Dynamic Programming

Dynamic Programming

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Basics

- A divide-and-conquer algorithm
  - Partition the problem into disjoint subproblems
  - Solve the subproblems recursively
  - Combine their solutions to solve the original problem
  - When a problem shares sub-problems a divide-and-conquer algorithm does more work than necessary
    - It repeatedly solves the common sub-sub-problems over and over again

Basics

- Dynamic programming
  - Like the divide-and-conquer method, solves problems by combining the solutions to subproblems
  - Applies when sub-problems overlap, i.e., when sub-problems share sub-sub-problems
    - Solves each sub-sub-problem just once
    - Saves its result in a table
    - Uses such a result every time it encounters the same sub-sub-problem, thus avoiding the work of recomputing the answer

Basics

- Dynamic programming is typically applied to optimization problems
  - These problems can have many possible solutions
    - Each solution is characterized by a “fitness” value
    - We wish to find a solution with the optimal (minimum or maximum) value
  - We call such a solution **an** optimal solution to the problem, as opposed to **the** optimal solution, since there may be several solutions that achieve the optimal value

Basics

- Thus, to develop a dynamic-programming algorithm, we need to
  - Characterize the structure of an optimal solution
  - Recursively define its value
  - Compute this value (typically in a bottom-up fashion) and construct an optimal solution from it

A motivating example

- Fibonacci numbers

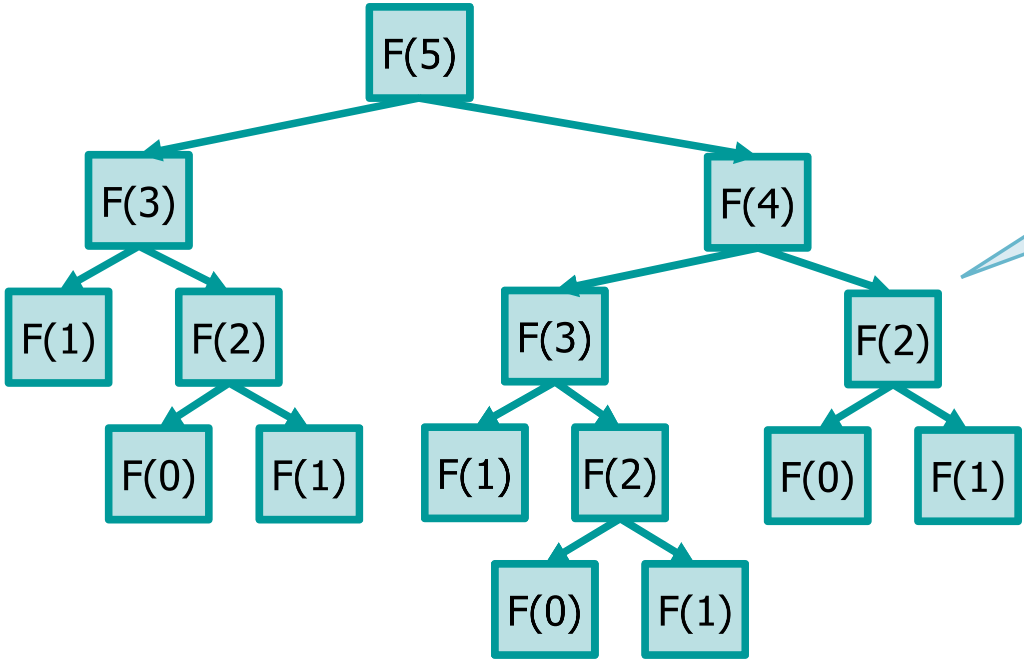
$$
\begin{cases}
F(n) = F(n-2) + F(n-1) & n>1 \\
F(0) = 0 \\
F(1) = 1
\end{cases}
$$

- Divide-and-conquer approach

```
long int fib (long int n) {
  if (n == 0 || n == 1)
    return (n);
  return (fib(n-2) + fib(n-1));
}
```

A motivating example

```
long int fib (long int n) {
  if (n == 0 || n == 1)
    return (n);
  return (fib(n-2) + fib(n-1));
}
```

