CS207 Design and Analysis of Algorithms

Sajith Gopalan

Indian Institute of Technology Guwahati

sajith@iitg.ac.in

January 26, 2022

Dynamic Programming

Dynamic Programming

- Particularly useful for optimization problems. Solution space. Constraints. Many feasible solutions, each of which satisfies the constraints, and has a value. We wish to find one feasible solution that minimises (maximizes) value.
- Steps:
  - Formulate the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution, typically in a bottom-up fashion
  - Construct an optimal solution from computed information
- Works particularly when the following properties hold:
  - Optimal Substructure: an optimal solution to the problem contains within it optimal solutions to subproblems
  - Overlapping subproblems: recursive algorithm for the problem solves the same subproblems repeatedly

Dynamic Programming

- is not exclusively for optimization problems
- Fibonacci numbers are defined as follows recursively:
- $F(0) = 0$, $F(1) = 1$
- For $n > 1$, $F(n) = F(n-1) + F(n-2)$
- Computation of Fibonacci numbers demonstrates the concepts of Dy Programming

Fibonacci Numbers

- Has the following properties:
  - Optimal Substructure: an optimal solution to the problem contains within it optimal solutions to subproblems; $F(n)$ can be computed using $F(n-1)$ and $F(n-2)$
  - Overlapping subproblems: recursive algorithm for the problem solves the same subproblems repeatedly; $F(n) = F(n-1) + F(n-2)$; $F(n-1) = F(n-2) + F(n-3)$; $F(2)$ occurs in the computation of $F(n-1)$, and then again in the computation of $F(n)$
- Fibonacci numbers are defined as follows recursively:
- $F(0) = 0$, $F(1) = 1$
- For $n > 1$, $F(n) = F(n-1) + F(n-2)$

Fibonacci Numbers

Algorithm 1 F(n)

```
if (n = 0 or n = 1) return n;
return F(n − 1) + F(n − 2)
```

Analysis

- Let $T(n)$ denote the time complexity of a call to $F(n)$
- $T(0) = T(1) = 1$
- At each level of recursion, a constant amount of time is spent, say one unit
- So, for $n > 1$, $T(n) = T(n-1) + T(n-2) + 1$
- Claim: for $n > 0$, $T(n) = 2F(n) - 1$
- Basis: $T(1) = 1 = 2 * 1 - 1 - 2F(1) - 1$
- Hypothesis: for all m, $1 \leq m < n$ the claim holds
- Step: $T(n) = T(n-1) + T(n-2) + 1 = 2F(n-1) - 1 + 2F(n-2) - 1 + 1 = 2F(n) - 1$

Analysis

- When $\phi = \frac{1+\sqrt{5}}{2}$, $F(n) = \frac{(\phi^n - (-\phi)^{-n})}{(2\phi - 1)}$
- Therefore, Algorithm 1 runs in exponential time

Bottom-Up Algorithm

Algorithm 2 Fibonacci(n)

```
Get an array A of n locations
Set A[0] = 0, A[1] = 1
for i = 2 to n do
    A[i] = A[i − 1] + A[i − 2]
end for
return A[n]
```

Analysis

- The Algorithm 2 runs in $O(n)$ time
- Of course, we can do just the same with three variables: *present*, *previous*, *temp*
- Repeat: $temp = present + previous$; $previous = present$; $present = temp$;
- Algorithm 2, nevertheless, demonstrates the Bottom-Up approach, where an array is used to store subproblems' solutions

Memoization

- if the recursive solution must be used, then memoization is the solution
- memoization is the trick of storing the result of the first invocation of a function on each input, and returning the cached result on the subsequent invocations with the same input

Fibonacci numbers using memoization

A is a global array that is initialized to -1 in all locations

Algorithm 3 *MemoizedF*(n)

if $(A[n] = -1)$ **then**
    if $(n = 0 \text{ or } n = 1)$ **then**
        $A[n] = n$
    else
        $A[n] = MemoizedF(n-1) + MemoizedF(n-2)$
    end if
    return $A[n]$
end if

Analysis

- Let $T(n)$ denote the time complexity of a call to $MemoizedF(n)$
- $T(0) = T(1) = 1$
- At each level of recursion, a constant amount of time is spent, say one unit
- This because when $MemoizedF(n-2)$ is called, $MemoizedF(n-1)$ would have already finished
- $MemoizedF(n-1)$ has inside it an invocation to $MemoizedF(n-2)$, which is the first such invocation
- The second invocation merely does a table-lookup
- So, for $n > 1$, $T(n) = T(n-1) + 1$
- That is, $T(n) = O(n)$