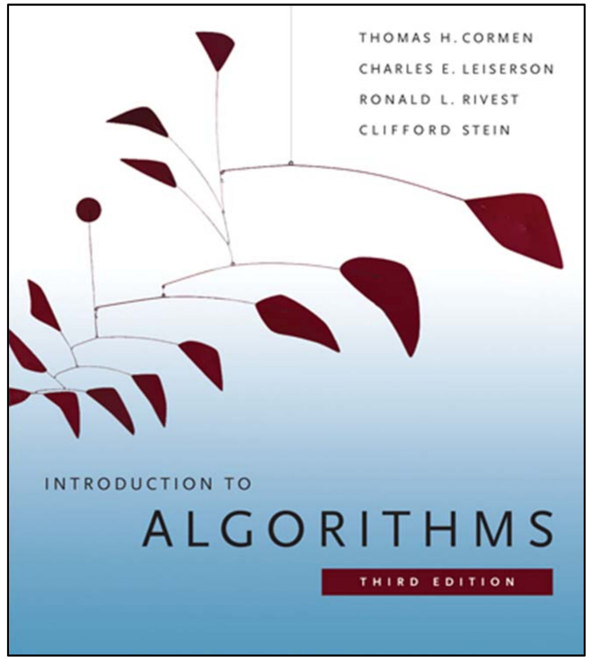
6.006

Introduction to Algorithms



Lecture 20: Dynamic Programming III

Prof. Erik Demaine

Today

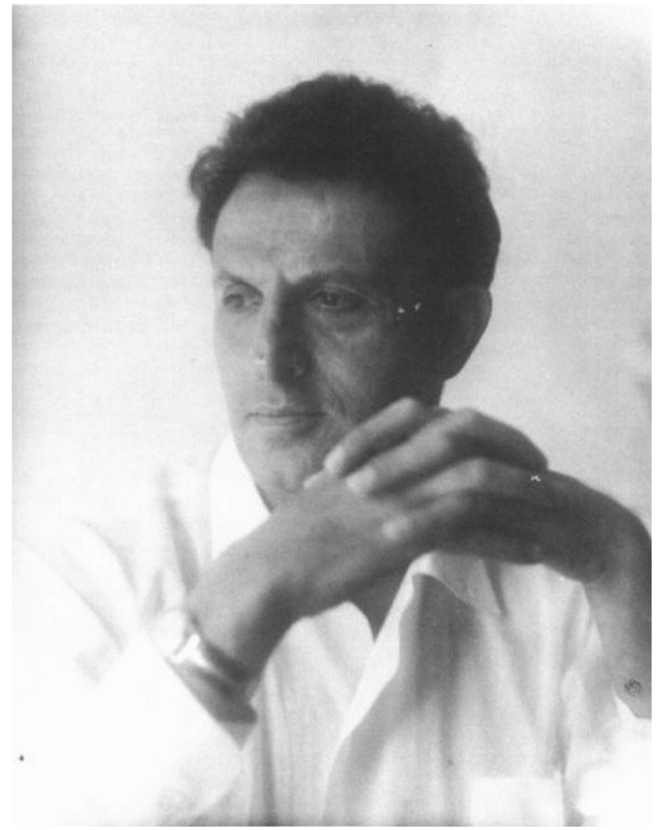
- Dynamic programming review
- **Guessing**
  - Within a subproblem
  - Using additional subproblems
- Parenthesization
- Knapsack
- Tetris training

Dynamic Programming History

'Bellman … explained that he invented the name "dynamic programming" to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who "had a pathological fear and hatred of the term, research." He settled on "dynamic programming" because it would be difficult give it a "pejorative meaning" and because "It was something not even a Congressman could object to." '

[John Rust 2006]

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2819&rep=rep1&type=pdf



Richard E. Bellman (1920–1984)

IEEE Medal of Honor, 1979

http://www.amazon.com/Bellman-Continuum-Collection-Works-Richard/dp/9971500906

What is Dynamic Programming?

- "Controlled" brute force / exhaustive search
- Key ideas:
  - **Subproblems:** like original problem, but smaller
    - Write solution to one subproblem in terms of solutions to smaller subproblems – acyclic
  - **Memoization:** remember the solution to subproblems we've already solved, and re-use
    - Avoid exponentials
  - **Guessing:** if you don't know something, guess it! (try all possibilities)

How to Dynamic Program

Five easy steps!

