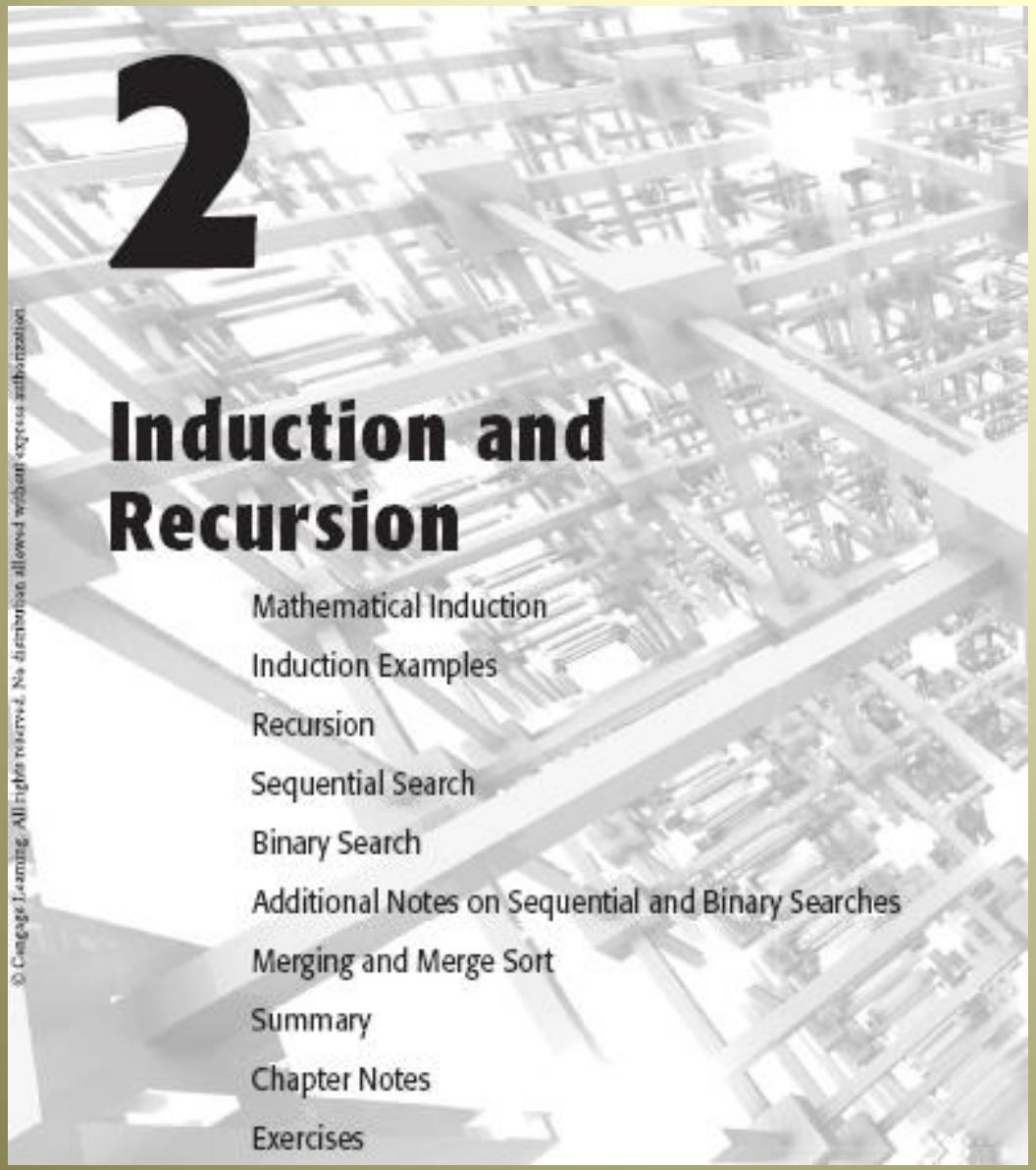


Fundamental mathematical techniques reviewed:

- **Mathematical induction**
- **Recursion**

Typically taught in courses such as Calculus and Discrete Mathematics.

