Bucket Sort, Radix Sort, Dynamic Programming, Fibonacci Sequence

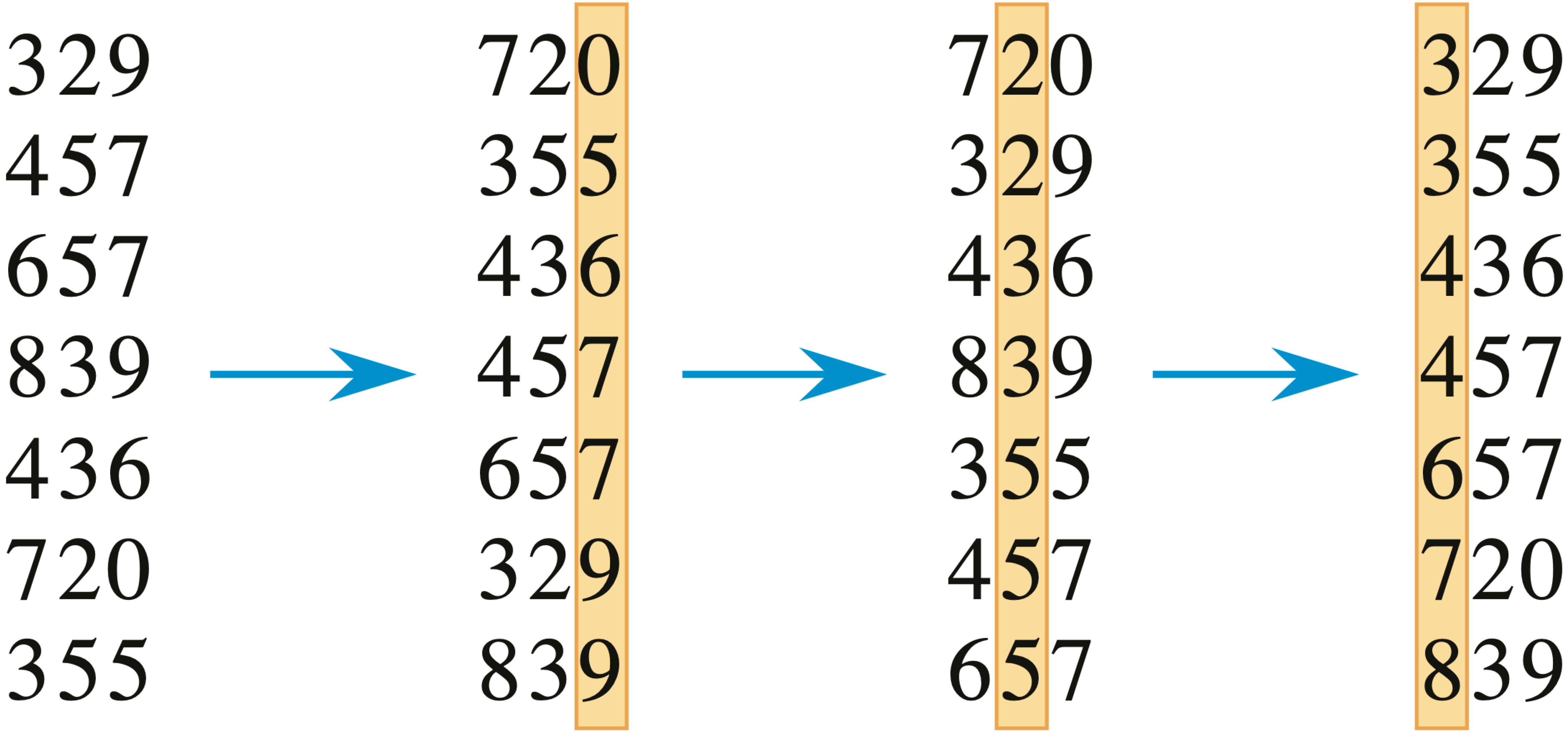
(Class 16)

Radix Sort From Book's Page No 211 (Chapter 8)

- Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums.
- The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places.
- The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.
- An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.
- In order for radix sort to work correctly, the digit sorts must be stable.

- In a typical computer, we sometimes use radix sort to sort records of information that are keyed by multiple fields.
- For example, we might wish to sort dates by three keys: year, month, and day.
- We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days.
- Alternatively, we could sort the information three times with a stable sort: first on day (the “least significant” part), next on month, and finally on year.

