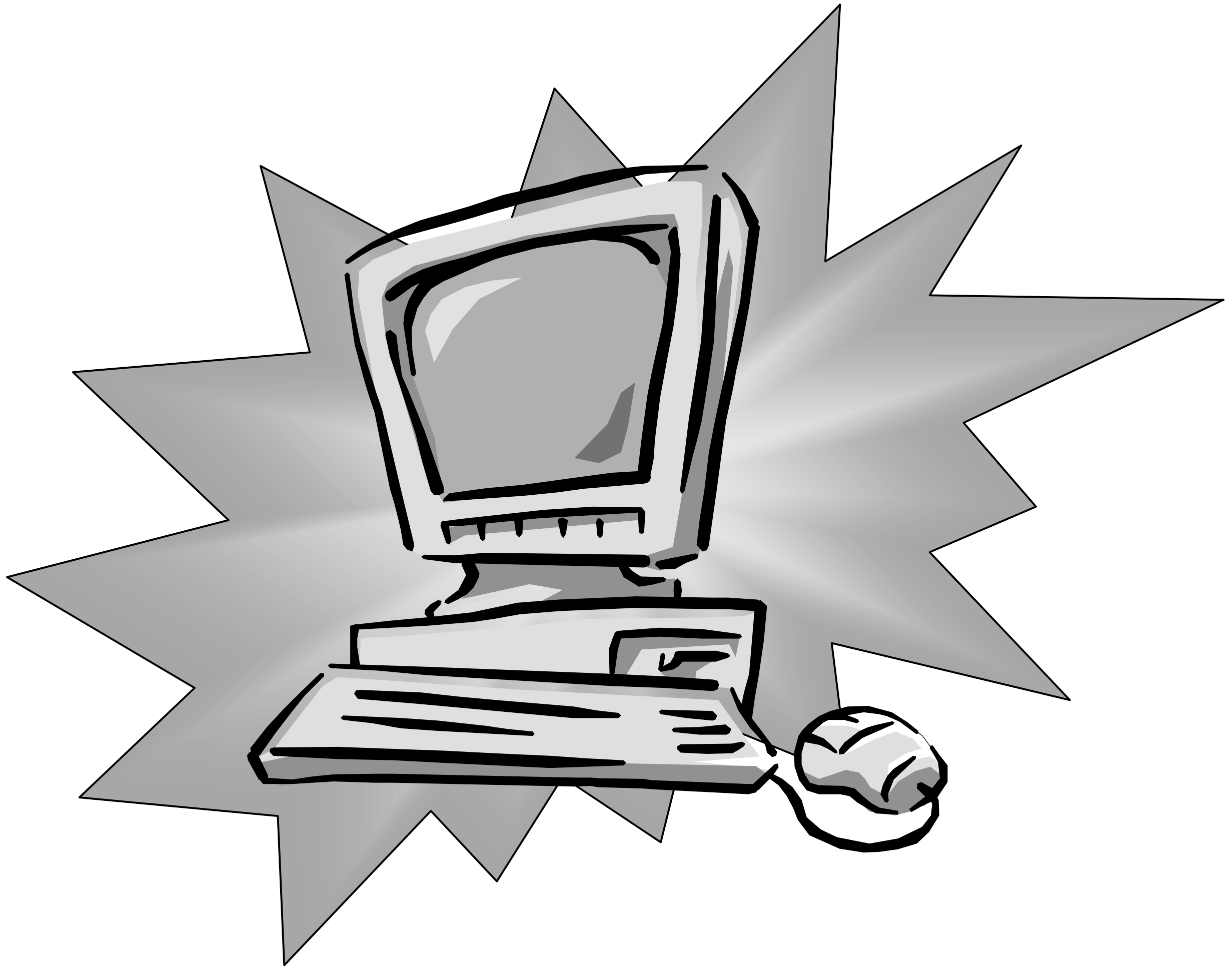
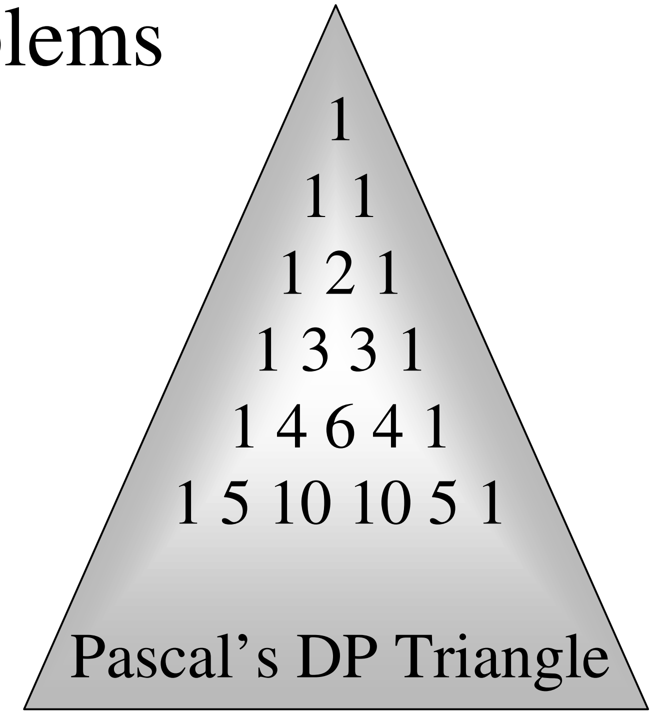
Dynamic Programming



