8.1.1

Common: Both solve a problem by breaking the problem into subproblems.

Difference: The difference is that in dynamic programming the subproblems overlap and are stored, while in divide and conquer they are not stored.

8.1.5

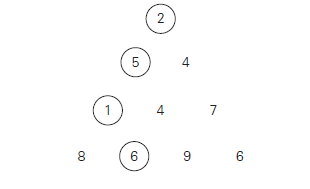
|  | x |  | • |  |  |
| --- | --- | --- | --- | --- | --- |
| • |  |  | x | • |  |
|  | • |  | x | • |  |
|  |  |  | • |  | • |
| x | x | x |  | • |  |

To modify the algorithm for inaccessible cells, a few clauses need to be added. If a cell is inaccessible for the robot, don’t compute a value for that cell. If the robot would have to pass through an inaccessible cell, pass along the x to show that that cell is inaccessible too. If there is one valid cell next to one in question, but one invalid, only compute for the valid.

Using this method, the largest number of coins the robot can collect is 4 which can be done in 12 different ways.

| 0 | x | x | x | x | x |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | x | x | x |
| 1 | 2 | 2 | x | x | x |
| 1 | 2 | 2 | 3 | 3 | 4 |
| x | x | x | 3 | 4 | 4 |

8.1.8

****

Start at the top of the triangle and store that value in a new tree. Then for each node on the second layer compute the sum of it and the adjacent node above it and store the smallest value of the sums. Continue this process until you are at the base of the triangle. The node in the base with the smallest value is your smallest sum.

Your sum triangle would look like this:

2

7 6

8 10 13

16 14 19 19

The time efficiency of the algorithm is quadratic (O(n²))

8.2.1

a.

| item | weight | value |
| --- | --- | --- |
| 1 | 3 | 25 |
| 2 | 2 | 20 |
| 3 | 1 | 15 |
| 4 | 4 | 40 |
| 5 | 5 | 50 |

Capacity = 6

|  | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 25 | 25 | 25 | 25 |
| 2 | 0 | 20 | 25 | 25 | 45 | 45 |
| 3 | 15 | 20 | 35 | 40 | 45 | 60 |
| 4 | 15 | 20 | 35 | 40 | 55 | 60 |
| 5 | 15 | 20 | 35 | 40 | 55 | 65 |

Using items 3 and 5, the maximum value of 65 can be reached

b.

The optimal subset found in part a is a unique solution

c.

We can backtrack through the table removing the current item and seeing if the new maximum value is the same as the final one.

ie does F[i - 1, j] equal vi + F[i - 1, j - wi]

8.2.6

|  | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 25 | 25 | 25 | 25 |
| 2 | 0 ii | 20 | 25 ii | 25 ii | 45 | 45 |
| 3 | 15 | 20 ii | i | i | i | 60 |
| 4 | 15 ii | i | i | i | i | 60 ii |
| 5 | i | i | i | i | i | 65 |

i) never computed ii)retrieved but not recomputed

8.3.2

=

1. OptimalBST makes two arrays, C of size (n+1) by (n+1) and R of size n by n. Each array is only filled half way. This mean the algorithm’s space efficiency is in .

8.3.3

Algorithms GenerateOptionalTree(i, j)

// Input: For indices i and j of the key set of a list. Compose a tree.

//

// Output: The indices of nodes of an optimal binary search tree.

//

If i <= j:

K <- R[i, j]

Print <- k

GenerateOptionalTree(i, k - 1)

GenerateOptionalTree(k + 1, j)

8.3.9

Let a1, a2, a3, …, an be a list of n keys. All the keys are distinct and sorted. Let p be the probability for searching for any key ai where i = 1, 2, 3, …, n. Let q be the know probability of searching for a key and not finding it for all ai and ai+1. By repeating the derivation equation for a tree like this yields a recurrence relation for the expected number of key comparisons.

Where 1 <= i < j <= n with the initial condition being for all i from 1 to n. Otherwise the algorithm is the same.