- The main shortcoming of counting sort is that it is useful for small integers, i.e., $1 \ldots k$ where k is small.
- If k were a million or more, the size of the rank array would also be a million.
- Radix sort provides a nice work around this limitation by sorting numbers one digit at a time.


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
839

- Here is the algorithm that sorts $A[1 \dots n]$ where each number is d digits long.

```
RADIX-SORT (array A, int n, int d)
for i ← 1 to d
    use a stable sort to sort array A[1…n] on digit i
```

- Although the pseudocode for RADIX-SORT does not specify which stable sort to use.
- But COUNTING-SORT is commonly used.
- The running time of the radix sort is:

$$T(n) = d \times n$$

$$\boldsymbol{T(n) = O(n)}$$

Bucket or Bin Sort From Book's Page No 215 (Chapter 8)

- Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$.
- Like counting sort, bucket sort is fast because it assumes something about the input.
- Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0,1)$.

- Bucket sort divides the interval $[0,1)$ into n equal-sized subintervals, or buckets.
- And then distributes the n input numbers into the buckets.
- Since the inputs are uniformly and independently distributed over $[0,1)$, we do not expect many numbers to fall into each bucket.
- To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

- Assume that the keys of the items that we wish to sort lie in a small, fixed range and that there is only one item with each value of the key.
- Then we can sort with the following procedure:
  - Set up an array of “bins” - one for each value of the key - in order,
  - Examine each item and use the value of the key to place it in the appropriate bin.

- Now our collection is sorted, and it only took n operations, so this is an $O(n)$ operation.
- However, note that it will only work under very restricted conditions.
- To understand these restrictions, let's be a little more precise about the specification of the problem and assume that there are m values of the key.
- To recover our sorted collection, we need to examine each bin.

- This adds a third step to the algorithm above,
  - Examine each bin to see whether there’s an item in it.
- Which requires m operations.
- So, the algorithm’s time becomes:

