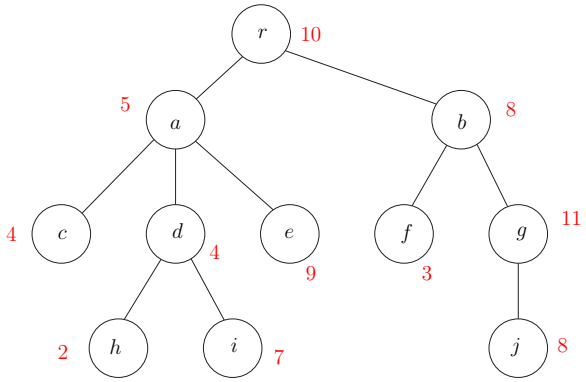
1. Given a graph $G = (V, E)$, a *vertex cover* of G is a subset $S \subseteq V$ of vertices such that for each edge $e = (u, v)$ in G, u or v is in S. That is, the vertices in S *cover* all the edges. Given a tree $T = (V, E)$ and a non-negative weight $w(v)$ for each vertex $v \in V$, give an algorithm that computes the minimum weight vertex cover of T; the weight of a cover S is the sum of the weights of the vertices in S. In the tree below, $\{B, E, G\}$ is a vertex cover while $\{C, E, F\}$ is not a vertex cover. It is helpful to root the tree.



Solution: We use dynamic programming. Root the tree at some arbitrary node r. For a node v let T_v be the subtree rooted at v. We also use $C(v)$ to denote the children of v and $G(v)$ to denote the grandchildren of v.

Let $MinVC(v)$ be the minimum weight vertex cover for the tree T_v. We make the following observations.

(a) $MinVC(v) = 0$ if v is a leaf.

(b) Suppose v is not a leaf. Consider an optimal vertex cover A for T_v.

- If v is in A, $A - \{v\}$ is an optimal vertex cover for the forest consisting of T_u, for all $u \in C(v)$. Since the trees in this forest are disjoint, $(A - \{v\}) \cap T_u$ is an optimal vertex cover for T_u and thus the cost of A is $\mathrm{w}(v) + \sum_{u \in C(v)} MinVC(u)$.
- If v is not in A, all of v's children are in A, since each edge (v, u) is covered by A. Thus $A - C(v)$ is an optimal vertex cover for the forest consisting of T_u, for all $x \in G(v)$. Since the trees in this forest are disjoint, $(A - C(v)) \cap T_x$ is an optimal vertex cover for T_x and thus the cost of A is $\sum_{u \in C(v)} w(u) + \sum_{x \in G(v)} MinVC(x)$.

Therefore we have the following recurrence

$$MinVC(v) = \begin{cases} 0 & v \text{ is a leaf} \\ \min \left\{ \begin{array}{c} w(v) + \sum_{u \in C(v)} MinVC(u), \\ \sum_{u \in C(v)} w(u) + \sum_{x \in G(v)} MinVC(x) \end{array} \right\} & \text{otherwise} \end{cases}$$

The above recurrence can be used to compute $MinVC(r)$ via memoization. Number of subproblems is n. Each subproblem can be computed in $O(n)$ time from previous subproblems which leads to an $O(n^2)$ time algorithm. However, just as we argued with the dynamic programming for max weight independent set in a tree, one can

reduce the time to $O(n)$. Space requirement is also $O(n)$ since there are only n subproblems. ■

Solution: We saw in lecture on reductions that for any graph $G = (V, E)$, S is a vertex cover iff $V \setminus S$ is an independent set. Thus, one can find a min-weight vertex cover in G by finding a max-weight independent set in G and taking its complement. We already saw an algorithm for finding a max-weight independent set in a tree via dynamic programming and hence we can use that to compute a min-weight vertex cover in a tree. It is therefore not surprising that the above dynamic programming solution looks quite similar to the one for max-weight independent set in a tree. ■

Solution: The solution for maximum weight independent set in lecture, and for minimum weight vertex cover that we described above, are based on a simple recursive approach that considers two cases based on whether to include the root or not. In more complex problems it is necessary to introduce an additional parameter that allows a clean decomposition of the problem into subproblems. Here we outline such a solution for the minimum weight vertex cover problem in a tree.

