Chapter 3. Dynamic Programming

Algorithmics and Programming III

FIB

Albert Oliveras Enric Rodríguez

Q1 2019–2020

Version October 21, 2019

Chapter 3. Dynamic Programming

Chapter 3. Dynamic Programming

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$
- $F_5 = 8$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$
- $F_5 = 8$
- $F_6 = 13$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$
- $F_5 = 8$
- $F_6 = 13$
- $F_7 = 21$

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$
- $F_5 = 8$
- $F_6 = 13$
- $F_7 = 21$
- ...

Example: Fibonacci Numbers

We want to compute the Fibonacci numbers, defined as:

$$F_k = \begin{cases} 1 & \text{if } k = 0, 1 \\ F_{k-1} + F_{k-2} & \text{if } k > 1 \end{cases}$$

- $F_0 = 1$
- $F_1 = 1$
- $F_2 = 2$
- $F_3 = 3$
- $F_4 = 5$
- $F_5 = 8$
- $F_6 = 13$
- $F_7 = 21$
- ...

How many spirals (in both directions)?

Example: Fibonacci Numbers

- A program that directly translates the recursive definition:

```
int F(int k) {
  if (k <= 1) return 1;
  return F(k-1) + F(k-2);
}
int main() {
  int k;
  cin >> k;
  cout << F(k) << endl;
}
```

Example: Fibonacci Numbers

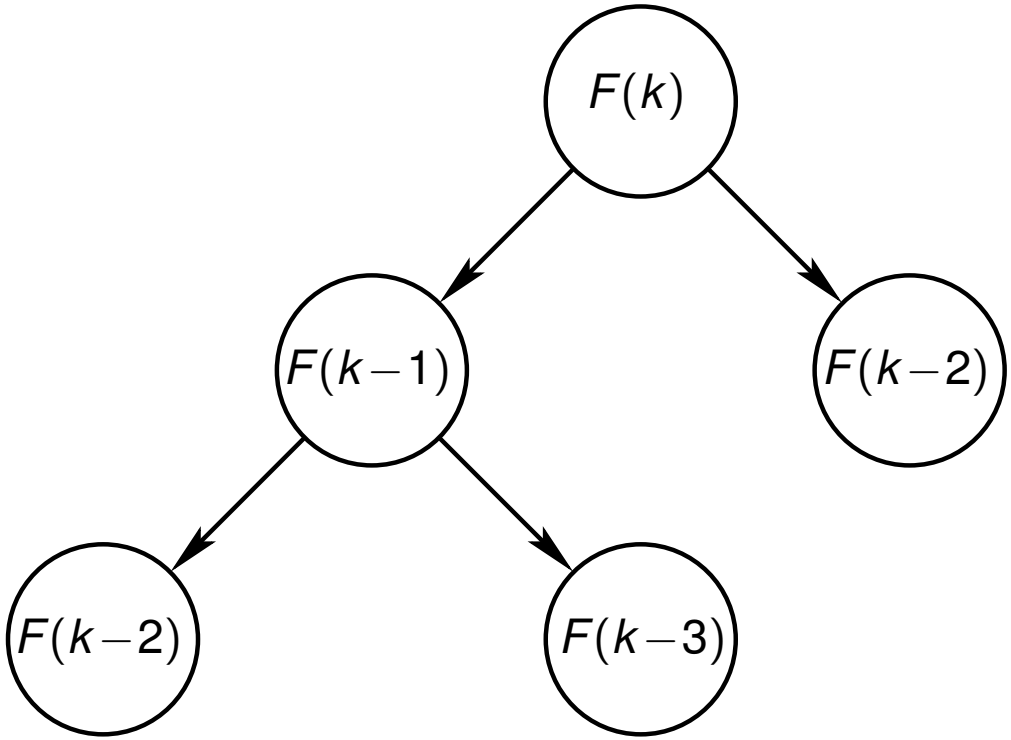
- A program that directly translates the recursive definition:

```
int F(int k) {
  if (k <= 1) return 1;
  return F(k-1) + F(k-2);
}
int main() {
  int k;
  cin >> k;
  cout << F(k) << endl;
}
```

- Let us analyse the cost of this program
- Let $C(k)$ be the no. of times code in red is run when computing $F(k)$
- For $k = 0, 1$ it is run exactly once: $C(0) = C(1) = 1$
- For $k > 1$ the number is $C(k) = C(k-1) + C(k-2)$
- So $C(k) = F(k) = \Theta(\phi^k)$ where $\phi = \frac{1+\sqrt{5}}{2} \approx= 1.618033...$ (golden ratio)

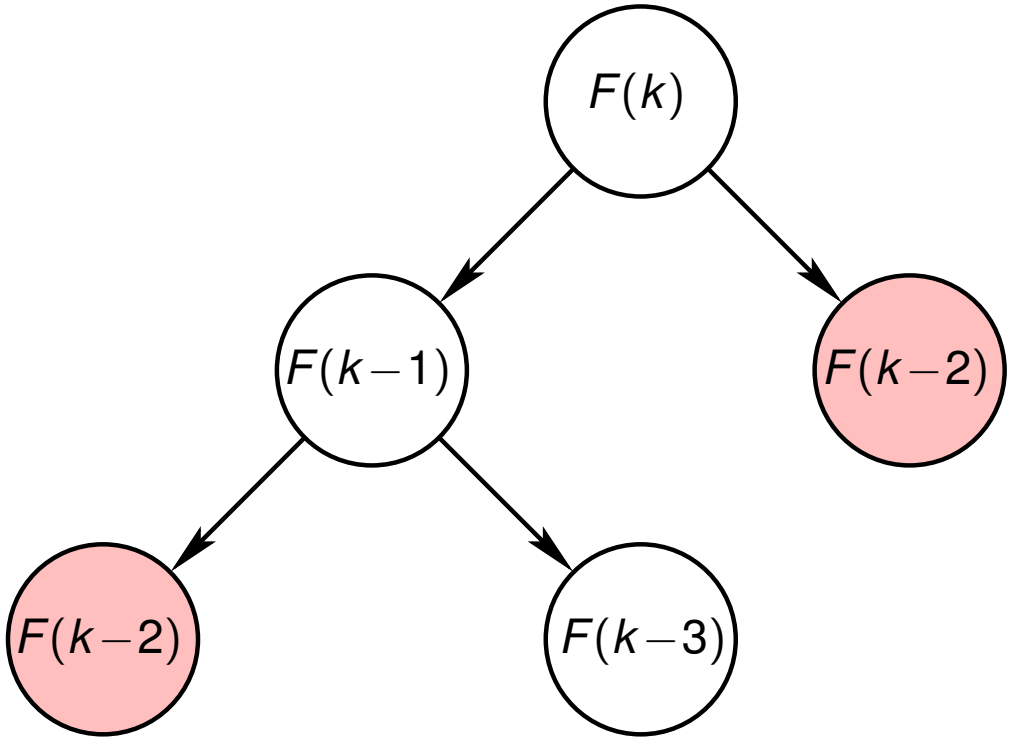
Example: Fibonacci Numbers

- Let us try to make a more efficient program
- Consider the tree of recursive calls



Example: Fibonacci Numbers

- Let us try to make a more efficient program
- Consider the tree of recursive calls



- We are repeating computations!

Example: Fibonacci Numbers

- Idea: memorize previous computations
- E.g. in an array of $k + 1$ integers (a position for each value in $0 \ldots k$)

```
const int UNDEF = -1;
vector<int> f;

int F(int k) {
  int& res = f[k];
  if (res != UNDEF) return res; // already computed: return
  if (k <= 1) return res = 1;   // store computed value
  return res = F(k-2) + F(k-1); // store computed value
}
int main() {
  int k;
  cin >> k;
  f = vector<int>(k+1, UNDEF);  // all UNDEFined at first
  cout << F(k) << endl;
}
```

- The cost of computing $F(k)$ is $\Theta(k)$

Memoization

- In top down dynamic programming (aka memoization)
  we extend a recursive algorithm
  by storing the solutions of the previously solved problems

- Hence, if a subproblem has already occurred, we do not need to recompute the solution: just reuse the one that was found before!

