Chapter 8: Dynamic Programming

It was invented by Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. R. Bellman mentioned a general principle that relates to optimization problems, called the *principle of optimality*:

"An optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances".

In computer science, this strategy is used to solve problems with overlapping subproblems and there is a recurrence relation between the solution of the larger problem and the solutions of its smaller subproblems.

Example: Computing the n^{th} Fibonacci number.

Example: Computing the binomial coefficient $\binom{n}{k}$.

The binomial coefficient $\binom{n}{k}$ is given by the formula:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

where $0 \le k \le n$ or by the following recursive formula:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \lor k = n \end{cases}$$

Algorithm (Recursive version)

```
Binomial(n, k) {
  if (k == 0 || k == n)
    return 1;
  return Binomial(n - 1, k - 1) + Binomial(n - 1, k);
}
```

The generalized formula for computing the binomial coefficient $\binom{n}{k}$ is as follows:

$$\binom{i}{j} \Leftrightarrow C[i,j] = \begin{cases} C[i-1,j-1] + C[i-1,j] & 0 < j < i \\ 1 & j = 0 \lor j = i \end{cases}$$

	0	1	2	…	$j-1$	j	…	$k-1$	k
0	1								
1	1	1							
2	1	2	1						
⋮									
$i-1$					$C[i-1, j-1]$	$C[i-1, j]$			
i						$C[i, j]$			
⋮									
k	1								1
⋮									
$n-1$	1							$C[n-1, k-1]$	$C[n-1, k]$
n	1								$C[n, k]$

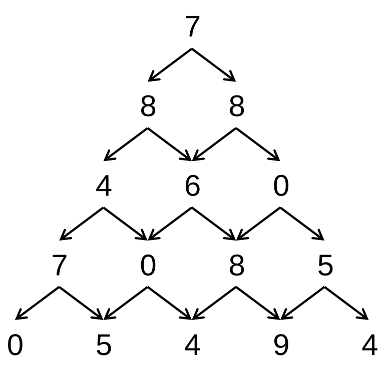
Algorithm (Dynamic programming version)

```
Binomial(n, k) {
  C[0 .. n, 0] = C[0 .. k, 0 .. k] = 1;
  for (i = 1; i ≤ n; i++)
    for (j = 1; j < (i ≤ k ? i : k + 1); j++)
      C[i, j] = C[i - 1, j - 1] + C[i - 1, j];
  return C[n, k];
}
```

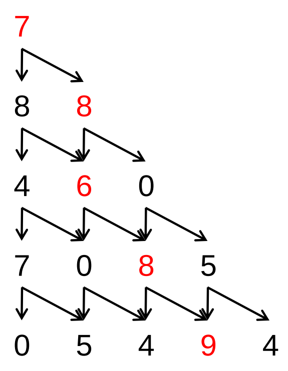
Analysis of the algorithm: $A(n, k) \in \Theta(nk)$

Example: Maximum sum of a path in a right number triangle.

Given a right triangle of numbers, find the largest of the sum of numbers that appear on a path starting from the top towards the base. The next number on the path is located directly below or below-and-one-place-to-the-right. Show the path discovered.



(a)



(b)

	0	1	2	3	4	5
0	0	0				
1	0	7	0			
2	0	15	15	0		
3	0	19	21	15	0	
4	0	26	21	29	20	0
5	0	26	31	33	38	24

(c)

Let's denote $S(i,j)$ the largest of the sum of numbers that appear on a path starting from the top to the location with row index i and column index j.

$$S(i,j) = \begin{cases} \max\big(S(i-1,j-1), S(i-1,j)\big) + a_{i,j} & i \neq j \wedge i,j \neq 1 \\ a_{i,j} & i = j = 1 \\ S(i-1,j-1) + a_{i,j} & i = j \wedge i,j \neq 1 \\ S(i-1,j) + a_{i,j} & i \neq j \wedge j = 1 \end{cases}$$

A two-dimensional array S of size $(n+1) \times (n+1)$ is used to compute all potential paths starting from the top. Initially,

$$S[0..n, 0] = 0, S[j, j+1] = 0$$

where $0 \leq j \leq n-1$.

