Decomposing into subproblems

Disjoint subproblems — Divide & Conquer

Inductive structure, overlapping subproblems

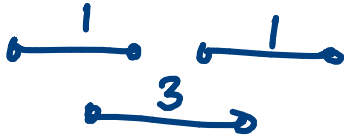
Memoization / Dynamic Programming

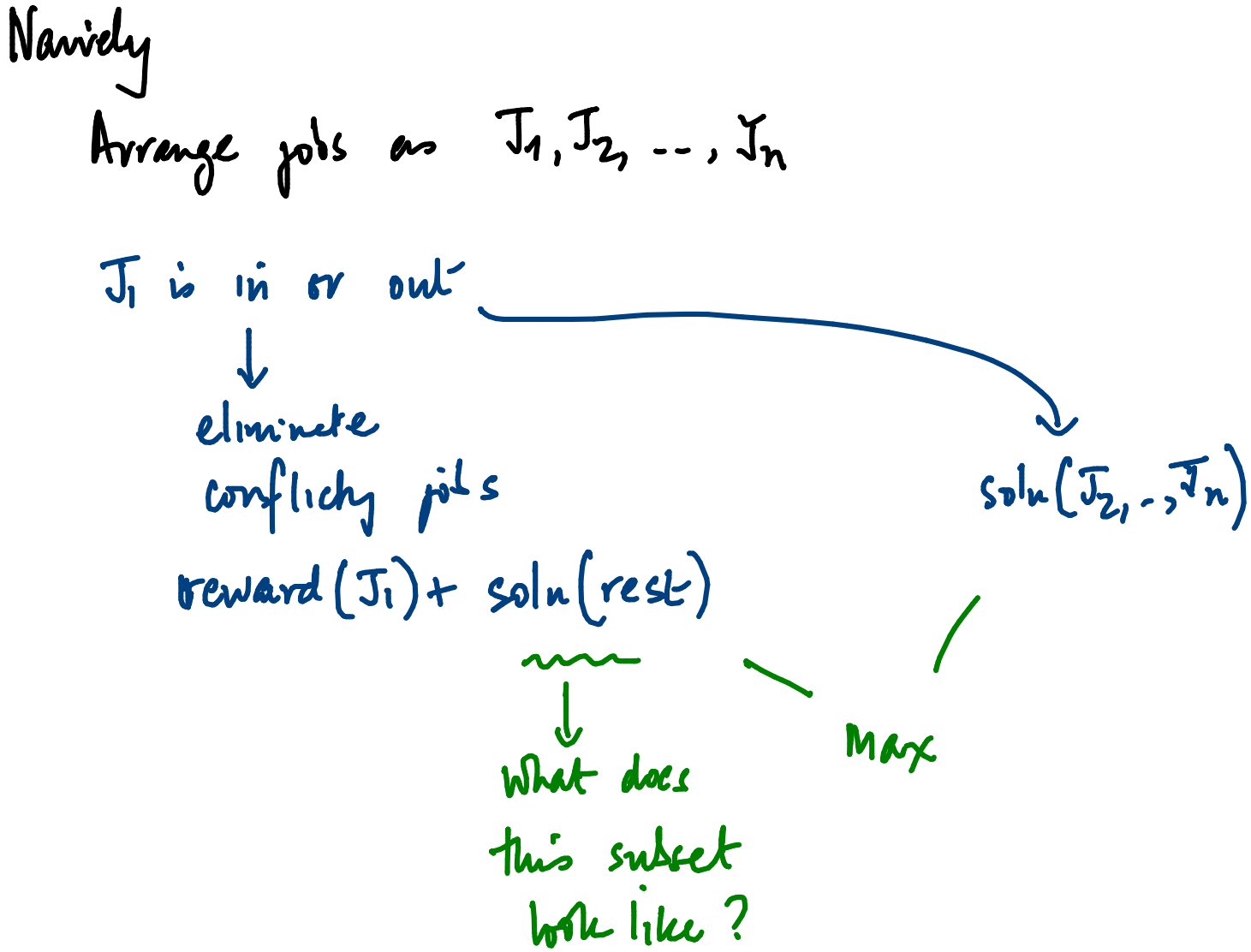
Efficient ways of handling recursion

Weighted Interval Scheduling

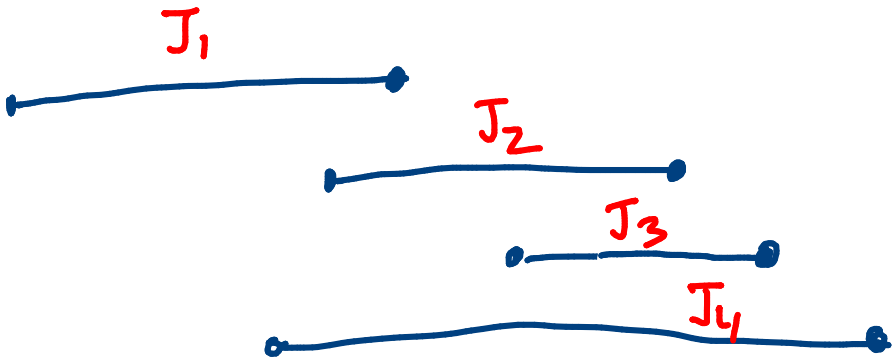
Each job has start, finish times and a reward

Select subset to maximize reward



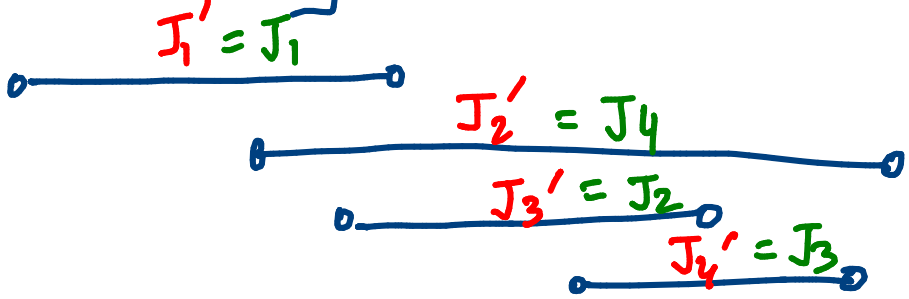

Naively
Arrange jobs as $J_1, J_2, \ldots, J_n$
J_1 is in or out
eliminate conflicty jobs
reward(J_1) + soln(rest)
soln($J_2, \ldots, J_n$)
Max
What does this subset look like?