Memoization

- Solutions are cached in a dictionary which, given a key that represents a subproblem, returns the corresponding solution

Memoization

- Solutions are cached in a dictionary which,
  given a key that represents a subproblem,
  returns the corresponding solution
- The term "*(dynamic) programming*" comes from the fact that
  the dictionary is usually implemented with a (multidimensional) table

  Frequently the (parameters of the) subproblems are identified with

  - naturals
  - pairs of naturals
  - triplets of naturals

  so the dictionary is

  - a vector
  - a matrix
  - a cube
- This way lookups of previous problems can be made in constant time

Memoization

- Initially each entry of the table has a special undefined value that indicates that the entry is still to be filled in

Memoization

- Initially each entry of the table has a special undefined value that indicates that the entry is still to be filled in
- When a subproblem is found for 1st time in the unfolding of recursion, its solution is computed and stored in the table at its respective position

Memoization

- Initially each entry of the table has a special undefined value
  that indicates that the entry is still to be filled in
- When a subproblem is found for 1st time in the unfolding of recursion,
  its solution is computed and stored in the table at its respective position
- Afterwards, every time this subproblem is found,
  the value stored in the table is looked up and returned

Chapter 3. Dynamic Programming

Example: Fibonacci Numbers

An alternative iterative solution for the Fibonacci numbers:

1. start filling the vector for 0, 1 following the base case of the definition
2. fill the vector for 2, 3, . . . , k in this order using the recurrence

Example: Fibonacci Numbers

An alternative iterative solution for the Fibonacci numbers:

1. start filling the vector for 0, 1 following the base case of the definition
2. fill the vector for 2, 3, . . . , k in this order using the recurrence

```
vector<int> f;

int main() {
  int k;
  cin >> k;
  f = vector<int>(k+1);
  f[0] = f[1] = 1;              // base cases
  for (int i = 2; i <= k; ++i)
    f[i] = f[i-1] + f[i-2]; // already found for i-1, i-2

  cout << f[k] << endl;
}
```

Example: Fibonacci Numbers

An alternative iterative solution for the Fibonacci numbers:

1. start filling the vector for 0, 1 following the base case of the definition
2. fill the vector for 2, 3, . . . , k in this order using the recurrence

```
vector<int> f;

int main() {
  int k;
  cin >> k;
  f = vector<int>(k+1);
  f[0] = f[1] = 1;              // base cases
  for (int i = 2; i <= k; ++i)
    f[i] = f[i-1] + f[i-2]; // already found for i-1, i-2

  cout << f[k] << endl;
}
```

The cost of computing $F(k)$ is $\Theta(k)$

Tabulation

- In bottom up dynamic programming (aka tabulation),
  the computation follows an order among subproblems, so that
  none is tackled before having solved the smaller problems it depends on

Tabulation

- In bottom up dynamic programming (aka tabulation),
  the computation follows an order among subproblems, so that
  none is tackled before having solved the smaller problems it depends on

- The base cases of the recursion are the first to be solved
- The problem that we want to solve is the last to be solved

Example: Knapsack

Knapsack problem

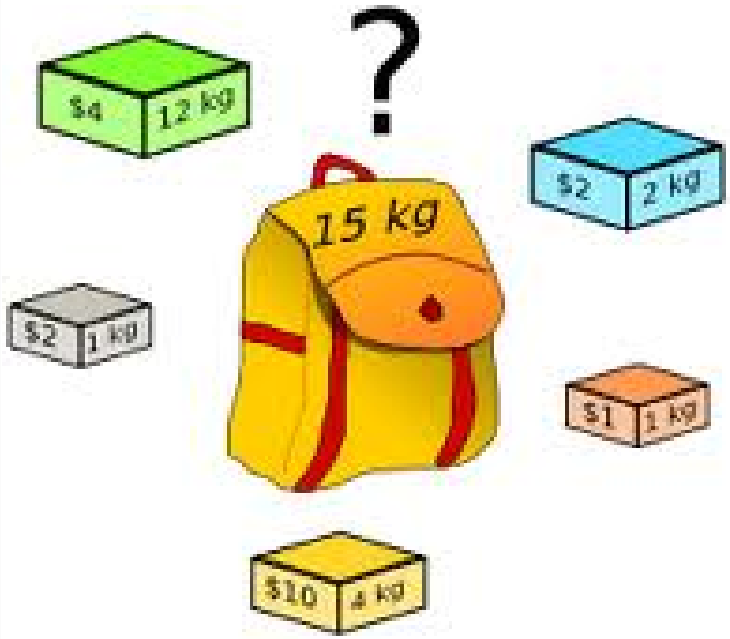
Given

- a knapsack that can store a weight W, and
- n objects with
  - weights $w_1, w_2, \ldots, w_n$, and
  - values $v_1, v_2, \ldots, v_n$

find a selection $S \subseteq \{1, \ldots, n\}$ of the objects

- with maximum value $\sum_{i \in S} v_i$, and
- which does not exceed the capacity:

$$\sum_{i \in S} w_i \leq W.$$



Example: Knapsack

- A recursive algorithm:

```
int n, W;
vector<int> v, w;

// Returns max value using first k objects with capacity W
int opt(int k, int W) {
  if (k == 0) return 0;
  if (w[k-1] > W) return opt(k-1, W);           // too heavy
  else return max(opt(k-1, W-w[k-1]) + v[k-1], // take it
                  opt(k-1, W));                 // do not
}

int main() {
  cin >> n >> W;
  v = w = vector<int>(n);
  for (int& x : v) cin >> x;
  for (int& x : w) cin >> x;
  cout << opt(n, W) << endl;
}
```

Example: Knapsack

- Top down:

```
int n, W;
vector<int> v, w;
vector<vector<int>> m;
const int UNDEF = -1;

int opt(int k, int W) {
  int& res = m[k][W];
  if (res != UNDEF) return res;
  if (k == 0) return res = 0;
  if (w[k-1] > W) return res = opt(k-1, W);
  else return res = max(opt(k-1, W - w[k-1]) + v[k-1],
                        opt(k-1,W)); }
int main() {
  cin >> n >> W;
  v = w = vector<int>(n);
  m = vector<vector<int>>(n+1, vector<int>(W+1, UNDEF));
  for (int& x : v) cin >> x;
  for (int& x : w) cin >> x;
  cout << opt(n, W) << endl; }
```

Example: Knapsack

- Bottom up:

```
int n, W;
vector<int> v, w;

int main() {
  cin >> n >> W;
  v = w = vector<int>(n);
  vector<vector<int>> m(n+1, vector<int>(W+1, 0));
  for (int& x : v) cin >> x;
  for (int& x : w) cin >> x;
 // m[k][j] = max value with first k objects and capacity j
  for (int k = 1; k <= n; ++k) {
    for (int j = 0; j <= W; ++j)
      if (w[k-1] > j)
        m[k][j] = m[k-1][j];
      else
        m[k][j] = max(m[k-1][j-w[k-1]] + v[k-1], m[k-1][j]);
  }
  cout << m[n][W] << endl;
}
```

Chapter 3. Dynamic Programming

Time and Space

- Time
  - When can top down be more efficient than bottom up?

    When one does not have to solve all subproblems,
    as top down only solves the subproblems that are strictly needed

Time and Space

- Time
  - When can top down be more efficient than bottom up?

    When one does not have to solve all subproblems,
    as top down only solves the subproblems that are strictly needed
  - When can bottom up be more efficient than top down?

    When all subproblems have to be solved at least once,
    due to the cost of the recursion
  - Usually bottom up is more efficient than top down
    (even when not all subproblems have to be solved at least once)

Time and Space

- Time
  - When can top down be more efficient than bottom up?

    When one does not have to solve all subproblems,
    as top down only solves the subproblems that are strictly needed
  - When can bottom up be more efficient than top down?

    When all subproblems have to be solved at least once,
    due to the cost of the recursion
  - Usually bottom up is more efficient than top down
    (even when not all subproblems have to be solved at least once)
- Space
  - A bottom up program often allows space optimizations as one can discard the solutions of the subproblems that are no longer needed
  - Not possible with top down, as the space is always necessary

Time and Space

- Bottom up program for Fibonacci with space optimization:

```
int main() {
  int n;
  cin >> n;
  if (n <= 1) cout << 1 << endl;

  int p = 1; // Only need to keep track of the last two
  int f = 1;
  for (int k = 2; k <= n; ++k) {
    int t = p;
    p = f;
    f += t;
  }
  cout << f << endl;
}
```

- We use $\Theta(1)$ additional space instead of $\Theta(k)$

Example: Catalan Numbers

Parenthesizations and Catalan numbers

- A parenthesization of length n is
  a sequence of n characters (and) that match correctly
- For example, ((())), ()(()), ()()() are parenthesizations of length 6
- For example,)()(is not a parenthesization
- The n-th Catalan number C_n is the no. of parenthesizations of length $2n$

Example: Catalan Numbers

Parenthesizations and Catalan numbers

- A parenthesization of length n is a sequence of n characters (and) that match correctly
- For example, ((())), ()(()), ()()() are parenthesizations of length 6
- For example,)()(is not a parenthesization
- The n-th Catalan number C_n is the no. of parenthesizations of length $2n$

- $C_0 = 1$: the empty sequence
- $C_1 = 1$: ()
- $C_2 = 2$: ()(), (())
- $C_3 = 5$: ((())), ()(()), (())(), (()()), ()()()

Example: Catalan Numbers

Parenthesizations and Catalan numbers

- A parenthesization of length n is a sequence of n characters (and) that match correctly
- For example, ((())), ()(()), ()()() are parenthesizations of length 6
- For example,)()(is not a parenthesization
- The n-th Catalan number C_n is the no. of parenthesizations of length $2n$

- $C_0 = 1$: the empty sequence
- $C_1 = 1$: ()
- $C_2 = 2$: ()(), (())
- $C_3 = 5$: ((())), ()(()), (())(), (()()), ()()()

Parenthesizations can be obtained recursively:

- $\emptyset$ is a parenthesization
- If E_1 and E_2 are parenthesizations, then so is $(E_1)E_2$.