Algorithm

```
… Initialize table S with the input data …

S[0 .. n, 0] = S[0 .. n – 1, 1 .. n] = 0;
for (i = 1; i ≤ n; i++)
  for (j = 1; j ≤ i; j++)
    S[i][j] = max(S[i - 1][j - 1], S[i-1][j]) + a[i][j];
return "the maximum value on the nth row"
```

Analysis of the algorithm: $T(n) \in \Theta(n^2)$

The change-making problem

Given k denominations: $d_1, d_2, \dots, d_k$. where $d_1 = 1$. Find the minimum number of coins (of certain denominations) that add up to a given amount of money n. The smallest denomination is always a one-cent coin. Show the exact coins that build up the amount of money.

Let's denote $C(n)$ the minimum number of coins (of certain denominations) that add up to n. Then,

$$C(n) = \min_{1 \le i \le k} C(n - d_i) + 1$$

where $n \ge d_i, C(0) = 0$

Algorithm

```
int  changeCoinsDP(int d[], int k, int money) {
  int C[0 .. money] = 0;

  for (cents = 1; cents ≤ money; cents++) {
    int minCoins = cents;
    for (i = 1; i ≤ k; i++) {
      if (d[i] > cents)
        continue;
      if (C[cents - d[i]] + 1 < minCoins)
        minCoins = C[cents - d[i]] + 1;
    }
    C[cents] = minCoins;
  }
  return C[money];
}
```

Analysis of the algorithm: $\Theta(kn)$

Longest Monotonically Increasing Subsequence (LMIS) problem

Find the length of the longest subsequence of a given sequence of positive integers such that all elements of the subsequence are in monotonically increasing order. Print the longest subsequence.

Note: There may be more than one *LMIS* combination, it is only necessary for you to return the length.

Formally we look for the longest sequence of indexes $i_1, i_2, \ldots, i_k$ such that:

$$1 \le i_1 < i_2 < \cdots < i_k \le n \wedge a_{i_1} < a_{i_2} < \cdots < a_{i_k}$$

Let's denote $L(i)$ the length of the longest increasing subsequence whose last element is a_i. Of course, a_i is greater than all other elements in this subsequence.

$$L(i) = \max_{1 \le j < i:\, a_j < a_i} \big(L(j)\big) + 1$$

where $L(1) = 1$.

Algorithm (recursive version)

```
lis(a[1 .. n], i) {
   if (i == 1)
      return 1;

   tmpMax = 1;
   for (j = 1; j < i; j++)
      if (a[j] < a[i]) {
         res = lis(a, j);
         if (res + 1 > tmpMax)
            tmpMax = res + 1;
      }

   if (max < tmpMax) // global variable
      max = tmpMax;
   return tmpMax;
}
for (i = 1; i ≤ n; i++)
   lis(a, i);
print(max);
```

Analysis of the algorithm:

The complexity of the algorithm depends on the distribution of an input data. For simplicity, let's assume that the given sequence contains no duplicate values.

Let's denote $T(i)$ the running time of function `lis(a, i)`, $\forall i \in [1, n]$

- The worst case: $\Theta(2^n)$
- The best case: $\Theta(n^2)$
- The average case: …

Now, we design a dynamic programming algorithm for this problem. Let's denote $L[i]$ the length of the longest increasing subsequence whose last element is a_i. It's not difficult to obtain the following formula:

$$L[i] = \max_{1 \le j < i:\, a_j < a_i} \{L[j]\} + 1$$

Algorithm (Dynamic programming version)

```
lis_DP(a[1..n]) {
  L[1 .. n] = 1;

  for (i = 2; i ≤ n; i++)
    for (j = 1; j < i; j++)
      if ((a[j] < a[i]) && (L[j] + 1 > L[i]))
        L[i] = L[j] + 1;

  return "The biggest element in L";
}
cout << lis_DP(a);
```

Analysis of the algorithm: $\Theta(n^2)$

Note: A *subsequence* of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

Longest Common Subsequence (LCS) Problem

Given two strings $S = s_1 s_2 \dots s_m$ and $T = t_1 t_2 \dots t_n$, find the length of their longest common subsequence and print it.

Note: A *common subsequence* of two strings is a subsequence that is common to both strings.

Formally we look for the longest sequence of indexes $i_1, i_2, \dots, i_k$ and $j_1, j_2, \dots, j_k$ such that:

$$1 \le i_1 < i_2 < \cdots < i_k \le m \land j_1 < j_2 < \cdots < j_k \land s_{i_h} = t_{j_h}, \forall h \in [1, k]$$

Example: Given $S = XYXZPQ, T = YXQYXP$. The longest common subsequence is $XYXP$ of the length 4.

