized as $(x.y).attr$.
- Objects are treated as references, like in Java. If x and y denote objects, then the assignment $y = x$ makes x and y reference the same object. It does not cause attributes of one object to be copied to another.
- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object's attributes are.

- The boolean operators "and" and "or" are ***short-circuiting***: if after evaluating the left-hand operand, we know the result of the expression, then we don't evaluate the right-hand operand. (If x is FALSE in "x and y" then we don't evaluate y. If x is TRUE in "x or y" then we don't evaluate y.)

Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time.

In order to predict resource requirements, we need a computational model.

Random-access machine (RAM) model

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll 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.

The RAM model uses integer and floating-point types.

- We don't 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 $\log_2 n$.)
  - $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  - c is a constant $\Rightarrow$ 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 1000 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'll 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 i takes the same amount of time c_i.
- 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. Example: "sort the points by x-coordinate."

Analysis of insertion sort

[Now add statement costs and number of times executed to INSERTION-SORT *pseudocode.]*

- Assume that the ith line takes time c_i, which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \ldots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j.
- 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.

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.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) . \end{aligned}$$

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

Best case

The array is already sorted.

- Always find that $A[i] \le key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- 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_i) $\Rightarrow T(n)$ is a *linear function* of n.

Worst case

The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare *key* with all elements to the left of the jth position $\Rightarrow$ compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\displaystyle\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\displaystyle\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1)$.
- $\displaystyle\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, and equation (A.1) shows that it equals $\dfrac{n(n+1)}{2}$.
- Since $\displaystyle\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\dfrac{n(n+1)}{2} - 1$.

  [The parentheses around the summation are not strictly necessary. They are there for clarity, but it might be a good idea to remind the students that the meaning of the expression would be the same even without the parentheses.]
- Letting $k = j - 1$, we see that $\displaystyle\sum_{j=2}^{n} (j-1) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$.
- 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 statement costs) $\Rightarrow T(n)$ 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's 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 $A[j]$ is less than half the elements in $A[1 \,.\,.\, j-1]$ and it's greater than the other half.
  $\Rightarrow$ On average, the **while** loop has to look halfway through the sorted subarray $A[1 \,.\,.\, j-1]$ to decide where to drop *key*.
  $\Rightarrow t_j \approx j/2$.

  Although the average-case running time is approximately half of the worst-case running time, it's 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 $\Rightarrow an^2$.
Ignore constant coefficient $\Rightarrow n^2$.

But we cannot say that the worst-case running time $T(n)$ *equals* n^2.

It *grows like* n^2. But it doesn't *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 $A[1\,..\,j-1]$, place $A[j]$ correctly, so that $A[1\,..\,j]$ 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.

[It would be a good idea to make sure that your students are comfortable with recursion. If they are not, then they will have a hard time understanding divide and conquer.]

Combine the subproblem solutions to give a solution to the original problem.

Merge sort

A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.

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'll define a procedure MERGE(A, p, q, r).

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

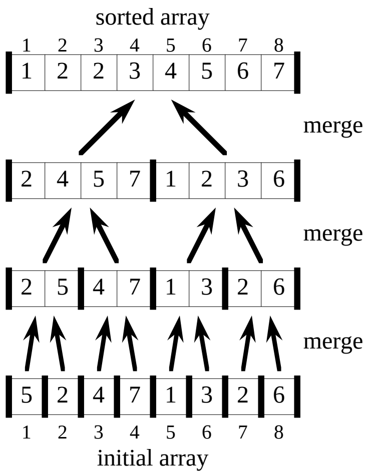
```
MERGE-SORT(A, p, r)
 if p < r                          // check for base case
     q = ⌊(p + r)/2⌋               // divide
     MERGE-SORT(A, p, q)           // conquer
     MERGE-SORT(A, q + 1, r)       // conquer
     MERGE(A, p, q, r)             // combine
```

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

[It is astounding how often students forget how easy it is to compute the halfway point of p and r as their average $(p + r)/2$. We of course have to take the floor to ensure that we get an integer index q. But it is common to see students perform calculations like $p + (r - p)/2$, or even more elaborate expressions, forgetting the easy way to compute an average.]