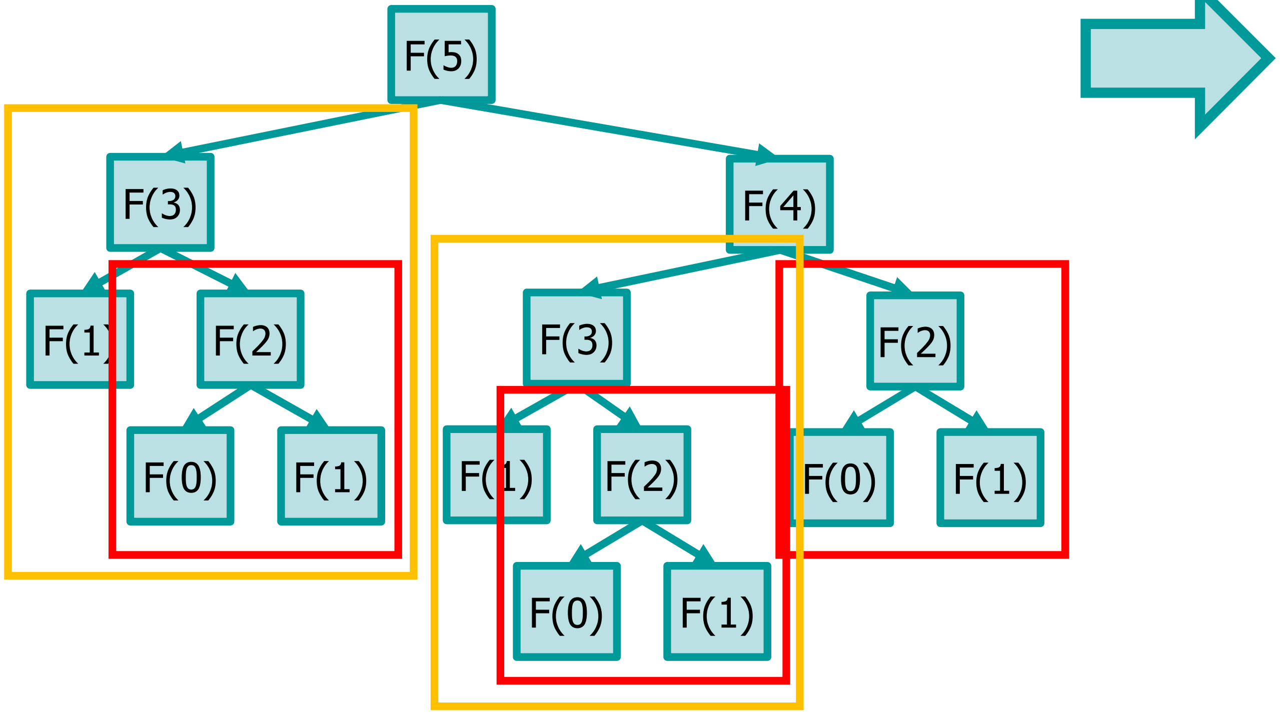


Sub-problems are not independent

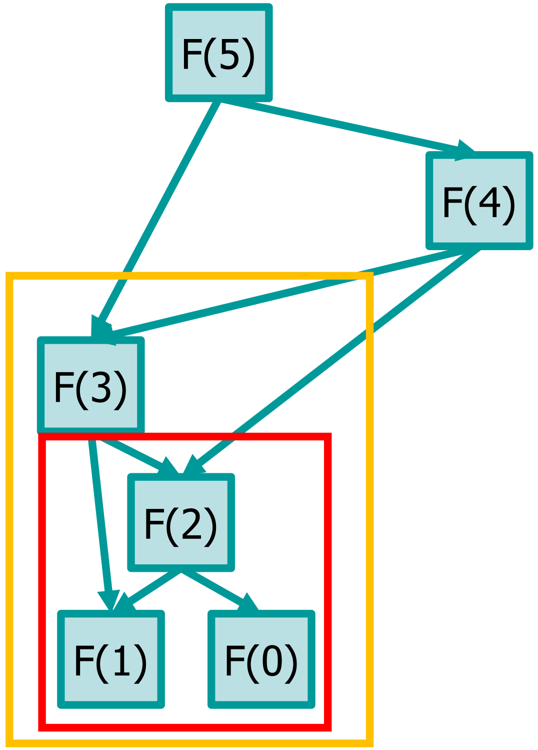
Recomputations lead to inefficiency

A motivating example

```
long int fib (long int n) {
  if (n == 0 || n == 1)
    return (n);
  return (fib(n-2) + fib(n-1));
}
```