Let's denote $L(i,j)$ the length of the longest common subsequence which exists in two substrings $s_1 s_2 \dots s_i$ and $t_1 t_2 \dots t_j$.

- If $s_i = t_j$: $L(i,j) = 1 + L(i-1, j-1)$
- If $s_i \ne t_j$: $L(i,j) = \max\{L(i-1,j), L(i,j-1)\}$

where $L(i,0) = L(0,j) = 0$.

Algorithm (recursive version)

```
int  LCS(char S[], int i, char T[], int j) {
   if ((i == 0) || (j == 0))
      return 0;

   if (S[i] == T[j])
      return 1 + LCS(S, i - 1, T, j - 1);
   else
      return max(LCS(S, i - 1, T, j), LCS(S, i, T, j - 1));
}
cout << LCS(S, m, T, n);
```

Analysis of the algorithm: $\Theta(2^n)$

Now, let's design a dynamic programming algorithm for this problem. A two-dimensional array L of the size $m \times n$ is used to hold the results of subproblems.

The cell $L[i, j]$ contains the length of the longest common subsequence which exists in two substrings $s_1 s_2 \ldots s_i$ and $t_1 t_2 \ldots t_j$:

- If $s_i = t_j$: $L[i, j] = L[i - 1, j - 1] + 1$
- If $s_i \neq t_j$: $L[i, j] = \max\{L[i - 1, j], L[i, j - 1]\}$

where $L[0, j] = L[i, 0] = 0$.

Obviously, $L[m, n]$ contains the final result.

Algorithm

```
LCS_Dyn(char S[], int m, char T[], int n) {
  L[1 .. m][0] = L[0][1 .. n] = 0

  for (i = 1; i ≤ m; i++)
    for (j = 1; j ≤ n; j++)
      if (S[i] == T[j])
        L[i][j] = 1 + L[i - 1][j - 1];
      else
        L[i][j] = max(L[i - 1][j], L[i][j - 1]);

  return L[m][n];
}
```

Analysis of the algorithm: $\Theta(mn)$.

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the all-pairs shortest-paths problem asks to find the distances - i.e., the lengths of the shortest paths - from each vertex to all other vertices.

Assuming that the given graph has n vertices. A two-dimensional array D of the size $n \times n$, called the *distance matrix*, is used to record the lengths of shortest paths: the element d_{ij} indicates the length of the shortest path from the vertex i to the vertex j ($1 \leq i \neq j \leq n$).

Floyd's algorithm computes the distance matrix D of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, D^{(1)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)} \equiv D$$

The element $d_{ij}^{(k)} \in D^{(k)}$ ($k = 0,1, \ldots, n; i, j = 1,2, \ldots, n$) is equal to the length of the shortest path among all paths from the the vertex i to the vertex j with each intermediate vertex, if any, numbered not higher than k. It's reasonable if we say that

$$d_{ij}^{(k)} = \min_{1 \leq k \leq n} \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}$$

Algorithm

```
Floyd(W[1 .. n, 1 .. n]) {
  D = W;

  for (k = 1; k ≤ n; k++)
    for (i = 1; i ≤ n; i++)
      for (j = 1; j ≤ n; j++)
        D[i, j] = min{D[i, j], D[i, k] + D[k, j]};

  return D;
}
```

Analysis of the algorithm: $\Theta(n^3)$

Matrix Chain Multiplication

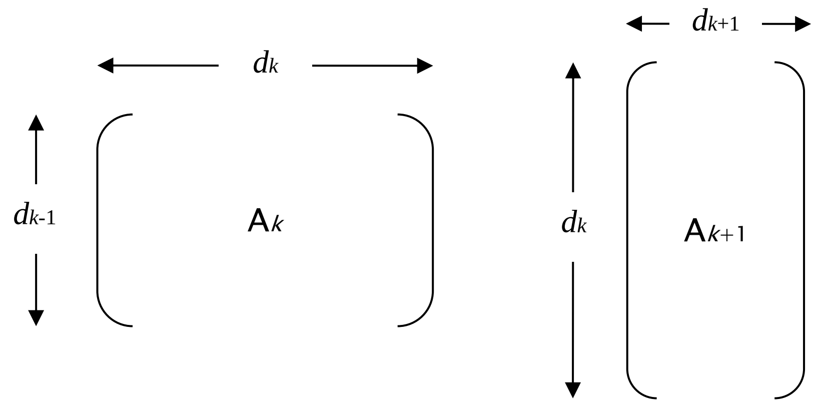
Find the most efficient way to multiply a sequence of n matrices $A_1 \times A_2 \times \ldots \times A_n$. Assuming that the sizes (in order) of these matrices are $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$, respectively.

