# Algorithms - Spring '25

Dynamic Programming:
LIS

Recap

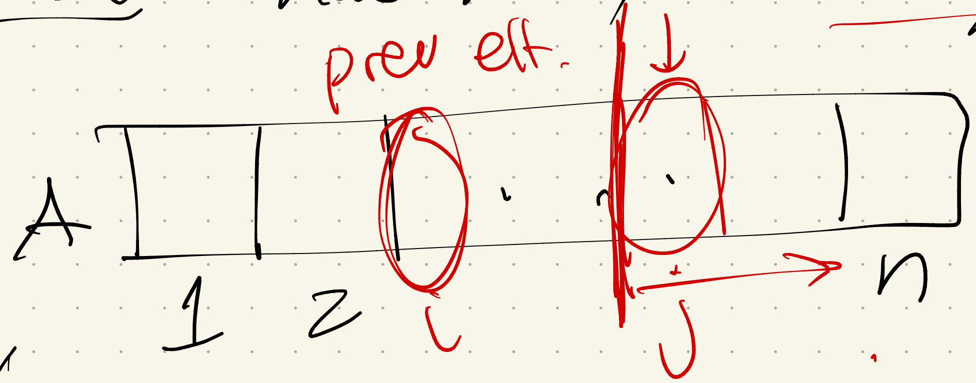
- HW0 grades is posted
  - ↳ Regrade: email to me
- Next HW due Monday
- Reading on Monday

# Recap: Longest Increasing Subsequence

Why "jump to the middle"?

Need a recursion!

First: how many subsequences?

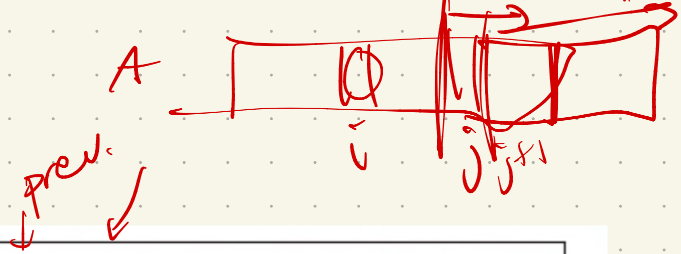



↳ could use or skip each #, so $\leq 2^n$ worst case

Backtracking approach:

At index $j$: decide if we can include $j$. If so, try both with & w/out $A[j]$

# Result:




> Given two indices $i$ and $j$, where $i < j$, find the longest increasing subsequence of $A[j\,..\,n]$ in which every element is larger than $A[i]$.

Store last "taken" index $i$.

Consider including $A[j]$:

- if $A[i] \geq A[j]$
  ↳ must skip!
- if $A[i]$ is less:
  try both options

# Recursion:

skip $j$ & move to $j+1$

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \boxed{\text{if } A[i] \geq A[j]} \\ \max \begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Include $A[j]$   skip $A[j]$

Code version: (helper function)

```
LISBIGGER(i, j):
    if j > n
        return 0
    else if A[i] ≥ A[j]
        return LISBIGGER(i, j + 1)
    else
        skip ← LISBIGGER(i, j + 1)
        take ← LISBIGGER(j, j + 1) + 1
        return max{skip, take}
```

must skip A[j]

try both

Problem - what did we want??

LIS(A[1..n]).

