ASSIGNMENT-4
SER-501

SNEHA VIDHYASHEKAR
ASU ID : 1211274841

(1)

Greedy algorithm - making change for 'n' cents using min no of coins.

Given,

Quarter = 25 ¢
Dime = 10 ¢
Nickel = 5 ¢
Penny = 1 ¢

(1) Greedy choice :

Always selects the coin with greater denomination at first choice.

Keep adding denominations to the remaining cents while remaining value > 0

(2) optimal solution :

Once the greedy choice is made, it can yield many solutions but always consider an optimal solution with lesser number of coins.

Let,

denominations be stored in array

$d = \{25, 10, 5, 1\}$ → decreasing order.

total ⇒ be an array to count the number of coins for each denomination.

total[0] → Quarters  total[1] → dimes

total[2] → Nickel  total[3] → pennies

Value → be the amount for which we make change.

```
MakeChange ( Value)
    for i=0 to 3
        while( Value ≥ denom[i] )
            total[i] = total[i]+1
            Value = Value - denom [i]
        end while
    end for
end function MakeChange
```

1(b) Let the denominations be,

$denom = \{4, 3, 1\}$

Value (money) = 6 cents

By using greedy,

(i) considers highest denomination.

1 coin → 4 ¢

value = 6 - 4 = 2

2 coins → 1 ¢

∴ Total No of coins = 3 coins

without applying greedy algorithm, optimal solution is,

2 coins → 3 ¢

∴ Total No of coins = 2 coins

3

## Rod cutting problem

$i \rightarrow$ length of the rod

$P_i \rightarrow$ value for that length $i$

density $= P_i / i$

$=$ value per inch

$n \rightarrow$ Total length of the rod.

By using greedy strategy,

(i) For the rod of length 'n' cuts off a piece of length $i$
where $1 \leq i \leq n$ having maximum density

(ii) applies greedy strategy to the remaining piece of length $n-i$

Example where greedy strategy does NOT yield an optimal solution.

| length $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Price $P_i$ | 1 | 22 | 36 | 40 |
| density $P_i/i$ | 1 | 11 | 12 | 10 |

Rod length $= n$

$n = 4$

By greedy choice.

(i) selecting 1 with highest density.

$P_i / i = 12 \longrightarrow$ length $i = 3$

First we cut the rod of length $n = 4$ into length $3 = \underline{\$36}$

(ii) Remaining length = 1 inch
= \$1

Total amount = 36+1 = \$37

But the optimal way of cutting the rod of length $n=4$ is into 2 pieces of length $i=2$
which yields = 22 + 22
= \$44

---

4. Fibonacci numbers given by

$F_0 = 0$

$F_1 = 1$

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

Dynamic programming algorithm
Time complexity = $O(n)$
compute $n^{th}$ Fibonacci number.

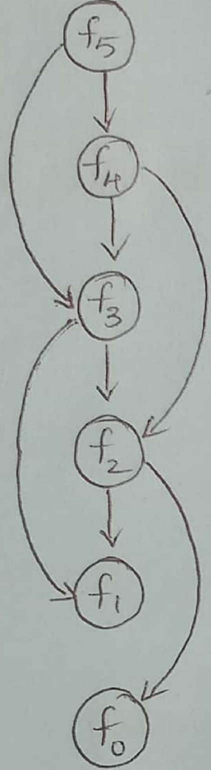
```
Fibonacci(num)

    fib[0]=0
    fib[1]=1
    for i=2 to n
        fib[i] = fib[i-1] + fib[i-2]
    return fib[num]
```

In Dynamic Programming,
store the value calculated so far and fetch when required to solve similar sub-problem.

## Sub-problem graph

when num = 5

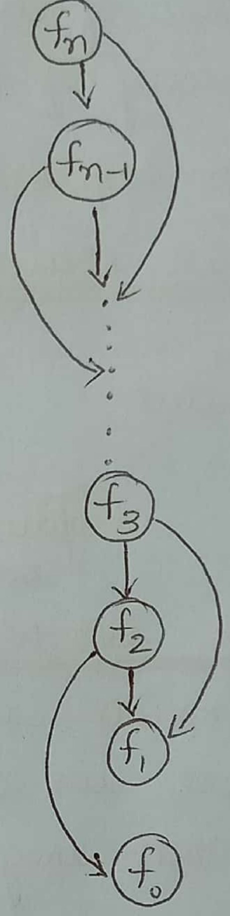



vertices = 6

Edges = 8

for num = 5

In General notation,




For $n^{th}$ fib

vertices = $(n+1)$

Edges = $(n-1) * 2$

---

(5) ## 0-1 Knapsack problem.