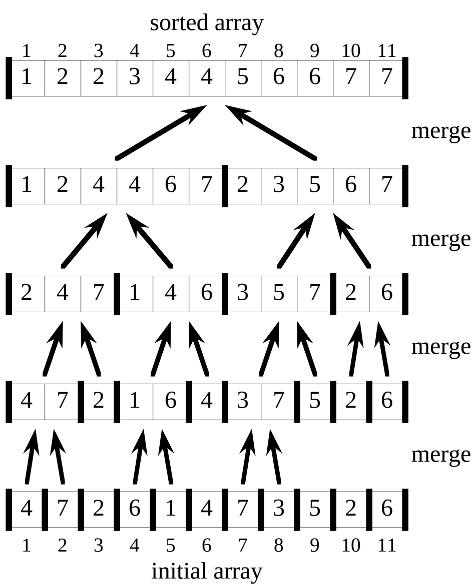
Example

Bottom-up view for $n = 8$: *[Heavy lines demarcate subarrays used in subproblems.]*



[Examples when n is a power of 2 are most straightforward, but students might also want an example when n is not a power of 2.]

Bottom-up view for $n = 11$:



[Here, at the next-to-last level of recursion, some of the subproblems have only 1 element. The recursion bottoms out on these single-element subproblems.]

Merging

What remains is 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 sorted 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? Until now, n has stood for the size of the original 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 into a single sorted pile, face-down on the table.
- A basic step:
  - Choose the smaller of the two top 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 check just the two top cards.
- There are $\le n$ basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles.
- Therefore, this procedure should take $\Theta(n)$ time.

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

- Put on the bottom of each input pile 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 have 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.

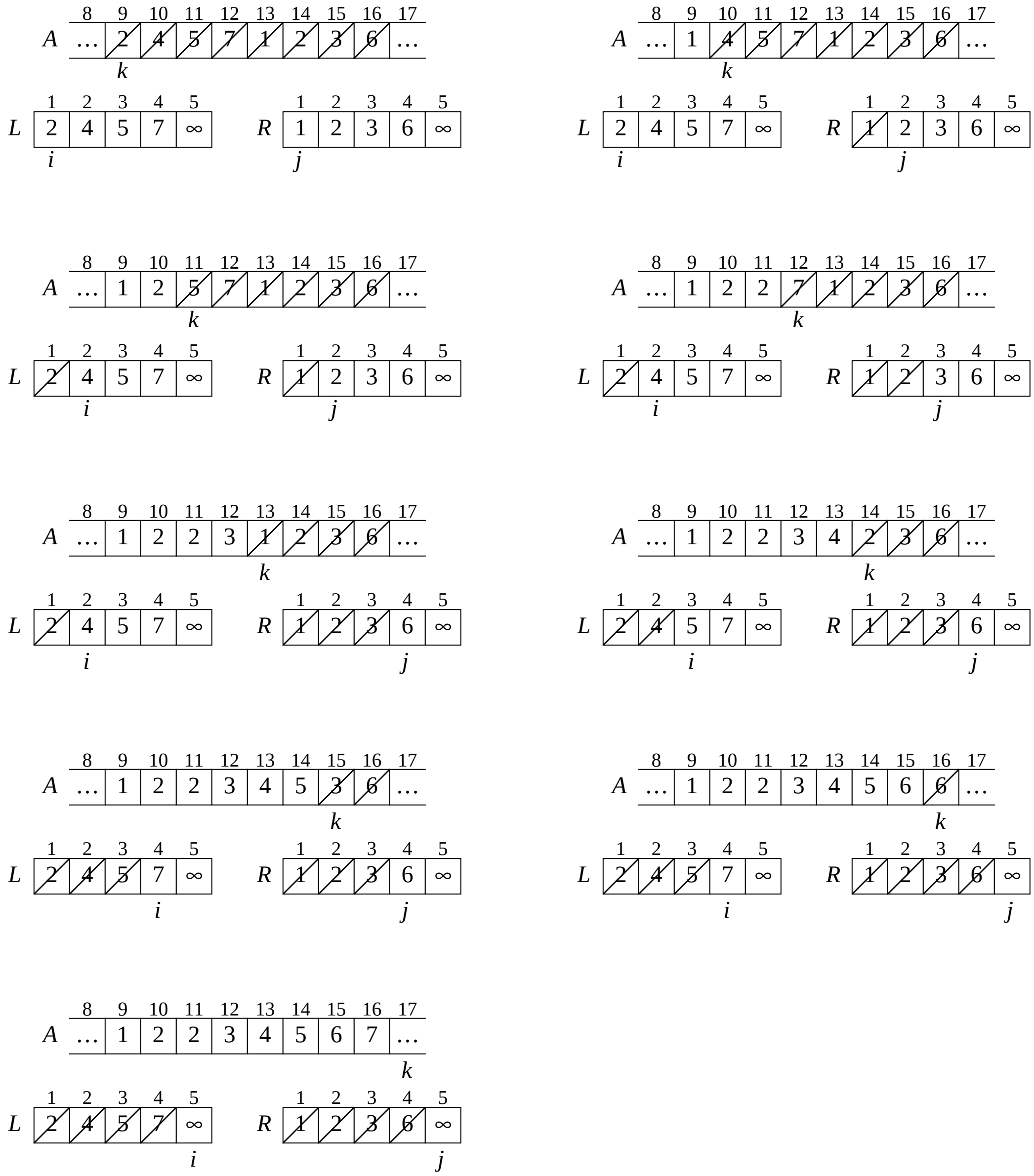
Pseudocode

```
MERGE(A, p, q, r)
  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
```

