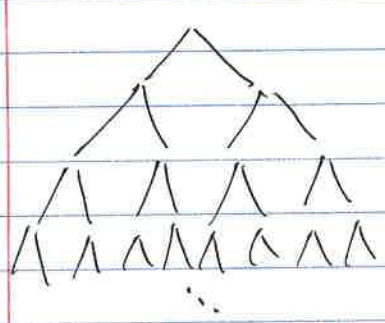
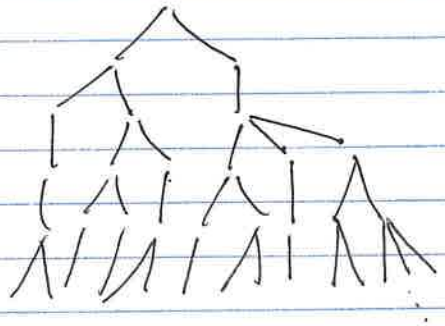
Dynamic Programming

DP is an extremely important technique. Think of it like recursion, except instead of top down, it's bottom up.





divconquer works well when the problem always divides naturally, its recursion tree has such a nice arity and structure. DP has to go bottom up. insert some base cases, decide on a recurrence, iteratively fill up a table and then return one element.

```
fib1(n)
  if n = 0,1 return n
  else return fib1(n-1) + fib1(n-2)
```

```
fib2(n)
  init dp of n+1 0's
  dp[0] = 0  dp[1] = 1
  for i in range(2, n+1):
      dp[i] = dp[i-1] + dp[i-2]
  return dp[n].
```

That's DP! It has all the telltale signs. Decide on an array, fill in base cases, choose recurrence, return one element. We will spend today only doing light problems, to dip your toes into the DP technique.

Problem: Suppose you have n stair steps. You can leap, one, two, or three steps at a time. What is the number of combinations, the number of ways you could reach the nth step?

Solution: array: dp[0...n] with dp[i] = num ways to reach step i
base cases: dp[0] = 1, dp[1] = 1, dp[2] = 2 (2 or 1+1).
recurrence: To reach step n, there was some last step. You either jumped 1, 2, or 3 steps, and you did so from 1, 2, or 3 steps away. So to go from 0 to n is the number of ways to go from 0 to n-1 and 0 to n-2 and 0 to n-3. That's just dp[n-1], dp[n-2], dp[n-3]. So the number dp[i-3] of ways to reach step i is dp[i] = dp[i-1] + dp[i-2] +

Implementation:

```
def numsteps(n):
    dp = [0]*(n+1)
    dp[0]=1, dp[1]= 1, dp[2] = 2
    for i in range(3, n+1)
        dp[i] =dp[i-1] + dp[i-2] + dp[i-3]
    return dp[n]
```

i	0	1	2	3	4	5	6	7	8	9
dp[i]	1	1	2	4	7	13	24	44	81	149

runtime: usually size of table x time per cell, so O(n).
if using fixed point arithmetic

problem: Min operations. Suppose you have only two operations. $X *= 2$ and $X += 1$. What is the minimum number of operations to go from 0 to n?

solution: array: $dp[0..n]$ with $dp[i]$ = min ops to go from 0 to i.
base cases: $dp[0]=0$, $dp[1]=1$, $dp[2]=2$, $dp[3]=3$
recurrence: to construct i, the last step was either adding one or multiplication by two. If i is odd, it had to be adding by one. If i was even, we could have multiplied by two, reaches bigger numbers faster. So

$$dp[i] = \begin{cases} dp[i-1]+1 & \text{if } i \text{ is odd} \\ dp[i/2]+1 & \text{if } i \text{ is even} \end{cases}$$

but what if $dp[i-1]+1$ is less than $dp[i/2]+1$ for i even?

implementation:

```
def minop(n)
    dp = [0]*(n+1)
    dp[0]=0, dp[1]=1
    for i in range(2, n+1)
        dp[i] = dp[i-1]+1
        if i is even
            dp[i] = min(dp[i], dp[i/2]+1)
    return dp[n]
```

i	0	1	2	3	4	5	6	7	8	9
dp[i]	0	1	2	3	3	4	4	5	4	5

runtime: n sized array, O(1) work at each level, linear time.

problem: House robber: given an array houses = [...] each of value, you want to rob them, but you can't rob the adjacent houses or alarms will go off. What is the max amount you can steal? [2, 7, 9, 3, 1] would be 2+9+1 = 12.

