Example of Chain Product

- We deine $s[i, j]$ to be k, the point at which $A_i A_{i+1} \ldots A_j$ should be split so that $m[i, j]$ have least value.
- A straightforward recursive algorithm based on the recurrence relation would require exponential time to compute $m[1, n]$ for multiplying $A_1 A_2 \ldots A_n$.
- But we observe that number of possible subproblems depends on each choice of the pair of indices i and j satisfying $1 \leq i \leq j \leq n$
- Total number $= \binom{n}{2} + n = \Theta(n^2)$ (includes $m[i, i]$)
- A recursive solution will encounter each subproblem several times in different branches of its recursion tree.

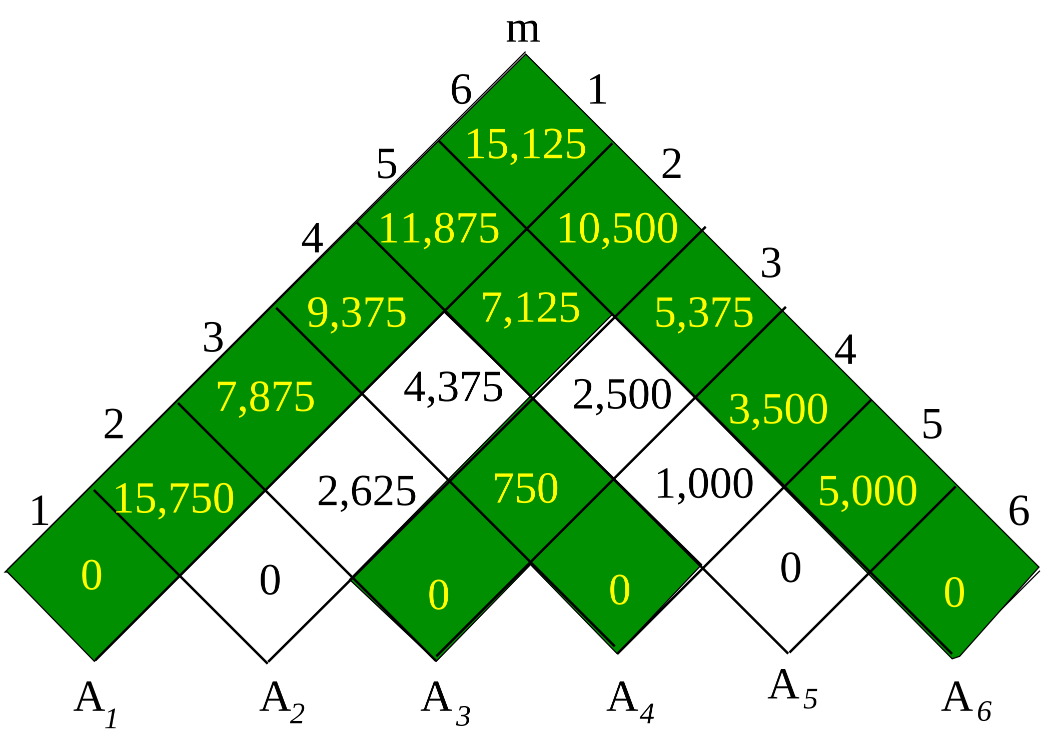
Matrix Chain Product

- If we save solutions for each new subproblem and reuse the same later, then the running time will be O(n^3)

```
n = lenght[p] − 1;
for (i = 1; i ≤ n; i++) m[i, i] = 0;
for (l = 2; l ≤ n; l++)
    for (i = 1; i <= n − l + 1; i++) {
        j = i + l − 1;
        m[i, j] = i + l − 1;
        for (k = i; k ≤ j − 1; k++) {
            q = m[i, k] + m[k + 1, j]
                +p_{i−1} p_k p_j;
            if (q < m[i, j]) {
                m[i, j] = q; s[i, j] = k;
            }
        }
    }
return m, s
```

Matrix Chain Product

- Dimensions of matrices are 30×35, 35×15, 15×5, 5×10, 10×20, 20×25 repsectively.



Matrix Chain Product

- For example, in order to compute $m[2,5]$, we need to consider the following ordering
  - $A_2 \times (A_3, A_4, A_5)$ – gives 13,000 multiplications.
  - $(A_2, A_3) \times (A_4, A_5)$ – requires 7,125 multiplications.
  - $(A_2, A_3, A_4) \times A_5$ – requires 11,375 multiplications.
- A_i has dimension $p_{i-1} \times p_i$.
- Implies that $m[2,5] = m[2,3] + m[4,5] + p_1 p_3 p_5$
- Thus $m[2,5] = 0 + 2500 + 35.5.20 = 7125$
- For computing $m[2,5]$ only unshaded elements of the table are used.
- Note that grouping $(A_4 \times A_5)$ already exist in the first and the last grouping mentioned above.

