Placing Parentheses

You are given an arithmetic expression containing n real numbers and $n-1$ operators, each either $+$ or $*$. Your goal is to perform the operations in an order that maximizes the value of the expression. That is, insert parentheses into the expression so that its value is maximized.

For example:

- For the expression $6 * 3 + 2 * 5$, the optimal ordering is to add the middle numbers first, then perform the multiplications: $(6 * (3 + 2)) * 5 = 150$.
- For the expression $0.1 * 0.1 + 0.1$, the optimal ordering is to perform the multiplication first, then the addition: $(0.1 * 0.1) + 0.1 = 0.11$.
- For the expression $(-3) * 3 + 3$, the optimal ordering is $((-3) * 3) + 3) = -6$.

Dynamic Programming Solution

This problem is *very* similar to the matrix chain multiplication problem done in lecture.

First, we define some notation. Denote the numbers by $a_1, a_2, \ldots, a_n$, and the operators by $op_1, op_2, \ldots, op_{n-1}$, so the given expression is $a_1\ op_1\ \ldots\ op_{n-1}\ a_n$.

Let $M[i, j]$ be the *maximum* value obtainable from the subexpression beginning at a_i and ending at a_j (i.e., $a_i\ op_i\ \ldots\ op_{j-1}\ a_j$), and let $m[i, j]$ be the *minimum* value obtainable from the subexpression beginning at a_i and ending at a_j.

We must keep track of both the minimum and the maximum because the maximal value of an expression may result from multiplying two negative subexpressions.

To solve the subexpression $a_a \ldots a_b$, we can split it into two problems at the kth operator, and recursively solve the subexpressions $a_a \ldots a_k$ and $a_{k+1} \ldots a_b$. In doing so, we must consider all combinations of the minimizing and maximizing subproblems.

The base cases are $M[i, i] = m[i, i] = a_i$, for all i.

$$\begin{aligned} M[a,b] = \max_{a \le k < b} (\max(& M[a,k]\ op_k\ M[k+1,b], \\ & M[a,k]\ op_k\ m[k+1,b], \\ & m[a,k]\ op_k\ M[k+1,b], \\ & m[a,k]\ op_k\ m[k+1,b])) \end{aligned}$$

$$\begin{aligned} m[a,b] = \min_{a \le k < b} (\min(& M[a,k]\ op_k\ M[k+1,b], \\ & M[a,k]\ op_k\ m[k+1,b], \\ & m[a,k]\ op_k\ M[k+1,b], \\ & m[a,k]\ op_k\ m[k+1,b])) \end{aligned}$$

Runtime analysis

There are $O(n^2)$ subproblems, two for each pair of indices $1 \leq a \leq b \leq n$. The subproblem $M[a, b]$ must consider $O(b - a) = O(n)$ smaller subproblems. Thus the total running time is $O(n^3)$.

The Knapsack Problem (without repetition)

You find yourself in a vault chock full of valuable items. However, you only brought a knapsack of capacity W pounds, which means the knapsack will break down if you try to carry more than W pounds in it). The vault has n items, where item i weighs w_i pounds, and can be sold for v_i dollars. You must choose which items to take in your knapsack so that you'll make as much money as possible after selling the items. The weights, w_i, are all integers.

Decisions and State

A solution to an instance of the Knapsack problem will indicate which items should be added to the knapsack. The solution can be broken into n true / false decisions $d_0 \dots d_{n-1}$. For $0 \leq i \leq n - 1$, d_i indicates whether item i will be taken into the knapsack.

In order to decide whether to add an item to the knapsack or not, we need to know if we have enough capacity left over. So the "current state" when making a decision must include the available capacity or, equivalently, the weight of the items that we have already added to the knapsack.

Dynamic Programming Solution

$V[i][w]$ is the maximum value that can be obtained by using a subset of the items $i \dots n - 1$ (last $n - i$ items) which weighs **at most** w pounds. When computing $V[i][w]$, we need to consider all the possible values of d_i (the decision at step i):

1. Add item i to the knapsack. In this case, we need to choose a subset of the items $i+1 \dots n-1$ that weighs at most $w-w_i$ pounds. Assuming we do that optimally, we'll obtain $V[i+1][w-w_i]$ value out of items $i + 1 \dots n - 1$, so the total value will be $v_i + V[i + 1][w - w_i]$.