Dynamic Programming



Fibonacci: Dynamic programming

- To get a more efficient solution, we must
  - Store solutions to subproblems as soon as they are found
  - Before solving a problem, check whether it has already been solved
  - Reuse existing partial solutions
  - In this way, we can improve the Fibonacci function from being $O(2^n)$ to $O(n)$

Fibonacci: Dynamic programming

```
int main (void) {
  int *known, i, n;
  fprintf(stdout, "Input n: ");
  scanf("%d", &n);
  known = (int *) malloc ((n+1)*sizeof(int));
  if (known==NULL){
    fprintf (stderr, "Memory allocation error.\n");
    exit(EXIT_FAILURE);
  }
  for (i=0; i<=n; i++) {
    known[i] = -1;
  }
  fprintf(stdout, "Fibonacci %d-th term = %d\n",
    n, fib_dp(n, known));
  free(known);
  return EXIT_SUCCESS;
}
```

We define an array **know**

Fibonacci: Dynamic programming

```
int fib_dp (int n, int *known) {
  if (known[n] < 0) {
    if (n==0 || n==1) {
      known[n] = n;
    } else {
      known[n] = fib_dp (n-1, known) +
                 fib_dp (n-2, known);
    }
  }

  return known[n];
}
```

We avoid all recomputations

We store partial results into array **know**

Dynamic programming

- Dynamic programming uses additional memory to save computation time
  - Time-memory trade-off
  - Savings may be dramatic, i.e., exponential-time solution may be transformed into polynomial-time solutions
- There are usually two equivalent ways to implement a dynamic-programming approach

Approach 1

Memoization not memorization !

- Top-down with memoization
  - We write the standard recursive procedure
  - We modify it to
    - Save the result of each subproblem (usually in an array or a hash table)
    - Check whether each problem has previously been solved
      - If so, we return the saved value, saving further computation at this level
      - If not, the procedure computes the value in the usual manner
  - The recursive procedure has been memorized, i.e., it "remembers" what results it has computed previously

Approach 2

- Bottom-up
  - As any sub-problem depends only on "smaller" sub-problems
    - We sort the sub-problems by size
    - We solve them in size order, smallest first
      - We solve each subproblem only once
      - When we encounter it, we have already solved all of its prerequisite sub-problems and we have saved their solutions

Matrix-chain multiplication

- Given a sequence (chain) of matrices M_i (with $i \in [1,n]$) to be multiplied

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_n$$

(where matrix M_i has dimension $r_i \cdot c_i$)

- Fully parenthesize the product in a way that minimizes the number of scalar multiplications necessary to compute M

Example

Please, remind that we know how to multiply two matrices

- Let us suppose we need to compute

$$M = M_1 \cdot M_2 \cdot M_3$$

Matrix size for M_1, M_2, and M_3

$(10 \cdot 100)$ $(100 \cdot 5)$ $(5 \cdot 50)$

Number of products

- We have two possible solutions
  - $(M_1 \cdot M_2) \cdot M_3$
    - Cost = $r_1c_2c_1 + r_1c_3c_2 = 10 \cdot 5 \cdot 100 + 10 \cdot 50 \cdot 5 = 7500$
  - $M_1 \cdot (M_2 \cdot M_3)$
    - Cost = $r_2c_3c_2 + r_1c_3c_1 = 100 \cdot 50 \cdot 5 + 10 \cdot 50 \cdot 100 = 75000$
- Thus, computing the product according to the first parenthesization is **10 times faster**

The standard algorithm

```
for (i=0; i<r1; i++) {
  for (j=0; j<c2; j++) {
    m[i][j] = 0;
    for (k=0; k<c1; k++) {
      m[i][j] += m1[i][k] * m2[k][j];
    }
  }
}
```

- How can we evaluate the product of n matrices?
  - We can use the **standard algorithm** for multiplying **pairs** of matrices

The standard algorithm

```
for (i=0; i<r1; i++) {
  for (j=0; j<c2; j++) {
    m[i][j] = 0;
    for (k=0; k<c1; k++) {
      m[i][j] += m1[i][k] * m2[k][j];
    }
  }
}
```