Example: Catalan Numbers

- Top-down:

```
vector<int> t;

int num(int i) {   // Returns #parenthesizations of length i
  int& res = t[i]; // I.e., (i/2)-th Catalan number
  if (res != UNDEF) return res;
  if (i == 0) return res = 1; // Empty word.
  else {                      // Rule "(E1) E2".
    res = 0;
    for (int len1 = 0; len1 <= i-2; ++len1) {
      int len2 = i-2-len1;           // |E1| + |E2| + 2 = i.
      res += num(len1) * num(len2); // Cartesian product.
    }
    return res;
} }

int main() {
  int n; cin >> n;
  t = vector<int>(n+1, UNDEF);
  cout << num(n) << endl;}
```

Example: Catalan Numbers

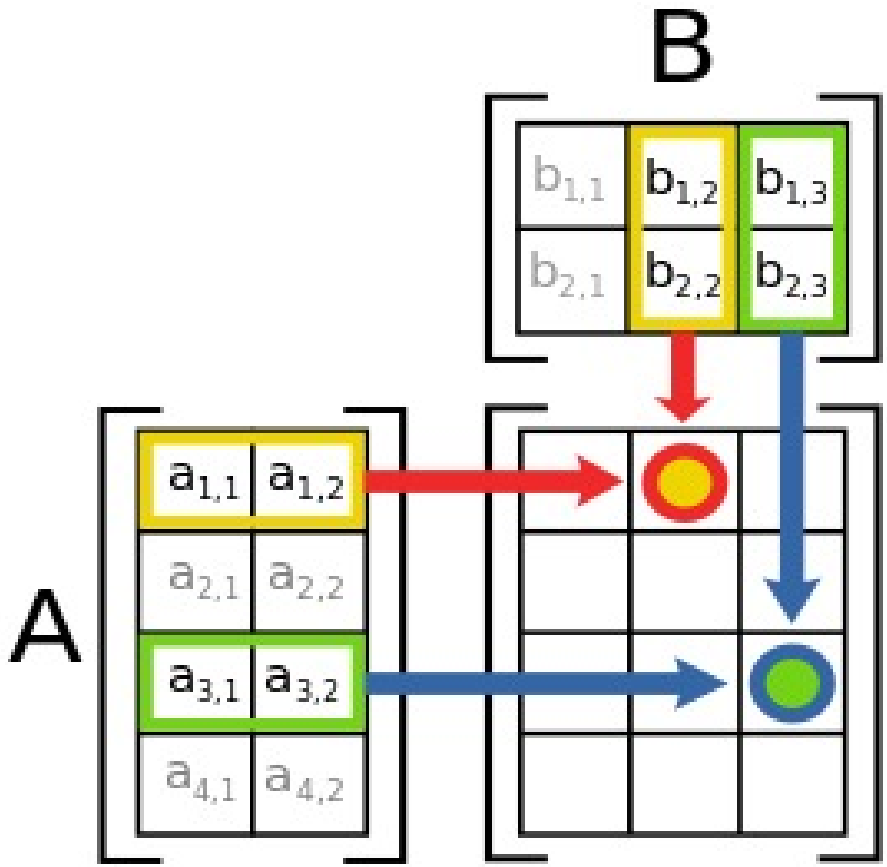
- Bottom-up:

```
int main() {
  int n;
  cin >> n;

  vector<int> t(n+1, 0);          // t[i] == #paren. of length i
  t[0] = 1;                       // Empty word.
  for (int i = 1; i <= n; ++i) // Rule "(E1) E2".
    for (int len1 = 0; len1 <= i-2; ++len1) {
      int len2 = i-2-len1;
      t[i] += t[len1] * t[len2];
    }

  cout << t[n] << endl;
}
```

Example: Matrix Sequence Multiplication



- We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B.
- If A is a $p \times q$ matrix and B is a $q \times r$ matrix, then AB is a $p \times r$ matrix.

Example: Matrix Sequence Multiplication

- Program for the product of a $p \times q$ matrix A and a $q \times r$ matrix B

```
vector<vector<int>> prod(const vector<vector<int>>& A,
                         const vector<vector<int>>& B) {
  int p = A.size();
  int q = B.size();
  int r = B[0].size();

  vector<vector<int>> AB(p, vector<int>(r, 0));
  for (int i = 0; i < p; ++i)
    for (int j = 0; j < r; ++j)
      for (int k = 0; k < q; ++k)
        AB[i][j] += A[i][k] * B[k][j];
  return AB;
}
```

Example: Matrix Sequence Multiplication

- Program for the product of a $p \times q$ matrix A and a $q \times r$ matrix B

```
vector<vector<int>> prod(const vector<vector<int>>& A,
                         const vector<vector<int>>& B) {
  int p = A.size();
  int q = B.size();
  int r = B[0].size();

  vector<vector<int>> AB(p, vector<int>(r, 0));
  for (int i = 0; i < p; ++i)
    for (int j = 0; j < r; ++j)
      for (int k = 0; k < q; ++k)
        AB[i][j] += A[i][k] * B[k][j];
  return AB;
}
```

- Time in `prod` dominated by no. of scalar multiplications in red: pqr.
  In what follows, we will consider that the cost of `prod` is pqr.

Example: Matrix Sequence Multiplication

- Given a sequence $A_0, A_1, \ldots, A_{m-1}$ of m matrices to be multiplied, we wish to compute the product $A_0 \cdot A_1 \cdots A_{m-1}$

Example: Matrix Sequence Multiplication

- Given a sequence $A_0, A_1, \ldots, A_{m-1}$ of m matrices to be multiplied, we wish to compute the product $A_0 \cdot A_1 \cdots A_{m-1}$
- For example, let us consider a sequence A_0, A_1, A_2 of 3 matrices of dimensions 50×5, 5×100 and 100×10, respectively
- We can parenthesize in two different ways:
  - $((A_0\ A_1)\ A_2)$
  - $(A_0\ (A_1\ A_2))$

Example: Matrix Sequence Multiplication

- Given a sequence $A_0, A_1, \ldots, A_{m-1}$ of m matrices to be multiplied, we wish to compute the product $A_0 \cdot A_1 \cdots A_{m-1}$
- For example, let us consider a sequence A_0, A_1, A_2 of 3 matrices of dimensions 50×5, 5×100 and 100×10, respectively
- We can parenthesize in two different ways:
  - $((A_0\ A_1)\ A_2)$
  - $(A_0\ (A_1\ A_2))$
- It turns out that different parenthesizations have different costs!
  - $((A_0\ A_1)\ A_2)$ has cost
    $(50 \cdot 5 \cdot 100) + (50 \cdot 100 \cdot 10) = 25000 + 50000 = 75000$
  - $(A_0\ (A_1\ A_2))$ has cost
    $(5 \cdot 100 \cdot 10) + (50 \cdot 5 \cdot 10) = 5000 + 2500 = 7500$
- $(A_0(A_1A_2))$ is 10 times faster than $((A_0A_1)A_2)$!

Example: Matrix Sequence Multiplication

- Given a sequence $A_0, A_1, \ldots, A_{m-1}$ of m matrices to be multiplied, we wish to compute the product $A_0 \cdot A_1 \cdots A_{m-1}$
- For example, let us consider a sequence A_0, A_1, A_2 of 3 matrices of dimensions 50×5, 5×100 and 100×10, respectively
- We can parenthesize in two different ways:
  - $((A_0\ A_1)\ A_2)$
  - $(A_0\ (A_1\ A_2))$
- It turns out that different parenthesizations have different costs!
  - $((A_0\ A_1)\ A_2)$ has cost
    $(50 \cdot 5 \cdot 100) + (50 \cdot 100 \cdot 10) = 25000 + 50000 = 75000$
  - $(A_0\ (A_1\ A_2))$ has cost
    $(5 \cdot 100 \cdot 10) + (50 \cdot 5 \cdot 10) = 5000 + 2500 = 7500$
- $(A_0(A_1A_2))$ is 10 times faster than $((A_0A_1)A_2)$!
- In general, what is the best parenthesization?

Example: Matrix Sequence Multiplication

Matrix sequence multiplication

Let us consider a sequence $A_0, A_1, \ldots, A_{m-1}$,
where each of the A_i is a matrix of dimensions $n_i \times n_{i+1}$

What is the minimum cost of computing $A_0 \cdot A_1 \cdots A_{m-1}$?