Techniques introduced:

- **Divide-and-Conquer**

Algorithms Covered:

- **Binary Search**
- **Merge Sort**

Solutions From Simpler Cases

- **Mathematical induction** is a technique for proving statements about sets of consecutive integers. One can view this as being done by *inducing* our knowledge of the next case from that of its predecessor.
- **Recursion** is a technique where a solution to a problem consists of utilizing solutions to smaller versions of the problem.
- **Divide-and-Conquer** is a recursive technique:
  - *i.* *Divide* a large problem into smaller subproblems.
  - *ii.* *Solve* the subproblems *recursively*, unless the problems are small enough to be solved directly.
  - *iii.* *Combine* the solutions to the subproblems in order to obtain a solution to the original problem.

Mathematical Induction

- Given a statement about positive integers, show that the statement is always true.
- Let $P(n)$ be a *predicate,* a statement that is true or false, depending on its argument n. Assume n is a positive integer.
- Example of Predicate: "The product of the positive integers from 1 to n is divisible by 10."
  - This predicate is true for $n = 5$ since $1 \times 2 \times 3 \times 4 \times 5 = 120$, which is divisible by 10.
  - This predicate is false for $n = 4$ since $1 \times 2 \times 3 \times 4 = 24$, which is **not** divisible by 10.

Principle of Mathematical Induction

Given a predicate that it is true for all positive integers n, the predicate can typically be proved as follows.

- Let $P(n)$ be a predicate, where *n* is an arbitrary positive integer. Suppose the following can be accomplished.
  1. Show that $P(1)$ is *true*.
  2. Show that whenever $P(k)$ is *true*, we can derive that $P(k+1)$ is also *true.*
- If these two goals can be achieved, then it follows that $P(n)$ is *true* for all positive integers *n*.

Why Does Mathematical Induction Work?

- Suppose the two steps on the previous slide have been proven.
  - Then from step 1, $P(1)$ is true, and thus by step 2,
    $P(1+1) = P(2)$ is true, and thus by step 2,
    $P(2+1) = P(3)$ is true, and thus by step 2,
    $P(3+1) = P(4)$ is true, and so forth.
  - That is, step 2 allows for the induction of the truth of $P(n)$ for every positive integer n from the truth of $P(1)$.
- **Base Case**: The statement $P(1) = true$ is referred to as the *base case* of the problem being considered.
- **Inductive Hypothesis**: The assumption in Step 2 that $P(k) = true$ is called the *Inductive Hypothesis.* This is due to the fact that Step 2 is typically used to *induce* the conclusion that the statement $P(k+1) = true$.

Example: Prove That For All Positive Integers *n*, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

- First, it might be *guessed* that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, as follows.
- Let $S = \sum_{i=1}^{n} i$.
- Then $S = 1 + 2 + \cdots + (n-1) + n$.
- Write S in reverse order: $S = n + (n-1) + \cdots + 2 + 1$.
- So, if these two equations are added by combining the first terms of the right sides, the second terms of the right sides, and so on, the result is

$$2S = (n+1) + (n+1) + \cdots + (n+1) = n(n+1),$$

$$\text{or } S = \frac{n(n+1)}{2}.$$

- Again, note that this exposition is not a proof, due to the imprecision of the "…" notation.

(continued)

Now, a formal proof is given of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

- The equation claims that the sum of the first n positive integers is $\frac{n(n+1)}{2}$.
- *Base case*: For $n = 1$, the left side of the asserted equation is $\sum_{i=1}^{1} i = 1$ and the right side of the asserted equation is $\frac{1(1+1)}{2} = 1$.
- Thus, for $n = 1$, the asserted equation is true. Therefore, the base case of the induction proof is achieved.

(continued)

- *Inductive case*: Suppose the asserted equation is valid for $n = k$, for some positive integer k. Notice that it is justifiable to state this assumption due to the demonstration above that the case $n = k = 1$ is an instance for which the assumption is valid.
- Now, the asserted equation must be proven true for the next case, namely, $n = k + 1$. That is, by using the assumption for $n = k$, it must be proven that

$$\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}.$$

- Notice that the left side of the latter equation can be rewritten as

$$\sum_{i=1}^{k+1} i = \left(\sum_{i=1}^{k} i\right) + (k+1).$$

- Substituting from the inductive hypothesis gives

$$\sum_{i=1}^{k+1} i = \frac{k(k+1)}{2} + (k+1) = \frac{(k+1)(k+2)}{2},$$

as desired. Thus, the proof is complete.

