6.1 Reviewing substitution method

Recurrence: $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$ and $T(1) = 1$.

This recurrence resembles merge sort, however, at each level, for the input size n, it takes n^2 cost, where the original problem is broken into two subproblems each working on $n/2$ size of input, n.

Using **master method**, we have $a = 2$, $b = 2$ and $f(n) = n^2$. Now, we calculate $n^{\log_b a} = n^{\log_2 2} = n$. Therefore, for $\epsilon = 1$, $f(n) = n^{\log_2 2+1} = n^2 = \Omega(n^2)$. $af(n/b) = 2f(n/2) = 2(n/2)^2 = \frac{n^2}{2} \le cf(n) = cn^2$, for $c = 1/2 < 1$. Therefore, this fits case 3.

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

.

From master method, we already know that $T(n) = \Theta(n^2)$.

Let's go over it again using substitution method. We guess $T(n) = O(n^2)$. To show $T(n) = O(n^2)$, we need to show that $T(n) \le cn^2$.

Let's assume that the inductive hypothesis is $T(m) \le cm^2$ for all $m < n$.

In order for this proof to work, we must come to an *exact form* of the inductive hypothesis when we make the substitution of the inductive hypothesis into the original equation, $T(n) = 2T(\lfloor n/2 \rfloor) + n^2$.

For $m = \lfloor n/2 \rfloor$, where $m \le n$. Thus, the inductive hypothesis will be:

$$T(\lfloor n/2 \rfloor) \le c(\lfloor n/2 \rfloor)^2$$

For the inductive hypothesis, let's check the base case $n = 1$, for $c \ge 1$, $T(1) = 1 \le c$. Therefore, the base case holds.

Let's make the substitution:

$$\begin{aligned} T(n) &\le 2c(\lfloor n/2 \rfloor)^2 + n^2 \\ &\le 2c(n/2)^2 + n^2 \\ &= cn^2/2 + n^2 \\ &= (c/2 + 1)n^2 \\ &\le cn^2 \qquad \text{for } c \ge 2 \end{aligned}$$

Picking $c = 2$, the inductive step is proven.

6.1.1 Subtleties

Let's go over another example where we are not able to come to the exact form of inductive hypothesis despite following the right steps.

Consider the recurrence,

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

.

We guess $T(n) = O(n)$, and we try to show that $T(n) \le cn$ for an appropriate choice of constant c. Substituting our guess in the recurrence, we obtain

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

which does not imply $T(n) \leq cn$ for any choice of c. $cn + 1$ will always be greater than cn for any choice of c. Thus, we are failing to get to the *exact form* of inductive hypothesis, which is $T(n) \leq cn$.

In this situation, it might be tempting to guess a larger guess, $T(n) = O(n^2)$, however, our original guess, $T(n) \leq cn$, is already a correct one.

We are off only by the constant 1, a lower-order term. We overcome our difficulty by *subtracting* a lower-order term from our previous guess. Our new guess is $T(n) \leq cn - d$, where $d \geq 0$ is a constant.

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &= cn - d + 1 - d \\ &\leq cn - d \quad \text{for } d \geq 1 \end{aligned}$$

As before, we must choose the constant c large enough to handle the boundary condition.

With the inductive hypothesis $T(n) \leq cn - d$, the substitution led to the same form, proving that $T(n) \leq cn - d \leq cn$, which shows $T(n) = O(n)$.

To show $T(n) = O(n)$, we should note that $T(n) \leq cn - d$ is a stronger guess for all $d > 0$, for $T(n) \leq cn - d \leq cn$ for all $d > 0$.

Similarly, to show $T(n) = O(n^2)$, a stronger guess can be $T(n) \leq cn^2 - dn$ or $T(n) \leq cn^2 - d$ for all $d > 0$, depending on how your offset looks like.

Similarly, to show $T(n) = O(n \lg n)$, a normal guess would be $T(n) \leq cn \lg n$, however, if this guess does not pan out, we can go for $T(n) \leq cn \lg n - d$ or $T(n) \leq c(n - d)\lg(n - d)$ for all $d \geq 0$.