- Notice that, each matrix pair must be compatible, i.e., computing

$$M = M_1 \cdot M_2$$

  - Is feasible only if c1 = r2 and M will have size (r·c) such that r=r1 and c=c2

The standard algorithm

```
for (i=0; i<r1; i++) {
  for (j=0; j<c2; j++) {
    m[i][j] = 0;
    for (k=0; k<c1; k++) {
      m[i][j] += m1[i][k] * m2[k][j];
    }
  }
}
```

- Moreover, the time to compute M is dominated by the number of scalar multiplications
- For each product pair, this number is equal to

$$r_1 \cdot c_2 \cdot c_1$$

Parenthesization

- When we have **n** matrices, we need to resolve all ambiguities in how the matrices are multiplied together

A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parenthesis

- We need to **parenthesized** the product
  - Matrix multiplication is **associative**, and so all parenthesizations yield the same product
  - As we have seen the **cost** in terms of the number of scalar multiplications is not always the same

Parenthesization

- Given the product

$$M = M_1 \cdot M_2 \cdot M_3 \cdot M_4$$

- How many parenthesitazions are possible?
  - $(((M_1 \cdot M_2) \cdot M_3) \cdot M_4)$
  - $((M_1 \cdot M_2) \cdot (M_3 \cdot M_4))$
  - $((M_1 \cdot (M_2 \cdot M_3)) \cdot M_4)$
  - $(M_1 \cdot ((M_2 \cdot M_3) \cdot M_4))$
  - $(M_1 \cdot (M_2 \cdot (M_3 \cdot M_4)))$

Parenthesization

- It can be proved that given n matrices
  - The number of possible parethesization is $\Omega(2^n)$
  - The number of solutions is thus exponential in n
  - Exhaustively checking (brute-force) all possible parenthesizations does not yield an efficient algorithm

Applying dynamic programming

- The structure of an optimal parenthesization
  - Let us suppose we need to optimize
    - $M_i \cdot M_{i+1} \cdot M_{i+2} \cdot \ldots \cdot M_j$
  - We can suppose to split the product for some value of k, i.e.,
    - $(M_i \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot \ldots \cdot M_j)$
  - If this product is optimum, the 2 subproducts must be optimum

Applying dynamic programming

- A recursive solution
  - Let us call m[i,j] the minimum cost to compute
    - $M_i \cdot M_{i+1} \cdot M_{i+2} \cdot \ldots \cdot M_j$
  - As we compute it as
    - $(M_i \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot \ldots \cdot M_j)$
  - We have that
    - $m[i, j] = 0$ if $i=j$
    - $m[i, j] = m[i,k]+m[k+1,j]+r_i \cdot c_k \cdot c_j$ if $i<j$
  - We define s[i,j] the value of k at which we split the product in optimal parenthesization

