Fibonacci Revisited

Consider again the recursive function for computing the nth Fibonacci number.

```
int Fibr(int n) {
  if (n <= 1) return 1;           // Base case
  return Fibr(n-1) + Fibr(n-2);    // Recursive call
}
```

Cost is Exponential. Why?

If we could eliminate redundancy, cost would be greatly reduced.

- Keep a table

```
int Fibrt(int n, int* Values) {
  // Assume Values has at least n slots, and all
  // slots are initialized to 0
  if (n <= 1) return 1;         // Base case
  if (Values[n] == 0)           // Compute and store
    Values[n] = Fibrt(n-1, Values) + Fibrt(n-2, Values);
  return Values[n];
}
```

Cost?

We don't need table, only last 2 values.

- Key is working bottom up.

Dynamic Programming

The issue of avoiding recomputation of subproblems comes up frequently.

- General solution: Store a table to avoid recomputation.
- Can work bottom up (fill table from smallest to largest)
- Can work top down (recursively), remembering any subproblems that happen to be solved (check table first).

This approach is called
Dynamic Programming

- Name comes from the field of dynamic control systems
- There, the act of storing precomputed values is referred to as "programming".

Dynamic Programming is an alternative to Divide and Conquer

- D&C: Split problem into subproblems, solve independently, and recombine.
- DP: Pay bookkeeping costs to remember solutions to shared subproblems.

A Knapsack Problem

Problem: Given an integer capacity K and n items such that item i has integer size k_i, find a subset of the n items whose sizes exactly sum to K, if possible.

Formally: Find $S \subset \{1, 2, ..., n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example:

- $K = 163$
- 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101.

What if K is 164?

Instead of parameterizing problem just by n, parameterize with n and K.

- $P(n, K)$ is the problem with n items and capacity K.

Solving the Knapsack Problem

Think about divide and conquer (alternatively, induction).

What if we know how to solve $P(n-1, K)$?

- If $P(n-1, K)$ has a solution, then it is a solution for $P(n, K)$.
- Otherwise, $P(n, K)$ has a solution $\Leftrightarrow$ $P(n-1, K-k_n)$ has a solution.

What if we know how to solve $P(n-1, k)$ for $0 \leq k \leq K$?

Cost: $T(n) = 2T(n-1) + c$.

$T(n) = \Theta(2^n)$.

BUT... there are only $n(K+1)$ subproblems to solve!

Solution

Clearly, there are many subproblems being solved repeatedly.

Store a $n \times K + 1$ matrix to contain the solutions for all $P(i, k)$.

Fill in the rows from $i = 0$ to n, left to right.

> If $P(n-1, K)$ has a solution,
> Then $P(n, K)$ has a solution
> Else If $P(n-1, K-k_n)$ has a solution
>   Then $P(n, K)$ has a solution
>   Else $P(n, K)$ has no solution.

Cost: $\Theta(nK)$.

Knapsack Example

$K = 10$.

Five items: 9, 2, 7, 4, 1.

	0	1	2	3	4	5	6	7	8	9	10
$k_1=9$	O	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	I	$-$
$k_2=2$	O	$-$	I	$-$	$-$	$-$	$-$	$-$	$-$	O	$-$
$k_3=7$	O	$-$	O	$-$	$-$	$-$	$-$	I	$-$	I/O	$-$
$k_4=4$	O	$-$	O	$-$	I	$-$	I	O	$-$	O	$-$
$k_5=1$	O	I	O	I	O	I	O	I/O	I	O	I

Key:

-: No solution for $P(i,k)$.
O: Solution(s) for $P(i,k)$ with i omitted.
I: Solution(s) for $P(i,k)$ with i included.
I/O: Solutions for $P(i,k)$ with i included AND omitted.

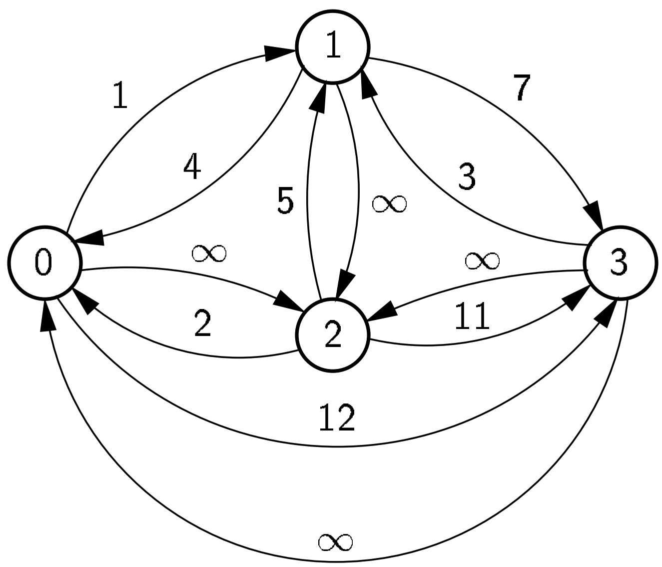
Example: $M(3,9)$ contains O because $P(2,9)$ has a solution. It contains I because $P(2,2) = P(2,9-7)$ has a solution.

How can we find a solution to $P(5,10)$?
How can we find ALL solutions to $P(5,10)$?

All Pairs Shortest Paths

For every vertex $u, v \in$ V, calculate d(u, v).

Define a **k-path** from u to v to be any path whose intermediate vertices all have indices less than k.




```
void Floyd(Graph& G) {       // All-pairs shortest paths
  int D[G.n()][G.n()];       // Store distances
  for (int i=0; i<G.n(); i++) // Initialize D
    for (int j=0; j<G.n(); j++)
      D[i][j] = G.weight(i, j);
  for (int k=0; k<G.n(); k++) // Compute all k paths
    for (int i=0; i<G.n(); i++)
      for (int j=0; j<G.n(); j++)
        if (D[i][j] > (D[i][k] + D[k][j]))
          D[i][j] = D[i][k] + D[k][j];
}
```