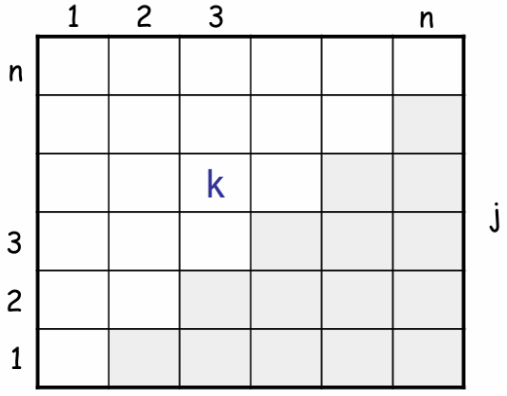
```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n          ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m and s
```

$O(N^3)$

- In a similar matrix s we keep the optimal values of k
- **s[i, j] = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}**




```
PRINT-OPT-PARENS(s, i, j)
if i = j
  then print "A"_i
  else print "("
       PRINT-OPT-PARENS(s, i, s[i, j])
       PRINT-OPT-PARENS(s, s[i, j] + 1, j)
       print ")"
```

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

```
MatrixChainOrder(p[0..n]):
    for i = 1 to n:
        m[i][i] = 0
    return LookupChain(m, p, 1, n)
```

```
LookupChain(m, p, i, j):
    if m[i][j] is already computed:
        return m[i][j]
    if i == j:
        m[i][j] = 0
    else:
        m[i][j] = ∞
        for k = i to j-1:
            q = LookupChain(m, p, i, k) +
                LookupChain(m, p, k+1, j) +
                p[i-1] * p[k] * p[j]
            if q < m[i][j]:
                m[i][j] = q
    return m[i][j]
```

Rod Cutting – Simple Recursion

```
function rodCutRecursive(n, p)
    if n == 0:
        return 0
    maxRevenue = -∞
    for i from 1 to n:
        revenue = p[i] + rodCutRecursive(n - i, p)
        if revenue > maxRevenue:
            maxRevenue = revenue
    return maxRevenue
```

TopDown

```
function rodCutMemoized(n, p):
    memo = array[0..n] filled with NEG_INF
    memo[0] = 0

    function helper(k):
        if memo[k] != NEG_INF:
            return memo[k]
        maxR = -∞
        for i from 1 to k:
            val = p[i] + helper(k - i)
            if val > maxR:
                maxR = val
        memo[k] = maxR
        return maxR
```

BottomUp

```
function rodCutBottomUp(n, p):
    r = array[0..n]
    s = array[0..n]      // s[j] =
    r[0] = 0

    for j from 1 to n:
        maxR = -∞
        best_i = 1
        for i from 1 to j:
            val = p[i] + r[j - i]
            if val > maxR:
                maxR = val
                best_i = i
        r[j] = maxR
        s[j] = best_i
```

```
function reconstructCuts(n, s):
    cuts = []
    while n > 0:
        cuts.append(s[n])
        n = n - s[n]
    return cuts   // sequence of piece-lengths
```

Rod Cutting with cut cost

Recursion

```
function rodCutRecursiveWithCost(n, p, c):
    if n == 0:
        return 0
    maxR = p[n]   // option: do not cut at all
    for i from 1 to n-1:
        val = p[i] + rodCutRecursiveWithCost(n - i, p, c) - c
        if val > maxR:
            maxR = val
    return maxR
```

TopDown

```
  memo = array[0..n] initialized to NEG_INF
  memo[0] = 0
  choice = array[0..n]

  function helper(k):
      if memo[k] != NEG_INF:
          return memo[k]
      best = p[k]       // no cut option
      best_i = k
      for i from 1 to k-1:
          val = p[i] + helper(k - i) - c
          if val > best:
              best = val
              best_i = i
      memo[k] = best
      choice[k] = best_i
      return best

  return (helper(n), choice)
```

BottomUp

```
function rodCutBottomUpWithCost(n, p, c):
    r = array[0..n]
    s = array[0..n]
    r[0] = 0
    s[0] = 0

    for j from 1 to n:
        best = p[j]     // option of no cut
        best_i = j
        for i from 1 to j-1:
            val = p[i] + r[j - i] - c
            if val > best:
                best = val
                best_i = i
        r[j] = best
        s[j] = best_i

    return (r, s)
```

