Design & Analysis of Algorithms (Dynamic Programming)

Memoization (top-down)
vs.
Tabulation (bottom-up)

1

Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into sub-problems and stores the results of sub-problems to avoid computing the same results again.

Two main properties of a problem that suggest that the given problem can be solved using Dynamic programming are:

1) Overlapping Sub-problems
2) Optimal Sub-structure

2

Dynamic Programming

> Dynamic Programming, like Divide and Conquer, combines solutions to sub-problems.

> DP is mainly used when solutions of the same sub-problems are needed again and again.

> In dynamic programming, computed solutions to sub-problems are stored in a table so that these do not have to be recomputed. However, when required, can be reused.

> DP is not useful when there are no common (overlapping) sub-problems because there is no point storing the solutions if they are not needed again.

A DP problem can be solved using two approaches:

i) Tabulation (Bottom-up) approach

ii) Memoization (Top-down) approach

3

Memoization

In computing, **memoization** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Memoization ensures that **a function/method doesn't run for the same inputs more than once** by keeping a record of the results for the given inputs (usually in an indexed table).

4

DP - Memoization

- *Memoization* is another way to deal with overlapping subproblems in dynamic programming
  - » After computing the solution to a subproblem, store it in a table
  - » Subsequent calls just do a table lookup
- With memoization, we implement the algorithm recursively:
  - » If we encounter a subproblem we have seen, we look up the answer
  - » If not, compute the solution and add it to the list of subproblems we have seen.
- Must useful when the algorithm is easiest to implement recursively
  - » Especially if we do not need solutions to all subproblems.

5

DP - Bottom Up

```
Fib_BottomUp(n)
{
 if(n === 1 || n === 0)
   return n;

 t1 = 0;
 t2 = 1;

 for( i = 1; i <= n; i++)
  {
    fib = t1 + t2;
    t1 = t2;
    t2 = fib;
  }
 return fib;
}
```

6

Tabulation - Bottom Up

```
Fib_BottomUp(n)
{
  fib[1]=0;
  fib[2]=1;
  if(n==1|| n==2)
     return fib[n];

  for( i = 3; i <= n; i++)
     fib[i]=fib[i-1]+fib[i-2];

   return fib[n];
}
```

7

Memoization

```
// Initialize a global lookup table
 fib_tab[1]=0;
 fib_tab[2]=1;
 for (i=3; i<=n; i++)
     fib_tab[i]=NULL;

Fib_memo(n)
 {
   if(n == 1 || n==2)
      return fib_tab[n];

   if(fib_tab[n]!=NULL)
      return fib_tab [n];

   fib_tab[n] = Fib_memo(n - 2) + Fib_memo(n - 1);
   return fib_tab[n];
}
```

8

Example – Factorial Recursive

```
int factorial (n)  //n is a non-negative integer
{
 if n is 0
   then return 1;
 else
   return (factorial (n–1) * n);
}
```


Example – Factorial using Bottom up

```
int bu_factorial (n)
{
   fact_tab[0]=1;  // initialize
   if (n == 0 )   // boundary condition for recursion
      return fact_tab[n];

   for(i=1; i<=n; i++)
      fact_tab[i] = i * fact_tab[i-1];
   return fact_tab[n];
}
```

Example – Factorial using Memoization

```
// Initialize a global lookup table
for(i=1; i<=n; i++)
    fact_tab[i] = -1;  // initialize

int memo_factorial (n)
{
  if (n == 0)   // boundary condition for recursion
     return 1;
  else if (fact_tab[n] !=-1)  // check in the lookup-table
     return  fact_tab[n];    //n-th slot in the lookup-table
  else
   {
     fact_tab[n] = memo_factorial (n–1) * n;
     return fact_tab[n];  //store in lookup-table in the nth slot
    }
}
```

11

End of Lecture

12