2. Don't add item i to the knapsack, so we'll re-use the optimal solution for items $i+1 \dots n-1$ that weighs at most w pounds. That answer is in $V[i + 1][w]$.

We want to maximize our profits, so we'll choose the best possible outcome.

$$V[i][w] = \max \begin{pmatrix} V[i+1][w] \\ V[i+1][w-w_i] + v_i \quad \text{if } w \geq w_i \end{pmatrix}$$

To wrap up the loose ends, we notice that $V[n][w] = 0, \forall 0 \leq w \leq W$ is a good base case, as the interval $n \dots n - 1$ contains no items, so there's nothing to add to the knapsack, which means the total sale value will be 0. The answer to our original problem can be found in $V[0][W]$. The

value in $V[i][w]$ depends on values of $V[i+1][k]$ where $k < w$, so a good topological sort would be:

$$\begin{array}{cccc} V[n][0] & V[n][1] & \dots & V[n][W] \\ V[n-1][0] & V[n-1][1] & \dots & V[n-1][W] \\ \vdots & \vdots & & \vdots \\ V[0][0], & V[0][1] & \dots & V[0][W] \end{array}$$

This topological sort can be produced by the pseudo-code below.

```
KNAPSACKDPTOPSORT(n, W)
1  for i in {n, n − 1 . . . 0}
2      for w in {0, 1 . . . W}
3          print (i, w)
```

The full pseudo-code is straight-forward to write, as it closely follows the topological sort and the Dynamic Programming recurrence.

```
KNAPSACK(n, W, s, v)
 1  for i in {n, n − 1 . . . 0}
 2      for w in {0, 1 . . . W}
 3          if i == n
 4              V[i][w] = 0 // initial condition
 5          else
 6              choices = []
 7              APPEND(choices, V[i + 1][w])
 8              if w ≥ w_i
 9                  APPEND(choices, V[i + 1][w − w_i] + v_i)
10              V[i][w] = MAX(choices)
11  return V[0][W]
```

DAG Shortest-Path Solution

The Knapsack problem can be reduced to the single-source shortest paths problem on a DAG (directed acyclic graph). This formulation can help build the intuition for the dynamic programming solution.

The state associated with each vertex is similar to the dynamic programming formulation: vertex (i, w) represents the state where we have considered items $0 \dots i$, and we have selected items whose total weight is at most w pounds. We can consider that the DAG has $n+1$ layers (one layer for each value of i, $0 \le i \le n$), and each layer consists of $W+1$ vertices (the possible total weights are $0 \dots W$).

As shown in figure **??**, each vertex (i, w) has the following outgoing edges:

- Item i is not taken: an edge from (i, w) to $(i+1, w)$ with weight 0

- Item i is taken: an edge from (i, w) to $(i + 1, w + w_i)$ with weight $-v_i$ if $w + w_i \leq W$

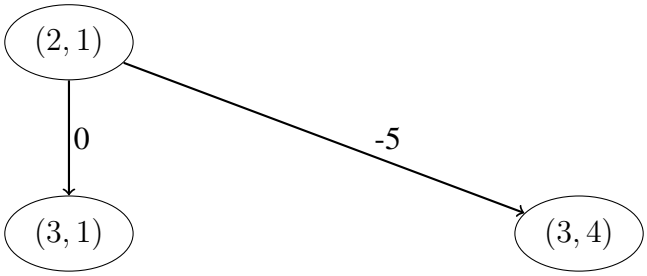

Figure 1: Edges coming out of the vertex $(2, 1)$ in a Knapsack problem instance where item 2 has weight 3 and value 5. If item 2 is selected, the new total weight will be $1 + 3 = 4$, and the total value will increase by 5. Edge weights are negative so that the shortest path will yield the maximum knapsack value.

The source vertex is $(0, 0)$, representing an initial state where we haven't considered any item yet, and our backpack is empty. The destination is the vertex with the shortest path out of all vertices (n, w), for $0 \leq w \leq W$.

