Dynamic Programming (DP)

- not coding
- term "Programming" refers to a tabular method (filling tables)
- optimization problems use it

- other "programming" methods in mathematical optimization are
  1) linear prog
  2) integer prog
  3) convex prog
  4) semidefinite prog

- not coding

- similar to divide and conquer
  → sub prob
  → solve sub prob recursives
  → combine soln to those.

- 03 LAB ← G
07 LAB + Tut
06 LAB ← G
CC2 quiz
1015 LAB ← G

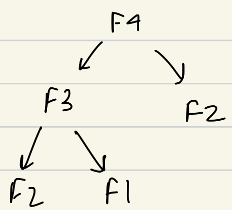
DP

- optimal substructure:
  → combo of optimal solutions to its sub problems

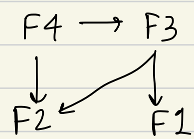
- fibonacci:
  $F_i = F_{i-1} + F_{i-2}$

- memoization: store optimal solutions to sub-problems in table (or memory or cache)
  ⇒ if sub-problems are independent, DP is not useful

DP = recursion + memoization



$\Longrightarrow$



$\Theta(2^n) \Rightarrow \Theta(n^p)$

helps transform from exponential to polynomial

- top down approach: recursion, memoize soln and reuse

- bottom up approach: figure out order of calc, solve subproblem to build up soln to larger problem

- applications:

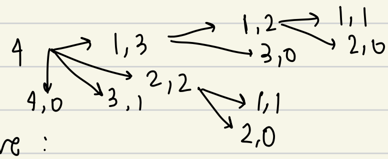
→ string algorithms
→ graph algorithms : Bellman-Ford / Floyd's
→ chain matrix multiplication

→ rod cutting
→ 0/1 knapsack
→ travelling salesman
→ subset sum ← lab / assignment.

Rod cutting Problem

- given a rod of length x and price of rod of diff len, det. max revenue by cutting the rod.

- many permutations



- naive top-down recursive :

```
cut rod (p,n) begin
 if n==0 return 0
 q ← -∞
 for i = 1 to n do
     q ← max(q, p[i] + cutrod(p, n-i))
 return q
end.
```

$\theta(2^n)$

p[i] = price of len i

q stores max price for every split and then returns it up the chain bottom up & top down have the same asymptomatic time

top down recursive w/ memoization
↑
result of each sub-problem is stored and reused

```
Cut-rod (p, n)
begin
    r[1, ..... n] ← {0}        // initialise
    return Mem-Cut-Rod-Aux (p, n, r)
end

Mem-Cut-Rod-Aux (p, n, r)
    if (n == 0)                          len is zero
        return 0
    if r[n] > 0                          the len 'n' has been solved before
        return r[n]
    else
        q ← -∞
        for i ← 1, ...., n do
            q ← max(q, p[i] + Mem-Cut-Rod-Aux (p, n-i, r))
        end for
        r[n] ← q                         update array
    end else
    return q
end fn
```

same as topdown naive

2 variable : only length is altered
subproblem has only remaining length

- bottom up w/ memoization ← not recursive

basically just filling up the table

```
DP-Cut-Rod (p, n)
begin
    r[1, .... n] ← 0
    for j = 1 to n do
        for i = 1 to j do
            r[j] ← max ( r[j] , p[i] + r[j-i])
    return r[n]
end
```

will always be already filled

$\Theta(n^2)$

p is array of prices
n is og len.
r is array that has the max price for each len.
→ each length can be divided whichever way you'd like ← no restriction on that

↑

RUN and OBSERVE

0/1 knapsack

- bag of capacity C ← constraint
- n items
  each item has a ① size (s_i)
  ② value (v_i)

  * each item has a unique size and value
  ∴ can't be repeated

- find largest total value that fits in the bag

$$\max \sum_{i=1}^{n} v_i x_i \quad : \quad \sum_{i=1}^{n} s_i x_i \leq C$$

$x_i \in \{0,1\}$ → item is chosen or not.