6.1.2 Pitfalls

It is easy to err in the use of asymptotic notation. For example, in the recurrence, $T(n) = 2T(\lfloor n/2 \rfloor) + n$ we can falsely "prove" $T(n) = O(n)$ by guessing $T(n) \leq cn$ and then arguing

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq cn + n \\ &= (c + 1)n \\ &= O(n), \Leftarrow \textit{wrong}!! \end{aligned}$$

since c is a constant. **The error is that we have not proved the *exact form* of the inductive hypothesis, that is, that** $T(n) \leq cn$. We therefore will explicitly prove that $T(n) \leq cn$ when we want to show that $T(n) = O(n)$.

6.2 Partition procedure

We will go over QUICKSORT later. The key to this algorithm is the PARTITION procedure, which rearranges the subarray $A[p \mathinner{.\,.} r]$ in place.

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

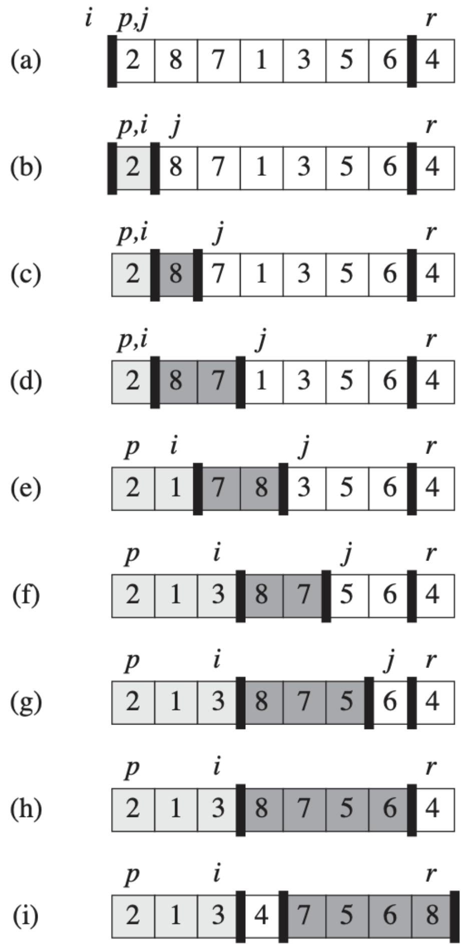

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

6.2.1 Proof of correctness

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p \mathinner{..} r]$. As the procedure runs, it partitions the array into four (possibly) empty regions. At the start of each iterations of the **for** loop in lines 3-6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

At the beginning of each iteration of the loop of lines 3-6, for any array index k,

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

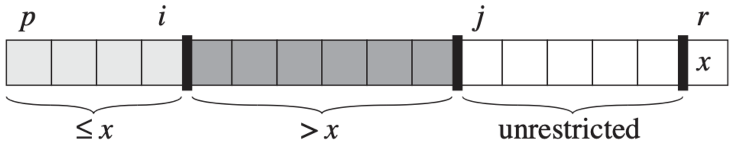

Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p \mathinner{..} r]$. The values in $A[p \mathinner{..} i]$ are all less than or equal to x, the values in $A[i+1 \mathinner{..} j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j \mathinner{..} r-1]$ can take on any values.

The indices between j and $r - 1$ are not covered by any the three cases, and the values in these entries have no particular relationship to the pivot x.

We need to show that

- this loop invariant is true prior to the first iteration
- that each iteration of the loop maintains the invariant,
- and, the invariant provides a useful property to show correctness when the loop terminates.

Initialization: Prior to the first iteration of the loop, $i = p-1$ and $j = p$. Because no values lie between p and i and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.
Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j. After j is incremented, condition 2 holds for $A[j - 1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \leq x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j. Because of the swap, we now that $A[i] \leq x$, and condition 1 is satisfied. Similarly, we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by the loop invariant, greater than x.
Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