Arrange in sorted order by earliest finish time



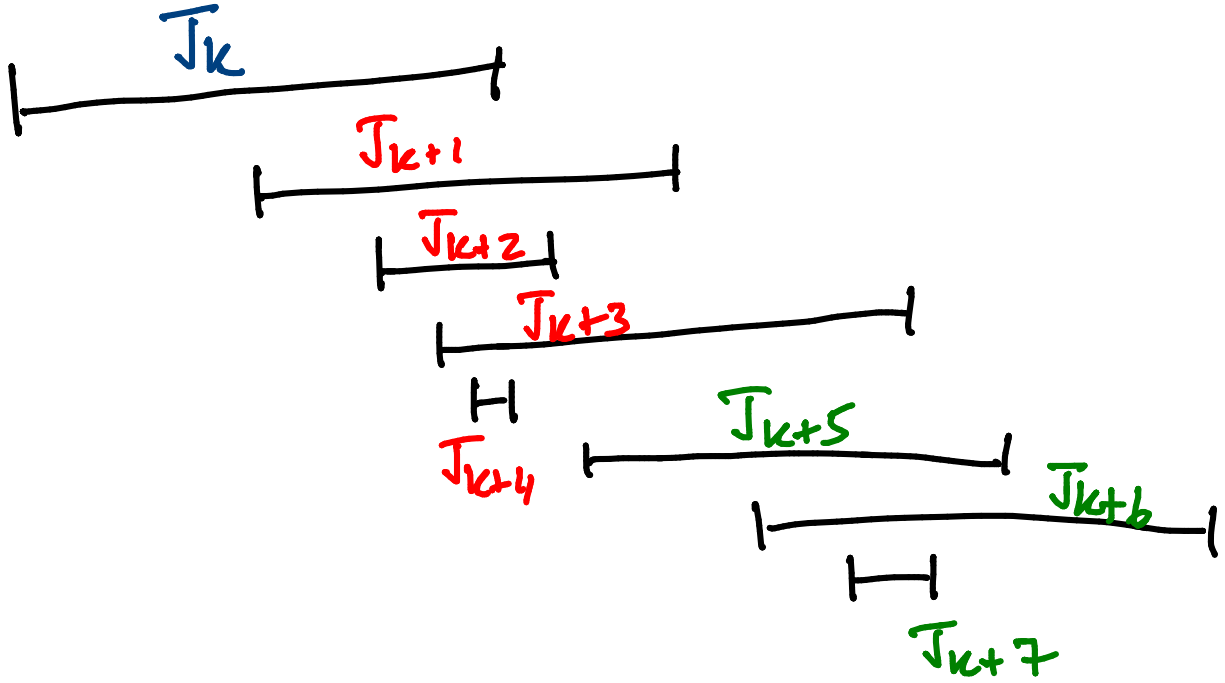
J_1 conflicts with? J_2, J_4

Instead sort by start time



Soln(k) stands for optimal soln for $J_k, J_{k+1}, \ldots, J_n$

$$\text{Soln}(k) = \max \begin{cases} \text{keep } J_k, \text{ drop conflicting jobs} \\ \text{drop } J_k, \text{ solve } k+1 \ldots n \end{cases}$$



note: Next(n) not needed

Define Next(k) = min ℓ s.t. Start(ℓ) $\geq$ Finish(k)

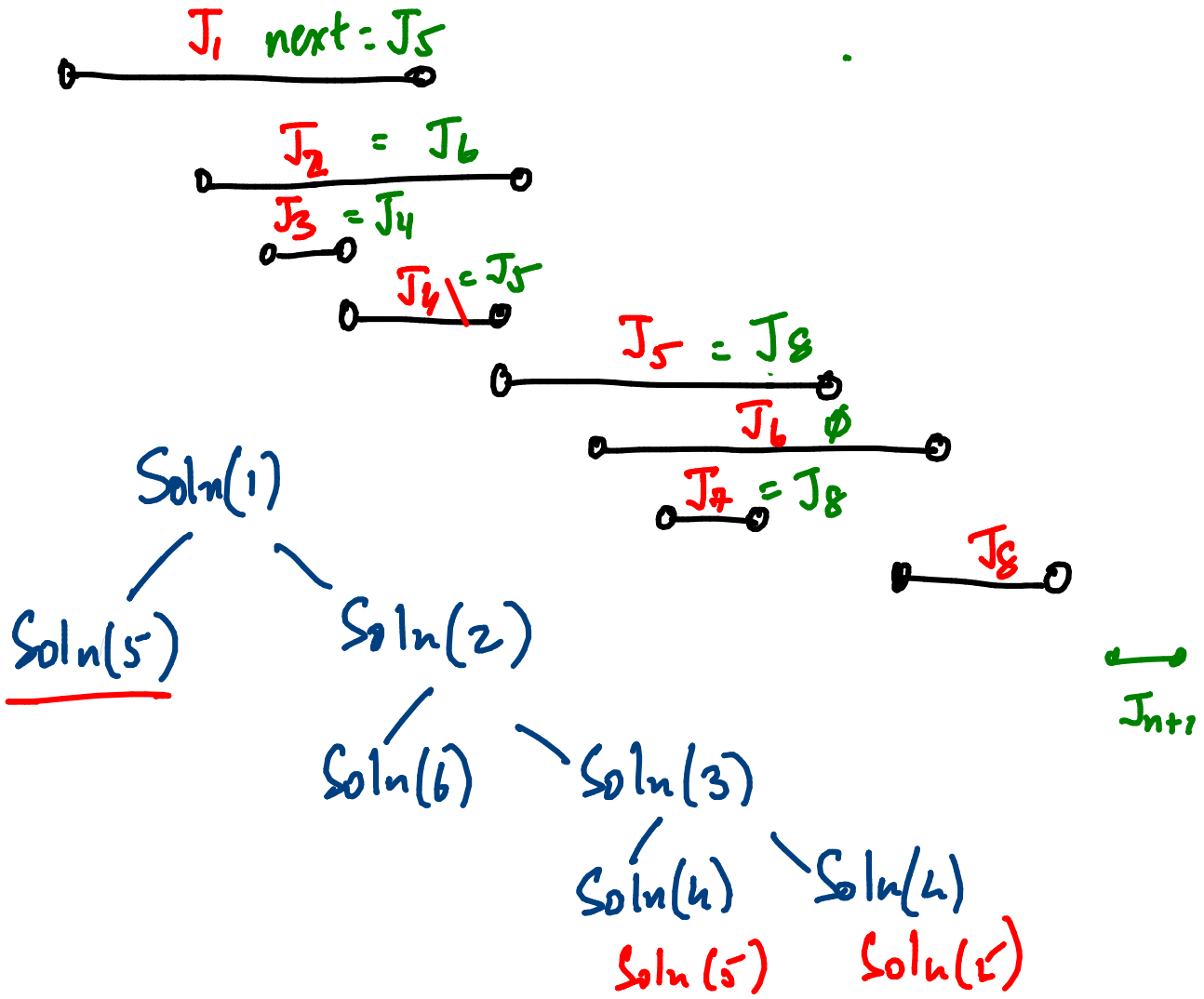
Compute using binary search after sorting by Start(i)