What is dynamic programming?

‘Programming’ - a mathematical term

- Break problem into subproblems
- Work backwards
- Can use ‘recursion’



Comparison to normal recursion

- Explicitly solve subproblems first
- Avoid recalculation
- Not as flexible

Eg: Fibonacci numbers

$$f(1) = f(2) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

1 1 2 3 5 8 13 21 35…

Recursive Fibonacci

```
int fibonacci(int n)
{
        if (n <= 2)
                return 1;
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

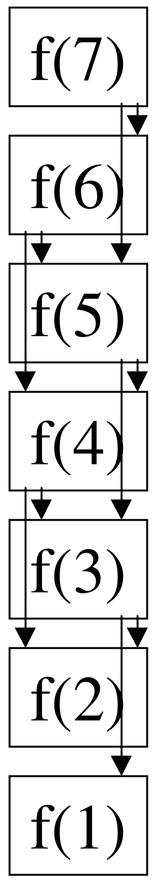
The problem

$$\begin{tikzcd}
& & & & & & & & f(7) \arrow[dlll] \arrow[drrrr] \\
& & & & & f(6) \arrow[dll] \arrow[drr] & & & & & & & f(5) \arrow[dll] \arrow[dr] \\
& & & f(5) \arrow[dl] \arrow[dr] & & & & f(4) \arrow[dl] \arrow[dr] & & & f(4) \arrow[dl] \arrow[dr] & & & f(3) \arrow[dl] \arrow[dr] \\
& & f(4) \arrow[dl] \arrow[dr] & & f(3) \arrow[d] \arrow[dr] & & f(3) \arrow[d] \arrow[dr] & & f(2) & f(3) \arrow[d] \arrow[dr] & & f(2) & f(2) & & f(1) \\
& f(3) \arrow[dl] \arrow[dr] & & f(2) & f(2) & f(1) & f(2) & f(1) & & f(2) & f(1) \\
f(2) & & f(1)
\end{tikzcd}$$

The solution: variables instead of functions

✓

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



DP: Arrays and ‘for’ loops

Golden Ratio Fibonacci

Is the DP solution always the most efficient

?

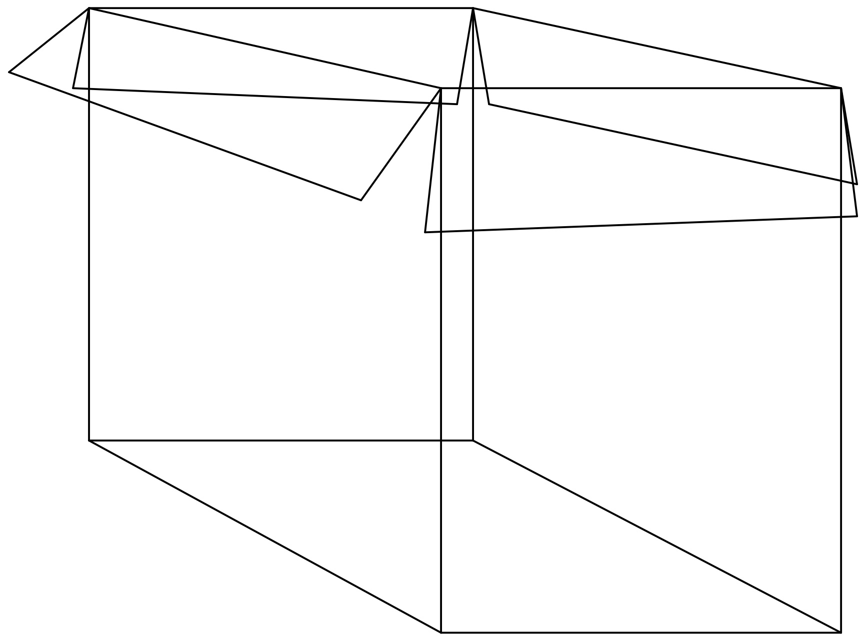
✗ Drawbacks | Advantages ✓

- Space – the ‘curse of dimensionality’
- Can be conceptually difficult

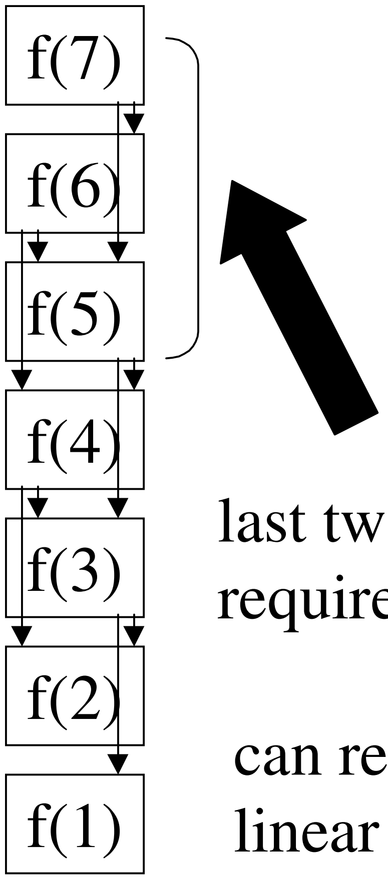
- Speed – polynomial (unlike recursion)