1. Define **subproblems**
2. **Guess** something (part of solution)
3. Relate subproblem solutions (**recurrence**)
4. Recurse and **memoize** (top down)
   or Build DP table bottom up
5. **Solve** original problem via subproblems
   (usually easy)

How to Analyze Dynamic Programs

Five easy steps!

1. Define subproblems ***count # subproblems***
2. Guess something ***count # choices***
3. Relate subproblem solutions
   analyze time per subproblem
4. ***DP running time*** = # subproblems · time per subproblem
5. Sometimes ***additional running time*** to solve original problem

Fibonacci Number F_n

1. **Subproblems:** F_k for $1 \le k \le n$ } n subprob.
2. **Guess:** nothing
3. **Recurrence:** $F_n = F_{n-1} + F_{n-2}$; $F_1 = F_2 = 1$ } O(1)/subprob.
4. **DP time** = # subprobloms · time/subproblem
   = n · O(1) = O(n)
5. **Original problem** = F_n

photo by Robobobobo
http://www.flickr.com/photos/45493477@N05/4178075187/

Crazy Eights

7♣ 7♥ K♣ K♠ 2♣ 8♥

1. **Subproblems:** trick(i) = length of longest trick ending with card i, for $1 \le i \le n$ } n
2. **Guess:** previous card j in trick(i) } $\le n$ choices
3. **Recurrence:** trick(i) = 1 + max $\{0\} \cup \{$ trick(j) for $1 \le j < i$ if cards i, j match$\}$ } $O(n)$
4. **DP time** = # subproblems · time/subproblem

   $n \cdot O(n) = O(n^2)$
5. **Original problem** =

   max(trick(i) for $1 \le i \le n$) } $O(n)$

Crazy Eights

7♣ 7♥ K♣ K♠ 2♣ 8♥

recurse + memoize

```
memo = {}
def trick(i):
    if i not in memo:
        memo[i] = 1 + max {0} ∪
            trick(j) for 1 ≤ j < i
            if cards i, j match}
    return memo[i]
return max(
    trick(i) for 1 ≤ i ≤ n)
```

DP table

```
trick = {}
for i from 1 to n:
    trick[i] = 1 + max {0} ∪
        trick[j] for 1 ≤ j < i
        if cards i, j match}
return max(
    trick[i] for 1 ≤ i ≤ n)
```

Sequence Alignment
(LCS, Edit Distance, etc.)