$O(n \log n)$ Sorting

$O(n \log n)$ compute Next(k)

$$Soln(k) = \max \begin{cases} reward(k) + Soln(Next(k)) \\ Soln(k+1) \end{cases}$$

$$Soln(n) = reward(n)$$


J1 next: J5
J2 = J6
J3 = J4
J4 = J5
J5 = J8
J6 ϕ
J7 = J8
J8
Jn+1
Soln(1)
Soln(5)
Soln(2)
Soln(6)
Soln(3)
Soln(4)
Soln(4)
Soln(5)
Soln(5)

Soln(1) depends on Soln(2), Soln(5)

⋮

Dependencies are acyclic ⟶ DAG

Want to evaluate Soln(j) in an order

compatible with DAG

- topological sort

Observe then $J_n, J_{n-1}, \ldots J_1$ always works for this problem

Dynamic Programming

Dynamic Programming

Inductive soln to the given problem

Subproblem dependencies form a DAG

Solve subproblems respecting DAG structure

Alternative strategy

Inductive soln → recursive implementation

Tabulate all recursive calls "remember"

Memoization "Memo table"

Canonical example:

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

$fib(0) = 0$

$fib(1) = 1$

0, 1, 1, 2, 3, 5, 8, 13, ...

```
def fib(n):
    if (n == 0):
        return(0)
    if (n == 1):
        return(1)
    return (fib(n-1) + fib(n-2))
```

```
def memofib(n, table):
    if (table[n] >= 0):
        return (table[n])
    if (n==0):
        table[0] = 0
        return (0)
    if (n==1):
        table[1] = 1
        return (1)
    val = memofib(n-1) + memofib(n-2)
    table[n] = val
    return (val)
```

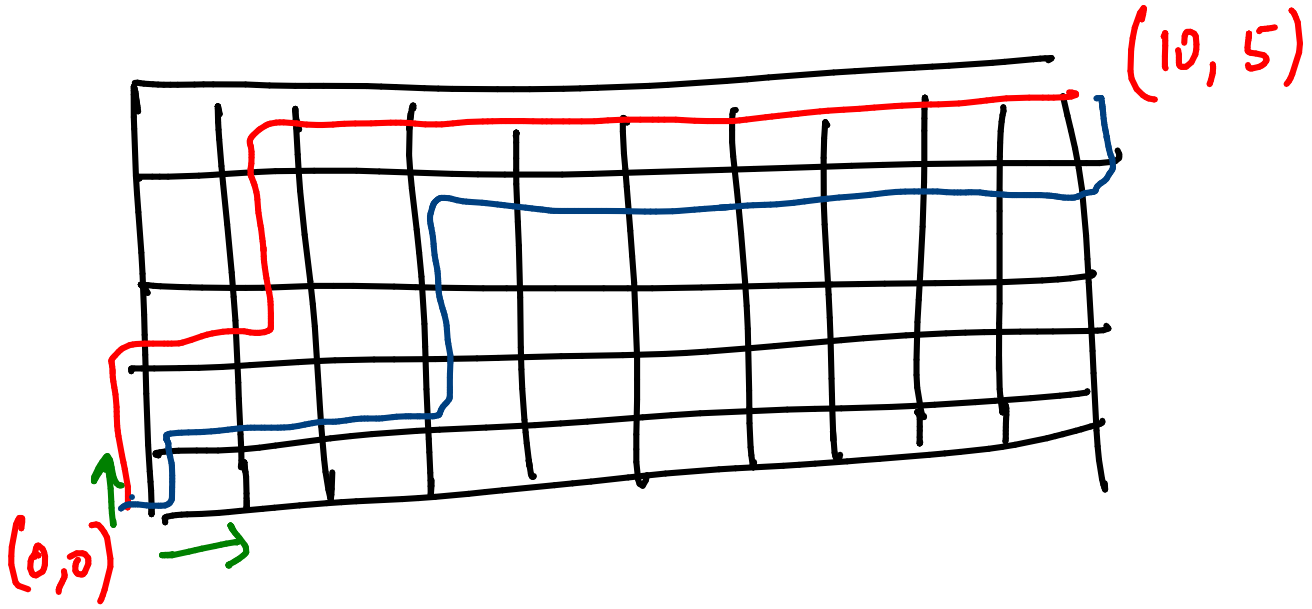
Assume table initialized to -1

Dynamic programming (version 2)