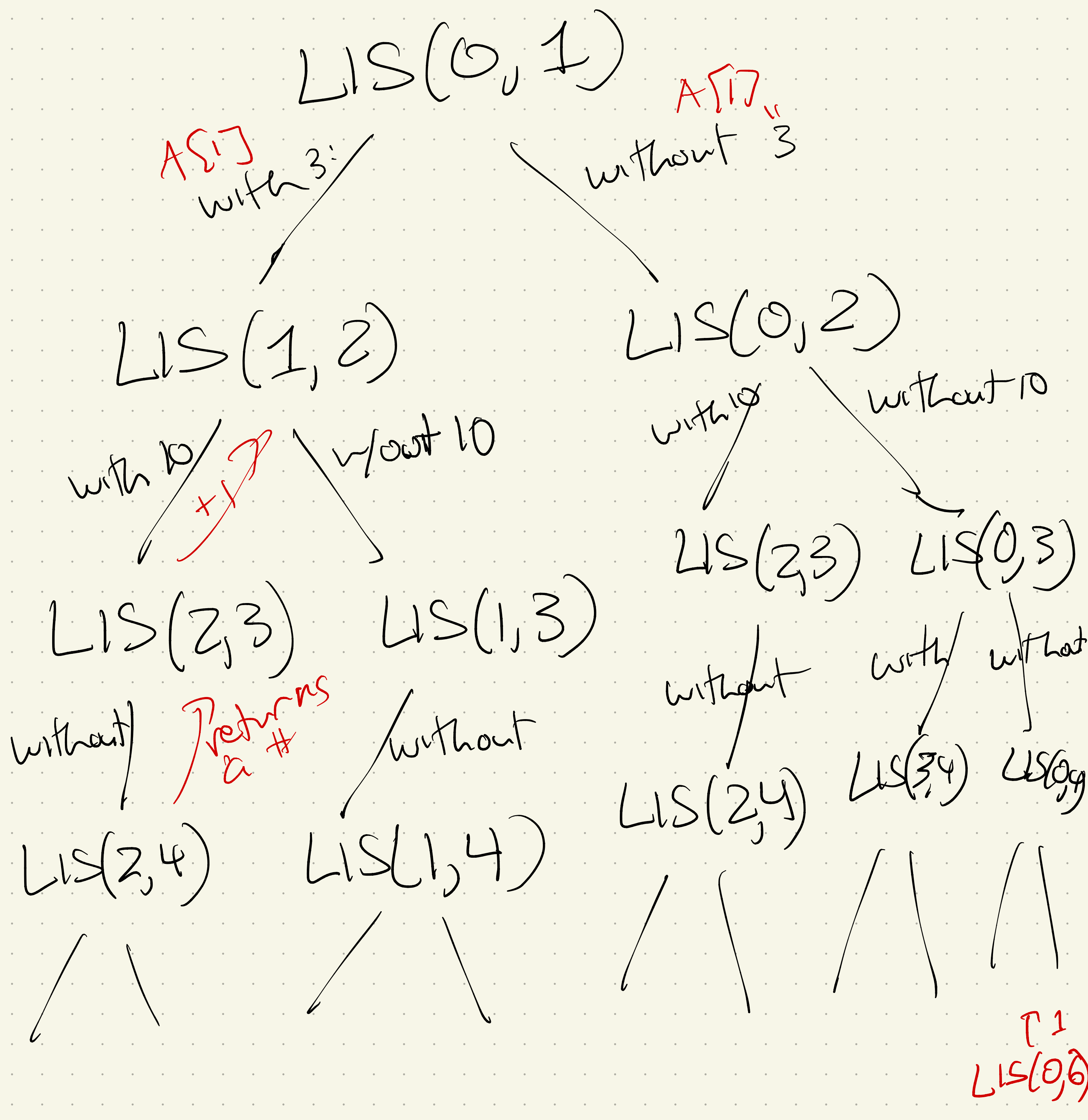
So: don't forget our "main":

```
LIS(A[1..n]):
    A[0] ← −∞
    return LISBIGGER(0, 1)
```

Example: A: [3, 10, 2, 11, 5, 7]

↳ [$-\infty$, 3, 10, 2, 11, 5, 7]

dummy at A[0]



Next: memoize?

What sort of calls are we making often?
Can we save them, & avoid recomputing over and over?

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \begin{Bmatrix} LISbigger(\underline{i, j+1}) \\ 1 + LISbigger(\underline{j, j+1}) \end{Bmatrix} & \text{otherwise} \end{cases}$$

```
LISBIGGER(i, j):
    if j > n
        return 0
    else if A[i] ≥ A[j]
        return LISBIGGER(i, j + 1)
    else
        skip ← LISBIGGER(i, j + 1)
        take ← LISBIGGER(j, j + 1) + 1
        return max{skip, take}
```

LISbigger(i, j)

$\hookrightarrow$ store these

Here:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{cases} & \text{otherwise} \end{cases}$$

This is a recursion, but think for a moment of it as a function.

After computing, store values!