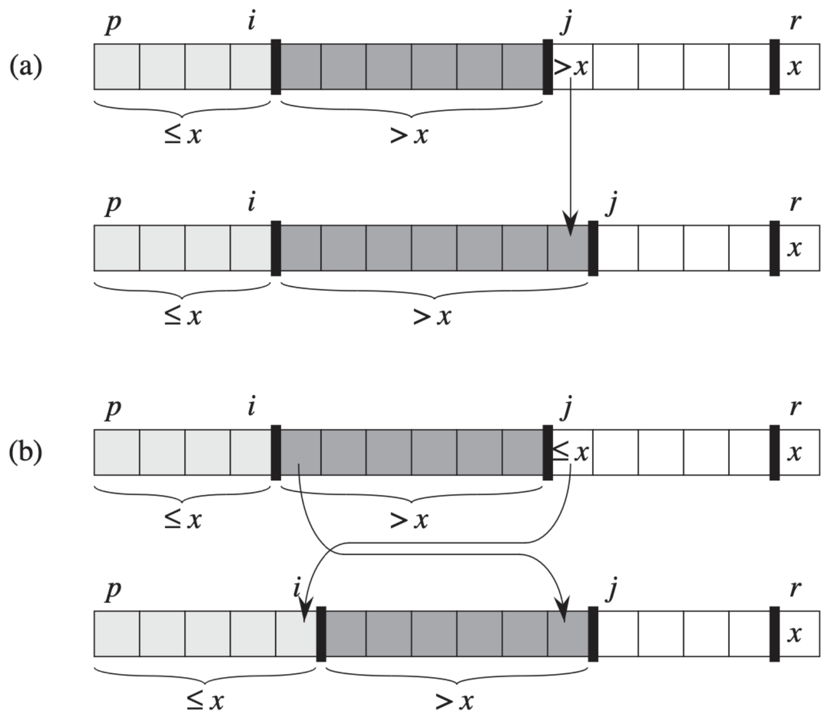

Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Classwork

- Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.
- Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

6.3 Quicksort

Quicksort, like merge sort, applies the divide-and-conquer paradigm introduced earlier. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p \mathinner{.\,.} r]$:

Divide: Partition (rearrange) the array $A[p \mathinner{.\,.} r]$ into two subarrays $A[p \mathinner{.\,.} q-1]$ and $A[q+1 \mathinner{.\,.} r]$ such that each element of $A[p \mathinner{.\,.} q-1]$ is less than or equal to $A[q]$, whic is, in turn, less than or equal to each element of $A[q+1 \mathinner{.\,.} r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p \mathinner{.\,.} q-1]$ and $A[q+1 \mathinner{.\,.} r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p\,..\,r]$ is now sorted.

Putting the three steps together:

```
QUICKSORT($A, p, r$)
1 if $p < r$
2         $q$ = PARTITION(A, P, R)
3         QUICKSORT($A, p, q - 1$)
4         QUICKSORT($A, q + 1, r$)
```

6.4 Performance of quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.

6.4.1 Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Intuitively, if we sum the costs incurred at each level of the recursion, we get an arithmetic series, which evaluates to $\Theta(n^2)$. We can use substitution method to show that $T(n) = T(n-1) + \Theta(n)$ is $\Theta(n^2)$.

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted - a common situation in which insertion sort runs is $On(n)$ time.

6.4.2 Best-case partitioning

The best-case would be where PARTITION produces two subproblems, each of size no more than $n/2$, since one of the size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. The recurrence for the running time can be expressed as

$$T(n) = 2T(n/2) + \Theta(n),$$

where we tolerate the sloppiness from ignoring the floor and ceiling and from subtracting 1. Using case 2 of the master theorem, the solution to this recurrence is $T(n) = \Theta(n \lg n)$.

6.4.3 Balanced partitioning

The average-case running time of quicksort is much closer to the best case than to the worst-case. The key to understanding why is to understand how the balance of the partitioning is reflected in the recurrence that describes the running time.

Suppose, for example, that the partitioning algorithm always produces 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$T(n) = T(9n/10) + T(n/10) + cn$,

on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term. Figure 7.4 shows the recursion tree for this recurrence. Notice that every level of the tree has cost cn, until the recursion reaches a boundary condition at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost at most cn. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. The total cost of quicksort is therefore $O(n \lg n)$.