Example: Matrix Sequence Multiplication

Matrix sequence multiplication

Let us consider a sequence $A_0, A_1, \ldots, A_{m-1}$,
where each of the A_i is a matrix of dimensions $n_i \times n_{i+1}$

What is the minimum cost of computing $A_0 \cdot A_1 \cdots A_{m-1}$?

- Let us make a top down program for solving this problem

Example: Matrix Sequence Multiplication

```
vector<int> n;              // A_i has dimensions n[i] x n[i+1]
vector<vector<int>> t;      // stores previous calls to cost(i,j)

int cost(int i, int j) { // min cost of computing A_i.. A_j
  int& r = t[i][j];
  if (r == UNDEF) {
    if (j == i) r = 0;      // computing A_i has cost 0
    else {
      r = INT_MAX;                      // INT_MAX acts as +∞
      for (int k = i+1; k <= j; ++k)//(A_i.. A_(k-1)(A_k.. A_j))
        r = min(r, cost(i,k-1) + cost(k,j) + n[i]*n[k]*n[j+1]);
  } }
  return r;
}
int main() {
  int m;
  while (cin >> m) {         // input is the number of matrices
    n = vector<int>(m+1); // and the sequence of dimensions
    for (int& x : n) cin >> x;
    t = vector<vector<int>>(m, vector<int>(m, UNDEF));
    cout << cost(0, m-1) << endl; } }
```

Example: Matrix Sequence Multiplication

- For a bottom up program: how to order the subproblems? Not trivial!

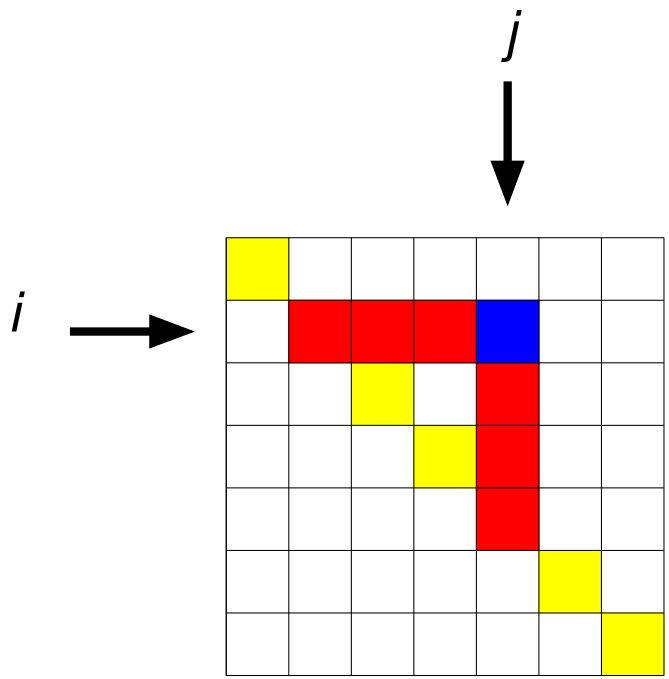
```
for (int k = i+1; k <= j; ++k) // Recall: i <= j
   r = min(r, cost(i,k-1) + cost(k,j) + n[i]*n[k]*n[j+1]);
```

Example: Matrix Sequence Multiplication

- For a bottom up program: how to order the subproblems? Not trivial!

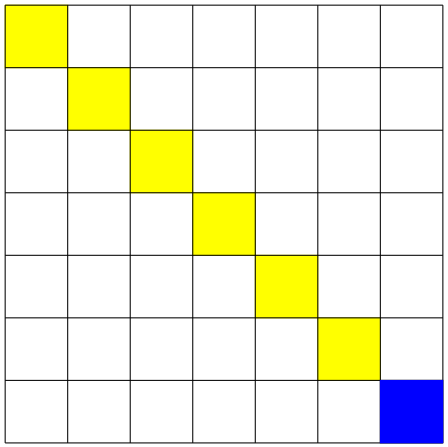
```
for (int k = i+1; k <= j; ++k) // Recall: i <= j
    r = min(r, cost(i,k-1) + cost(k,j) + n[i]*n[k]*n[j+1]);
```

- Before filling (i, j), we must have filled
  - $(i, i), \ldots, (i, j-1)$
  - $(i+1, j), \ldots, (j, j)$



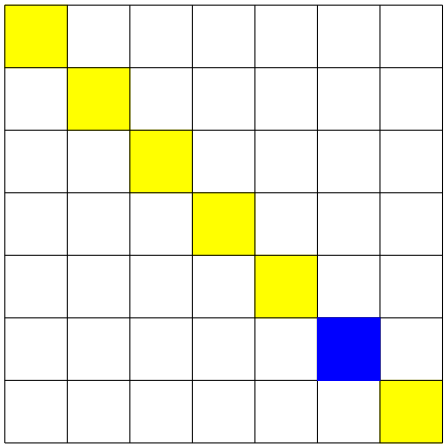
Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



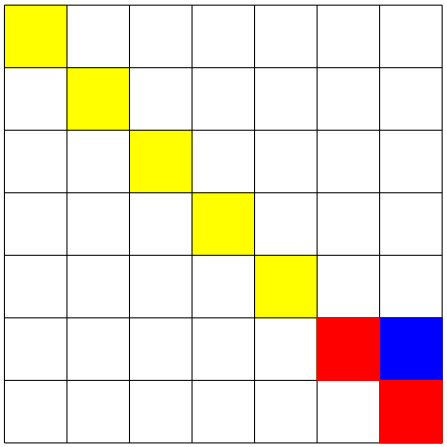
Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



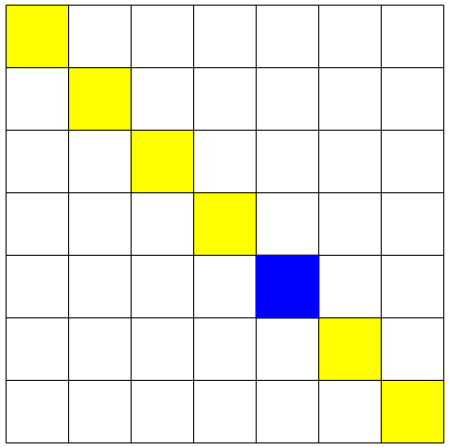
Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



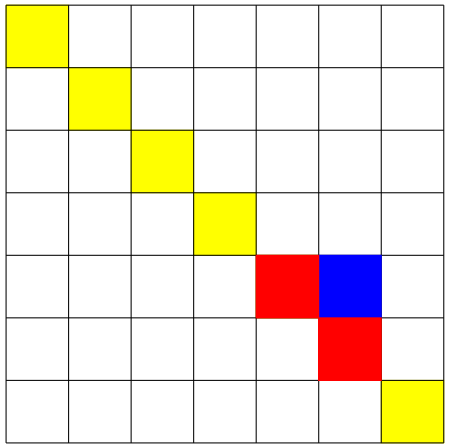
Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



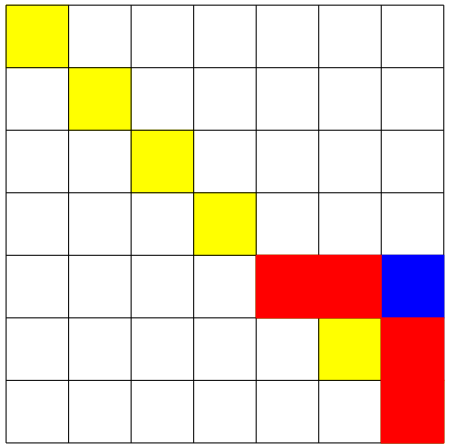
Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



Example: Matrix Sequence Multiplication

- A solution:
  by rows from bottom to top, and in each row by columns from left to right



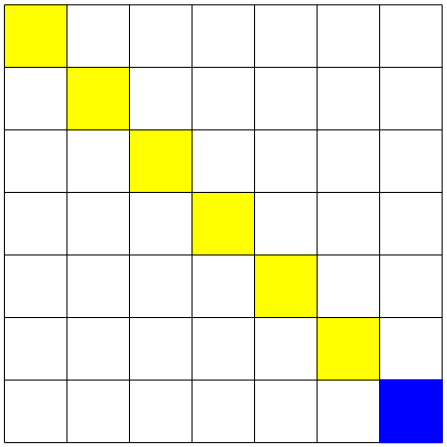
Example: Matrix Sequence Multiplication

```
int main() {
  int m;
  while (cin >> m) {
    vector<int> n(m+1);
    for (int& x : n) cin >> x;
    vector<vector<int>> t(m, vector<int>(m));
    for (int i = m-1; i >= 0; i--){// by rows bottom..top
      t[i][i] = 0; // Base case
      for (int j = i+1; j < m; j++) {// by cols left..right
        t[i][j] = INT_MAX;
        for (int k = i+1; k <= j; ++k)
          t[i][j] = min(t[i][j], t[i][k-1] + t[k][j] +
                                 n[i] * n[k] * n[j+1]);
    } }
    cout << t[0][m-1] << endl;
} }
```

Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

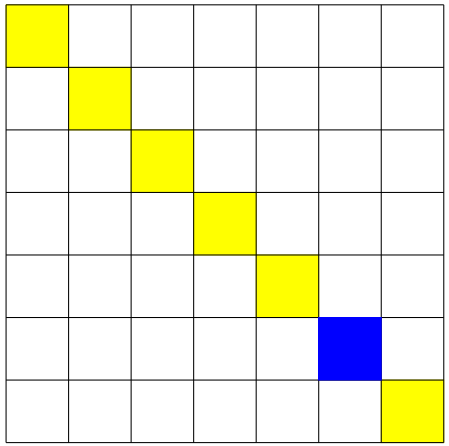
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

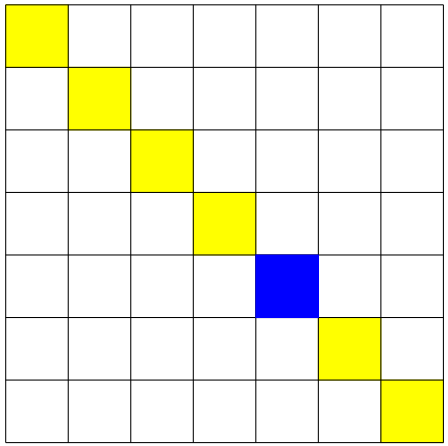
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

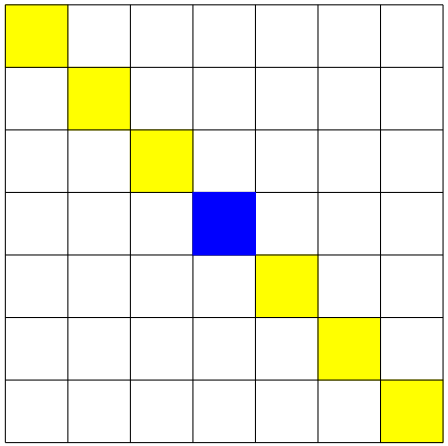
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

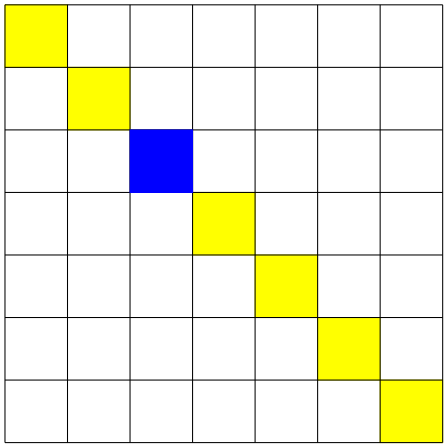
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

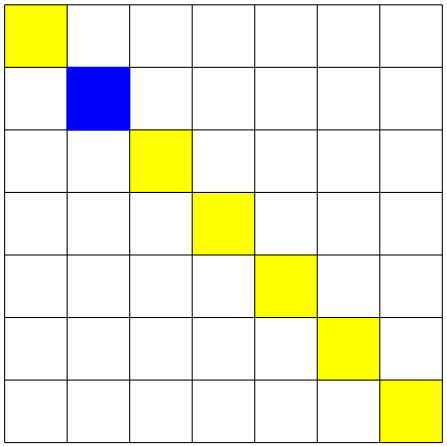
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

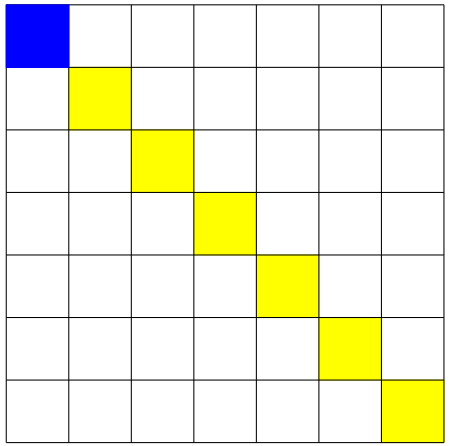
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

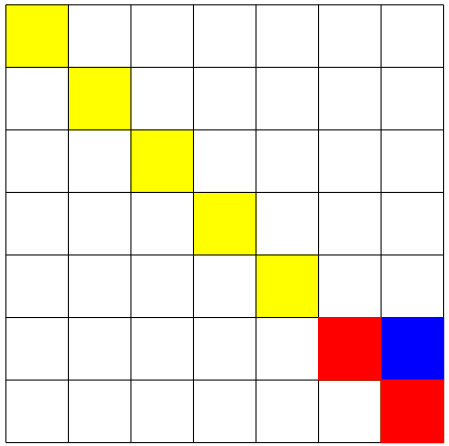
First length 1...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

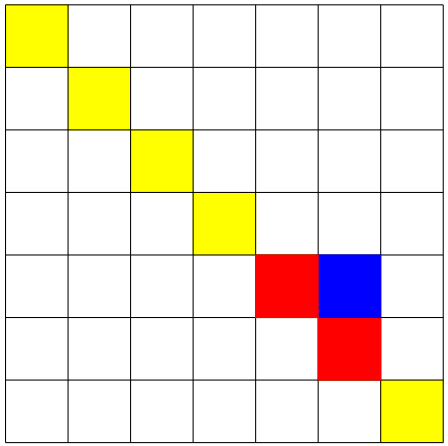
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

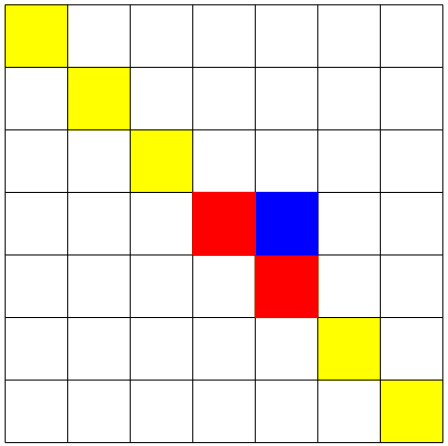
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

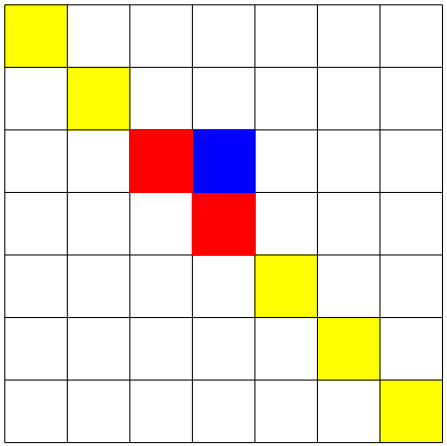
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

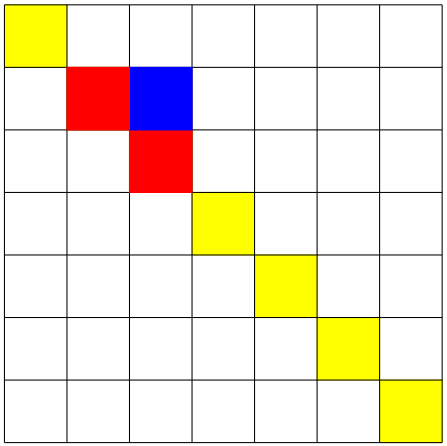
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

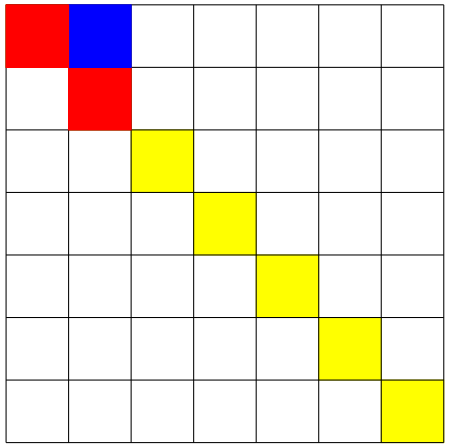
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

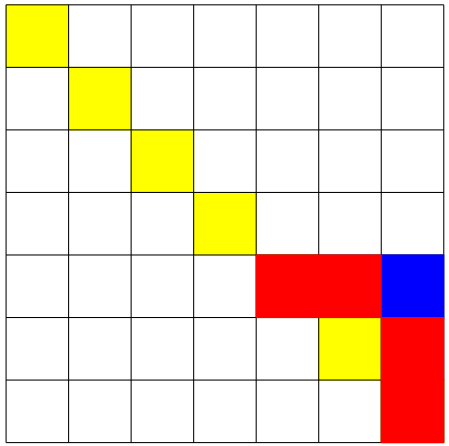
Then length 2...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

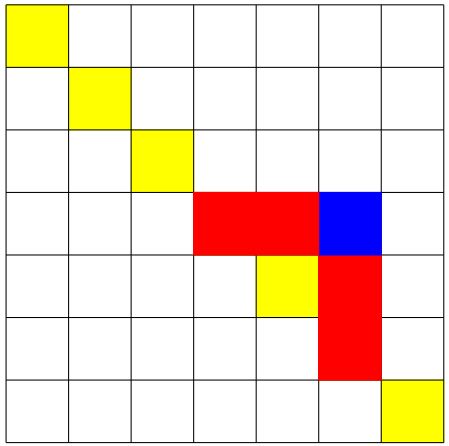
Then length 3...



