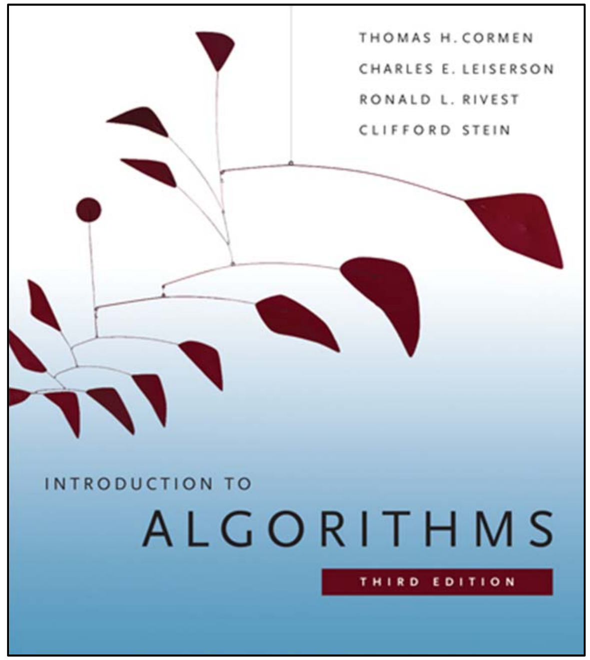
6.006
Introduction to Algorithms



Lecture 21: Dynamic Programming IV

Prof. Erik Demaine

Today

- Piano fingering
- Platform video games
- Structural dynamic programming
- Vertex cover
- Widget layout

<u>Recall:</u> What is Dynamic Programming?

- "Controlled" brute force / exhaustive search
- <u>Key ideas:</u>
  - **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)

Recall:
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)

<u>Recall:</u> 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

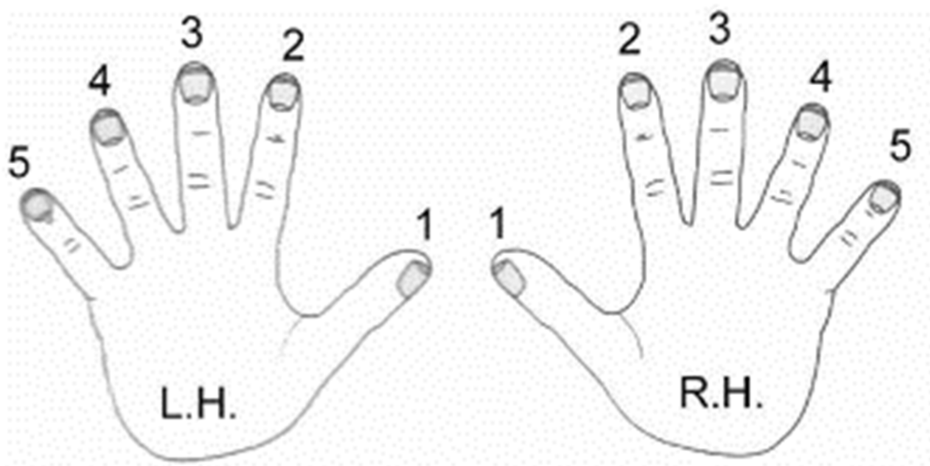
Two Kinds of Guessing

1. **Within a subproblem**
   – Crazy Eights: previous card in trick
   – Sequence alignment: align/drop one character
   – Bellman-Ford: previous edge in path
   – Floyd-Warshall: use vertex k?
   – Parenthesization: last multiplication
   – Knapsack: include item i?
   – Tetris training: how to place piece i
2. **Using additional subproblems**
   – Knapsack: how much space left in knapsack
   – Tetris training: current board configuration

Piano Fingering

photo by Brian Richardson (seriousbri), 2009
http://www.flickr.com/photos/seriousbri/4148739768/

Piano Fingering



images from http://www.piano-lessons-central.com/music-notation/how-to-read-music/

Piano Fingering

[Parncutt, Sloboda, Clarke, Raekallio, Desain 1997;
Hart, Bosch, Tsai 2000; Al Kasimi, Nichols, Raphael 2007]