We assume that the tree T is rooted and let T_u denote the subtree of T rooted at a given node u. For $u \in V$ and $b \in \{0, 1\}$ we define $MinVC(u, b)$ as follows. $MinVC(u, 0)$ denotes the weight of a min-weight vertex cover of T_u. $MinVC(u, 1)$ denotes the weigth of a min-weight vertex cover of T_u among all vertex covers that include u.

We give a recursive definition of $MinVC(u, b)$. The base cases are when u is a leaf, in which case the minimum-weight vertex cover is 0 if $b = 0$ and $w(u)$ if $b = 1$. For the recursive case, if we have to include u in the solution, then we are not required to include u's children, but if $b = 0$ and we decide to not include u in the solution, then we are required to include all of u's children. Let $C(u)$ denote the set of u's children in the tree. We have

$$MinVC(u,b) = \begin{cases} 0 & \text{if } u \text{ is a leaf and } b = 0 \\ w(u) & \text{if } u \text{ is a leaf and } b = 1 \\ w(u) + \sum_{v \in C(u)} MinVC(v, 0) & \text{if } b = 1 \\ \min \left\{ \begin{array}{c} w(u) + \sum_{v \in C(u)} MinVC(v, 0), \\ \sum_{v \in C(u)} MinVC(v, 1) \end{array} \right\} & \text{otherwise.} \end{cases}$$

Note that the introduction of the parameter b meant that $MinVC(u, b)$ relied only on values at the children of u. Finally, the solution can be computed by calling $MinVC(r, 0)$.

Now, the memoization data structure will be a $2d$-array of size $2n$, since there are n vertices but b can only take two values. Suppose VC is an array that, at position $VC[u, b]$ stores the value $MinVC(u, b)$. Like before, $MinVC(u, b)$ depends only on values of $MinVC$ for vertices that are in the subtree T_u, and thus a correct evaluation order will be an ordering generated by a post-order traversal of the tree. We can fill in the array VC in this ordering with a simple for-loop. The running time of the algorithm can be shown to be $O(n)$, by noticing that every element of the array

participates in one assignment and one addition, so 2 operations, and we have $2n$ elements in the array in total, for a total bound of $4n$ on the number of operations, which is $O(n)$. ■

2. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is equal to n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above. The running time of your algorithm should be bounded by a small polynomial function of n.

Solution: Let $\mathit{Min1s}(n)$ denote the minimum number of 1s in a basic arithmetic expression with value n. This function obeys the following recurrence:

$$\mathit{Min1s}(n) = \begin{cases} 1 & \text{if } n = 1 \\ \min\left\{ \begin{array}{l} \min\left\{ \mathit{Min1s}(m) + \mathit{Min1s}(n-m) \;\middle|\; 1 \le m \le n/2 \right\} \\ \min\left\{ \mathit{Min1s}(m) + \mathit{Min1s}(n/m) \;\middle|\; \begin{array}{l} 1 < m \le \sqrt{n} \text{ and} \\ n/m \text{ is an integer} \end{array} \right\} \end{array} \right\} & \text{otherwise} \end{cases}$$

Here are the first twenty values of this function; more values can be found at *The Online Encyclopedia of Integer Sequences* (sequence A005245).