Recursion

- A subprogram that calls upon itself, either directly or indirectly, is called *recursive*. Formally, an algorithm exhibits recursive behavior when it can be defined by two properties.
  - A simple base case or cases.
  - A set of rules that reduce all other cases towards the base case.
- Recursive calls are made with a smaller/simpler set of data.
- When a call is made with a sufficiently small/simple enough set of data, the call is resolved directly.
- Notice the similarity of mathematical induction and recursion.
  - Just as mathematical induction is a technique for inducing conclusions for "large *n*" from our knowledge of "small *n*," recursion allows for the processing of large or complex data sets based on the ability to process smaller or less complex data sets.

Recursively Defined Factorial Function

Definition of *n!*: Let n be a nonnegative integer. Then $n!$ is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0; \\ n \times (n-1)! & \text{otherwise.} \end{cases}$$

- **Theorem:** For $n > 0$, $n!$ is the product of the integers from 1 to n.
- So, $n!$ can be computed using a tight loop.
- Note that the definition of $n!$ is recursive and lends itself to a recursive calculation.

Example: Compute $3!$

The definition of $n!$ is used to compute $3!$ as follows.

- From the recursive definition, $3! = 3 \times 2!$.
- Thus, the value of $2!$ needs to be determined.
- Using the second line of the recursive definition, $3! = 3 \times 2! = 3 \times 2 \times 1! = 3 \times 2 \times 1 \times 0!$.
- Notice that the first line of the Definition of *n* Factorial yields $0! = 1$.
- This is the simplest case of n considered by the definition of $n!$, a case that does not require further use of recursion and therefore is a *base case*.

Example: Compute $3!$ (continued)

A recursive definition or algorithm may have more than one base case. It is the existence of one or more base cases, and logic that drives the computation toward base cases, that *prevent recursion from producing an infinite loop*.

In the example, substitute 1 for $0!$ in order to resolve our calculations. If one were to proceed in the typical fashion of a person calculating with pencil and paper, this would yield

$$3! = 3 \times 2 \times 1 \times 0! = 3 \times 2 \times 1 \times 1 = 6.$$

Pseudocode for $n!$

Integer function factorial (integer n)

Input: n is assumed to be a nonnegative integer.

Algorithm: Produce the value of $n!$ by using recursion.

Action:

If $n = 0$, then return 1

Else return $n \times factorial(n-1)$

- -

How does one analyze the running time of this function?

Analysis of Factorial Function

- Let $T(n)$ denote the running time of the procedure with input value n.
- From the base case of the recursion, $T(0) = \Theta(1)$, since the time to compute $0!$ is constant.
- From the recurrence given above, the time to compute $n!$, for $n > 0$, can be defined as $T(n) = T(n-1) + \Theta(1)$.
- The conditions

$$\left.\begin{array}{c} T(0) = \Theta(1); \\ T(n) = T(n-1) + \Theta(1) \end{array}\right\} \quad (1)$$

  form a *recursive relation*.
- Now, evaluate $T(n)$ in such a way as to express $T(n)$ without recursion.

Analysis of Factorial Function (continued)

- Repeated substitution of the recursive relation results in

$$T(n) = T(n-1) + \Theta(1) =$$
$$T(n-2) + \Theta(1) + \Theta(1) = T(n-2) + 2\Theta(1) =$$
$$T(n-3) + \Theta(1) + 2\Theta(1) = T(n-3) + 3\Theta(1).$$

- The emerging pattern is

$$T(n) = T(n-k) + k\Theta(1).$$

- Such a pattern will lead us to conjecture that

$$T(n) = T(0) + n\Theta(1),$$

  which, by the base case of the recursive definition, yields

$$T(n) = \Theta(1) + n\Theta(1) = \Theta(n).$$

- Note the above *is not a proof*.

Analysis of factorial function (continued)

Observe that the Θ-notation in condition (1) is a generalization of proportionality. Suppose the simplified recursive relation is considered

$$\left\{\begin{array}{c} T(0) = 1; \\ T(n) = T(n-1) + 1. \end{array}\right\} \qquad (2)$$

Previous observations leads one to suspect that this yields $T(n) = n + 1$. *A proof by mathematical induction follows.*

- For $n = 0$, the assertion is $T(0) = 1$, which is true.

Analysis of factorial function (continued)

- Suppose the assertion is true for some nonnegative integer k. Thus, the inductive hypothesis is the equation $T(k) = k + 1$.
- Now, one needs to show $T(k + 1) = k + 2$. Using the recursive relation (2) and the inductive hypothesis yields $T(k + 1) = T(k) + 1 = k + 1 + 1 = k + 2$, as desired.
- Since condition (1) is a generalization of (2), in which the Θ-interpretation is not affected by the differences between (1) and (2), it follows that condition (1) satisfies $T(n) = \Theta(n)$.