- Given musical piece to play
  - Say, sequence of single notes with right hand
  - (Can extend to both hands, multiple notes, etc.)
- Given metric $d(f, p, g, q)$ of ***difficulty*** going from finger f on note p to finger g on note q
  - **Crossing:** High if $1 < f < g$ and $p > q$
  - **Stretch:** High if $p \ll q$
  - **Legato:** ∞ if $f = g$
  - **Weak finger:** High if $g \in \{4, 5\}$
  - **3 $\leftrightarrow$ 4:** High if $\{f, g\} = \{3, 4\}$
  - …

References:
http://www.jstor.org/pss/40285730
http://www.jstor.org/pss/10.1525/mp.2001.18.4.505
http://www.cse.unsw.edu.au/~cs9024/05s2/ass/ass01/fingering.pdf
http://ismir2007.ismir.net/posters/ISMIR2007_p355_kasimi_poster.pdf

Piano Fingering DP

1. **Subproblems:** for $1 \leq i \leq n$:
   minimum difficulty possible for note$[i:]$
2. **Guess:** finger f for note$[i]$
3. **Recurrence:** $P(i) = \min(P(i+1) +$
   $d(\text{note}[i], f, \text{note}[i+1], \cdots ??? \cdots)$
   for f in fingers)

- *How to know fingering for next note $i + 1$?*
- ***Guess!***

Piano Fingering DP

1. **Subproblems:** for $1 \leq i \leq n$ & finger f: minimum difficulty possible for note$[i:]$ starting on finger f $\Big\}$ n·F
2. **Guess:** finger g for note$[i+1]$ $\}$ F choices
3. **Recurrence:** $P(i, f) = \min(P(i+1, g) + d(\text{note}[i], f, \text{note}[i+1], g)$ for g in fingers) $\Big\}$ O(F)
4. **DP time** = $\underbrace{\#\text{ subproblems}}_{nF} \cdot \underbrace{\text{time/subproblem}}_{O(F)} = O(nF^2)$
5. **Original problem** = $\min(P(1, f)$ for f in fingers)

http://media.ign.com/games/image/article/833/833615/ign-presents-the-history-of-super-mario-bros-20071108044755327.gif
http://images.inquisitr.com/wp-content/2011/03/super-mario-bros-modern-sound.jpg

Platform Video Games

- Given entire level: objects, enemies, etc.
- Anything outside $w \times h$ screen is reset
- ***Configuration*** = screen state, score, velocity, …
- Given transition function for each time step

$$\delta\colon (\text{config}, \text{action}) \mapsto \text{config}'$$

  – Movement, enemies, …

- <u>Goal:</u> Maximize score subject to surviving and reaching goal

Platform Video Game DP

configs.

1. **Subproblems:** for configuration C: best possible score starting from C $\Big\}\ = O(1)^{w \cdot h} \cdot n \cdot m \cdot S \cdot V$
2. **Guess:** which action to take (if any) $\}\ O(1)$
3. **Recurrence:**
   $$P(C) = \max\Big(P\big(\delta(C, A)\big) \text{ for } A \text{ in actions}\Big)$$
   $$P(\text{goal } C) = C.\text{score};\ P(\text{dead } C) = -\infty$$
   $\Big\}\ O(1)$
4. **DP time** = $\underbrace{\#\text{ subproblems}}_{O(1)^{w \cdot h} \cdot n \cdot m \cdot S \cdot V} \cdot \underbrace{\text{time/subproblem}}_{O(1)}$ — (pseudo) polynomial for $w \cdot h = O(\lg(nmSV))$
5. **Original problem** = $P(\text{init})$