Applying dynamic programming

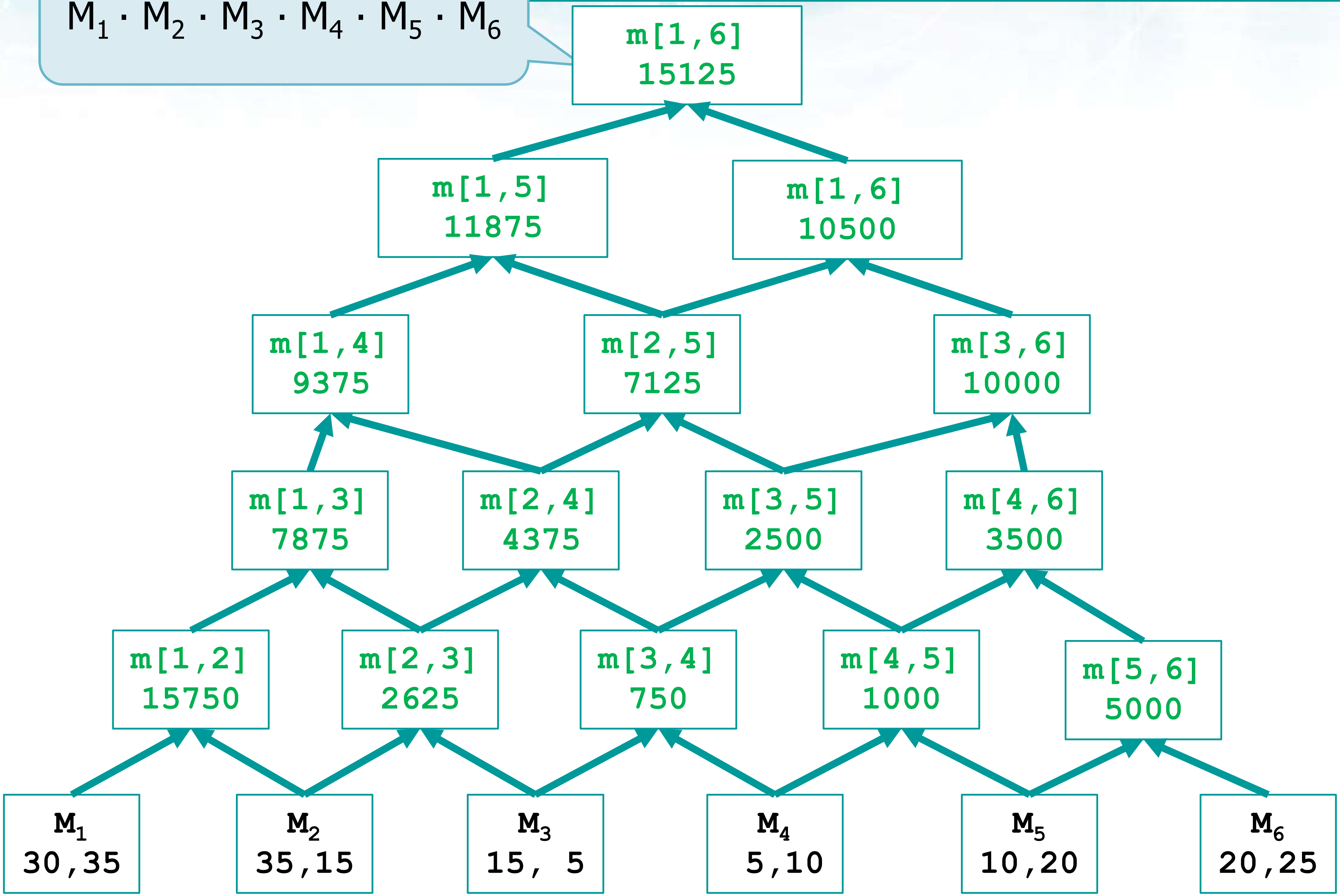
- Computing the optimal costs
  - Instead of computing the solution to the previous recurrence recursively, we compute the optimal cost by using a tabular, bottom-up approach
  - The procedure uses an auxiliary table m[1..n,1..n] to storing the cost to compute the product and and another auxiliary table s[1..n-1,2..n] to identify the value of k for such a cost

Applying dynamic programming

- In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing m[i,j]
  - The previous equation shows that the cost of computing a matrix-chain product depends only on the costs of computing matrix-chain products of fewer matrices
- Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length

Example

$M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5 \cdot M_6$



Example

```
m[1][2] =      0+     0+   30x   35x   15= 15750   MIN

m[2][3] =      0+     0+   35x   15x    5=  2625   MIN

m[3][4] =      0+     0+   15x    5x   10=   750   MIN

m[4][5] =      0+     0+    5x   10x   20=  1000   MIN

m[5][6] =      0+     0+   10x   20x   25=  5000   MIN

m[1][3] =      0+  2625+   30x   35x    5=  7875   MIN
           15750+     0+   30x   15x    5= 18000

m[2][4] =      0+   750+   35x   15x   10=  6000   MIN
            2625+     0+   35x    5x   10=  4375   MIN
```

Example

```
m[3][5] =       0+  1000+    15x    5x   20=  2500    MIN
              750+     0+    15x   10x   20=  3750

m[4][6] =       0+  5000+     5x   10x   25=  6250    MIN
             1000+     0+     5x   20x   25=  3500    MIN

m[1][4] =       0+  4375+    30x   35x   10= 14875    MIN
            15750+   750+    30x   15x   10= 21000
             7875+     0+    30x    5x   10=  9375    MIN

m[2][5] =       0+  2500+    35x   15x   20= 13000    MIN
             2625+  1000+    35x    5x   20=  7125    MIN
             4375+     0+    35x   10x   20= 11375

m[3][6] =       0+  3500+    15x    5x   25=  5375    MIN
              750+  5000+    15x   10x   25=  9500
             2500+     0+    15x   20x   25= 10000
```