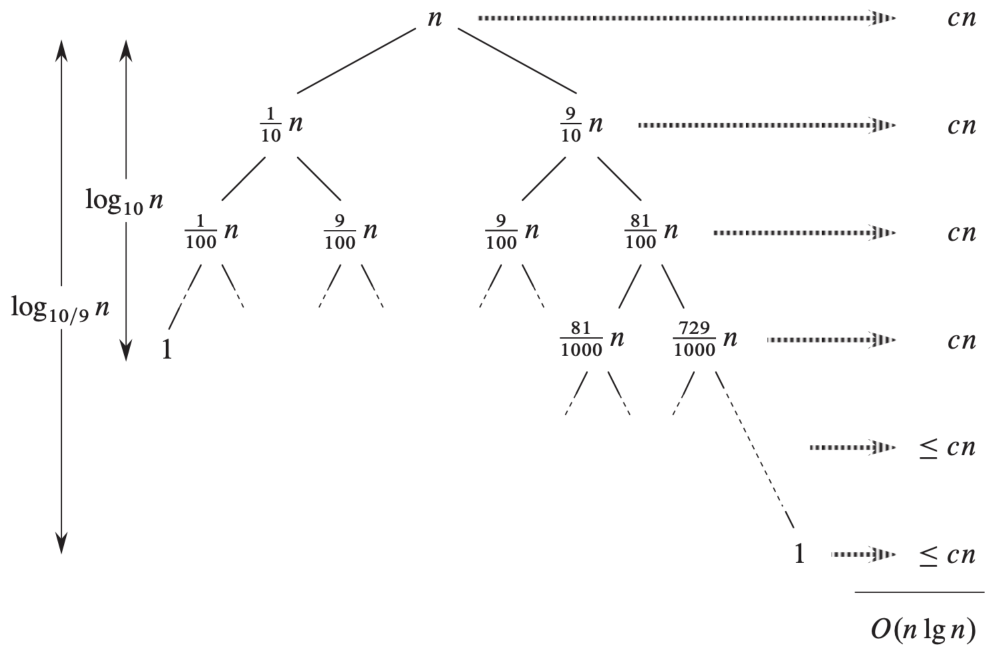

Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(n \lg n)$ time - asymptotically the same as if the split were right down the middle. Even if we have a split of 99-to-1 at each recursion level, it will yield an $O(n \lg n)$ running time. **In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality**.

Classwork

1. What is the running time of QUICKSORT when all elements of array A have the same value?
2. Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

6.5 Lower bounds of sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, .. a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. In this section, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made. We also note that the comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j. We therefore assume that all comparisons have the form $a_i \leq a_j$.

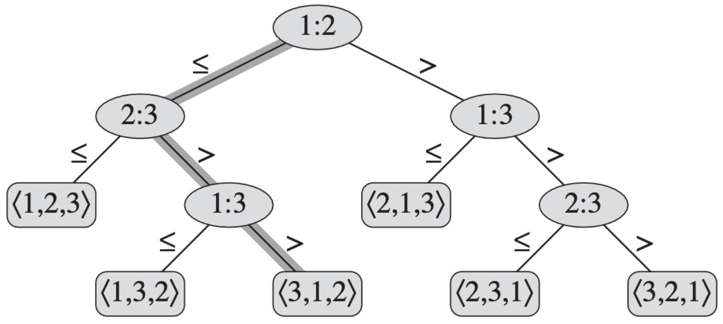

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

6.5.1 The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm operating on an input sequence of three elements.

In a decision tree, we annotate each internal node by $i : j$ for some i and j in the range $1 \leq i,\ j \leq n$, where n is the number of elements in the input sequence. The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i < a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. Thus, we shall consider only decision trees in which each permutation appears as a reachable leaf.

For instance, say, $A = \langle 4, 7, 10 \rangle$, then we first compare $A[1]$ and $A[2]$, which follows the left subtree. Then we compare $A[2]$ and $A[3]$, which also satisfies $a_i \leq a_j$, thus leading towards the left subtree. Finally, leading to $A[1] \leq A[2] \leq A[3]$, making $\langle 1, 2, 3 \rangle$ as the leaf. If we have an array $A = \langle 6, 8, 5 \rangle$, first $A[1]$ and $A[2]$ is compared, leading to left subtree. Then, $A[2]$ and $A[3]$ are compared, where the $a_2 \leq a_3$ fails, therefore, it leads to the right subtree. Here, $A[1]$ and $A[3]$ are compared, which leads to the right subtree again for $a_1 > a_3$. The final ordering is $A[3] \leq A[1] \leq A[2]$, making the leaf $\langle 3, 1, 2 \rangle$.

Theorem 1. Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h and l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \leq l \leq 2^h$$

,

which, by taking logarithms, implies

$$\begin{aligned} h &\geq lg(n!) \\ &= \Omega(n \lg n) \text{ using eq. 3.19 from the book} \end{aligned}$$

Corollary Heapsort and merge sort are asymptotically optimal comparison sorts.

Proof The $O(n \lg n)$ upper bounds on the running times of heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from the theorem above.