Cycles in Subproblems

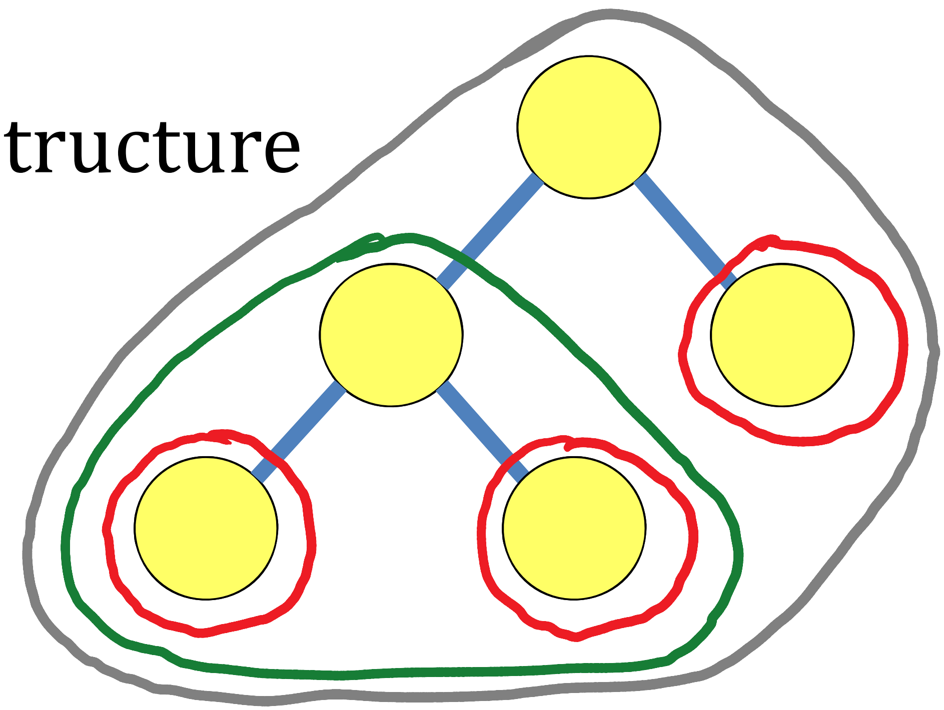
- $C_1 \to \delta(C_1, A_1) = C_2 \to \delta(C_2, A_2) = C_3 \to \cdots$ might lead to cycles
- In this problem, never helps to cycle
  - C captures entire state, including score
- So mark subproblem at start, and if cycle, ignore that subproblem
- OR: SMB timer in C, so actually no cycles

```
memo = {}
def mario(C):
    if C not in memo:
        memo[C] = −∞
        memo[C] = max(
            mario(δ(C, A))
            for A in actions)
return memo[C]
```

Structural Dynamic Programming

- Follow a combinatorial structure other than a sequence / a few sequences
  - Like structural vs. regular induction

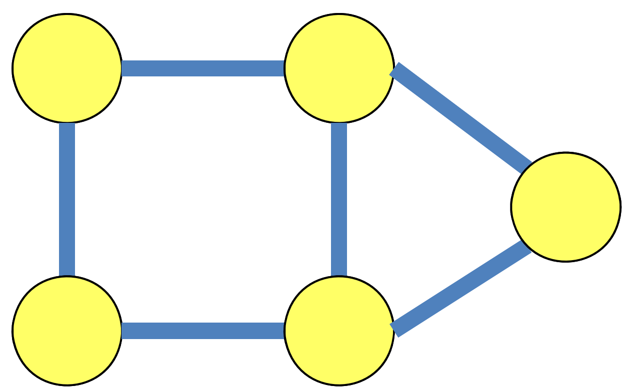
- Main example: Tree structure
- Useful subproblems:
  for every vertex v,
  subtree rooted at v



Vertex Cover

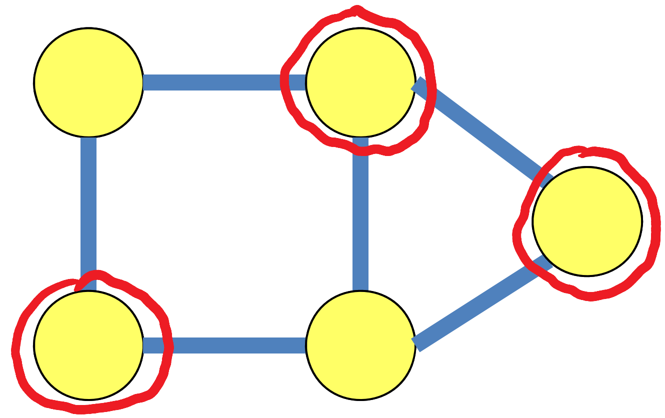
- Given an undirected graph $G = (V, E)$
- Find a minimum-cardinality set S of vertices containing at least one endpoint of every edge
  - Equivalently, find a minimum set of guards for a building of corridors, or (unaligned) streets in city

Example:

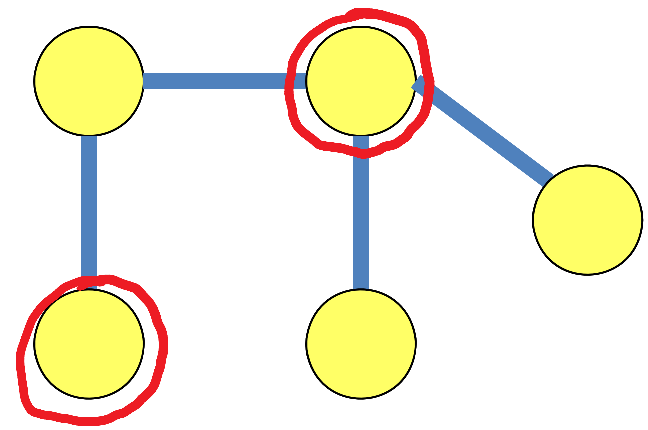


Vertex Cover Algorithms

- Extremely unlikely to have a polynomial-time algorithm, even for planar graphs (see Lecture 25)



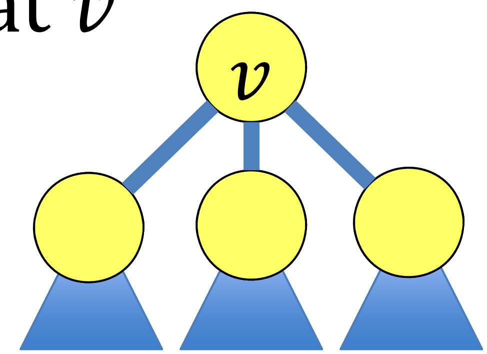
- But polynomially solvable on trees, using dynamic programming



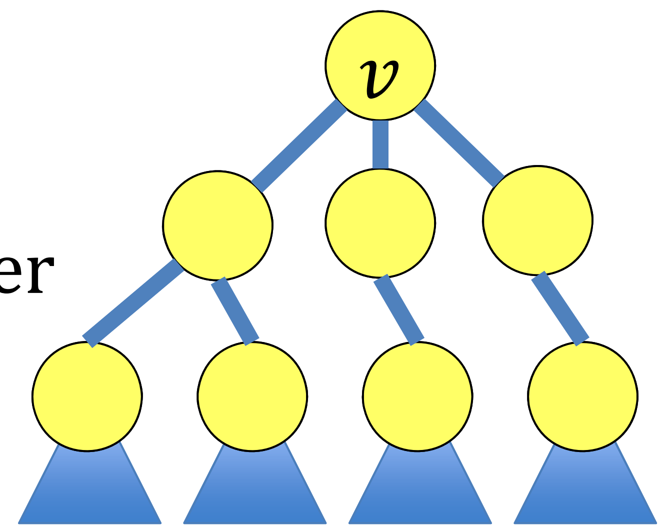
Vertex Cover in Tree DP

0. Root the tree arbitrarily.

1. **Subproblems:** for $v \in V$: size of smallest vertex cover in subtree rooted at v

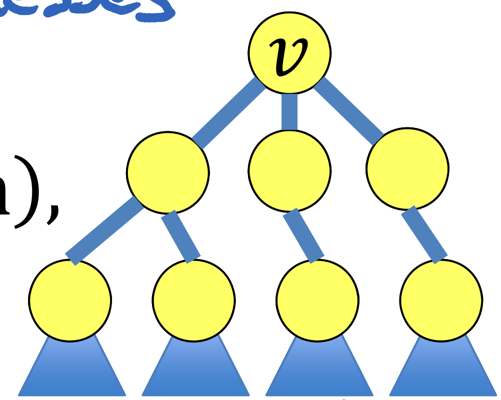


2. **Guess:** is v in the cover?
   - YES:
     - Cover children edges
     - Left with children subtrees
   - NO:
     - All children must be in cover
     - Left with grandchildren subtrees



Vertex Cover in Tree DP

1. **Subproblems:** for $v \in V$: size of smallest vertex cover in subtree rooted at v } $|V|$
2. **Guess:** is v in the cover? } 2 guesses
3. **Recurrence:** $V(v) = \min\{$
   YES: $1 + \text{sum}(V(c) \text{ for } c \text{ in } v.\text{children}),$
   NO: $\text{len}(v.\text{children}) +$
   $\text{sum}(V(g) \text{ for } c \text{ in } v.\text{children}$
   $\text{for } g \text{ in } c.\text{children})\}$ ~ $O(V)$