Dynamic programming algorithm

(1) optimal solution

(2) overlaping sub-problems

Solution for sub-problem is stored as and when it is calculated.

This stored solution can be used when same ~~po~~ sub-problem is encountered.

Given,

$n \rightarrow$ number of items ($i = 1$ to $n$)

$w_i \rightarrow$ weights associated with each item $i$

$v_i \rightarrow$ values of each item $i$

$W \rightarrow$ capacity of knapsack.

Bottom-up-approach. 2 properties of dynamic prog,

(1) <u>optimal sub-structure :</u>

case(i) $\rightarrow$ item included

item NOT included $\leftarrow$ case(ii)

optimal set

(2) <u>overlapping sub-problem</u>

$\rightarrow$ There exists sub-problems which overlap.

Hence we use an array $V[i, w]$ to store the values as and when they are calculated.

$i = 1$ to $n$ $\qquad$ $w = 0$ to $W$

$\rightarrow$ optimal solution is defined in terms of solutions to smaller sub-problems.

$$V[i, w] = \begin{cases} V[i-1, w] & \text{if } w_i > W \\ \max[\underbrace{V[i-1, w]}_{\text{Not including item}}, \underbrace{v_i + V[i-1, W-w_i]}_{\text{including that item}}] & \text{if } w_i \leq W \end{cases}$$

<u>Base case</u> :- if item $= 0$
$W$(Knapsack capacity) $= 0$ $\Big\}$ $V[i, w] = 0$.

for w from 0 to W

$V[0, w] = 0$

for i from ~~0~~ 1 to n

$V[i, 0] = 0$

for i from 1 to n

$O(nW)$ ← loop inside loop

for w from 0 to W

if ($w_i > W$) // if item weight exceeds capacity knapsack

$V[i, w] = V[i-1, w]$ // exclude that item

end if

else if ($w_i \leq W$)

if ( $v_i + V[i-1, w-w_i] > V[i-1, w]$ )

$V[i, w] = v_i + V[i-1, w-w_i]$

end if

else

$V[i, w] = V[i-1, w]$

end else

end else if

end for (w)

end for (i)

Time complexity = $\underline{O(nW)}$

$W \rightarrow$ goes from 1 to $W \rightarrow O(W)$

↳ repeated n times hence $O(nW)$

Scanned by CamScanner

item = 5 ; W = 20

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $w_i$ | 2 | 3 | 4 | 5 | 9 |
| $v_i$ | 3 | 4 | 5 | 8 | 10 |

| $w_i$ | 0 | 2 | 3 | 4 | 5 | 9 |
|---|---|---|---|---|---|---|
| W ↓ / i → | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | $\boxed{3}$ | 3 | 3 | 3 | 3 |
| 3 | 0 | 3 | 4 | 4 | 4 | 4 |
| 4 | 0 | 3 | 4 | 5 | 5 | 5 |
| 5 | 0 | 3 | 7 | 7 | 8 | 8 |
| 6 | 0 | 3 | 7 | $\boxed{8}$ | $\boxed{8}$ | 8 |
| 7 | 0 | 3 | 7 | 9 | 11 | 11 |
| 8 | 0 | 3 | 7 | 9 | 12 | 12 |
| 9 | 0 | 3 | 7 | 12 | 13 | 13 |
| 10 | 0 | 3 | 7 | 12 | 15 | 15 |
| 11 | 0 | ~~0~~3 | 7 | 12 | $\boxed{16}$ | 16 |
| 12 | 0 | 3 | 7 | 12 | 17 | 17 |
| 13 | 0 | 3 | 7 | 12 | 17 | 17 |
| 14 | 0 | 3 | 7 | 12 | 20 | 20 |
| 15 | 0 | 3 | 7 | 12 | 20 | 20 |
| 16 | 0 | 3 | 7 | 12 | 20 | 21 |
| 17 | 0 | 3 | 7 | 12 | 20 | 22 |
| 18 | 0 | 3 | 7 | 12 | 20 | 23 |
| 19 | 0 | 3 | 7 | 12 | 20 | 25 |
| 20 | 0 | 3 | 7 | 12 | 20 | $\boxed{26}$ |

→ ∵ No item of weight 1

(ignore this)

W = 20

maximum Value = <u>26</u>

Elements (items) included are,

(9, 10)

(5, 8)

(4, 5)

(2, 3)

6. <u>Stair-case problem.</u>

Each time it can climb → 1, 2 or 3 steps
$\{1, 2, 3\}$

$n$ → no of steps.

| when $n = 1$ | $n = 2$ | $n = 3$ | $n = 4$ |
|---|---|---|---|
| 1 | 1, 1 | 1, 1, 1 | 1, 1, 1, 1 |
| | 2 | 2, 1 | 2, 1, 1 |
| | | 3 | 2, 2 |
| | | | 3, 1 |
| | | | 1, 3 |

ways.