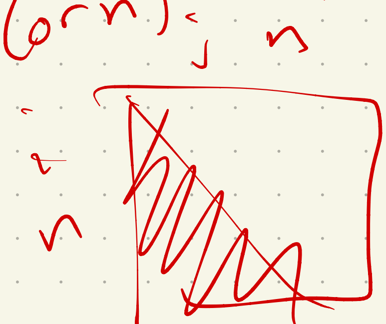
How many values to store?

LISbigger(i,j)

for i = 0 .. n-1 (or n), j = 1 .. n

where i < j

n×n array




$O(n^2)$

How long to compute each?

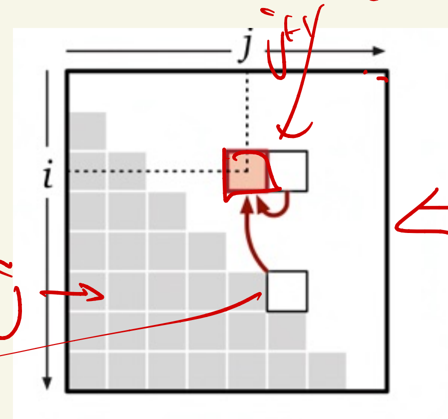
$O(1)$ time

Now, can we do the same trick as Fibonacci memoization, & convert to something loop-based?

Rethink:

To fill in L[i][j], what do I need?

without: i, j+1




So, go in that order!

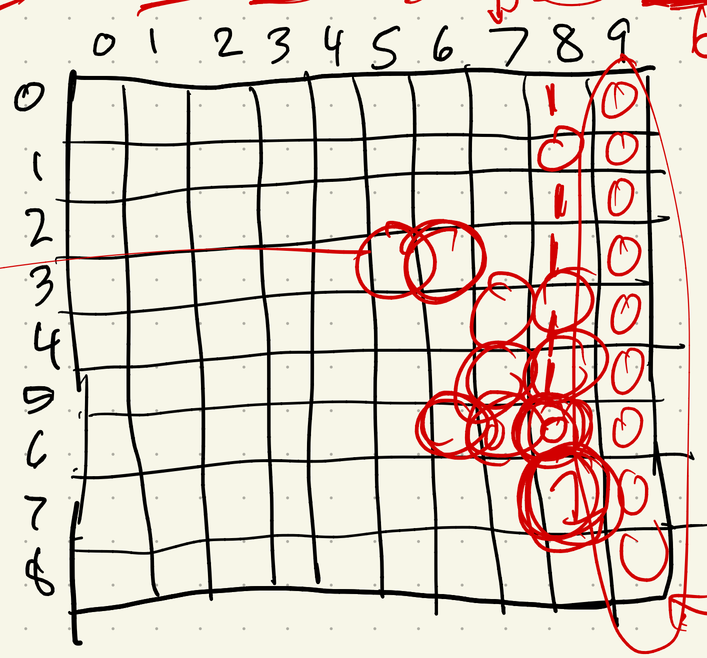
with j: j, j+1

Ex: A = [10 2 4 1 6 11 7 9]

$-\infty$

6 7 8 9

n=8




LIS(3,5)

LIS(6,7

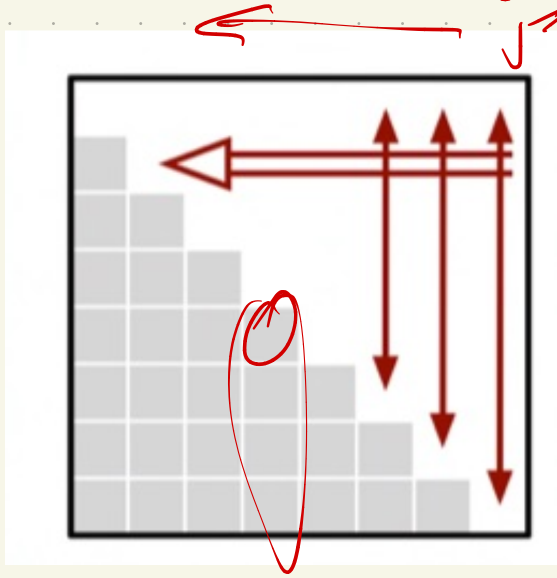
LIS(8,9)