[The book uses a loop invariant to establish that MERGE *works correctly. In a lecture situation, it is probably better to use an example to show that the procedure works correctly.]*

Example

A call of MERGE$(9, 12, 16)$



*[Read this figure row by row. The first part shows the arrays at the start of the "**for** $k = p$ **to** r" loop, where $A[p\,..\,q]$ is copied into $L[1\,..\,n_1]$ and $A[q+1\,..\,r]$ is copied into $R[1\,..\,n_2]$. Succeeding parts show the situation at the start of successive iterations. Entries in A with slashes have had their values copied to either L or R and have not had a value copied back in yet. Entries in L and R with slashes have been copied back into A. The last part shows that the subarrays are merged back into $A[p\,..\,r]$, which is now sorted, and that only the sentinels (∞) are exposed in the arrays L and R.]*

Running time

The first two **for** loops take $\Theta(n_1 + n_2) = \Theta(n)$ time. The last **for** loop makes n iterations, each taking constant time, for $\Theta(n)$ time.
Total time: $\Theta(n)$.

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

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

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 $n/2 \Rightarrow 2T(n/2)$.

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, we can show that this recurrence has the solution $T(n) = \Theta(n \lg n)$. *[Reminder:* $\lg n$ *stands for* $\log_2 n$*.]*

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.

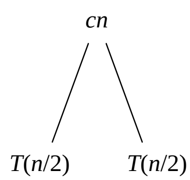
On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

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

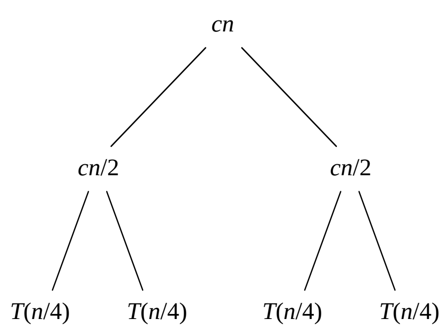
- Let c be a constant that describes the running time for the base case and also is the time per array element for the divide and conquer steps. *[Of course, we cannot necessarily use the same constant for both. It's not worth going into this detail at this point.]*
- We rewrite the recurrence as