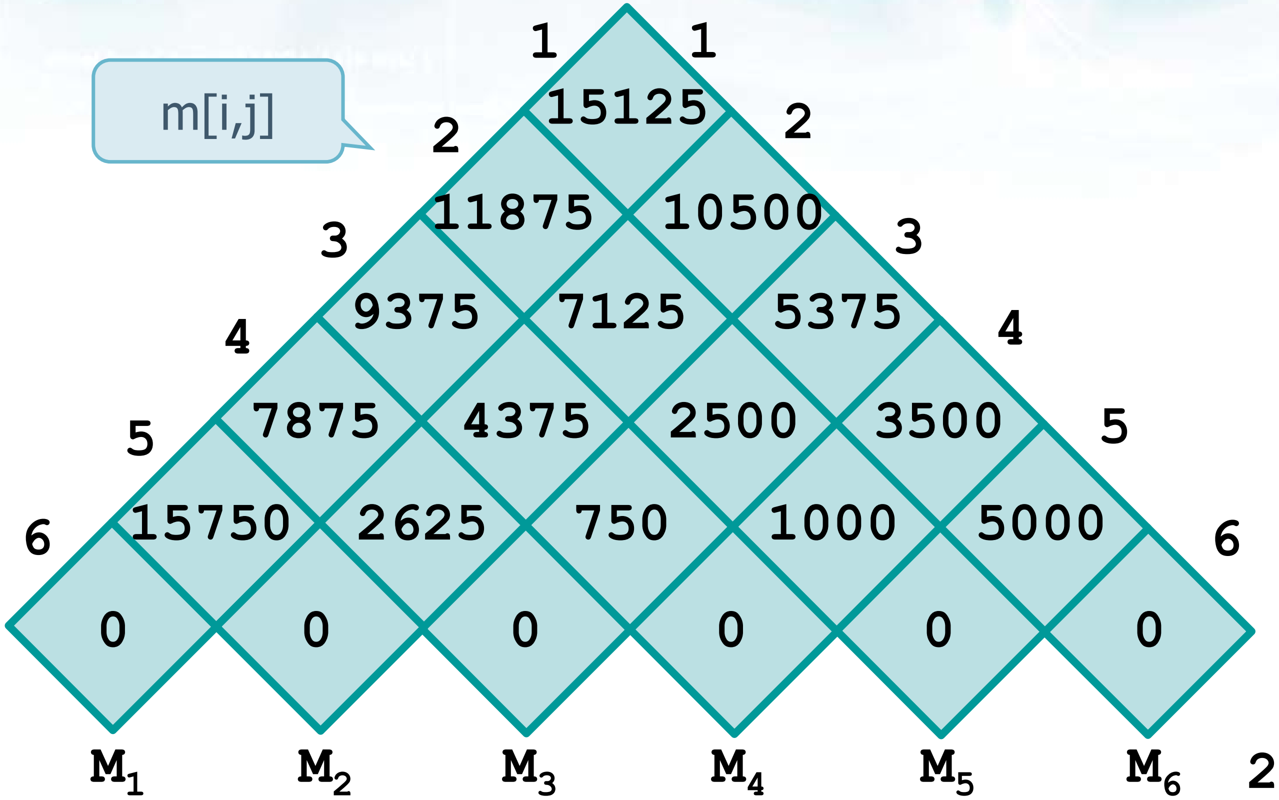
Example

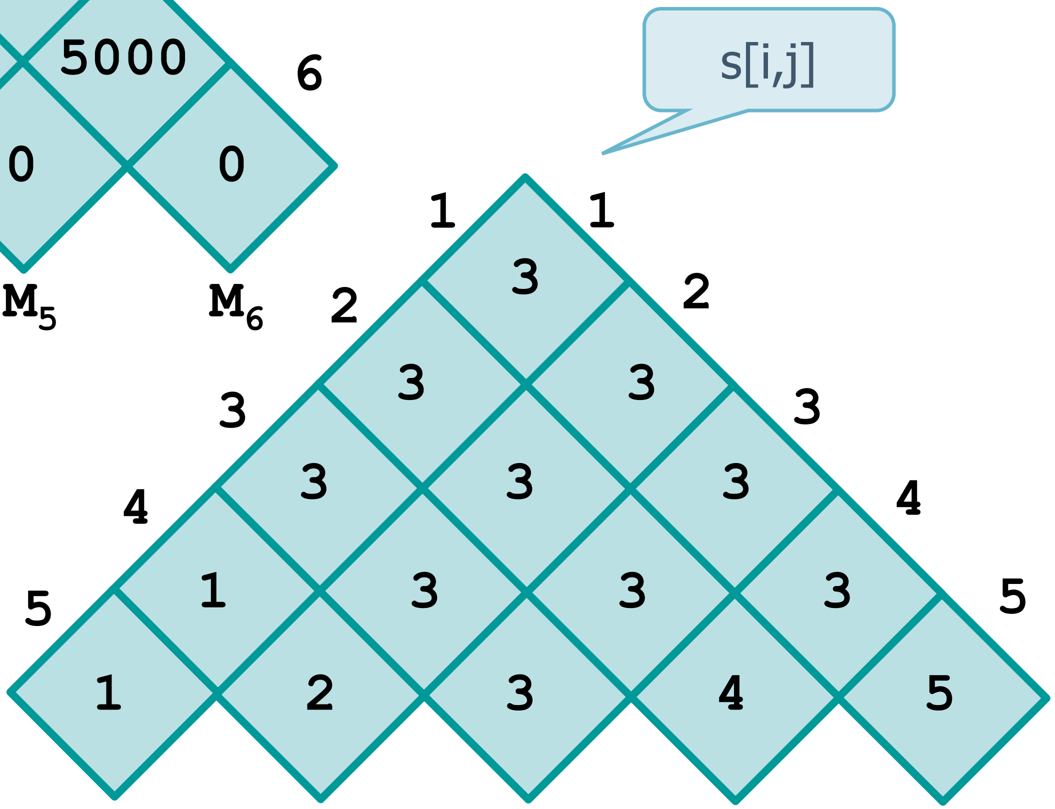
```
m[1][5] =      0+  7125+    30x    35x    20= 28125
           15750+  2500+    30x    15x    20= 27250
            7875+  1000+    30x     5x    20= 11875    MIN
            9375+     0+    30x    10x    20= 15375

m[2][6] =      0+  5375+    35x    15x    25= 18500
            2625+  3500+    35x     5x    25= 10500    MIN
            4375+  5000+    35x    10x    25= 18125
            7125+     0+    35x    20x    25= 24625

m[1][6] =      0+ 10500+    30x    35x    25= 36750
           15750+  5375+    30x    15x    25= 32375
            7875+  3500+    30x     5x    25= 15125    MIN
            9375+  5000+    30x    10x    25= 21875
           11875+     0+    30x    20x    25= 26875
```

Example


m[i,j]
1 1
15125
2 2
11875 10500
3 3
9375 7125 5375
4 4
7875 4375 2500 3500
5 5
15750 2625 750 1000 5000
6 6
0 0 0 0 0 0
M_1 M_2 M_3 M_4 M_5 M_6


s[i,j]
1 1
3
2 2
3 3
3 3
3 3 3
4 4
1 3 3 3
5 5
1 2 3 4 5

Implementation

```
p = readSizes (argv[1], &n);
m = (int **)util_matrix_alloc(n+1, n+1, sizeof(int));
for (i=0; i<=n; i++) {
  for (j=0; j<=n; j++) {
    m[i][j] = ((i==j) ? 0 : INT_MAX);
  }
}
best = matrixChainOrder (p, m, n);
matrixChainPrint (m, 1, n);
```

Implementation

```
int *readSizes(char *filename, int *num) {
  int i, n, *p;
  FILE *fp;

  fp = util_fopen(filename, "r");
  fscanf(fp, "%d", &n);
  p = (int *)util_malloc((n+1)*sizeof(int));
  for (i=0; i<n; i++) {
    fscanf(fp, "%dx%d", &p[i], &p[i+1]);
  }
  fclose(fp);
  *num = n;
  return p;
}
```