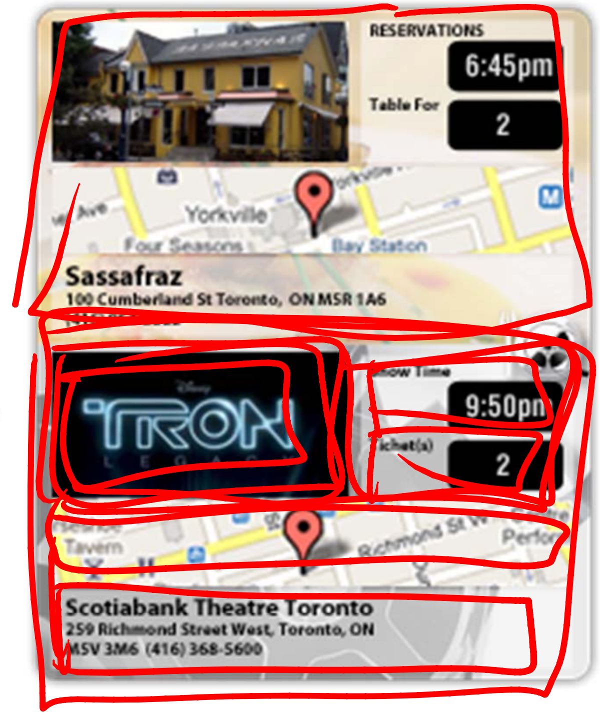
4. **DP time** = $\underbrace{\#\text{ subproblems}}_{|V|} \cdot \underbrace{\text{time/subproblem}}_{O(V)} = O(V^2)$
5. **Original problem** = $V(\text{root})$ actually $O(V)$ because each vertex visited twice: parent & grandpar.

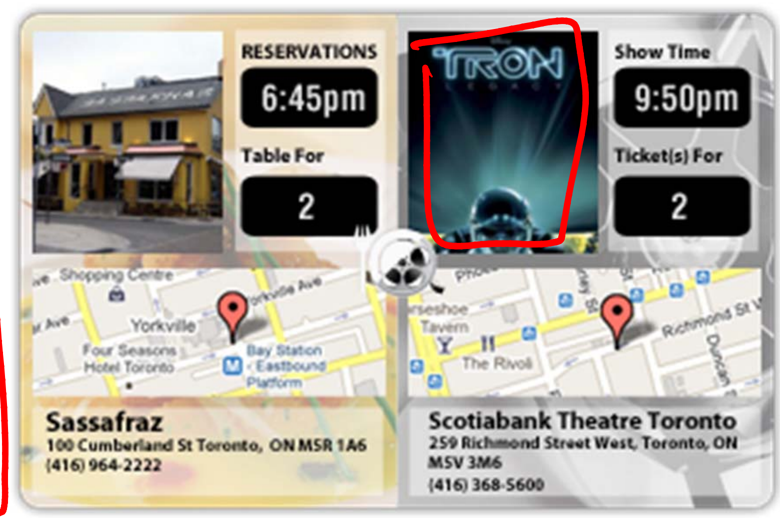
Improved Vertex Cover in Tree DP

1. **Subproblems:** for $v \in V$ & $y \in \{\text{YES}, \text{NO}, \text{MAYBE}\}$: $3|V|$
   size of smallest vertex cover S in subtree rooted at v such that $[v \in S?] = y$ *(unconstrained if* $y = \text{MAYBE}$*)*
2. **Guess:** Does MAYBE = YES or NO? $\} \leq 2$ choices
3. **Recurrence:**
   $V(v, \text{MAYBE}) = \min\{V(v, \text{YES}), V(v, \text{NO})\}$ — $O(1)$
   $V(v, \text{YES}) = 1 + \text{sum}(V(c, \text{MAYBE})$ for c in $v.\text{children})$
   $V(v, \text{NO}) = \text{sum}(V(c, \text{YES})$ for c in $v.\text{children})$
   \# children(v)
4. **DP time** = $\underbrace{\text{\# subproblems}}_{\sum_{v \in V} 3} \cdot \underbrace{\text{time/subproblem}}_{\text{\# children}(v)} = O(V)$
5. **Original problem** = $V(\text{root}, \text{MAYBE})$