Computing Fibonacci Numbers

RecursiveFibonacci(n)

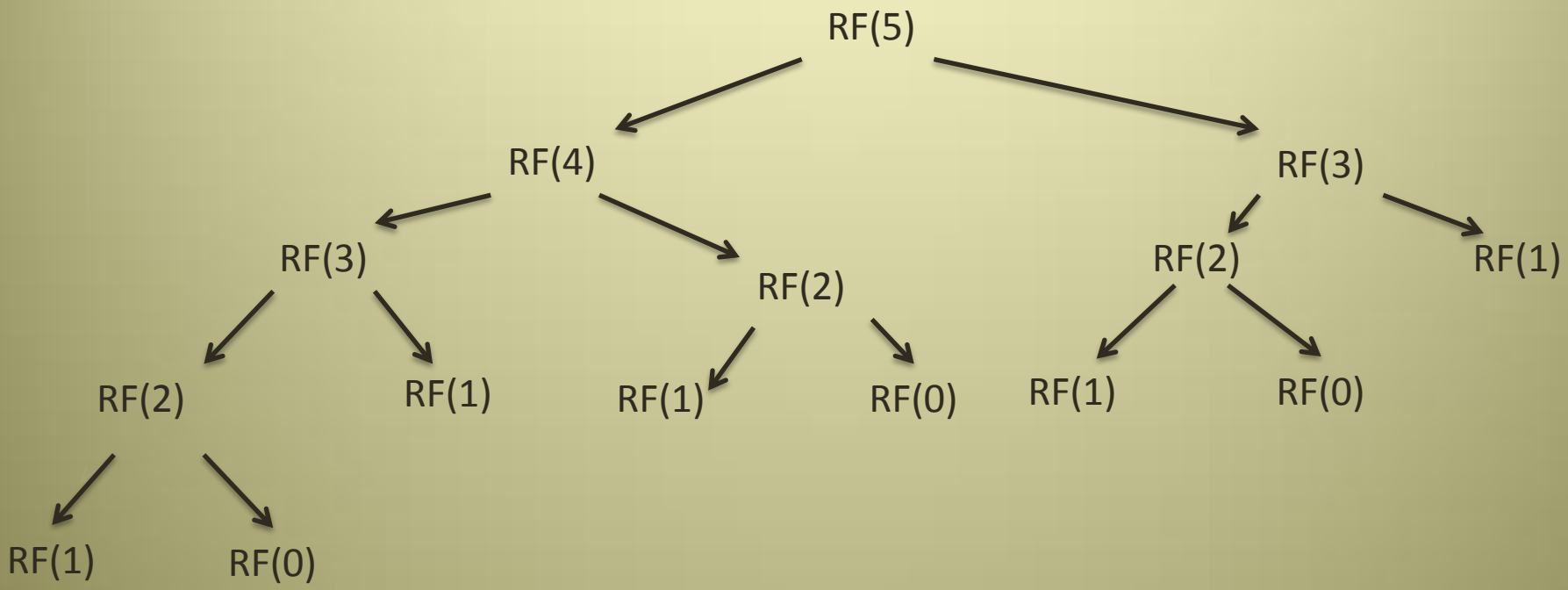
Input: a non-negative number n
Output: the Fibonacci number with index n

If n=0 then return 1
If n=1 then return 1

f =RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2)

Return(f)

Recursion Tree



Sequential Search (Non-Recursive)

The traditional non-recursive sequential search algorithm is presented so that it can be compared to the recursive implementation of binary search that is given later.

Consider the problem of searching an unordered set of data by a traditional sequential search.

- In the worst case, every item must be examined, since the item being sought *i*) might not exist or *ii*) might be the last item listed.
- So, without loss of generality, it is assumed that the sequential search starts at the beginning of the unordered data set and concludes based on one of the following conditions.
  - The search succeeds when the required item is located.
  - The search fails after every item has been examined without finding the item being sought.

Sequential Search (continued)

Since the data is not known to be ordered, the sequential examination of data items is necessary. Note that if any data item is skipped, that item could be the one being sought.