# Result:

```
FastLIS(A[1..n]):
  A[0] ← −∞                              ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                         ⟨⟨Base cases⟩⟩
      LISbigger[i, n+1] ← 0
  for j ← n down to 1
      for i ← 0 to j − 1                 ⟨⟨... or whatever⟩⟩
          keep ← 1 + LISbigger[j, j+1]
          skip ← LISbigger[i, j+1]
          if A[i] ≥ A[j]
              LISbigger[i, j] ← skip
          else
              LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

# Picture:

j=n



# Edit distance:

**HUGE** in bioinformatics!

One of the basic tools in sequence alignment.

(I have a book with an entire chapter on how to optimize.)

Also: spell checkers, word prediction, etc.

How to begin? (Recursively!)

ALGORITM

ALTRUISTIC

Start at end, + ask "obvious" question:

insert, delete, edit

try them all!

STOP

STOOP

edit

insert

instead;

insert(0)

STO P

STO +, O P

Let's try:

A: ALGORITM

B: ALTRUISTIC

Start at end:

Align: edit A[m] to B[n]

M +1 C

(or +1 if =)

Insert:

M

— +1 C

Delete:

— — M

— — C

—

Example: TGCATAT to ATCCGAT

TGCATAT
↓ delete last T
TGCATA
↓ delete last A
TGCAT
↓ insert A at the front
ATGCAT
↓ substitute C for G in the third position
ATCCAT
↓ insert a G before the last A
ATCCGAT

TGCATAT
↓ insert A at the front
ATGCATAT
↓ delete T in the sixth position
ATGCAAT
↓ substitute G for A in the fifth position
ATGCGAT
↓ substitute C for G in the third position
ATCCGAT

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | T | G | C | - | A | T | A | T |
| A | T | C | C | G | A | T | - | - |
| +1 | +0 | +1 | +0 | +1 | +0 | +0 | +1 | +1 |

= 5

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| - | T | G | C | A | T | A | T |
| A | T | C | C | G | - | A | T |
| +1 | +0 | +1 | +0 | +1 | +1 | +0 | +0 |

↳ cost 4

Input: $A[1..m]$
$B[1..n]$

$$\text{Edit}(\;,\;) = \min \left\{ \begin{array}{l} \\ \\ \\ \end{array} \right.$$

+ Base cases!

this way:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

So: what's our "memory" data structure?

Then, our algorithm:

- start w/ base case (row & column)
- Fill in:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

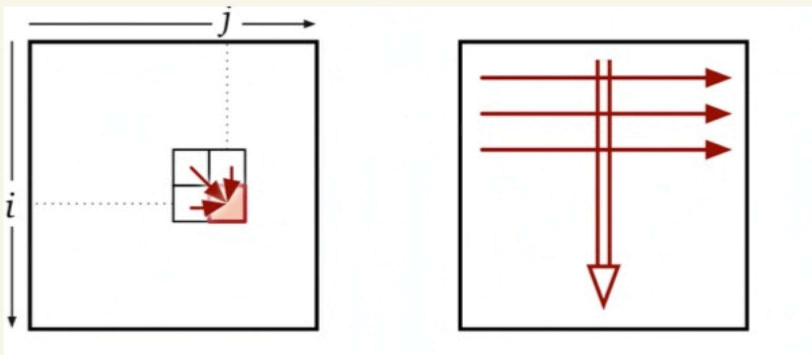
Result:

```
EditDistance(A[1..m], B[1..n]):
    for j ← 0 to n
        Edit[0, j] ← j

    for i ← 1 to m
        Edit[i, 0] ← i
        for j ← 1 to n
            ins ← Edit[i, j − 1] + 1
            del ← Edit[i − 1, j] + 1
            if A[i] = B[j]
                rep ← Edit[i − 1, j − 1]
            else
                rep ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min {ins, del, rep}

    return Edit[m, n]
```

Picture:



Question:

Can we do better?

A really good question!

Lots of attention in bioinformatics.

Clever divide an conquer can reduce space.

↳ but will give #, not sequence, w/out some nice tricks

# Subset sum (revisited)

Key takeaway (I think):
Sometimes, our backtracking recurrences can be memoized
(Note: sometimes, they can't!
Think $n$ queens.)

## Recall:

Given a set $X[1..n]$ of numbers + a target $T$, find a subset of $X$ whose sum is $= T$.

# Ch2 solution