Widget Layout

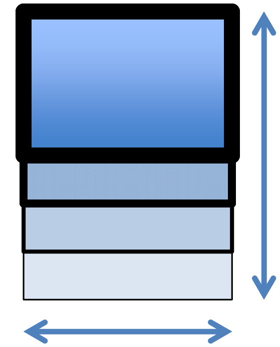
- Given a hierarchy of ***widgets***
- **Leaf** widget = button, image, …
  – List of possible rectangular sizes
- **Internal** widget = rectangular container
  – Can join children horizontally or vertically
- <u>Goal:</u> Fit into a given rectangular screen





http://lin-kuei.blogspot.com/2010/12/android-app-menu-widgets-and-site-map.html

Widget Layout DP



1. **Subproblems:** for $v \in V$ & $0 \leq w \leq W$: minimum h such that widget v fits into $w \times h$
2. **Guess:** Leaf v: Which size to use?
   Internal v: Horizontal or vertical?
3. **Recurrence:**

$$L(\text{leaf } v, w) = \min(h' \text{ for } (w', h') \text{ in } v.\text{sizes if } w' \leq w)$$

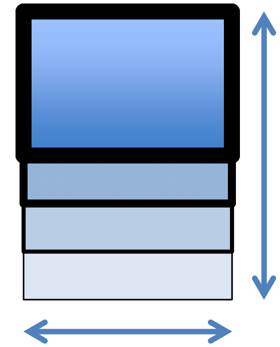
$$L(\text{internal } v, w) = \min\{\text{sum}(L(c, w) \text{ for } c \text{ in } v.\text{child}),\ H(v, w, 1)\}$$

Horizontal Layout DP

$\sum_{v \in V} W \cdot \deg(v) = W \cdot |E|$

1. **Subproblems:** for $v \in V$, $0 \le w \le W$, $1 \le i \le \text{len}(v.\text{children})$: minimum h such that horizontal layout of $v.\text{child}[i:]$ fits into $w \times h$ rectangle
2. **Guess:** Width $0 \le w' \le W$ of child i } W choices
3. **Recurrence:** $H(v, w, i) = \min($ $\max\{L(v.\text{child}[i], w'), H(v, w - w', i + 1)\}$ for $1 \le w' \le w)$ } $O(W)$
4. **DP time** = $\underbrace{\#\text{ subproblems}}_{W \cdot |E|} \cdot \underbrace{\text{time/subproblem}}_{O(W)} = O(W^2 E)$

Widget Layout DP

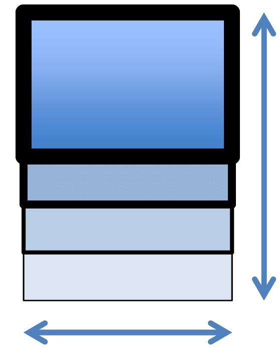


1. **Subproblems:** for $v \in V$ & $0 \leq w \leq W$: $\rightarrow |V| \cdot W$
   minimum h such that widget v fits into $w \times h$
2. **Guess:** Leaf v: Which size to use? deg(v)
   Internal v: Horizontal or vertical? 2
3. **Recurrence:**

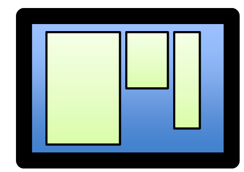
$$\left.\begin{aligned} L(\text{leaf } v, w) &= \cdots \text{for} \cdots \text{in } v.\text{sizes} \cdots \\ L(\text{internal } v, w) &= \cdots \text{for} \cdots \text{in } v.\text{child} \cdots \end{aligned}\right\} O(\deg(v))$$

4. **DP time** = $\underbrace{\#\text{ subproblems}}_{\leq \sum_{v \in V} W} \cdot$ time/subproblem $\cdot\, O(\deg(v)) = O(WE)$
5. **Original problem** $= S(\text{root}, W) \leq H$

Widget Layout Summary



- Two "levels" of dynamic programming
  1. Optimal height for given width of subtree rooted at v
  2. Optimal layout (partitioning) of children into horizontal arrangement



- Really just one bigger dynamic program
- Pseudopolynomial running time:

$$O(W^2E + WE) = O(W^2E)$$