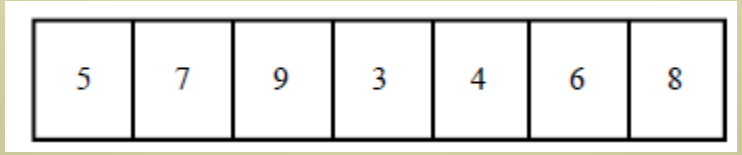


Figure 2-1. An example of sequential search. Given the array of data, a search for the value 4 requires five key comparisons. A search for the value 9 requires three key comparisons. A search for the value 1 requires seven key comparisons in order to determine that the requested value is not present.

Subprogram SequentialSearch (*X, searchValue, success, foundAt*) (continued)

Action:

position = 1

Do

success = (*searchValue* = *X*[*position*])

If *success*, then *foundAt* = *position*

Else *position* = *position* + 1

While (Not *success*) and (*position* $\leq n$) {End Do}

Return *success, foundAt*

End *Search*

Analysis of Sequential Search

- The set of instructions inside the loop runs in $\Theta(1)$ time since each instruction runs in $\Theta(1)$ time.
- In the worst case, the body of the loop will be executed n times.
  - This occurs when either the search is unsuccessful or when the item we seek is the last item in the array *X*.
  - Thus, the worst-case sequential search runs in $\Theta(n)$ time.
- Assuming that the data is ordered in a truly random fashion, then a successful search will, on average, succeed after examining half of the entries.
  - So, the average successful search runs in $\Theta(n)$ time.
- Finally, since the data is presented in a random fashion, it is possible that the item being sought is found immediately. So, the time required for the best-case search is $\Theta(1)$.

Binary Search

- Recursion is commonly used when every recursive call involves a significant reduction in the *size* of the current instance of the problem.
- An example of such a recursive algorithm is *Binary Search*.
- Consider the impact of performing a search on a sorted set of data. Think about designing an algorithm that mimics what you would do to find “Miller” in a hardcopy phone book.
  - That is, grab a bunch of pages and flip back and forth, each time grabbing fewer and fewer pages, until the desired item is located.
  - Notice that this method considers very few data values relative to the number considered by a sequential search.

Subprogram BinarySearch(*X, searchValue, success, foundAt, minIndex, maxIndex*)

Algorithm: Binary search algorithm to search ordered subarray for a *key* field equal to *searchValue*.

The algorithm is recursive. In order to search the entire array, assuming indices $1, \ldots, n$, the initial call is of the form *Search*(*X, searchValue, success, foundAt,* $1, n$).

If *searchValue* is found, return *success = true* and *foundAt* as an index at which *searchValue* is found; otherwise, return *success = false*.

Local variable: index *midIndex*

Action:

If *minIndex > maxIndex*, then {The subarray is empty}

success = false, foundAt = 0

Else {The subarray is nonempty}

BinarySearch Algorithm (continued)

```
midIndex = ⌊(minIndex + maxIndex) / 2 ⌋
If searchValue = X[midIndex].key, then
   success = true, foundAt = midIndex
Else {searchValue ≠ X[midIndex].key}
   If searchValue < X[midIndex].key, then
      BinarySearch(X, searchValue, success, foundAt,
                   minIndex, midIndex - 1)
   Else { searchValue > X[midIndex].key}
      BinarySearch(X, searchValue, success, foundAt,
                   midIndex + 1, maxIndex);
  End {Else searchValue ≠ X[midIndex].key}
 End {Subarray is nonempty}
End Search
```

Examples of Binary Search

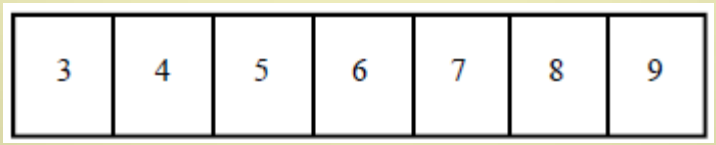


Figure 2-2. Given the array of data, a search for the value 4 requires two key comparisons (6,4). A search for the value 9 requires three key comparisons (6,8,9). A search for the value 1 requires three key comparisons (6,4,3) in order to determine that the value is not present.