hieroglyphology
Michaelangelo

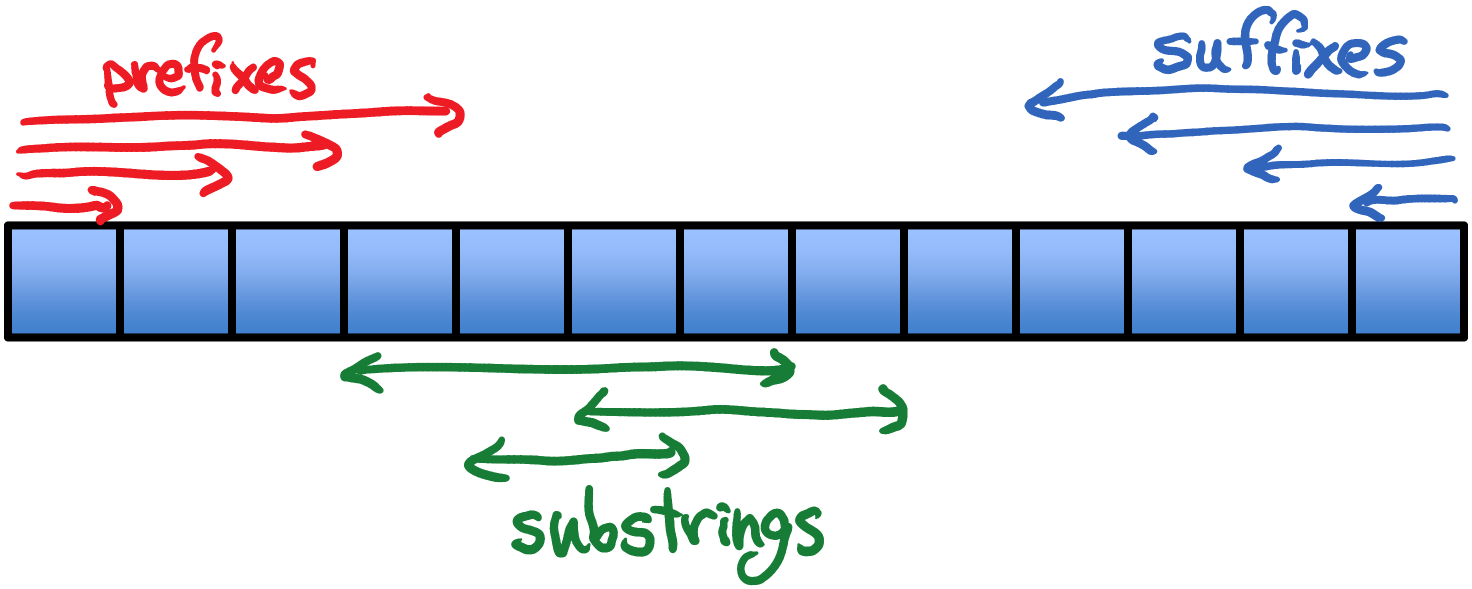
1. **Subproblems:** for $0 \le i \le m$ & $0 \le j \le n$:
   $A(i, j)$ = cost of best alignment of $s[:i]$ & $t[:j]$ } mn
2. **Guess:** how to align/drop $s[i]$ and $t[j]$ } 3 choices
3. **Recurrence:** $A(i, j) = \min\{$
   $A(i-1, j-1) +$ cost of aligning $s[i]$ & $t[j]$,
   $A(i-1, j) +$ cost of dropping $s[i]$,
   $A(i, j-1) +$ cost of inserting $t[j]\}$ } $O(1)$
4. **DP time** = # subproblems · time/subproblem
   mn · $O(1)$ $= O(mn)$
5. **Original problem** = $A(m, n)$

Choosing Subproblems

- For string/sequence/array x:
  - Suffixes $x[i:]$
  - Prefixes $x[:i]$
  - Substrings $x[i:j]$

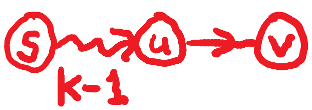
} $O(n) \Rightarrow$ preferred

} $O(n^2)$



Bellman-Ford
(single-source shortest paths)

1. **Subproblems:** for $0 \le k < |V|$ & $v \in V$: $\delta_k(s, v)$ = weight of shortest $s \to v$ path using $\le k$ edges $\Big\}\ |V|^2$
2. **Guess:** last edge in this path $\Big\}\ O(\text{in-degree}(v))$
3. **Recurrence:** $\delta_k(s, v) = \min\{\delta_{k-1}(s, v)\} \cup \{\delta_{k-1}(s, u) + w(u, v) : (u, v) \in E\}$



4. **DP time** = # subproblems · time/subproblem

   $\sum_{v \in V} |V| \cdot O(\text{in-deg}(v)) = O(VE)$

5. **Original problem** = $\delta_{|V|-1}(s, v)$ for $v \in V$

Floyd-Warshall
(all-pairs shortest paths)

$\{1, 2, \ldots, |V|\}$

1. **Subproblems:** for $0 \leq k \leq |V|$ & $i, j \in V$:
   $\delta_k(i, j)$ = weight of shortest $i \to j$ path
   using intermediate vertices in $\{1, 2, \ldots, k\}$ — $O(V^3)$
2. **Guess:** is vertex k in the path? — 2 choices
3. **Recurrence:** $\delta_k(i, j) =$
   $\min\{\delta_{k-1}(i, j), \delta_{k-1}(i, k) + \delta_{k-1}(k, j)\}$ — $O(1)$
4. **DP time** = # subproblems · time/subproblem
   $O(V^3) \cdot O(1) = O(V^3)$
5. **Original problem** = $\delta_{|V|}(i, j)$ for $i, j \in V$