For `n=4`:

j (rod length)	Try i=1	Try i=2	Try i=3	Try i=4	Best `r[j]`	Best cut `s[j]`
1	1	–	–	–	1	1
2	1+1=2	5	–	–	5	2
3	1+5=6	5+1=6	8	–	8	3
4	1+8=9	5+5=10	8+1=9	9	10	2

Example: `n=4`, `c=2`, `p=[1,5,8,9,…]`

j (rod length)	Try i=1	Try i=2	Try i=3	No Cut p[j]	Best `r[j]`	Best cut `s[j]`
1	–	–	–	1	1	1
2	1+1-2=0	–	–	5	5	2
3	1+5-2=4	5+1-2=4	–	8	8	3
4	1+8-2=7	5+5-2=8	8+1-2=7	9	9	4 (no cut)

```
rodCutRecursive(4)
 ├─ i=1 → p[1] + rodCutRecursive(3)
 │              ├─ i=1 → p[1] + rodCutRecursive(2)
 │              ├─ i=2 → p[2] + rodCutRecursive(1)
 │              └─ i=3 → p[3] + rodCutRecursive(0)
 ├─ i=2 → p[2] + rodCutRecursive(2)
 ├─ i=3 → p[3] + rodCutRecursive(1)
 └─ i=4 → p[4] + rodCutRecursive(0)
```

n	Best Revenue `r[n]`	First Cut `s[n]`
0	0	–
1	1	1
2	5	2
3	8	3
4	10	2

LCS Recursive

```python
LCS-Recursive(X, Y, i, j):
    if i == 0 or j == 0:
        return 0
    if X[i] == Y[j]:
        return 1 + LCS-Recursive(X, Y, i-1, j-1)
    else:
        return max(LCS-Recursive(X, Y, i-1, j),
                   LCS-Recursive(X, Y, i, j-1))
```

LCS TopDown

```
LCS-TopDown(X, Y, i, j, dp):
    if i == 0 or j == 0:
        return 0
    if dp[i][j] != -1:
        return dp[i][j]
    if X[i] == Y[j]:
        dp[i][j] = 1 + LCS-TopDown(X, Y, i-1, j-1, dp)
    else:
        dp[i][j] = max(LCS-TopDown(X, Y, i-1, j, dp),
                       LCS-TopDown(X, Y, i, j-1, dp))
    return dp[i][j]
```

LCS BottomUp

```
LCS-BottomUp(X, Y, m, n):
    create dp[0..m][0..n]
    for i from 0 to m:
        dp[i][0] = 0
    for j from 0 to n:
        dp[0][j] = 0
    for i from 1 to m:
        for j from 1 to n:
            if X[i] == Y[j]:
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

MaxSumSubarray

function FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high):

Returns (max_left, max_right, max_sum) where low ≤ max_left ≤ mid < max_right ≤ high

left_sum = -∞

sum = 0

max_left = mid

for i from mid down to low:

```
    sum = sum + A[i]
    if sum > left_sum:
      left_sum = sum
      max_left = i

  right_sum = -∞
  sum = 0
  max_right = mid + 1
  for j from mid + 1 to high:
    sum = sum + A[j]
    if sum > right_sum:
      right_sum = sum
      max_right = j

  return (max_left, max_right, left_sum + right_sum)



function FIND-MAX-SUBARRAY(A, low, high):
  # Returns (i, j, max_sum) for subarray A[i..j]
  if low == high:
    # base case: single element
    return (low, high, A[low])

  mid = floor((low + high) / 2)

  (left_i,  left_j,  left_sum)  = FIND-MAX-SUBARRAY(A, low, mid)
  (right_i, right_j, right_sum) = FIND-MAX-SUBARRAY(A, mid+1, high)
  (cross_i, cross_j, cross_sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
```

```
  if left_sum >= right_sum and left_sum >= cross_sum:
    return (left_i, left_j, left_sum)
  else if right_sum >= left_sum and right_sum >= cross_sum:
    return (right_i, right_j, right_sum)
  else:
    return (cross_i, cross_j, cross_sum)
```

Kadane

```
function KADANE(A):
  max_ending_here = A[0]
  start_temp = 0
  best_i = 0
  best_j = 0
  max_so_far = A[0]

  for k from 1 to n-1:
    if max_ending_here + A[k] < A[k]:
      max_ending_here = A[k]
      start_temp = k
    else:
      max_ending_here = max_ending_here + A[k]

    if max_ending_here > max_so_far:
      max_so_far = max_ending_here
      best_i = start_temp
      best_j = k

  return (best_i, best_j, max_so_far)
```