solution: array: dp[0...n] with dp[i] = max can steal only from houses house[0] ... house[i].

base cases: dp[0] = house[0], dp[1] = max(h[0], h[1])

recurrence: at the next house visited, he can choose to rob or not rob, which ever maximizes, so

$dp[i] = \max(dp[i-2] + h[i], dp[i-1])$.

choosing the previous house means you can't choose this one, but choosing this one means you can choose two houses back.

implementation:

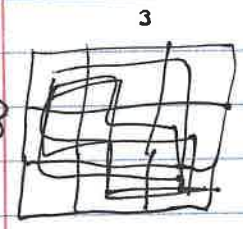
```
def houserobber(houses h[])
    dp = [0] * (n+1)
    dp[0] = h[0], dp[1] = max(h[0], h[1])
    for i in range 2...n+1
        dp[i] = max(dp[i-2] + h[i], dp[i-1])
    return dp[n]
```

h = [2, 7, 9, 3, 1]

dp[i] = [2, 7, 11, 10, 12]

runtime: O(n) element array with O(1) work at each so O(n) time.

problem Given n, m. what is the number of paths through an n×m grid from (1, 1) to n×m if you can only go down and right.→

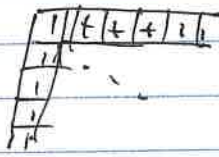


3×3 = 6 paths. Can be solved without DP, just some combo, but lets do it this way.

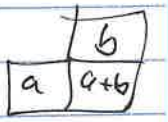
$\frac{(n+m)!}{n!\,m!}$

array: dp[0...n][0...m] with dp[i][j] = num paths to (1,1)→(i,i)

base cases: dp[0:n][1] and dp[1][0:m] = 1



Recurrence: consider the last step. It was either a D or a R. So its the number of paths coming from those directions, so $dp[i][j] = dp[i-1][j] + dp[i][j-1]$



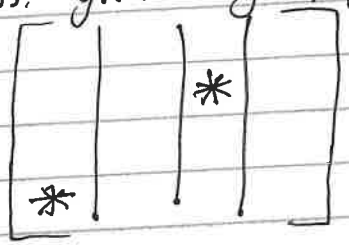
implementation:

```
def numpaths(i,j):
    dp = [1...n][1...m] all zeroes
    for i in range n  dp[i][1] = 1
    For i in rage m  dp[1][i] = 1
    for i in rangen (1...n):
        for j in rangem (1...m):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[n,m]
```

1	1	1	1	1	1	1
1	2	3	4	5	6	7
1	3	6	10	15	21	28
1	4	10	20	35	56	84
1	5	15	35	70	128	210
1	6	21	56	128	256	468
1	7	28	84	212	468	936
1	8	36	120	332	800	1736
1	9	45	165	497	1297	3073

runtime is O(n) by O(m) with O(1) work each level so O(nm).

problem: number of paths, but with bombs! given a grid, give number of paths but avoid bombs.



can't take a path over a bomb.
similar solution

solution array: $dp[1..n][1..m]$ with $dp[i][j]$ = num paths (i,j) that avoid bombs.
base cases: $dp[i][j] = 1$ if no bomb before. otherwise zero. bombs.
~~recurrence~~ $dp[i][i] = 1$ same.
recurrence:

if bomb[i-1][j] and bomb[i][j-1]
~~dp[i][j]~~ $dp[i][j] = 0$
if bomb[i-1][j]
$dp[i][j] = dp[i][j-1]$
if bomb[i][j-1]
$dp[i][j] = dp[i-1][j]$
if bomb[i][j], $dp[i][j] = 0$.

	1	1	*	0
1	2	*	0	0
1	3	3	*	0
*	3	6	6	6
0	3	9	13	19
0	3	12	25	44

Although DP may appear as a time-space tradeoff, go from exponential time and $O(1)$ space to $O(poly)$ time and $O(poly)$ space, it's not really a tradeoff as you can usually get the space back. Here is a Fibonacci example with only 3 variables. You only need space for the current and last two answers, not all previous n. This is bad coding advice though so don't do it. Just so there is no "tradeoff"

```
fib3(n)
  prev = 1
  curr = 0
  for i = 1..n
    next = prev+curr
    prev = curr
    curr = next
  return curr
```