Analysis of Binary Search

- Each recursive call processes an interval of about ½ the length of the interval processed by the previous call, so the running time, $T(n)$, of the binary search algorithm satisfies the recursive relation

$$T(1) = \Theta(1);$$
$$T(n) \leq T\left(\frac{n}{2}\right) + \Theta(1).$$

- Seeking a pattern: Notice in the worst case,

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) =$$
$$T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1) = T\left(\frac{n}{4}\right) + 2\Theta(1) =$$
$$T\left(\frac{n}{8}\right) + \Theta(1) + 2\Theta(1) = T\left(\frac{n}{8}\right) + 3\Theta(1).$$

- It appears $T(n) = T\left(n/2^k\right) + k\Theta(1)$, where the argument of T reaches the base value $1 = n/2^k$ when $2^k = n$, or $k = \log_2 n$. Such a pattern leads us to the conjecture that

$$T(n) = T(1) + \log_2 n \times \Theta(1) = \Theta(\log n).$$

Analysis of Binary Search (continued)

The conjecture that in the worst case, $T(n) = \Theta(\log n)$, and thus that in general, $T(n) = O(\log n)$, is established by showing, using mathematical induction, that the worst-case recursion derived from the previous discussion,

$$T(1) = 1;$$

$$T(n) = T\left(\frac{n}{2}\right) + 1,$$

resolves as $T(n) = 1 + \log_2 n$. The proof is left as an exercise.

Comparison of Sequential and Binary Searches

	Sequential search	Binary search
Data	Not ordered	Ordered by search key
Time for search that fails	$\Theta(n)$	$\Theta(\log n)$
Worst-case time for search that succeeds	$\Theta(n)$	$\Theta(\log n)$
Expected-case time for search that succeeds	$\Theta(n)$	$\Theta(\log n)$
Best-case time for search that succeeds	$\Theta(1)$	$\Theta(1)$