- Coding complexity (compared to full searches)

Problems can often be considered from different angles

Different angles – eg. knapsack problem



Reducing dimensions



last two required

Not all examples are as obvious as this one…

can reduce to constant amount of space instead of linear – removes necessity for dynamic array

Typical applications

- Find maximum/minimum possible…
- Calculate the number of ways in which…

Consider DP instead of recursion or a full search when:

- The time required is exponential and is not severely limited
- Working forwards yields too many complexities … (Guji paths…)

Eg: Subset Sums [Spring 98 USACO]

Summary: in how many ways can the set of numbers 1 to n be divided into two subsets with identical sums?

Eg with the set {1, 2, 3} there is one way: {1, 2} and {3} since (1 + 2) = (3)
Note: {3} and {1, 2} is not counted as another way…

The underlying DP problem:

The recursion

Let P[T, N] be the number of possible ways in which a total of T can be reached by adding together numbers up to and including N. (Numbers cannot be used more than once)

Thus P[T, N] = P[T, N – 1] + P[T – N, N - 1]

More specifically: if there are 2 ways of making 3 using numbers up to 4, then with the use of 5 you know there are 2 ways of making 8, and you can add this to the original 1 way you had of making 8 using numbers up to 4.

The process

	Numbers up to:
Total:	0
6	0
5	0
4	0
3	0
2	0
1	0
0	1

The process

	Numbers up to:	
Total:	0	1
6	0	0
5	0	0
4	0	0
3	0	0
2	0	0
1	0	1
0	1	1

The process

	Numbers up to:		
Total:	0	1	2
6	0	0	0
5	0	0	0
4	0	0	0
3	0	0	1