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\mathit{Min1s}(n)$	1	2	3	4	5	5	6	6	6	7	8	7	8	8	8	8	9	8	9	9

We can memoize this function into a one-dimensional array $\mathit{Min1s}[1..n]$. Each entry $\mathit{Min1s}[i]$ depends on all entries $\mathit{Min1s}[j]$ with $j < i$, so we can fill the array in increasing index order.

```
MinOnes(n):
  Min1s[1] ← 1
  for i ← 2 to n
      Min1s[i] ← i        ⟨⟨Easy upper bound⟩⟩
      for m ← 1 to i/2
          Min1s[i] ← min{Min1s[i], Min1s[m] + Min1s[i − m]}
          if ⌊i/m⌋ · m = i
              Min1s[i] ← min{Min1s[i], Min1s[m] + Min1s[i/m]}
  return Min1s[n]
```

The resulting algorithm runs in $O(n^2)$ ***time*** (assuming each arithmetic operation takes $O(1)$ time).

This is not the fastest algorithm known for this problem. ■

To think about later:

2. Suppose you are given a sequence of integers separated by $+$ and $-$ signs; for example:

$$1 + 3 - 2 - 5 + 1 - 6 + 7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1 + 3 - 2 - 5 + 1 - 6 + 7 = -1$$
$$(1 + 3 - (2 - 5)) + (1 - 6) + 7 = 9$$
$$(1 + (3 - 2)) - (5 + 1) - (6 + 7) = -17$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication as in $1 + 3(-2)(-5) + 1 - 6 + 7 = 33$.

Solution: Suppose the input consists of an array $X[0..2n]$, where $X[i]$ is an integer for every even index i and $X[i] \in \{+, -\}$ for every odd index i.

Let $\mathit{Max}(i,k)$ and $\mathit{Min}(i,k)$ respectively denote the maximum and minimum values obtainable by parenthesizing the subexpression $X[2i..2k]$. We need to compute $\mathit{Max}(0,n)$. These functions obey the following mutual recurrences:

$$\mathit{Max}(i,k) = \begin{cases} X[2i] & \text{if } i = k \\ \max\left\{ \max\left\{ \mathit{Max}(i,j) + \mathit{Max}(j+1,k) \;\middle|\; \begin{array}{c} i \le j < k \\ X[2j+1] = + \end{array} \right\} \right. & \\ \left. \quad \max\left\{ \mathit{Max}(i,j) - \mathit{Min}(j+1,k) \;\middle|\; \begin{array}{c} i \le j < k \\ X[2j+1] = - \end{array} \right\} \right\} & \text{otherwise} \end{cases}$$

$$\mathit{Min}(i,k) = \begin{cases} X[2i] & \text{if } i = k \\ \min\left\{ \max\left\{ \mathit{Min}(i,j) + \mathit{Min}(j+1,k) \;\middle|\; \begin{array}{c} i \le j < k \\ X[2j+1] = + \end{array} \right\} \right. & \\ \left. \quad \max\left\{ \mathit{Min}(i,j) - \mathit{Max}(j+1,k) \;\middle|\; \begin{array}{c} i \le j < k \\ X[2j+1] = - \end{array} \right\} \right\} & \text{otherwise} \end{cases}$$

We can memoize each of these functions into a two-dimensional array. Each entry $\mathit{Mxx}[i,k]$ depends on earlier entries in the same row of the same array, and later entries in the same column *in both arrays*. Thus, we can fill both arrays simultaneously, by considering rows from bottom to top in the outer loop, and considering each row from left to right in the inner loop.

The resulting algorithm (shown on the next page) runs in $O(n^3)$ ***time***.

```
MaxValue(X[0..2n]):
  for i ← n down to 0
      Max[i,i] ← X[2k]
      Min[i,i] ← X[2k]
      for k ← i+1 to n
          localMax ← −∞
          localMin ← ∞
          for j ← i to k−1
              if X[2j+1] = +
                  localMax ← max{localMax, Max[i,j] + Max[j+1,k]}
                  localMin ← min{localMin, Min[i,j] + Min[j+1,k]}
              else
                  localMax ← max{localMax, Max[i,j] − Min[j+1,k]}
                  localMin ← min{localMin, Min[i,j] − Max[j+1,k]}
          Max[i,k] ← localMax
          Min[i,k] ← localMin
  return Max[0,n]
```

■