

1 Boxing

Subproblem: Let $\text{OPT}(i)$ be the maximum number of boxes from boxes $1, \dots, i$ (sorted in non-decreasing order by height) one can nest, if forced to include box i .

Recurrence: $\text{OPT}(i) = 1 + \max_{j < i} n_{ij} \text{OPT}(j)$, where $n_{ij} = 1$ if box j can be nested inside box i , and 0 otherwise.

Proof of Recurrence: If forced to nest boxes inside box i , you must choose which box to nest next. Note that by the way we sorted the boxes, only lower-indexed boxes may be nested, and not all of them, at that. For any box j which can be nested, the number of boxes we get is 1 from box i , plus as many as we can fit inside box j , which is $\text{OPT}(j)$. If j cannot be nested inside i , then we're left with one nested box, i . Since we get to choose the best j , we take the maximum over all boxes with index less than i .

Base Cases: $\text{OPT}(1) = 1$. The box with the smallest height can't fit any other boxes. Note: we've constructed our recurrence in such a way that we don't actually need a base case. To get intuition for why, try running the algorithm below on an example, with the loop going from 1 to n instead of 2 to n , and omit the base case step.

Algorithm:

```

Memo[ ] = new int[n]

sort boxes by height, nondecreasing

Memo[1] = 1

for all  $i$  and  $j$ , compute  $n_{ij}$ 

for  $i$  from 2 to  $n$ 
    Memo[i] = 1 + max $_{j < i}$   $n_{ij}$  Memo[j]

return max $_i$  Memo[i]
```

Runtime Sorting boxes by height takes $O(n \log n)$ time. Computing n_{ij} requires $O(n^2)$ steps. The algorithm then fills in an array of size $O(n)$, and does $O(n)$ table lookups to fill in each entry. The total to fill in the array is therefore $O(n^2)$, and the total is $O(n^2)$ as well.

2 Tiny Times Tables

Subproblem: Let $\text{OPT}(i, j, s)$ be true if and only if there is a way of parenthesizing $p_i \dots p_j$ to get s .

Recurrence: $\text{OPT}(i, j, s) = \bigvee_{a, b: ab=s} (\bigvee_{k=i+1}^{j-1} \text{OPT}(i, k, a) \wedge \text{OPT}(k+1, j, b))$, where \bigvee denotes "OR" of all the terms indicated, and \wedge denotes "AND."

Proof of Recurrence: You can form s by parenthesizing $p_i \dots p_j$ only if there is some k , a , and b such that you can parenthesize $p_i \dots p_k$ to get a , $p_{k+1} \dots p_j$ to get b , and such that $ab = s$. The recurrence above exhaustively checks for all such a , b , and k .

Base Cases: $\text{OPT}(i, i, s) = T$ if $p_i = s$ and F otherwise.

Algorithm:

```

Memo[ ] = new int[n][n][m]
for  $i = 1, \dots, n$  Memo[i][i][p_i] = T
for  $i = 1, \dots, n$  Memo[i][i][s] = F for all  $s \neq p_i$ 
for all  $\ell$  from 1 to  $n - 1$ 
    for all  $s \in S$ 
        Memo[i][i +  $\ell$ ][s] =  $\bigvee_{a,b:ab=s} (\bigvee_{k=i+1}^{j-1} \text{Memo}[i][k][a] \wedge \text{Memo}[k+1][j][b])$ 
return Memo[1][n][t]
```

Runtime There are n^2m entries in our memo table. Filling in an entry requires us to search all pairs of symbols in S and iterate through all possible values of k for each pair, for a per-entry cost of m^2n . The total runtime is therefore n^3m^3 .