Implementation

```
int matrixChainOrder(int *p, int **m, int n) {
  int i, j, k, l, cost;
  for (l=2; l<=n; l++) {
    for (i=1; i<=n-l+1; i++) {
      j = i+l-1;
      for (k=i; k<=j-1; k++) {
        cost = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
        if (cost < m[i][j]) {
          m[i][j] = cost;
          m[j][i] = k;
        }
      }
    }
  }
  return m[1][n];
}
```

Implementation

```
void matrixChainPrint (int **m, int i, int j) {
  int k;

  if (i == j) {
    printf("A%d", i);
    return;
  }

  k = m[j][i];
  printf("(");
  matrixChainPrint(m, i, k);
  printf(" * ");
  matrixChainPrint(m, k+1, j);
  printf(")");
}
```

Elements of dynamic programming

- When the method applies, i.e., when should we look for a dynamic-programming solution to a problem?
- Two key ingredients
  - Optimal substructure
  - Overlapping subproblems

Optimal substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems
  - In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems

Overlapping subproblems

- The space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems
  - Typically, the total number of distinct subproblems is a polynomial in the input size

Longest common subsequence

- A subsequence of a given sequence is just the given sequence with zero or more elements left out
  - For example, {A, E, F, H} is a subsequence of {A, B, C, D, E, F, G, H}
- Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y
  - For example, {D, F, H} is a common subsequence of {A, B, C, D, E, F, G, H} and {D, F, G, H, I, L, M, N}

Longest common subsequence

- In the longest-common-subsequence (LCS) problem, we are given two sequences
  - $X = \{x_1, x_2, \ldots, x_m\}$
  - $Y = \{y_1, y_2, \ldots, y_n\}$
- And wish to find a maximum-length common subsequence of X and Y

Brute-force solution

- In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find
- Each subsequence of X corresponds to a subset of the indices {1, 2, …, m} of X
- Because X has 2m subsequences, this approach requires exponential time, making it impractical for long sequences

Implementation

```
... given strX and strY ...

lX = strlen(strX)+1;
l = strlen(strY)+1;
b = (int **)2D_malloc(lX, lY, sizeof(int));
c = (int **)2D_malloc(lX, lY, sizeof(int));

l = lcsLength(strX, strY, b, c);
printf("LCS length: %d\n", length);

printf("LCS: ");
lcsPrint(strX, b, c, lengthX-1, lengthY-1);
printf("\n");

2D_dispose((void ***)b, lX, lY, NULL);
2D_dispose((void ***)c, lX, lY, NULL);
```

Implementation

```
int lcsLength (
  char *strX, char *strY, int **b, int **c) {
  int i, j, m = strlen(strX), n = strlen(strY);
  for (i=1; i<=m; i++) {
    for (j=1; j<=n; j++) {
      if (strX[i-1] == strY[j-1]) {
         c[i][j] = c[i-1][j-1] + 1; b[i][j] = DIAG;
      } else {
         if (c[i-1][j] >= c[i][j-1]) {
          c[i][j] = c[i-1][j]; b[i][j] = UP;
        } else {
          c[i][j] = c[i][j-1]; b[i][j] = LEFT;
        }
      }
    }
  }
  return c[m][n];
}
```

Implementation

```
void lcsPrint (
  char *strX, int **b, int **c, int i, int j) {
  if (i!=0 && j!=0) {
    if (b[i][j] == DIAG) {
      lcsPrint(strX, b, c, i-1, j-1);
      printf("%c", strX[i-1]);
    } else {
      if (b[i][j] == UP) {
        lcsPrint(strX, b, c, i-1, j);
      } else {
        lcsPrint(strX, b, c, i, j-1);
      }
    }
  }
}
```

Example

- Find the LCS between the two strings
  - X = {A, B, C, B, D, A, B}
  - Y = {A, D, C, A, B, A}

	j	0	1	2	3	4	5	6
i		yi	B	D	C	A	B	A
0	xi	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	1	←1	1
2	B	0	1	←1	←1	↑1	2	←2
3	C	0	↑1	↑1	2	←2	↑2	↑2
4	B	0	1	↑1	↑2	↑2	↑3	↑3
5	D	0	↑1	2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	3	↑3	4
7	B	0	1	↑2	↑2	↑3	4	↑4