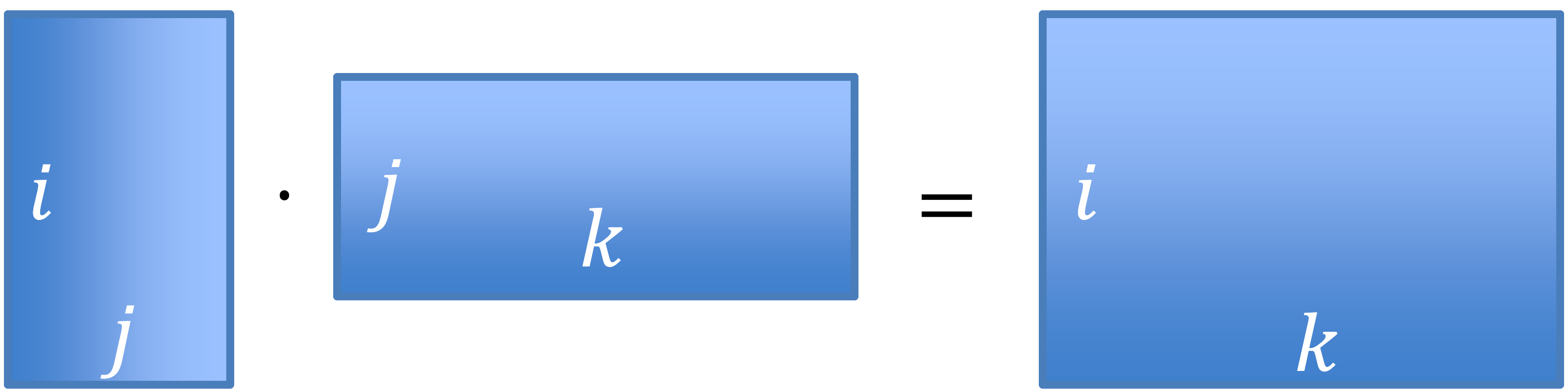
Bottom-Up Floyd-Warshall

for i in V:
    for j in V:
        $d[i,j] = w(i,j)$ [∞ if no edge]
for k from 1 to $|V|$:
    for i from 1 to $|V|$:
        for j from 1 to $|V|$:
            if $d[i,j] > d[i,k] + d[k,j]$:
                $d[i,j] = d[i,k] + d[k,j]$