Recursive Sorting Algorithms

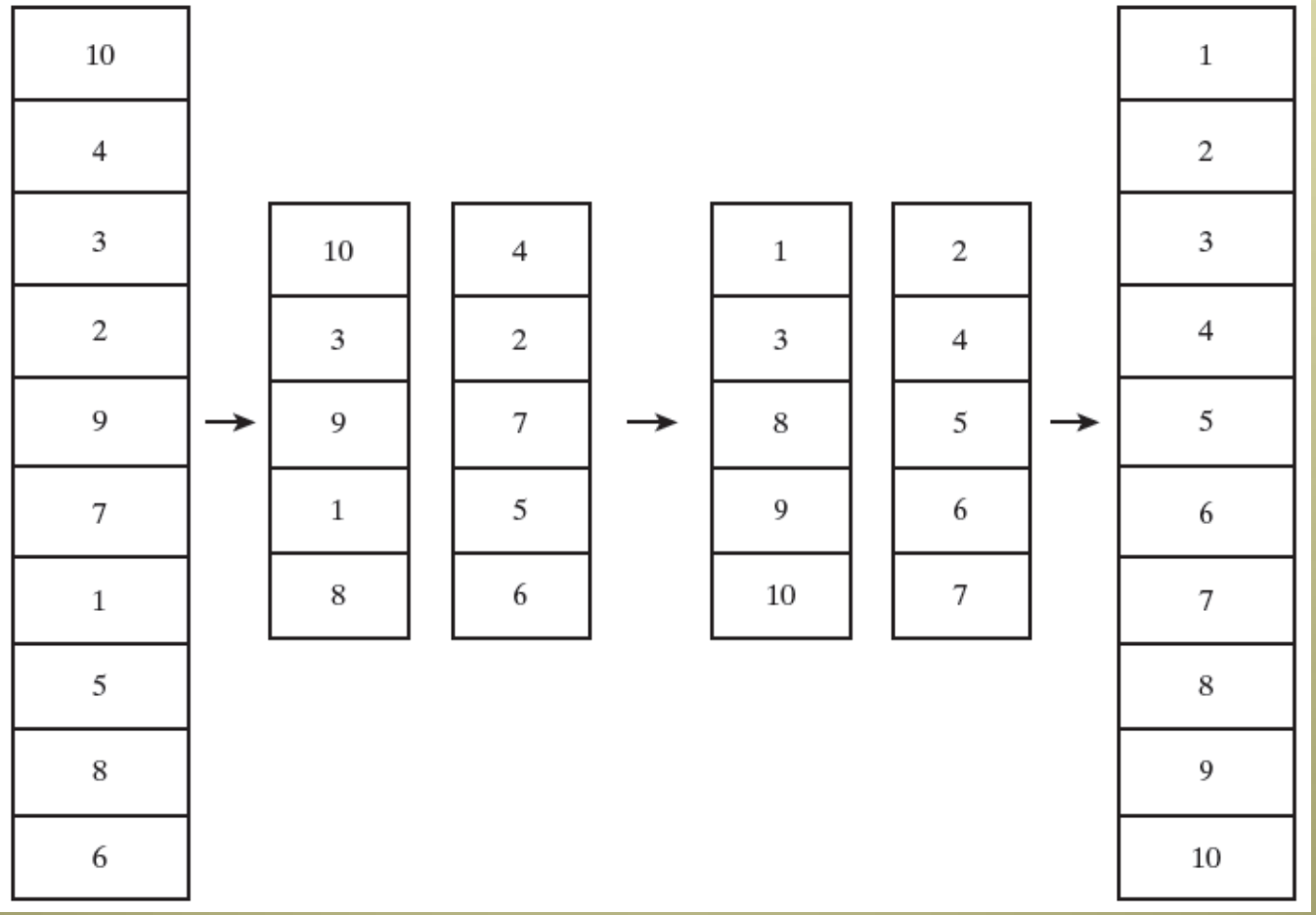


Figure 2-3. Recursively sorting a set of data. Take the initial list and divide it into two lists, each roughly half the size of the original list. Recursively sort each of the sublists. Merge these sorted sublists to create the final sorted list.

Analyzing a Recursive Sorting Algorithm

The recursive relation that describes the running time of such an algorithm is given by

$$T(1) = \Theta(1);$$
$$T(n) = S(n) + 2T(n/2) + C(n),$$

where $S(n)$ is the time used by the algorithm to split a list of n entries into two sublists of approximately $n/2$ entries apiece, and $C(n)$ is the time used by the algorithm to combine two sorted lists of approximately $n/2$ entries apiece into a single sorted list.

Discuss $S(n) = \Theta(n)$ since S = Split $= \Theta(n)$ and C = Merge = $\Theta(n)$

Merge Sort: The Split Step

- Divide the input list *X* into 2 other lists *Y* and *Z* in a simple fashion.
- A commonly used method: mimic how you might partition a deck of cards into 2 piles of equal size.
  - Elements of *X* alternate between going into *Y* and going into *Z*.
  - E.g., a member whose initial rank in *X* is odd goes into *Y* and a member whose initial rank in *X* is even goes into *Z*.
  - This is illustrated in Figure 2-3, above.

- Analysis: This step is essentially a scan of *X*, hence is performed in $S(n) = \Theta(n)$ time.