- brute force : $\Theta(2^n)$
- using DP

provided $j - s_i \geq 0$
else, $M(i-1, j)$

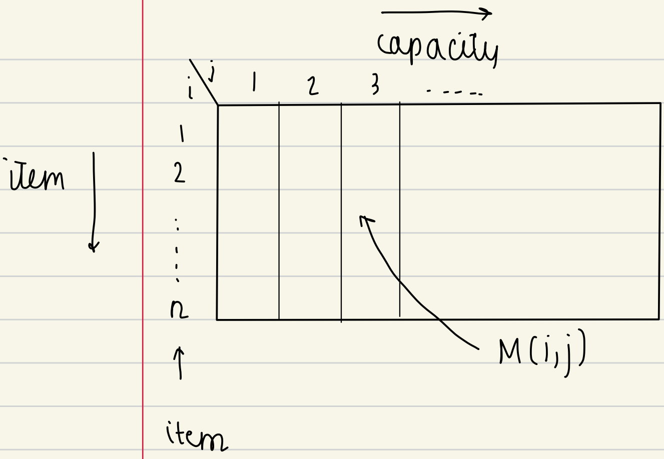
recursive formula : $M(i,j) = \max\{M(i-1,j),\ M(i-1, j-s_i) + v_i\}$

- $M(i,j)$ → computes max value
- $M(i-1,j)$ → item not used → max value chosen from $n-1$ items still fulfilling capacity j
- $M(i-1,j-s_i)$ → item used → (max chosen from $n-1$ items fulfilling capacity $j - s_n$) + value of n^{th} item

2 variables
$i : 1 \to n$
$j : 1 \to C$

→ no of available items, capacity.

→ subproblem has
a) suffix i → indicator of item
b) remaining capacity



- n x C matrix M
- bottom up approach
- time complexity : $\Theta(nC)$

- Knapsack (C, wt[], val[], n)

```
  K[n+1][C+1]        ←   n x C matrix

  for i : 0 → n]
   for w : 0 → C]
      if ( i==0 || w == 0)
         K[i][W] = 0;
      else if ( wt[i-1] <= w)
         K[i][w] = max (val[i-1] + K[i-1][w- wt[i-1]],
                        K[i-1][w]);
     else
       K[i][w] = K[i-1][W]
  end for
  end for
  return K[n][W]
end fn
```

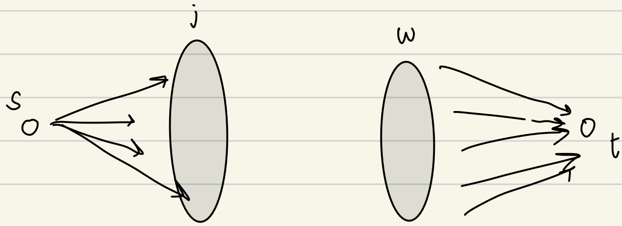
wt doesent exceed capacity

Matching Problem

- graph : set of vertexes V
  - ↙ ↘ 2 subsets
  - J W ← disjoint
- every edge connects a vertex in J to one in W
- this graph is Bipartite
- Matching prob : find a subset of edges that are mutually nonadjacent.
  - ⇓
  - no 2 edges have common endpoints
  - maximise no. of edges
- maximum matching == maximum flow from s to t

2 new vertices

source connected to every j vertex



← sink connected to every w vertex

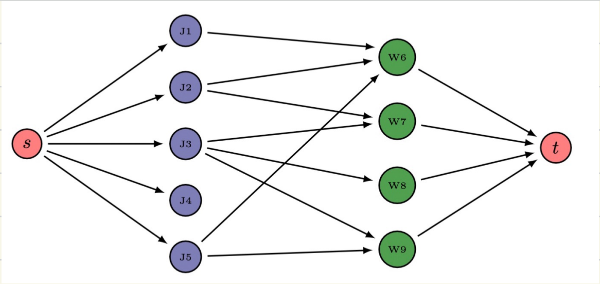
Ford - Fulkerson Method

- iterative improvement strategy
  to find an additional flow in the network
  ↳ match

a residual network is used to find the available flow

$$c_f(j,w) = c(j,w) - f(j,w)$$

residual graph — og network / graph — path we found.



- in each iteration:

→ find an 'augmenting' path p from s to t in G_f (residual network)
→ update og G by adding p → visited / traversed?
→ update G_f → reverse that path

Ford-Fulkerson

func. ford-fulkerson (Graph g, vertex s, vertex t)

for each edge $(u,v) \in E[G]$ do
    $f[u,v] \leftarrow 0$ initialising flow
    $f[v,u] \leftarrow 0$
end for

use BFS/DFS to find a path

while finding a path p from s to t in G_f do

for our ques all wts are 1

$c_{min}(p) \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$ ← i think this is finding the bottleneck

1 always for this prob

for each edge $(u,v) \in p$ do
    if $(u,v) \in E$ then
        $f[u,v] \leftarrow f[u,v] + c_{min}(p)$ add new flow
    else
        $f[v,u] \leftarrow f[v,u] - c_{min}(p)$

always

$c_f(u,v) \leftarrow c_f(u,v) - c_{min}(p)$ removes that path

$c_f(v,u) \leftarrow c_f(v,u) + c_{min}(p)$ introduces path in opp. direction

update G_f

end for
end while
end fn