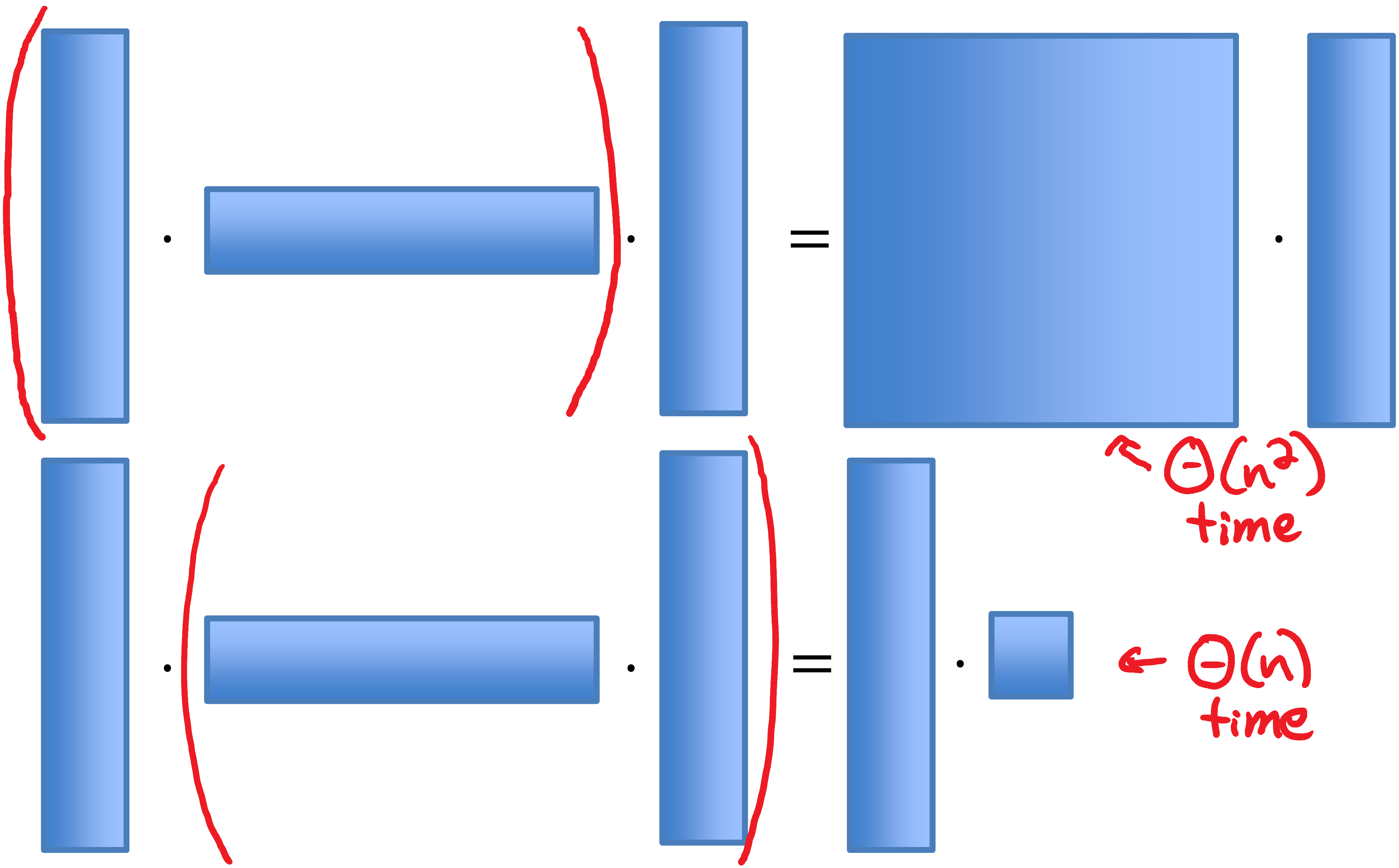
} relaxation of Δ inequality

Parenthesization Problem

- Given sequence of matrices $A_1, A_2, \ldots, A_n$ of dimensions $d_1 \times d_2, d_2 \times d_3, \ldots, d_n \times d_{n+1}$
- Compute associative product $A_1 \cdot A_2 \cdot \cdots \cdot A_n$ using sequence of normal matrix multiplies in the order that minimizes cost
- Cost to multiply $i \times j$ with $j \times k$ is $i\ j\ k$



Parenthesization Example



Parenthesization DP

1. **Subproblems:** for $1 \leq k \leq n$:
   cost of optimal multiplication of $A_1 \cdot \cdots \cdot A_k$
2. **Guess:** last multiplication to do:
   $$\left(A_1 \cdot \cdots \cdot A_j\right) \cdot (A_{j+1} \cdot \cdots \cdot A_k)$$
3. **Recurrence:** $M(k) =$
   $$\min(M(j) + \cdots ??? \cdots \text{ for } 1 \leq j < k)$$

- *Prefix/suffix not enough; use* ***substrings***

Parenthesization DP

1. **Subproblems:** for $1 \le i \le k \le n$:
   cost of optimal multiplication of $A_i \cdot \cdots \cdot A_k$ } n^2
2. **Guess:** last multiplication to do:
   $$(A_i \cdot \cdots \cdot A_j) \cdot (A_{j+1} \cdot \cdots \cdot A_k)$$ } $\le n$ choices
3. **Recurrence:** $M(i,k) = \min(n_i n_{j+1} n_{k+1}$
   $+M(i,j) + M(j+1,k)$ for $i \le j < k)$ } $O(n)$
4. **DP time** = # subproblems · time/subproblem
   $n^2 \cdot O(n) = O(n^3)$
5. **Original problem** = $M(1,n)$

Knapsack Problem

- Knapsack of integer size S
- Items $1, 2, \ldots, n$
- Item i has integer **size** s_i and **value** v_i
- Goal: Choose subset of items of maximum possible total value, subject to total size $\leq S$

photo by Erik Demaine

Knapsack DP

1. **Subproblems:** for $1 \le i \le n$:
   optimal packing of items $i, i+1, \ldots, n$
2. **Guess:** include item i?
3. **Recurrence:**
   $$K(i) = \max(K(i+1),$$
   $$K(i+1) + v_i \text{ if } s_i \le S' \ldots ???)$$

- *How to maintain remaining space in knapsack?*

Guess!