```
⟨⟨Does any subset of X sum to T?⟩⟩
SubsetSum(X, T):
  if T = 0
      return True
  else if T < 0 or X = ∅
      return False
  else
      x ← any element of X
      with ← SubsetSum(X \ {x}, T − x)      ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X \ {x}, T)          ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

```
⟨⟨Does any subset of X[1..i] sum to T?⟩⟩
SubsetSum(X, i, T):
  if T = 0
      return True
  else if T < 0 or i = 0
      return False
  else
      with ← SubsetSum(X, i − 1, T − X[i])   ⟨⟨Recurse!⟩⟩
      wout ← SubsetSum(X, i − 1, T)          ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

## The recursion:

(Note: same thing as code!!)

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \ \vee \ SS(i+1, t - X[i]) & \text{otherwise} \end{cases}$$

$SS(i, t) = T$ or $F$

$0 \le i \le n$

$0 \le t \le T$

<u>So</u>: another 2-d table!

To decide:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

look at these 2 cells.

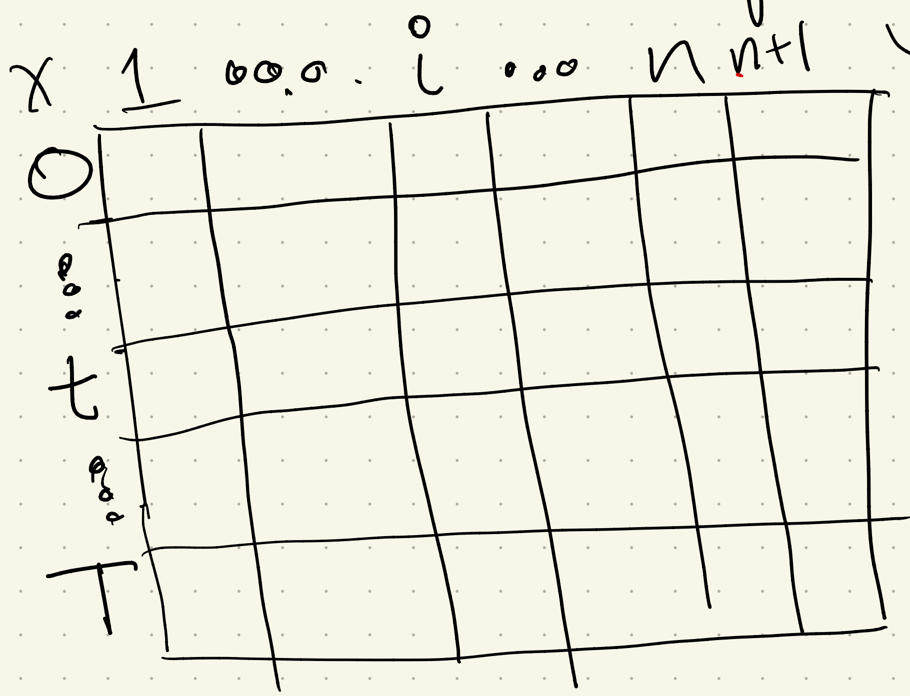
one note: if $t - X[i] < 0$, wasting time! Equivalent to:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

Now – need to code this:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

How should our loops go?




Fill:

## His code:

```
FastSubsetSum(X[1..n], T):
    S[n+1, 0] ← True
    for t ← 1 to T
        S[n+1, t] ← False

    for i ← n downto 1
        S[i, 0] = True
        for t ← 1 to X[i] − 1
            S[i, t] ← S[i+1, t]          ⟨⟨Avoid the case t < 0⟩⟩
        for t ← X[i] to T
            S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]

    return S[1, T]
```

## Correctness:

## Time/Space Analysis:

Note:

How big is this, & is it even a good idea??

input: number $T$ and array $X[1..n]$

table has a column for *every* number $1..T$.

How bad?

Well, $X$ could be a list of 5000 #s, but $T$ could be in the millions! (lots of empty columns, many of which are impossible to hit!)