Alternatively, we can create a virtual source vertex s, and connect it to all the vertices $(0, w)$ for $0 \leq w \leq W$, meaning that we can leave w pounds of capacity unused (the knapsack will end up weighing $W - w$ pounds). In this case, the destination is the vertex (n, W). This approach is less intuitive, but matches the dynamic programming solution better.

Running Time

The dynamic programming solution to the Knapsack problem requires solving $O(nW)$ sub-problems. The solution of one sub-problem depends on two other sub-problems, so it can be computed in $O(1)$ time. Therefore, the solution's total running time is $O(nW)$.

The DAG shortest-path solution creates a graph with $O(nW)$ vertices, where each vertex has an out-degree of $O(1)$, so there are $O(nW)$ edges. The DAG shortest-path algorithm runs in $O(V + E)$, so the solution's total running time is also $O(nW)$. This is reassuring, given that we're performing the same computation.

The solution running time is **not polynomial** in the input size, instead this algorithm is said to run in **pseudo-polynomial time**.

Polynomial Time vs Pseudo-Polynomial Time (optional)

The input for an instance of the Knapsack problem can be represented in a reasonably compact form as follows (see Figure **??**):

- The number of items n, which can be represented using $O(\log n)$ bits.

- n item weights. We notice that item weights should be between $0 \ldots W$ because we can ignore any items whose weight exceeds the knapsack capacity. This means that each weight can be represented using $O(\log W)$ bits, and all the weights will take up $O(n \log W)$ bits.
- n item values. Let V be the maximum value, so we can represent each value using $O(\log V)$ bits, and all the values will take up $O(n \log V)$ bits.

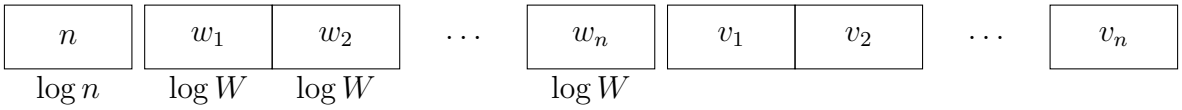

Figure 2: A compact representation of an instance of the Knapsack problem. The item weights, w_i, should be between 0 and W. An item whose weight exceeds the knapsack capacity ($w_i > W$) can be immediately discarded from the input, as it wouldn't fit the knapsack.

The total input size is $O(\log(n) + n(\log W + \log V)) = O(n(\log W + \log V))$. Let $b = \log W$, $v = \log V$, so the input size is $O(n(v + b))$. The running time for the Dynamic Programming solution is $O(nW) = O(n \cdot 2^b)$. Does this make you uneasy? It should, and the next paragraph explains why.

In this course, we use asymptotic analysis to understand the changes in an algorithm's performance as its input size increases – the corresponding buzzword is "scale", as in "does algorithm X scale?". Wo, how does our Knapsack solution runtime change if we double the input size?

We can double the input size by doubling the number of items, so $n' = 2n$. The running time is $O(nW)$, so we can expect that the running time will double. Linear scaling isn't too bad!

However, we can also double the input size by doubling v and b, the number of bits required to represent the item weights and values. v doesn't show up in the running time, so let's study the impact of doubling the input size by doubling b. If we set $b' = 2b$, the $O(n \cdot 2^b)$ result of our algorithm analysis suggests that the running time will increase quadratically!

To drive this point home, Table **??** compares the solution's running time for two worst-case inputs of $6,400$ bits against a baseline worst-case input of $3,200$ bits.

Metric	Baseline	Double n	Double b
n	100 items	200 items	100 items
b	32 bits	32 bits	64 bits
Input size	$3,200$ bits	$6,400$ bits	$6,400$ bits
Worst-case W	$2^{32} - 1 = 4 \cdot 10^9$	$4 \cdot 10^9$	$2^{64} - 1 = 1.6 \cdot 10^{19}$
Running time	$4 \cdot 10^{11}$ ops	$8 \cdot 10^{11}$ ops	$1.6 \cdot 10^{21}$ ops
Input size	1x	2x	2x
Time	1x	2x	$4 \cdot 10^9$x

Table 1: The amounts of time required to solve some worst-case inputs to the Knapsack problem.

The Dynamic Programming solution to the Knapsack problem is a **pseudo-polynomial** algorithm, because the running time will not always scale linearly if the input size is doubled.