http://3.bp.blogspot.com/_lrYZ590iyME/TA-aqU7MPnI/AAAAAAAABf8/KXABMduUVvM/s1600/Guess+Backpack.jpg

Knapsack DP

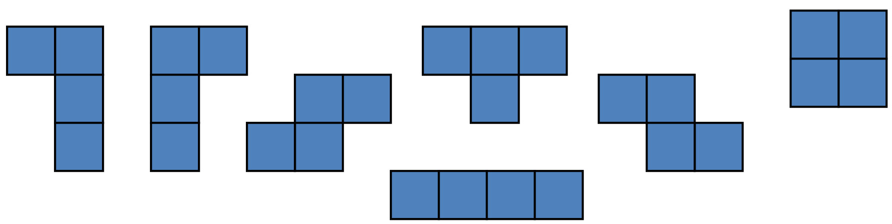
1. **Subproblems:** for $1 \leq i \leq n$ & $0 \leq X \leq S$: optimal packing of items $i, i+1, \ldots, n$ into knapsack of size X } nS
2. **Guess:** include item i? } 2 choices
3. **Recurrence:** } $O(1)$
   $$K(i) = \max(K(i+1, X),$$
   $$K(i+1, X - s_i) + v_i \text{ if } s_i \leq X)$$
4. **DP time** = # subproblems · time/subproblem
   $nS \cdot O(1) = O(nS)$
5. **Original problem** = $K(1, S)$

Pseudopolynomial Time

- $O(n\ S)$ running time is ***pseudopolynomial***
- In general: polynomial in n and the integers in the problem input
- Equivalently: polynomial in the input size if the integers were written in unary
- ***Polynomial time*** assumes encoded in binary
- Knapsack is extremely unlikely to have a polynomial-time algorithm (see Lecture 25)

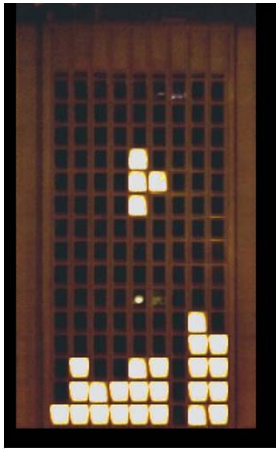
Tetris Training

- Given sequence of n pieces



- Given board of small width w and larger height h
- <u>Goal:</u> Place each piece in sequence to *survive* —
  stay within height h —
  without any holes/overhang

Tetris Training DP

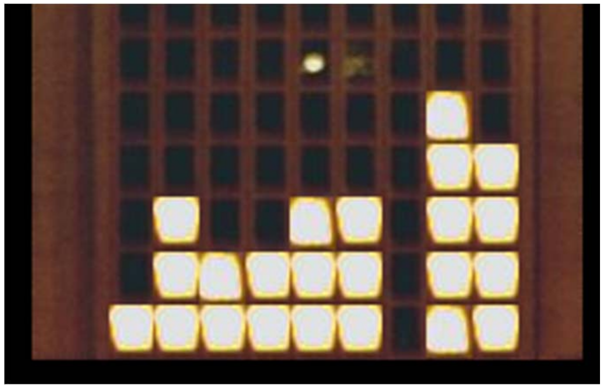


1. **Subproblems:** for $1 \leq i \leq n$:
   can you survive given pieces $i, i+1, \ldots, n$?
2. **Guess:** how to place piece i
3. **Recurrence:**

$$T(i) = \text{or}(T(i+1) \text{ for each possible move } \ldots ??? \ldots)$$

- *How to know valid moves for piece i?*
- ***Guess!***

Tetris Training DP



1. **Subproblems:** for $1 \leq i \leq n$ & $0 \leq h_1, h_2, \ldots, h_w \leq h$: can you survive given pieces $i, i+1, \ldots, n$ starting from columns with heights $h_1, h_2, \ldots, h_w$? – $\Theta(nh^w)$ subprobs.
2. **Guess:** how to place piece i } $O(w)$ choices
3. **Recurrence:**

$$T(i, h_1, \ldots, h_w) = \text{or}(T(i+1, h'_1, \ldots, h'_w)$$

for each possible placement of piece i) } $O(w^2)$

Time = $O(nh^w w^2)$

What's Next?

- Dynamic programming over combinatorial structures other than arrays
- More examples of the power of guessing