Fill up memo table in a "sensible" order

"Bottom up" evaluation of recursive program

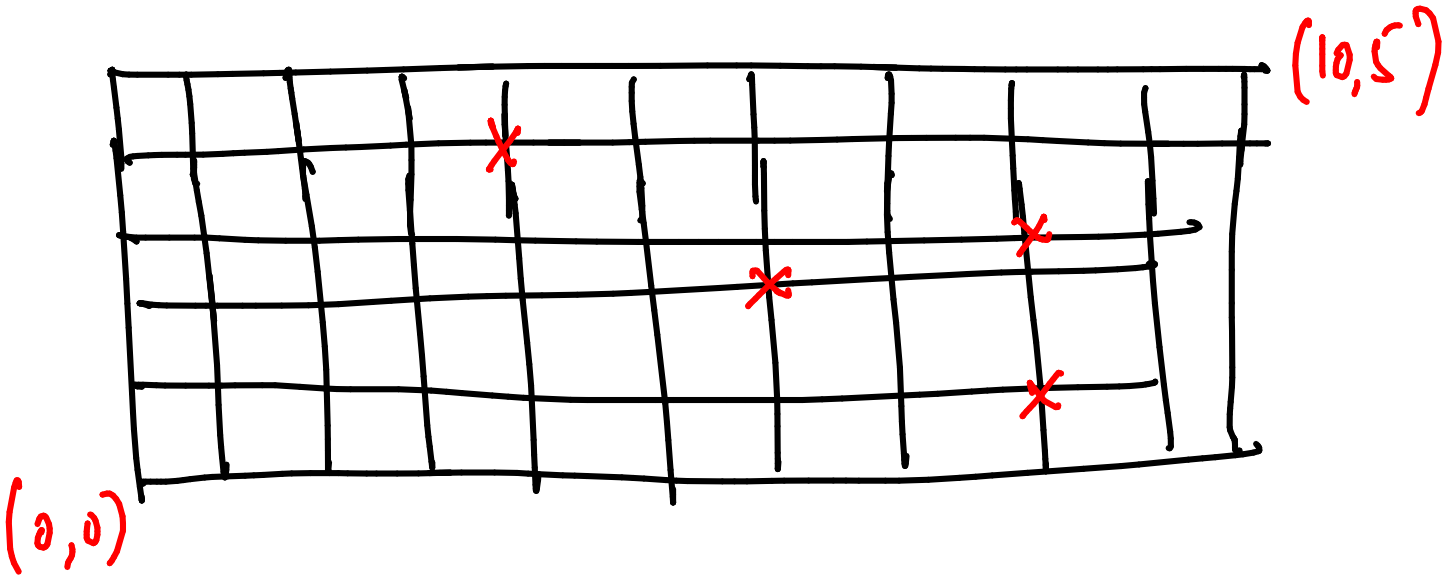
Grid Paths



How many paths from (0,0) to (10,5)?

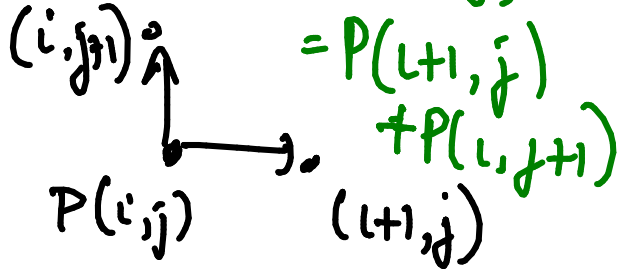
$$\binom{15}{5} = \binom{15}{10}$$

Blocked intersections



Using $\binom{n}{k}$, inclusion-exclusion

Instead $P(i,j)$ = # paths from (i,j) to $(10,5)$



$P(i,j) = P(i+1,j) + P(i,j+1)$

Boundary conditions etc

(0,0) to (m,n)