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

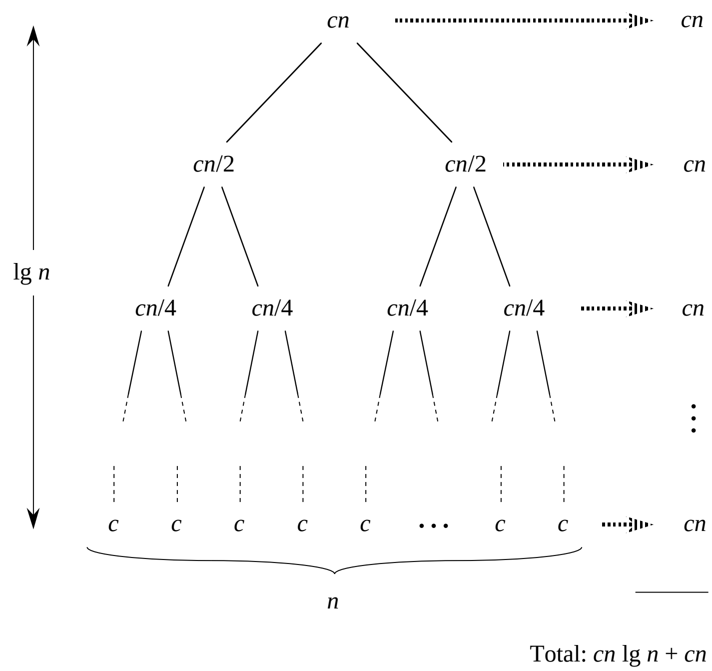
- Draw a ***recursion tree***, which shows successive expansions of the recurrence.
- For the original problem, we have a cost of cn, plus the two subproblems, each costing $T(n/2)$:



- 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:



- 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 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're done with the 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 constant coefficient $c \Rightarrow \Theta(n \lg n)$.

Solutions for Chapter 2: Getting Started

Solution to Exercise 2.2-2

This solution is also posted publicly

SELECTION-SORT(A)

```
n = A.length
for j = 1 to n - 1
    smallest = j
    for i = j + 1 to n
        if A[i] < A[smallest]
            smallest = i
    exchange A[j] with A[smallest]
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1 \,..\, j-1]$ consists of the $j-1$ smallest elements in the array $A[1 \,..\, n]$, and this subarray is in sorted order. After the first $n-1$ elements, the subarray $A[1 \,..\, n-1]$ contains the smallest $n-1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

Solution to Exercise 2.2-4

This solution is also posted publicly

Modify the algorithm so it tests whether the input satisfies some special-case condition and, if it does, output a pre-computed answer. The best-case running time is generally not a good measure of an algorithm.

Solution to Exercise 2.3-3

The base case is when $n = 2$, and we have $n \lg n = 2 \lg 2 = 2 \cdot 1 = 2$.

For the inductive step, our inductive hypothesis is that $T(n/2) = (n/2)\lg(n/2)$. Then

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(n/2)\lg(n/2) + n \\ &= n(\lg n - 1) + n \\ &= n\lg n - n + n \\ &= n\lg n \ , \end{aligned}$$

which completes the inductive proof for exact powers of 2.

Solution to Exercise 2.3-4

Since it takes $\Theta(n)$ time in the worst case to insert $A[n]$ into the sorted array $A[1\,..\,n-1]$, we get the recurrence

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

Although the exercise does not ask you to solve this recurrence, its solution is $T(n) = \Theta(n^2)$.

Solution to Exercise 2.3-5
This solution is also posted publicly

Procedure BINARY-SEARCH takes a sorted array A, a value v, and a range $[low\,..\,high]$ of the array, in which we search for the value v. The procedure compares v to the array entry at the midpoint of the range and decides to eliminate half the range from further consideration. We give both iterative and recursive versions, each of which returns either an index i such that $A[i] = v$, or NIL if no entry of $A[low\,..\,high]$ contains the value v. The initial call to either version should have the parameters $A, v, 1, n$.

```
ITERATIVE-BINARY-SEARCH(A, v, low, high)
  while low ≤ high
      mid = ⌊(low + high)/2⌋
      if v == A[mid]
          return mid
      elseif v > A[mid]
          low = mid + 1
      else high = mid − 1
  return NIL
```