It follows fibinocci sequence. By considering it has an <u>application of fibinocci series.</u>

Climbing ($n$)

$f[1] = 1$

$f[2] = 2$ // $f(1) + 1$

$f[3] = 4$ // $f(2) + f(1) + 1$

for $i = 4$ to $n$

$f[i] = f[i-1] + f[i-2] + f[i-3]$

return $f[n]$

Time complexity = $O(n)$

Let,

$n \to$ NO of steps

$m \to$ NO of ways of climbing steps ($n$)

```
Climbing(int n, int m)

        count[ ] = new int[n]

        count[0] = 0      //If NO of steps = 0 return 0
        Count[1] = 1      //If NO of steps = 1 return 1

        for i = 2 to n
            count[i] = 0    //initializing array values
            for j = 1 to m and j<=i
                count[i]+ = count[i-j]
            end for(j)
        end for(i)

        return count[n-1]
```

O(nm)

This is for the general case of 'm' ways of climbing 'n' number of steps.

This algorithm has time complexity of $\underline{\underline{O(nm)}}$

3. Given,

Set $= \{x_1, x_2, x_3, \ldots\ldots x_n\}$ // Points on real line

To determine the smallest set of unit-length closed intervals that contains all the given points.

Step 1: Sort the elements in the set S[ ] (array) in ascending order.
such that

S[0] ≤ S[1] ≤ S[2] and so on.

Step 2: Take first element of the array and add 1 to the element to get the interval of unit-length.

Step 3: Add all the elements that fall in the closed interval from S[0] and S[0]+1 to a new array result[0] = [S[0], S[0]+1] (elements in this closed interval)

Step 4: Now to check for next interval, consider the element after the first set of unit-length closed interval as pivot element.

Step 5: Repeat the step 3 and 4 by incrementing the index of result array and add all elements falling in that [S[i], S[i]+1] closed interval and add them to result[j].

Step 6: Repeat step 5 until the set $S = \{x_1, x_2 \ldots x_n\}$

becomes empty.

step 7: index (j+1) of results array will give the number of sets of that unit-length closed interval. ~~i.e~~ ~~to~~ i.e, result[j] ← last set.

(j+1) → no of sets of unit-length interval

Pseudocode :-

```
count_small_set (S)

    sort (S)   // sort all elements in array S[]
               in ascending order. → O(nlogn)

    result[0] = { S[0], S[0]+1 }  // closed interval
                                  // of unit length.
    // result is a new array to store the
    // set of elements in closed interval of unit
    // length.

    Pivot = S[0] + 1
    i = 0;
    for j from 1 to n        → O(n)
        if ( S[j] > pivot)

            Pivot = S[j]+1
            j = j+1
            result[i] = { S[j], S[j]+1 }
        end if
    end for
    return ~~result~~ ~~[i+1]~~ (i+1)  // (i+1) ← index contains
                                          count.
end count_small_set
```

Time complexity:

sorting elements in set "s" $\rightarrow O(n \log n)$

For loop $\rightarrow$ scanning every element in set "s" $\rightarrow O(n)$

Adding elements to new result array $\rightarrow$ constant time $\rightarrow O(1)$

$$\therefore T(n) = O(n \log n) + O(n) + O(1)$$

$$\hookrightarrow = \underline{\underline{O(n \log n)}}$$

---

Set $S = \{0.8, 2.3, 3.1, 1.7, 3.6, 4.0, 4.2, 5.5, 5.2, 1.0, 3.9, 4.7\}$

$S[0] + 1 = 0.8 + 1 = 1.8$

Pivot $= 1.8$

~~result[0] = result [0] = {0.8, 1~~

Sort the set S

$S = \{0.8, 1.0, 1.7, 2.3, 3.1, 3.6, 3.9, 4.0, 4.2, 4.7, 5.2, 5.5\}$

$j = 0$

Pivot = 1.8

j = 0

result[0] = {0.8, 1.0, 1.7}

Pivot = 2.3+1 = 3.3 ⇒ {S[3], S[3]+1}

result[1] = { ~~S[~~ 2.3, 3.1}

Pivot = S[5]+1

= 3.6+1 = 4.6

result[2] = {3.6, 3.9, 4.0, 4.2}

Pivot = S[9]+1

= 4.7+1 = 5.7

result[3] = {4.7, 5.2, 5.5}

result[ ] ⟶ set of unit-length closed intervals.

count ⟵ (index+1) of result array.

index of result is 3

index+1 = 3+1 = $\boxed{4}$ ⟵ NO of sets. (count)

Scanned by CamScanner