Example: Matrix Sequence Multiplication

- Another solution: by increasing length of subsequence

Then length 3...



Example: Matrix Sequence Multiplication

```
int main() {
  int m;
  while (cin >> m) {
    vector<int> n(m+1);
    for (int& x : n) cin >> x;
    vector<vector<int>> t(m, vector<int>(m, 0));
    for (int len = 2; len <= m; ++len) {// length of [i..j]
      for (int i = m-len; i >= 0; i--) {
        int j = i + len - 1;
        t[i][j] = INT_MAX;
        for (int k = i+1; k <= j; ++k)
          t[i][j] = min(t[i][j], t[i][k-1] + t[k][j] +
                                 n[i] * n[k] * n[j+1]);
    } }
    cout << t[0][m-1] << endl;
} }
```

Simplicity

- Writing top down programs is usually easier
  - recursive code expresses in a natural way the problem recurrence
  - adding memoization is mechanic:
    1. lookup in the dictionary
    2. if solution has already been computed, return it
    3. otherwise compute recursively and store the solution before returning
- Bottom up needs to order the subproblems, which may not be direct

Debugging

- A bottom up program is usually easier to debug
  - the programmer has a better control on the execution flow
  - often enough to print the results at each iteration to track the bug

- Debugging a top down program is more difficult because in general recursive programs are harder to debug than iterative ones

  E.g. it is not easy to distinguish the outputs of each recursive call

Chapter 3. Dynamic Programming

Applications

1. Counting
   - Often problems of exhaustive generation that are solved with a backtracking algorithm have a counting version that can be solved with dynamic programming
   - Problem: generate all binary sequences of length n with k ones

```
int n, k;
vector<int> sol;

void f(int i, int u) {
  if (u > k  or i-u > n-k) return;
  if (i == n) write();
  else {
    sol[i] = 0; f(i+1, u  );
    sol[i] = 1; f(i+1, u+1);
} }
int main() {
  cin >> n >> k;
  sol = vector<int>(n);
  f(0, 0);}
```

Applications

1. Counting
   - Often problems of exhaustive generation that are solved with a backtracking algorithm have a counting version that can be solved with dynamic programming
   - Problem: count all binary sequences of length n with k ones

```
vector<vector<int>> b;

int binomial(int n, int k) {
  if (k < 0 or k > n) return 0;
  if (n == 0)         return 1;
  int& res = b[n][k];
  if (res != -1) return res;
  return res = binomial(n-1, k) + binomial(n-1, k-1); }

int main() {
  int n, k;
  cin >> n >> k;
  b = vector<vector<int>>(n+1, vector<int>(n+1, -1));
  cout << binomial(n, k) << endl; }
```

Applications

1. Counting
   - Sometimes the numbers can be so big that, to avoid problems of overflow, it is necessary to use `long long int`'s instead of `int`'s

Applications

1. Counting
   - Sometimes the numbers can be so big that, to avoid problems of overflow, it is necessary to use `long long int`'s instead of `int`'s
2. Optimization
   - Examples seen so far
     - knapsack
     - matrix sequence multiplication
   - To reconstruct the optimal solution, one can store in an additional table which choice has been made to solve each subproblem, if this choice cannot be remade efficiently (e.g., in constant time)

Computation of Optimal Solution

- Recall the bottom up program for knapsack:

```
int n, W;
vector<int> v, w;

int main() {
  cin >> n >> W;
  v = w = vector<int>(n);
  vector<vector<int>> m(n+1, vector<int>(W+1, 0));
  for (int& x : v) cin >> x;
  for (int& x : w) cin >> x;
 // m[i][j] = max value with first i objects and capacity j
  for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= W; ++j)
      if (j >= w[i-1])
        m[i][j] = max(v[i-1] + m[i-1][j-w[i-1]], m[i-1][j]);
      else
        m[i][j] = m[i-1][j];
  }
  cout << m[n][W] << endl;
}
```

Computation of Optimal Solution

- Bottom up program extended with optimal solution computation:

```
int main() {
  ...
  for (int i = 1; i <= n; ++i) {
    for (int j = 0; j <= W; ++j)
      if (j >= w[i-1])
        m[i][j] = max(v[i-1] + m[i-1][j-w[i-1]], m[i-1][j]);
      else
        m[i][j] = m[i-1][j];
  }
  cout << m[n][W] << endl;

 // m[i][j] = max value with first i objects and capacity j
  int j = W;
  for (int i = n; i > 0; --i)
    if (m[i][j] != m[i-1][j]) {
      cout << i << endl;
      j -= w[i-1];
}   }
```

Computation of Optimal Solution

- Recall the last bottom up solution for matrix sequence multiplication:

```
vector<vector<int>> t;

int main() {
  int m;
  while (cin >> m) {
    vector<int> n(m+1);
    for (int& x : n) cin >> x;
    t = vector<vector<int>>(m, vector<int>(m, 0));
    for (int len = 2; len <= m; ++len) {// length of [i..j]
      for (int i = m-len; i >= 0; i--) {
        int j = i + len - 1;
        t[i][j] = INT_MAX;
        for (int k = i+1; k <= j; ++k)
          t[i][j] = min(t[i][j], t[i][k-1] + t[k][j] +
                                 n[i] * n[k] * n[j+1]);

    } }

} }
```

Computation of Optimal Solution

- Program extended with optimal solution computation:

```
vector<vector<int>> s, t;

int main() {
  int m;
  while (cin >> m) {
    vector<int> n(m+1);
    for (int& x : n) cin >> x;
    s = t = vector<vector<int>>(m, vector<int>(m, 0));
    for (int len = 2; len <= m; ++len) {// length of [i..j]
      for (int i = m-len; i >= 0; i--) {
        int j = i + len - 1;
        t[i][j] = INT_MAX;
        for (int k = i+1; k <= j; ++k)
          t[i][j] = min(t[i][j], t[i][k-1] + t[k][j] +
                                 n[i] * n[k] * n[j+1]);
          s[i][j] = k;
    } }
    solution(0, m-1);
} }
```

Computation of Optimal Solution

- Program extended with optimal solution computation:

```
...
void solution(int i, int j) {
  if (i == j) cout << i;
  else {
    int k = s[i][j];
    cout << "("; solution(i, k-1); cout << ")";
    cout << "("; solution(k, j);   cout << ")";
} }
```

- Choices cannot be recovered in constant time, so we store them in `s` (or we could reconstruct the choice, paying the corresponding cost)

Chapter 3. Dynamic Programming

Advanced Examples

Sometimes the object to be computed is not inductive. What to do?

Advanced Examples

Sometimes the object to be computed is not inductive. What to do?

1. Generalize by adding new dimensions to get an inductive object

   For example, for the matrix sequence multiplication:

   - The original problem is:
     compute optimal parenthesization of $A_0 \cdot A_1 \cdots A_{m-1}$
   - The problem we solved is:
     given i, j, compute optimal parenthesization of $A_i \cdot A_{i+1} \cdots A_j$

   Important for efficiency to introduce the minimum number of additional parameters to describe the subproblems!

