Dynamic Programming

Dynamic Programming

- Many problem can be solved by D&C
  - (in fact, D&C is a very powerful approach if you generalize it since MOST problems can be solved by breaking it into smaller parts)
- However, some might show special behavior
  - Optimal sub-structures
  - Overlapping sub-problems

Dynamic Programming

- Optimal sub structure
  - "the best" of sub-solutions constitute "the best" solution
    - E.g., MCS, Closest Pair
- Overlapping sub-problem
  - Some instances of sub-problem occur several times

Optimal sub structure

- The solution to the sub-problems directly constitute the solution of the original problem
  - Finding the best solutions for sub-problems helps solving the original problem

Overlapping Sub-problem

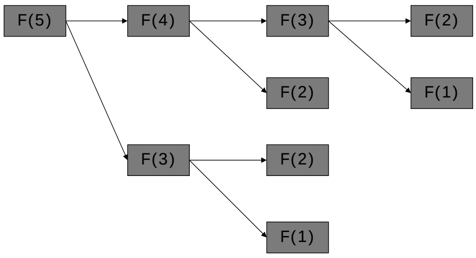
- When a sub-problem of some higher level problem is the same instance as a sub-problem of other higher level

Example Fibonacci

- Problem: compute F(N), the Fibonacci function of N

- Def: F(N) = F(N-1) + F(N-2)
- F(1) = 1
- F(2) = 1

Recursion Tree



Example

- F(1) = 1
- F(2) = 1
- F(3) = 2
- F(4) = 3
- F(5) = 5
- F(6) = 8
- F(7) = 13
- F(8) = 21
- F(9) = 34
- F(10) = 55

Example

- F(1) = 1
- F(2) = 1
- F(3) = F(2) + F(1)
- F(4) = F(3) + F(2)
- F(5) = F(4) + F(3)
- F(6) = F(5) + F(4)
- F(7) = F(6) + F(5)
- F(8) = F(7) + F(6)
- F(9) = F(8) + F(7)
- F(10) = F(9) + F(8)

Example

- F(1) = 1
- F(2) = 1
- F(3) = F(2) + F(1)
- F(4) = F(3) + F(2)
- F(5) = F(4) + F(3)
- F(6) = F(5) + F(4)
- F(7) = F(6) + F(5)
- F(8) = F(7) + F(6)
- F(9) = F(8) + F(7)
- F(10) = F(9) + F(8)

Key Idea

- If there are “overlapping” sub-problem,
  – Why should we do it more than once?
- Each sub-problem should be solved only once!!!

Dynamic Programming Method

- Top-down approach
  – Memoization
  – Remember what have been done, if the sub-problem is encountered again, use the processed result
- Bottom-up approach
  – Use some kind of “table” to build up the result from the sub-problem

Fibonacci Example: recursive

```
int fibo(int n) {
   if (n > 2) {
      return fibo(n-1) + fibo(n-2);
   } else
      return 1;
}
```

Fibonacci Example: Memoization

```
int fibo_memo(int n) {
   if (n > 2) {
       if (stored[n] == 0) {
           int value = fibo_memo(n-1) + fibo_memo(n-2);
           stored[n] = value;
       }
       return stored[n];
   } else
       return 1;
}
```

Stored is an array of size n, initialized as 0

Memoization

- Remember the solution for the required sub-problem
  - it's caching
- Need a data structure to store the result
  - Must know how to identify each sub-problem

Memoization : Defining Subproblem

- The subproblem must be uniquely identified
  - So that, when we need to compute a sub-problem, **we can lookup in the data structure** to see whether the problem is already solved
  - So that, when we solve a subproblem, **we can store the solution in the data structure**

Code Example : D&C

```
ResultType DandC(Problem p) {
    if (p is trivial) {
       solve p directly
       return the result
    } else {
       divide p into p1,p2,...,pn

       for (i = 1 to n)
         ri = DandC(pi)

       combine r1,r2,...,rn into r
       return r
    }
}
```

Trivial Case

Divide

Recursive

Combine

Code Example : Memoization

```
ResultType DandC(Problem p) {
    if (p is trivial) {
       solve p directly
       return the result
    } else {
       if p is solved
           return cache.lookup(p);
       divide p into p1,p2,...,pn

       for (i = 1 to n)
         ri = DandC(pi)

       combine r1,r2,...,rn into r
       cache.save(p,r);
       return r
    }
}
```

Look up

save

Memoization : Data Structure

- Usually, we use an array or multi-dimension array
- For example, the Fibonacci

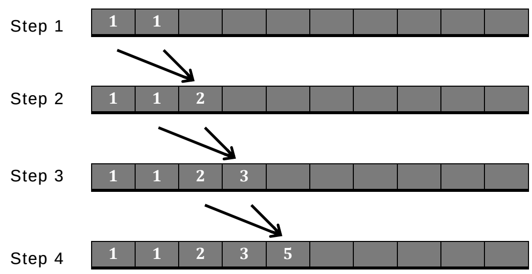
Fibonacci Example: Bottom up

- From the recurrent, we know that
  - F(n) needs to know F(n-1) and F(n-2)
  - i.e., if we know F(n-1) and F(n-2)
    - Then we know F(N)
- Bottom Up ➔ Consider the recurrent and fill the array from the initial condition to the point we need

Fibonacci Example: Bottom up

- Initial Condition:
  - F(1) = 1, F(2) = 2
  - i.e., stored[1] = 1; stored[2] = 1;
- From the recurrent
  - stored[3] = stored[2] + stored[1]
  - stored[4] = stored[3] + stored[2]
  - …

Fibonacci Example: Bottom up



Fibonacci Example: Bottom up

```
int fibo_buttom_up(int n) {
    value[1] = 1;
    value[2] = 1;
    for (int i = 3;i <= n;++i) {
        value[i] = value[i-1] + value[i-2];
    }
    return value[n];
}
```

Approach Preference

- Bottom up is usually better
  - But it is harder to figure out
- Memoization is easy
  - Directly from the recursive

Binomial Coefficient

- $C_{n,r}$ =how to choose r things from n things
  - We have a closed form solution
    - $C_{n,r} = n!/ (r!*(n-r)!)$

- $C_{n,r} = C_{n-1,r} + C_{n-1,r-1}$
- $= 1$; $r = 0$
- $= 1$; $r = n$
  - What is the subproblem?
  - Do we have overlapping subproblem?

Binomial Coefficient: sub-problem

- Described by two values (n,r)

- Data structure should be 2D array

Binomial Coefficient : Code

- Can you write the recursive version of the binomial coefficient?

- Can you change it into the memoization version?

Binomial Coefficient : Code

```
int bino_naive(int n,int r) {
	if (r == n) return 1;
	if (r == 0) return 1;

	int result = bino_naive(n-1,r) + bino_naive(n-1,r-1);
	return result;
}
```

Binomial Coefficient : Memoization

```
int bino_memoize(int n,int r) {
	if (r == n) return 1;
	if (r == 0) return 1;

	if (storage[n][r] != -1)
		return storage[n][r];

	int result = bino_memoize(n-1,r) + bino_memoize(n-1,r-1);
	storage[n][r] = result;

	return result;
}
```

Binomial Coefficient: bottom up

- Pascal Triangle