2	0	0	1
1	0	1	1
0	1	1	1

The process

	Numbers up to:						
Total:	0	1	2	3	4	5	6
6	0	0	0	1	2	3	4
5	0	0	0	1	2	3	3
4	0	0	0	1	2	2	2
3	0	0	1	2	2	2	2
2	0	0	1	1	1	1	1
1	0	1	1	1	1	1	1
0	1	1	1	1	1	1	1

You can remove a dimension by keeping a spare; but you can remove the spare if you order is correct

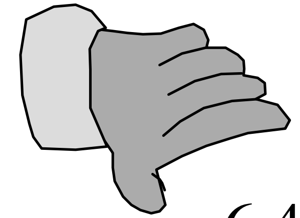
Missing unnecessary information

In the previous example we have no idea *which* combinations could produce those sums; only the *number* of possible combinations.

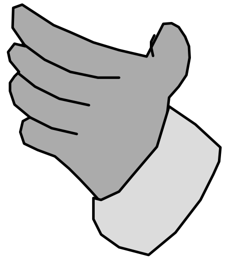
The fact that the details of the combinations were not specified as necessary for the solution is another clue pointing towards DP.

If we were required to calculate the combinations, either a full search or much more complicated DP would be necessary which kept track of all sub-possibilities…

IOI example: number game



6 46 17 31 22 41 21 43 27 8 20 26 3 16 45



Too many possibilities?

Work backwards…

Too many possibilities?

Work backwards…

21 43

Too many possibilities?

Work backwards…

21 43 27

Too many possibilities?

Work backwards…

First: A; Second: B

$6\ \boxed{46\ 17\ 31\ 22\ 41\ 21\ 43\ 27\ 8\ 20\ 26\ 3\ 16\ 45}$

First: C; Second: D

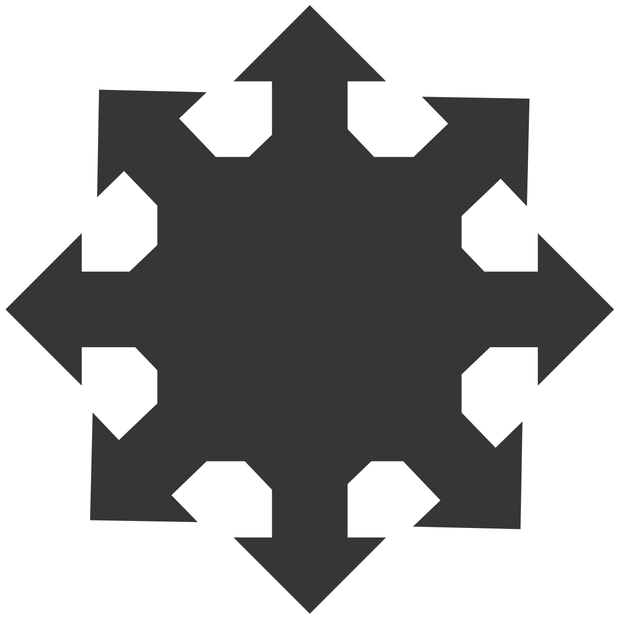
$\boxed{6\ 46\ 17\ 31\ 22\ 41\ 21\ 43\ 27\ 8\ 20\ 26\ 3\ 16}\ 45$

So…

You only need a 2D array which gives the optimum first and second player score for subsequences of different length. But this can be compressed into a 1D array and a spare, and with care the spare can be removed.

All thanks to DP!

Why not chess?



The End