2. Express the object in terms of inductive objects (e.g., as the sum)

Advanced Examples

Counting binary sequences

Count the no. of binary sequences of length n that
do not contain two consecutive zeros nor three consecutive ones

- For instance, there are 7 sequences for $n = 5$:

  01010 01011 01101 10101 10110 11010 11011

- This quantity is not recursive

Advanced Examples

Counting binary sequences

Count the no. of binary sequences of length n that
do not contain two consecutive zeros nor three consecutive ones

- For instance, there are 7 sequences for $n = 5$:

  01010 01011 01101 10101 10110 11010 11011

- This quantity is not recursive
- Let us consider how words in the language may end:
  possible suffixes are 10, 01, 11
- Let us count the no. of words of length n in the language that end with
  - 10: denote it by $w_{10}(n)$
  - 01: denote it by $w_{01}(n)$
  - 11: denote it by $w_{11}(n)$

Advanced Examples

- When we remove last digit of a word of length n ending with 10 we get
  - a word of length $n-1$ ending with 01, or
  - a word of length $n-1$ ending with 11

  So $w_{10}(n) = w_{01}(n-1) + w_{11}(n-1)$

Advanced Examples

- When we remove last digit of a word of length n ending with 10 we get
  - a word of length $n-1$ ending with 01, or
  - a word of length $n-1$ ending with 11

  So $w_{10}(n) = w_{01}(n-1) + w_{11}(n-1)$

- When we remove last digit of a word of length n ending with 01 we get
  - a word of length $n-1$ ending with 10

  So $w_{01}(n) = w_{10}(n-1)$

Advanced Examples

- When we remove last digit of a word of length n ending with 10 we get
  - a word of length $n-1$ ending with 01, or
  - a word of length $n-1$ ending with 11

  So $w_{10}(n) = w_{01}(n-1) + w_{11}(n-1)$

- When we remove last digit of a word of length n ending with 01 we get
  - a word of length $n-1$ ending with 10

  So $w_{01}(n) = w_{10}(n-1)$

- When we remove last digit of a word of length n ending with 11 we get
  - a word of length $n-1$ ending with 01

  So $w_{11}(n) = w_{01}(n-1)$

Advanced Examples

- A bottom up solution:

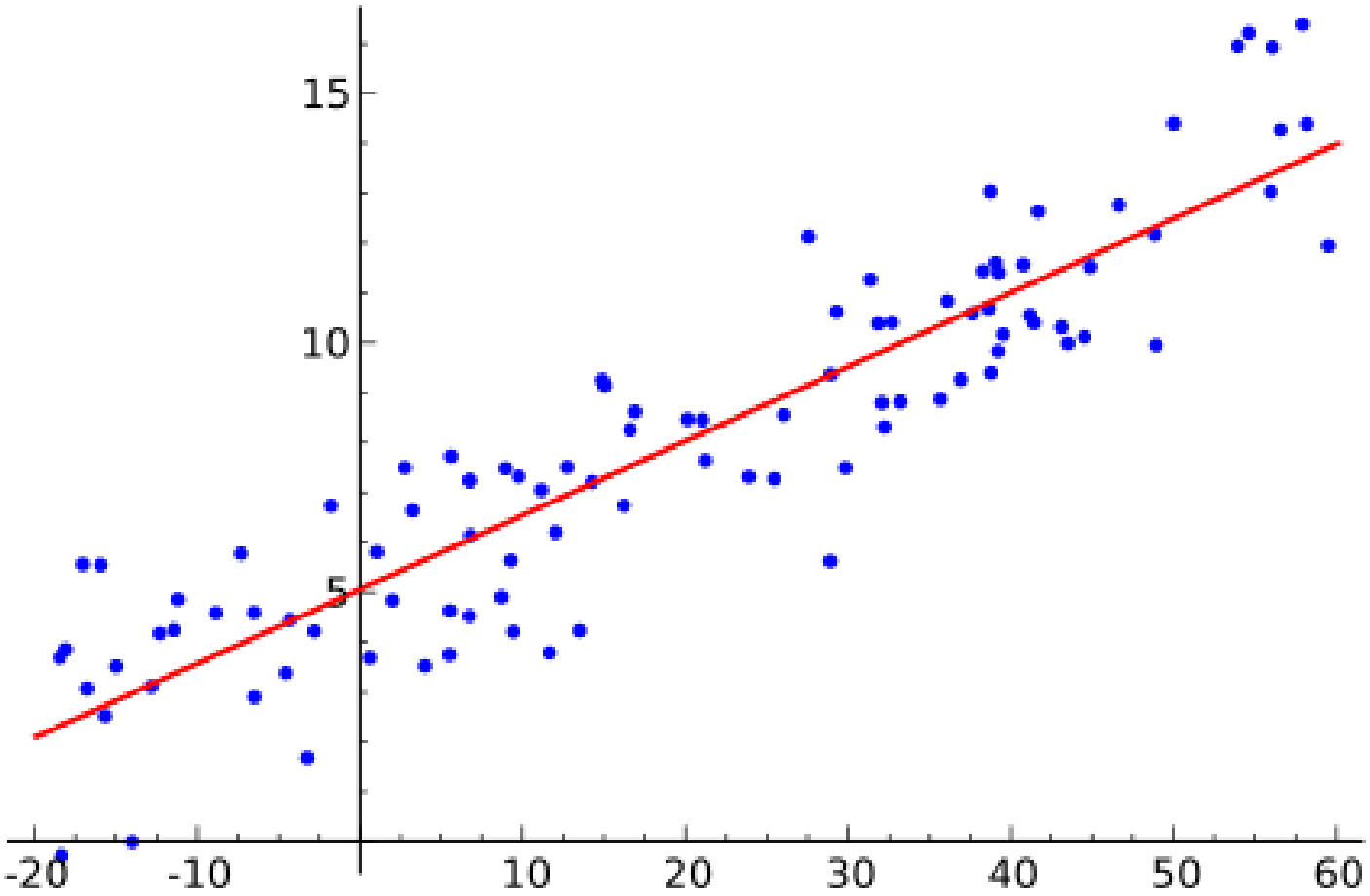
```
typedef long long  lint; //int's may not be enough
typedef vector<lint> VL;

// wX[i] counts words of length i >= 2 that end with X
VL w10, w11, w01;

int main() {
  const int N = 150;
  w10 = w11 = w01 = VL(N+1);
  w10[2] = w11[2] = w01[2] = 1; // words are 10, 11, 01 resp.
  for (int i = 3; i <= N; ++i) {
      w10[i] = w01[i-1] + w11[i-1];
      w11[i] = w01[i-1];
      w01[i] = w10[i-1]; }
  int n;
  while (cin >> n) {
    if (n <= 1) cout << n+1 << endl; // simple cases
    else cout << w10[n] + w11[n] + w01[n] << endl; } }
```

Advanced Examples

- Let us see an example of dynamic programming relevant to data science
- Often when looking at data plotted on the plane,
  one tries to pass a line of best fit through the data (linear regression)



Advanced Examples

- Suppose our data consists of a set P of n points on the plane, denoted (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) where $x_1 < x_2 < ... < x_n$
- Given a line L defined by the equation $y = ax + b$, the error of L wrt. P is the sum of its squared distances to points in P:

$$Error(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

- Which is the line with minimum error?

Advanced Examples

- Suppose our data consists of a set P of n points on the plane, denoted (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) where $x_1 < x_2 < ... < x_n$
- Given a line L defined by the equation $y = ax + b$, the error of L wrt. P is the sum of its squared distances to points in P:

$$Error(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

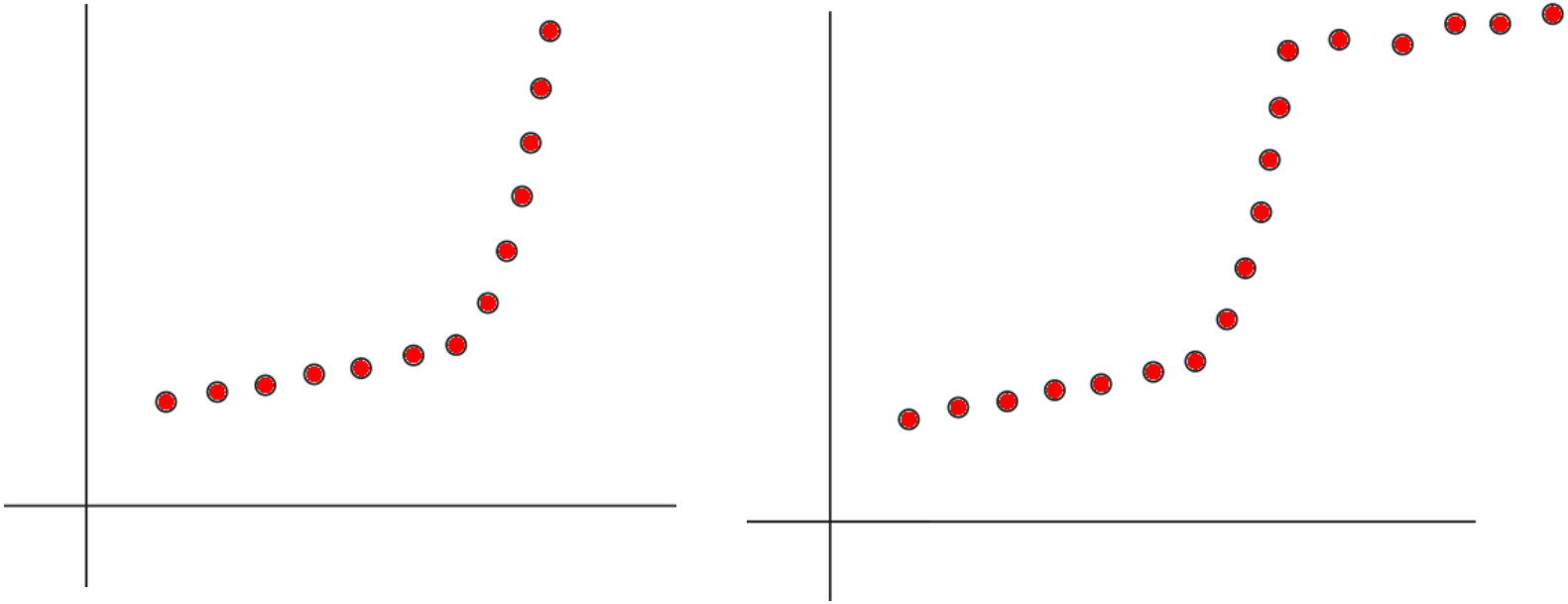
- Which is the line with minimum error?
- The line of minimum error (regression line) is $y = ax + b$, where

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \qquad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

- If $n > 1$ and $x_1 < x_2 < ... < x_n$ then $n \sum_i x_i^2 - (\sum_i x_i)^2 \neq 0$

Advanced Examples

- What if the points lie roughly on a sequence of different lines?