Matrix Chain Product

- We have only found the number of scalar multiplications.
- But we must also find the parenthesization order.
- Table $s[1:n, 1:n]$ can be used for this purpose.
- $s[i,j]$ records the value of k such that optimal parenthesization splits $A_{i\ldots j}$ as $A_{i\ldots k}$ and $A_{(k+1)\ldots j}$
- So $A_{1\ldots n}$ has to be split as $A_{1\ldots s[1,n]}$ and $A_{s[1,n]+1\ldots n}$ in optimal order.
- Recursive split means optimal split for computing $A_{1\ldots s[1,n]}$ should be $A_{1\ldots s[1,s[1,n]]}$ and $A_{s[1,s[1,n]]+1\ldots s[1,n]}$
- Similarly for othe other first level split the optimal split order should be $A_{s[1,n]+1,s[s[1,n],n]}$ and $A_{s[s[1,n]+1,n]+1,n}$.

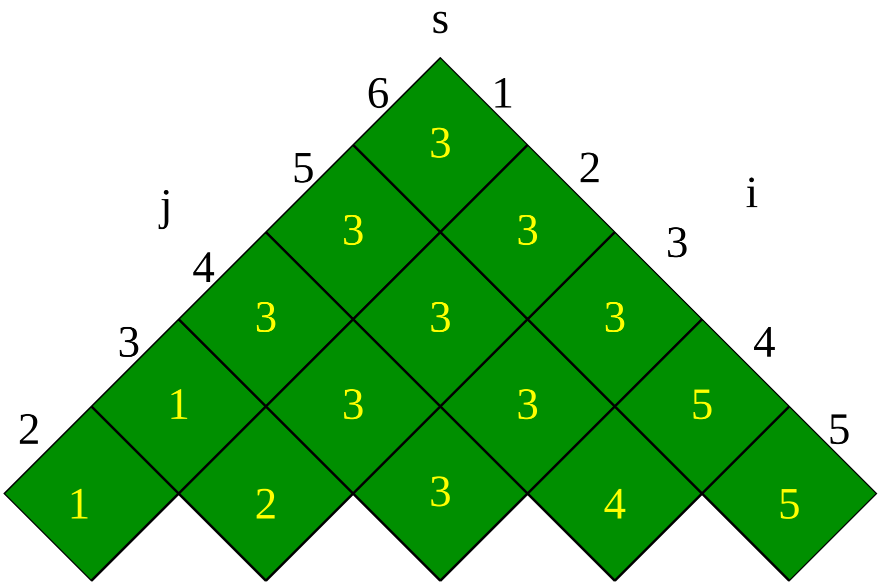
Matrix Chain Product

- Thus we can use the following recursive algorithm for finding product of matrix chain in optimal way.

```
matChainMult(A, s, i, j) {
    if (j > i) {
        X = matChainMult(A, s, i, s[i, j]);
        Y = matChainMult(A, s, s[i, j] + 1, j);
        return matrixMult(X, Y);
    }
    else return A_i
}
```

- Calling matChainMult$(A, s, 1, 6)$ for preceding example

Example



- $s[1,6] = 3$ so top-level split is $A_{1\ldots3}$ and $A_{4\ldots6}$
- $s[1,3] = 1$ so first split in second level is A_1 and $A_{2,3}$
- $s[4,6] = 5$ so second split in second level is $A_{4,5}$ and A_6
- Thus the the optimal order is $((A_1(A_2A_3)((A_4A_5)A_6))$

Steps of Dynamic Programming

- First step is - Optimal substructure.
- Second step - recursive formulation.
- Third step - computing optimal bottom up.
- Fourth step - finding optimal solution.

Overlapping Subproblems

- Only recursive formulation is not enough. The space of subproblems should be small.
- In other words the recursive formulation is such that same subproblems are solved over and over.
- Typically the number of distinct subproblems generated should be polynomial in input size.
- When a recursive algorithm revisits the same subproblems again and again, optimization problem has overlapping subproblems.

Overlapping Subproblems

```
recurMatChain(p, i, j) {
    if (i == j)
        return 0;
    m[i, j] = ∞;
    for (k = i; k < j; k++) {
        q = recurMatChain(p, i, k)
            + recurMatChain(p, k + 1, j)
            +p_{i-1} p_k p_j;
        if (q < m[i, j])
            m[i, j] = q;
    }
    return m[i, j];
}
```