Merge Sort: The Combine Step

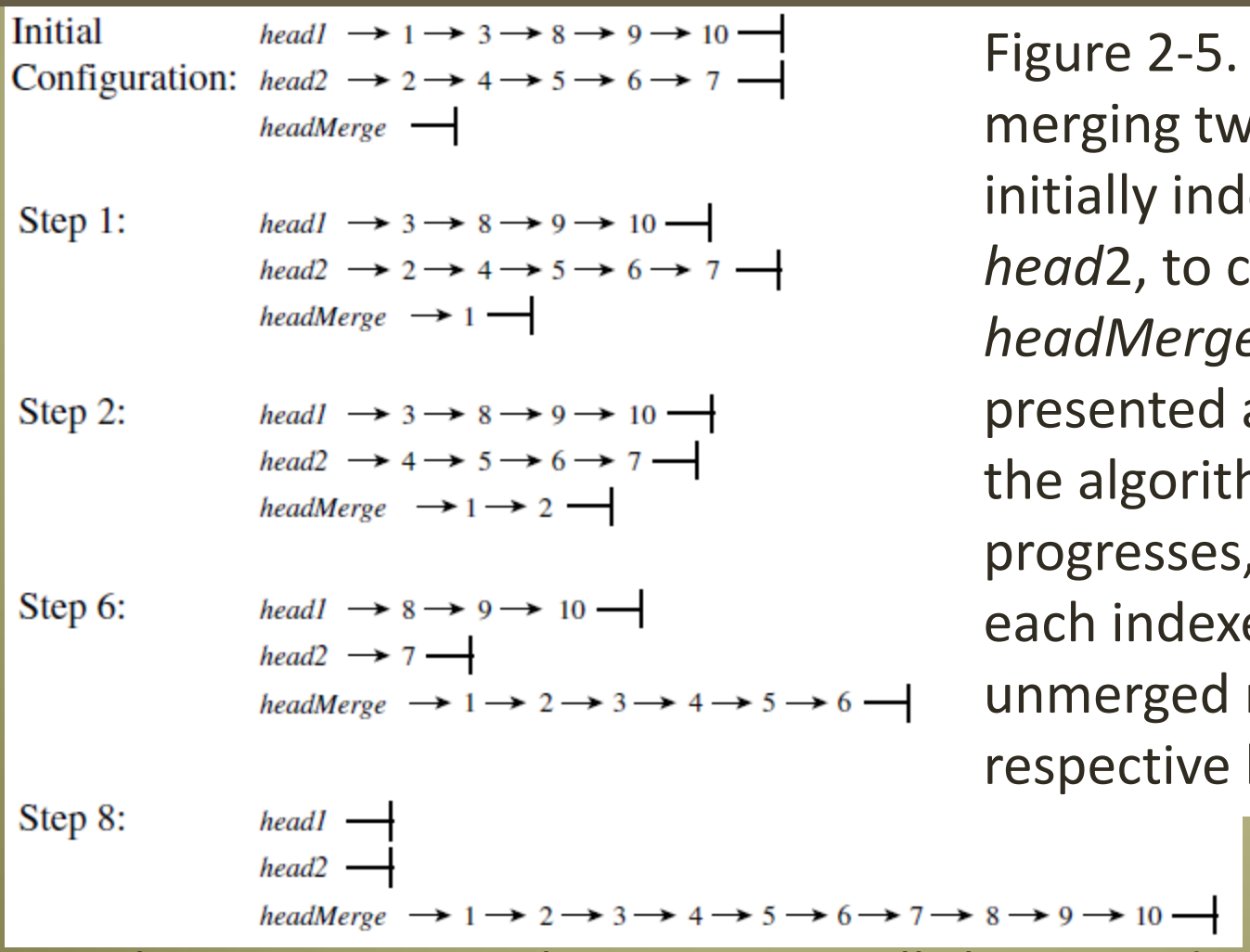


Figure 2-5. An example of merging two ordered lists, initially indexed by *head*1 and *head*2, to create an ordered list *headMerge*. Snapshots are presented at various stages of the algorithm. As the merge progresses, *head*1 and *head*2 each indexes the first unmerged node in their respective list.

Combine step: since the recursive calls have produced *Y* and *Z* as sorted lists, merge these sorted lists into a sorted list *X*.

Merging Two Lists

Input: lists *Y* and *Z* with a total of *n* members, each in ascending order
Output: list *X*, initially empty, is output with members that were the input members of *Y* and *Z* , in ascending order
Action:
While *Y* and *Z* are both non-empty, do the following.

Compare the first members of *Y* and *Z* . Whichever has the smaller key value is removed from its list and placed at the end of *X*. Note one of *Y* and *Z* shrinks, and *X* grows. This is done in $\Theta(1)$ time.

End While. The time required for this loop is bounded above by $O(n)$ and below by the length of the shorter of the input lists. In a general merge, the best case runs in $\Theta(1)$ time. In Merge Sort, the input *Y* and *Z* have approximately $n/2$ members each. Thus, the loop runs in $\Theta(n)$ time.

Now, one of *Y* and *Z* is empty, and the other isn't. Concatenate *X* and the non-empty one of *Y* and *Z* in $\Theta(1)$ time.

Merge Sort: Analysis

- The running time of Merge Sort satisfies

$$T(1) = \Theta(1);$$
$$T(n) = S(n) + 2T(n/2) + C(n),$$

  where the splitting time is $S(n) = \Theta(n)$ and the combine time is the time for a merge operation, $C(n) = \Theta(n)$.
- This yields

$$T(1) = \Theta(1);$$
$$T(n) = 2T(n/2) + \Theta(n).$$

} →Θ(*n* log *n*)

- The recursive relation is resolved as

$$T(1) = 1;$$
$$T(n) = 2T(n/2) + n,$$

  by substituting $n = 2^k$. This gives

$$T(1) = 1;$$
$$T(2^k) = 2T(2^{k-1}) + 2^k.$$

Merge Sort Analysis (continued)

- Using mathematical induction on k yields
$$T(n) = n(1 + \log_2 n).$$
The details of how this is done are left as an exercise.

- Hence, Merge Sort runs in $\Theta(n \log n)$ time.