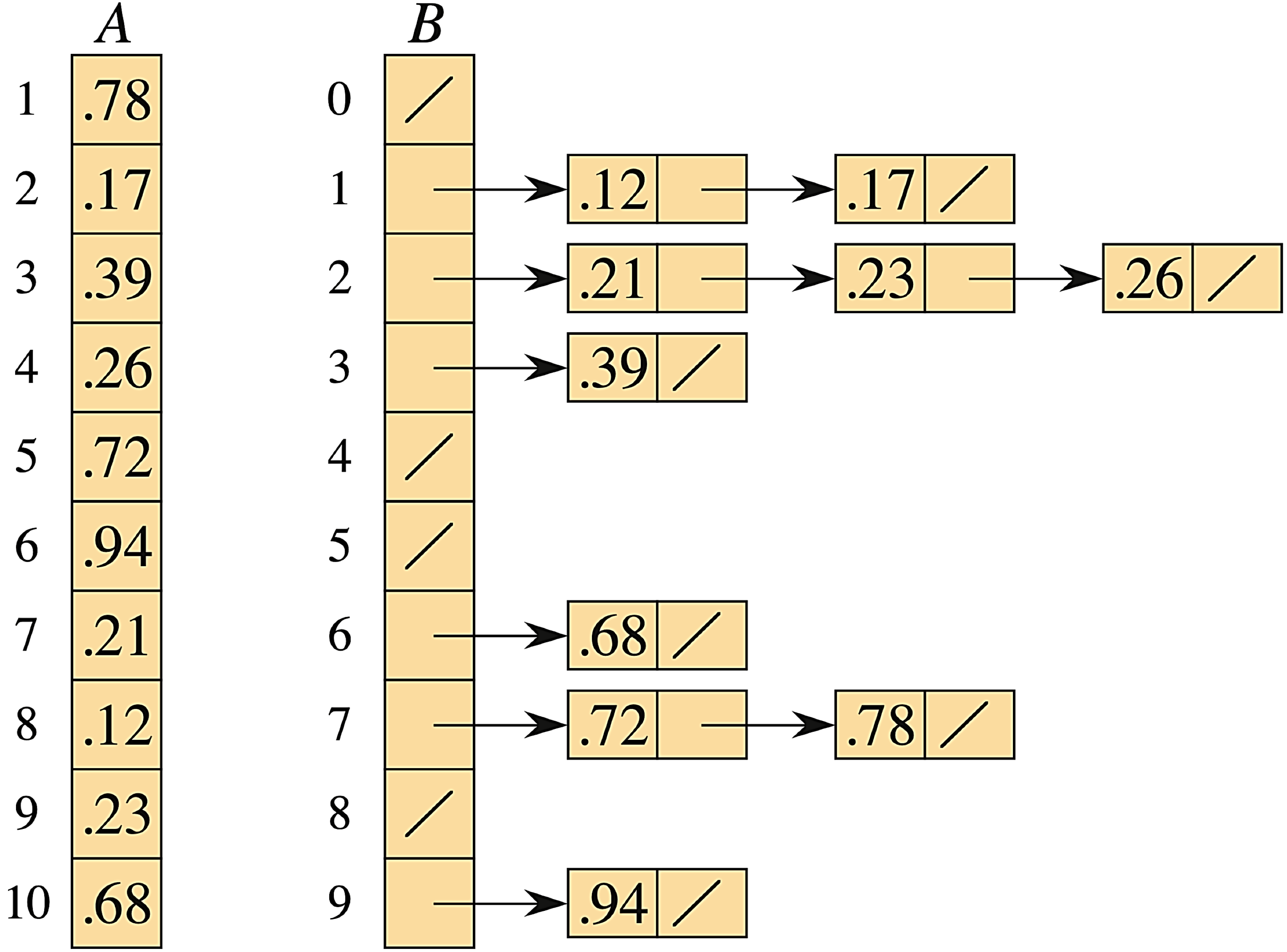
$$T(n) = c_1 n + c_2 m$$

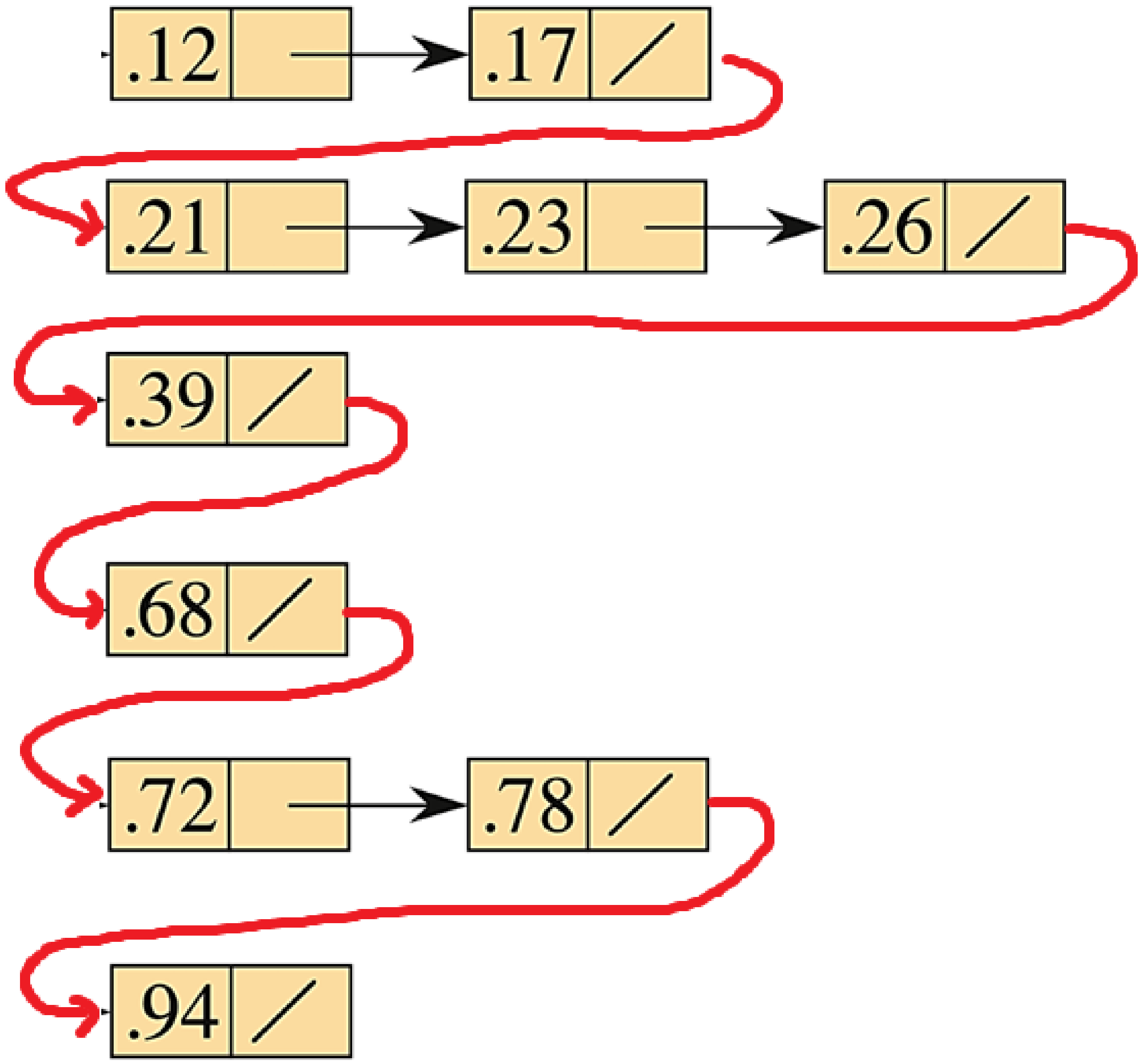
- It is strictly $O(n+m)$. If $m \leq n$, this is clearly $O(n)$.
- However, if $m \gg n$, then it is $O(m)$.

```
BUCKET-SORT (array A, n)
let B[0:n−1] be a new array
for i ← 0 to n−1
    make B[i] an empty list
for i ← 1 to n
    insert A[i] into list B[ ⌊n.A[i]⌋ ]
for i ← 0 to n−1
    sort list B[i] with insertion sort
concatenate the lists B[O], B[1], …, B[n−1] together in order
return the concatenated lists
```