Check in this order

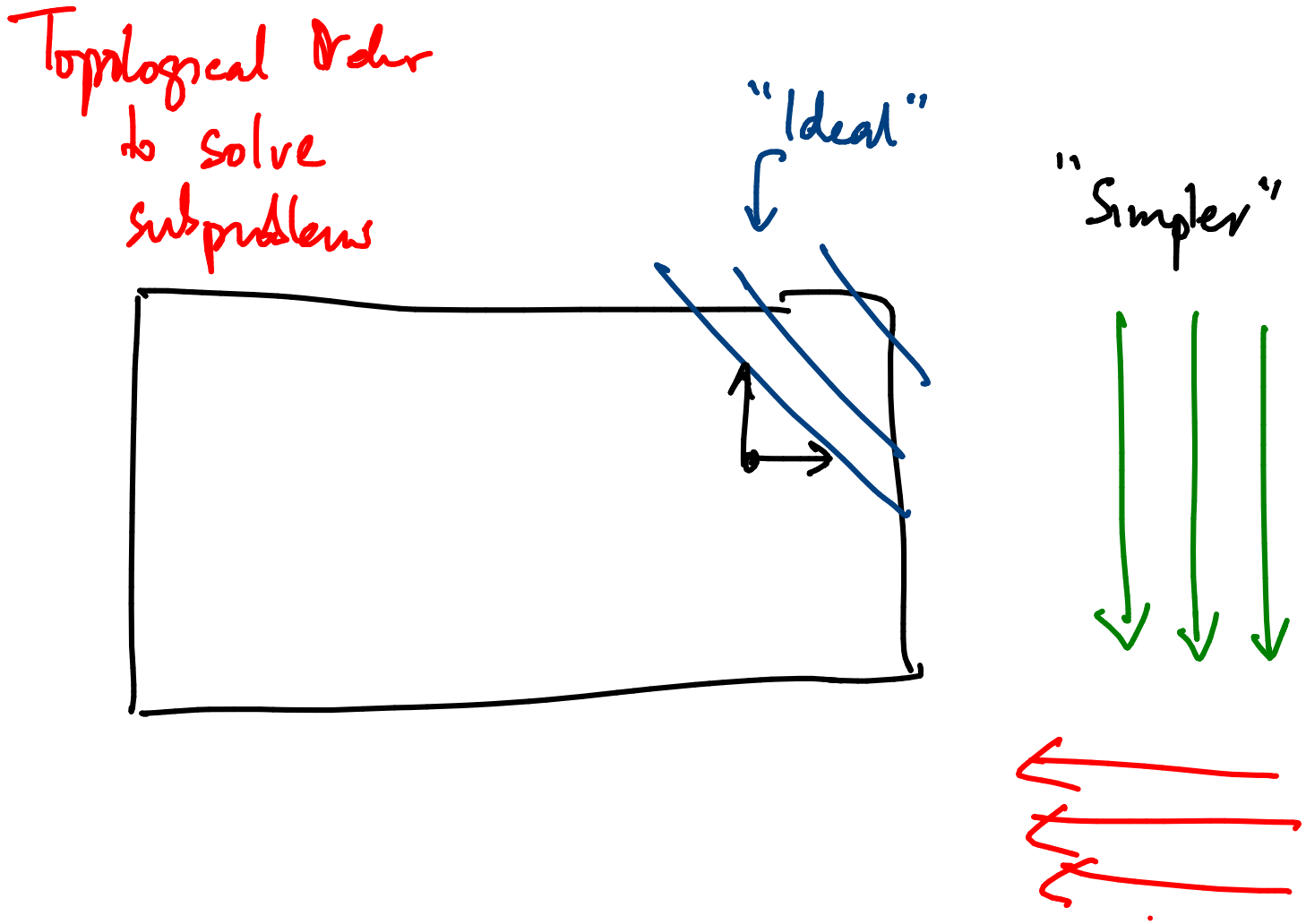
$P(m,n) = 1$

$P(i,j) = 0$ if (i,j) is blocked

$P(i,n) = P(i+1,n) \quad 0 \le i < m$

$P(m,j) = P(m,j+1) \quad 0 \le j < n$

$P(i,j) = P(i+1,j) + P(i,j+1)$


Topological Order
to solve
subproblems
"Ideal"
"Simpler"

Memoization O(m+n)

DP O(mn)