- Traditional linear regression is not suitable for these situations
- Instead we consider the problem of fitting the points well, using as few lines as possible

Advanced Examples

- Let us formally formulate the problem
- We are given a set P of $n > 1$ points $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ where $x_1 < x_2 < \ldots < x_n$
- The set of points P is partitioned into segments: a subset of the form $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i < j$
- For each segment S we compute the line minimizing the error with respect to the points in S, according to the formulas of linear regression
- The penalty of a partition is defined to be the sum of:
  - The number of segments into which we partition P, times a fixed multiplier $C > 0$ (by tuning C we can penalize the use of more lines)
  - For each segment, the error value of the optimal line of the segment

Segmented linear regression problem

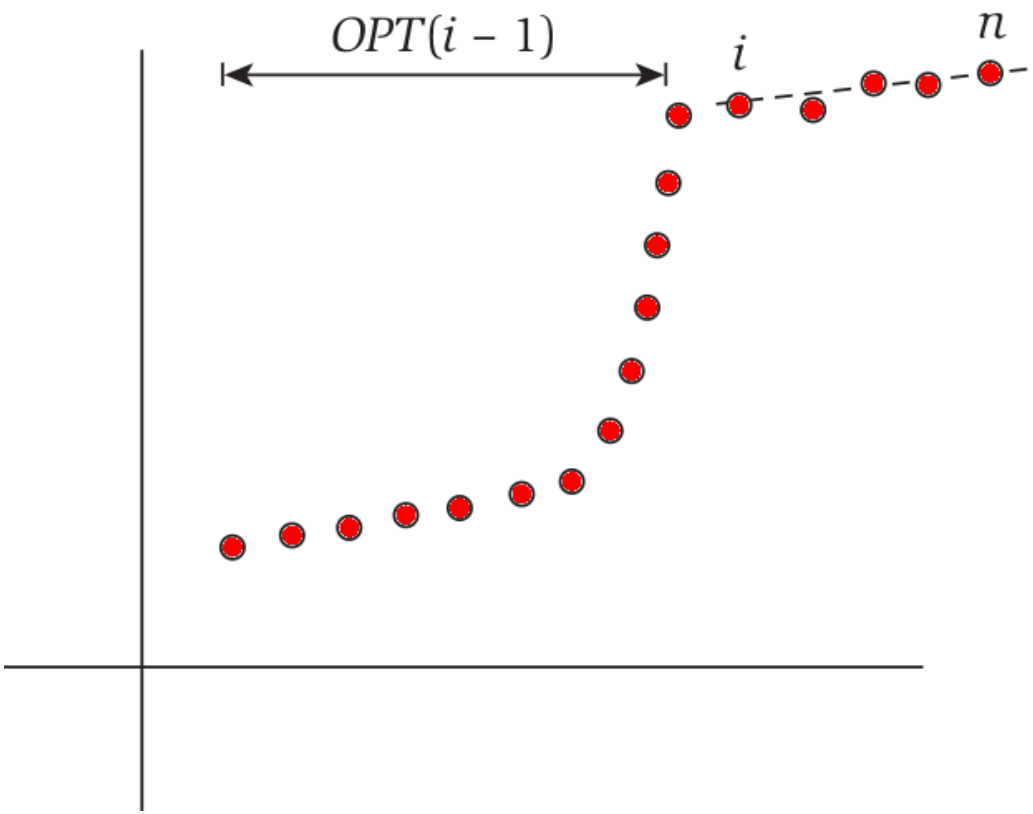
Find a partition of P into segments of minimum penalty

Advanced Examples

- The last point p_n belongs to a single segment in the optimal partition, and that segment begins at some earlier point p_i
- If we knew the last segment $p_i, \ldots, p_n$, then we could remove those points and recursively solve the problem on the points $p_1, \ldots, p_{i-1}$

Advanced Examples

- The last point p_n belongs to a single segment in the optimal partition, and that segment begins at some earlier point p_i
- If we knew the last segment $p_i, \ldots, p_n$, then we could remove those points and recursively solve the problem on the points $p_1, \ldots, p_{i-1}$
- Let $OPT(i)$ denote the minimum penalty for the points $p_1, \ldots, p_i$



Advanced Examples

- Let $OPT(j)$ denote the minimum penalty for the points $p_1, ..., p_j$
- Let $e_{i,j}$ be the minimum error of a regression line for $p_i, p_{i+1}, \ldots, p_j$
- By convention $OPT(0) = 0$, $e_{i,i} = +\infty$
- For the subproblem on the points $p_1, \ldots, p_j$:

$$OPT(j) = min_{1 \leq i \leq j}(e_{i,j} + C + OPT(i-1))$$

Advanced Examples

- A bottom up program with solution computation:

```
const double  C = 0.05;     // Cost of adding a new line
const double oo = DBL_MAX; // Infinity. Needs #include <cfloat>

int main() {

  // Read data
  int n;
  cin >> n;
  vector<pair<double, double>> P(n+1); // Position 0 not used
  for (int k = 1; k <= n; ++k)
    cin >> P[k].first >> P[k].second;

  // Precompute the error of fitting the segment pi, ..., pj
  vector<vector<double>> E(n+1, vector<double>(n+1, +oo));
  for (int i = 1; i <= n; ++i)
    for (int j = i+1; j <= n; ++j)
      E[i][j] = err(P, i, j);

  ...
```

Advanced Examples

```
  ...
  vector<double> opt(n+1,oo); // opt[j] = min penalty of p1..pj
  vector<int> idx(n+1);       // Last segment starts at idx[j]
  opt[0] = 0;
  for (int j = 1; j <= n; ++j)
    for (int i = 1; i <= j; ++i) {
      double cost = E[i][j] + C + opt[i-1];
      if (cost < opt[j]) {
        opt[j] = cost;
        idx[j] = i;   // Record the choice
    } }

  // Reconstruct the solution
  int j = n;
  while (j != 0) {
    int i = idx[j];
    cout << "Segment from #" << i << " to #" << j << ": ";
    double a, b, e;
    line(P, i, j, a, b, e);
    cout << "a = " << a << ", b = " << b << endl;
    j = i - 1;
} }
```

Advanced Examples

```
// Finds regression line y = ax + b for P[i]..P[j] and error e
void line(const vector<pair<double,double>>& P, int i, int j,
          double& a, double& b, double& e) {
  double sum_x = 0;          double sum_xx = 0;
  double sum_y = 0;          double sum_xy = 0;

  for (int k = i; k <= j; ++k) {
    double x = P[k].first;   double y = P[k].second;
    sum_x += x;              sum_xx += x*x;
    sum_y += y;              sum_xy += x*y;
  }

  int n = j - i + 1;
  a = (n*sum_xy - sum_x*sum_y) / (n*sum_xx - sum_x*sum_x);
  b = (sum_y - a * sum_x) / n;
  e = 0;

  for (int k = i; k <= j; ++k) {
    double x = P[k].first;   double y = P[k].second;
    e += (y - a*x - b) * (y - a*x - b);
  } }
```

Advanced Examples

```
void line(const vector<pair<double,double>>& P, int i, int j,
          double& a, double& b, double& e) {
  ...
}

double err(const vector<pair<double,double>>& P, int i, int j){
  double a, b, e;
  line(P, i, j, a, b, e);
  return e; }
```

Chapter 3. Dynamic Programming

Requirements for Applying Dynamic Programming

- The problem must have recursive structure:
  a solution to the problem "contains" solutions to smaller subproblems

  Hence the problem can be solved by means of a recurrence

Requirements for Applying Dynamic Programming

- The problem must have recursive structure:
  a solution to the problem "contains" solutions to smaller subproblems

  Hence the problem can be solved by means of a recurrence

- Subproblems must overlap:
  the same subproblem must be a subproblem of different problems