- If there are *duplicates*, then each bin can be replaced by a *linked list*.
- The third step then becomes:
  - Link all the lists into one list.

- We can add an item to a linked list in $O(1)$ time.
- There are n items requiring $O(n)$ time.
- Linking a list to another list simply involves making the tail of one list point to the other, so it is $O(1)$.
- Linking m such lists obviously take $O(m)$ time.
- So, the algorithm is still $O(n + m)$.


A
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
B
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94
(a) (b)


.12
.17
.21
.23
.26
.39
.68
.72
.78
.94

Dynamic Programming From Book's Page No 263 (Chapter 14)

- Dynamic programming typically applies to optimization problems in which you make a set of choices in order to arrive at an optimal solution.
- Each choice generates subproblems of the same form as the original problem, and the same subproblems arise repeatedly.

- The key strategy is to store the solution to each such subproblem rather than recompute it.
- Dynamic programming shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- "*Programming*" in this context refers to a tabular method, not to writing computer code.
- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

- In contrast, dynamic programming applies when the subproblems overlap that is, when subproblems share subsubproblems.
- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table.
- Thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

- Dynamic programming typically applies to *optimization problems*.
- Such problems can have many possible solutions.
- Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value.
- We call such a solution "*an*" optimal solution to the problem, as opposed to "*the*" optimal solution, since there may be several solutions that achieve the optimal value.

- To develop a dynamic-programming algorithm, follow a sequence of four steps:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

Fibonacci Sequence

- Suppose we put a pair of rabbits in a place.
- How many pairs of rabbits can be produced from that pair in a year?
- If it is supposed that every month each pair begets a new pair which from the second month on becomes productive.

- Resulting sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . where each number is the sum of the two preceding numbers.
- This problem was posed by Leonardo Pisano, better known by his nickname Fibonacci (1170-1250).
- This problem and many others were in posed in his book *Liber abaci*, published in 1202.
- The book was based on the arithmetic and algebra that Fibonacci had accumulated during his travels.

- The book, which went on to be widely copied and imitated, introduced the Hindu-Arabic place-valued decimal system and the use of Arabic numerals into Europe.
- The rabbit’s problem in the third section of *Liber abaci* led to the introduction of the Fibonacci numbers and the Fibonacci sequence for which Fibonacci is best remembered today.

- This sequence, in which each number is the sum of the two preceding numbers.
- It has proved extremely fruitful and appears in many different areas of mathematics and science.
- The *Fibonacci Quarterly* is a modern journal devoted to studying mathematics related to this sequence.
- The Fibonacci numbers F_i are defined as follows:

$$F_0 = 0$$

$$F_1 = 1$$

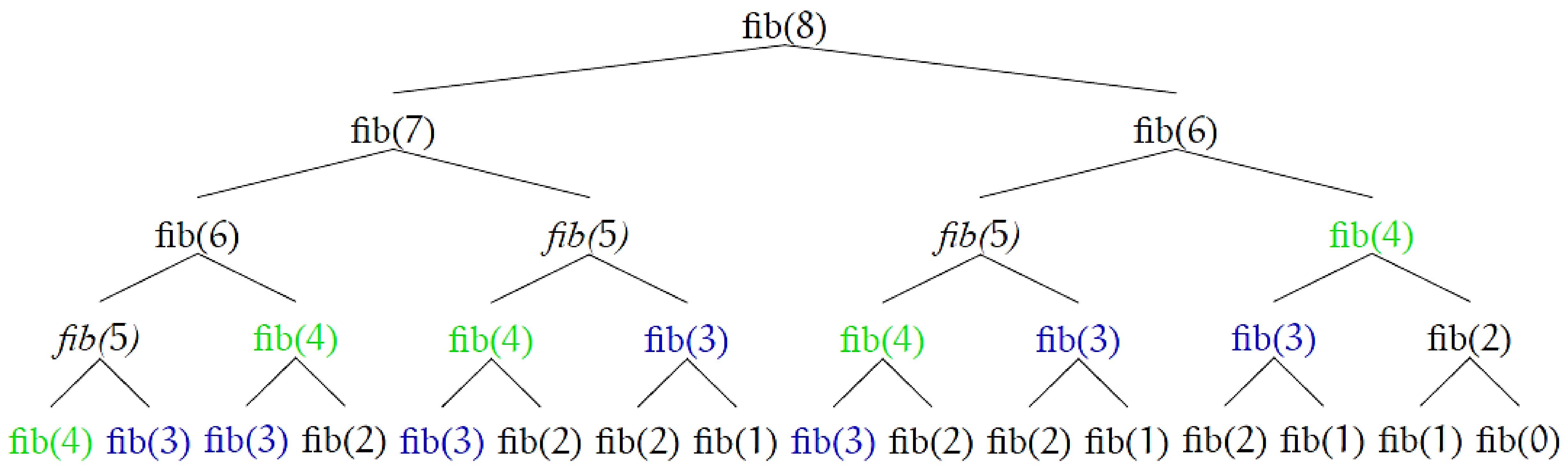
$$F_i = F_{i-1} + F_{i-2}$$

- We define the Fibonacci numbers F_i , for $i \geq 0$, as follows:

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2 \end{cases}$$

- The recursive definition of Fibonacci numbers gives us a recursive algorithm for computing them:

```
FIB(n)
if (n < 2)
    then return n
else return FIB(n − 1) + FIB(n − 2)
```



Recursive calls during computation of Fibonacci number

- We can see that there are multiple repeated calls for fib(4), fib(3), fib(5), etc.
- And for each call the FIB() algorithm will be executed recursively.
- Can we save the value of fib(3) once and every time there is recursive call of fib(3) we simply returned the saved value instead the recursive call?

- A single recursive call to fib(n) results in one recursive call to fib(n – 1), two recursive calls to fib(n – 2), three recursive calls to fib(n – 3), five recursive calls to fib(n – 4) and, in general, F_{k-1} recursive calls to fib(n – k) For each call, we’re recomputing the same Fibonacci number from scratch.
- We can avoid these unnecessary repetitions by writing down the results of recursive calls and looking them up again if we need them later.
- This process is called *memoization*.

- Save the result of each subproblem (usually in an array or hash table).
- The procedure now first checks to see whether it has previously solved this subproblem.
- If so, it returns the saved value, saving further computation at this level.
- If not, the procedure computes the value in the usual manner but also saves it.
- We say that the recursive procedure has been *memoized*: it “remembers” what results it has computed previously.

- Here is the algorithm with *memoization*.

```
MEMOFIB(n)
if (n < 2)
    then return n
if (F[n] is undefined)
    then F[n] ← MEMOFIB(n − 1) + MEMOFIB(n − 2)
    save the F[n]
return F[n]
```

- This approach is basically bottom-up approach.
- If we trace through the recursive calls to MEMOFIB, we find that array F gets filled from bottom up. i.e., first $F[2]$, then $F[3]$, and so on, up to $F[n]$.
- We can replace recursion with a simple for-loop that just fills up the array F in that order.
- So, we can also modify the above algorithm using iterations instead of recursions.
- This gives us our first explicit *dynamic programming* algorithm.

```
ITERFIB(n)
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
      do
      F[i] ← F[i − 1] + F[i − 2]
    return F[n]
```

- This algorithm clearly takes only $O(n)$ time to compute F_n.
- By contrast, the original recursive algorithm takes $O(\Phi^n)$, $\Phi = \frac{1+\sqrt{5}}{2} \approx 1.618$.
- Φ is called *golden ratio* of Fibonacci sequence.
- ITERFIB algorithm achieves an exponential speedup over the original recursive algorithm.