Note: The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Example: Multiplying 4 matrices $A \times B \times C \times D$ of the sizes (in order) $50 \times 20, 20 \times 1, 1 \times 10, 10 \times 100$. Here are three among five different orders to perform the multiplications:

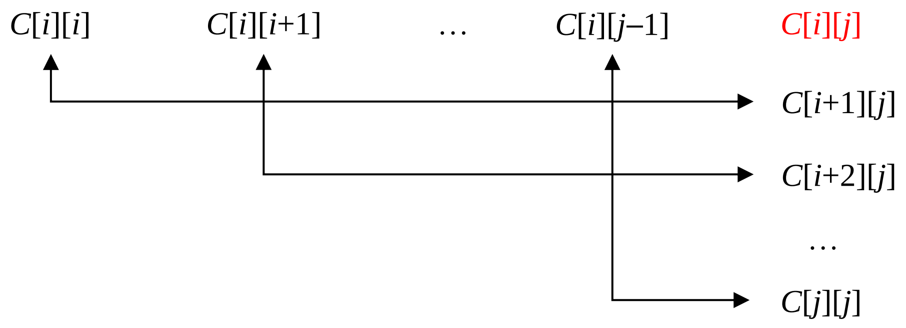
Multiplication order	*The number of operations*
$A \times ((B \times C) \times D)$	$20 \times 1 \times 10 + 20 \times 10 \times 100 + 50 \times 20 \times 100 = 120200$
$(A \times (B \times C)) \times D$	$20 \times 1 \times 10 + 50 \times 20 \times 10 + 50 \times 10 \times 100 = 60200$
$(A \times B) \times (C \times D)$	$50 \times 20 \times 1 + 1 \times 10 \times 100 + 50 \times 1 \times 100 = 7000$

Convention: If $A_k \times A_{k+1}$ where $1 \leq k < n$ then the sizes of A_k and A_{k+1} are $d_{k-1} \times d_k$ and $d_k \times d_{k+1}$, respectively. In addition, the size of the matrix which is the product of A_k and A_{k+1} is $d_{k-1} \times d_{k+1}$.



Let's consider the sequence of multiplications: $A_i \times A_{i+1} \times \ldots \times A_j$, where $1 \leq i \leq j \leq n$. Let's denote $C(i,j)$ the lowest cost of this sequence. We have the following recurrence relation:

$$C(i,j) = \begin{cases} \min\limits_{i \leq k < j} \{C(i,k) + C(k+1,j) + d_{i-1} \times d_k \times d_j\} & i < j \\ 0 & i = j \end{cases}$$



Algorithm

```
ChainMatrixMult(d[0 .. n], P[1 .. n][1 .. n]) {
  C[1 .. n, 1 .. n] = 0;
  for (diag = 1; diag < n; diag++)
    for (i = 1; i ≤ n - diag; i++) {
      j = i + diag;
      C[i, j] = $\min_{i \le k < j} \{C[i,k] + C[k+1,j] + d[i-1] \times d[k] \times d[j]\}$;
      P[i, j] = the value of k that minimizes C[i, j];
    }
  return C[1, n];
}
```

Analysis of the algorithm: $\Theta(n^3)$

How to print the most efficient way to multiply the sequence of matrices?

Algorithm

```
order(i, j) {
  if (i == j)
    cout << "A" <<  i;
  else {
    k = P[i][j];
    cout << "(";
    order(i, k);
    order(k + 1, j);
    cout << ")";
  }
}
order(1, n);
```

Optimal Binary Search Trees

An optional binary search tree is a special binary search tree for which the average number of comparisons in a search is the smallest possible value. Hence, in order to construct an optional binary search tree, probabilities of searching for elements of a set must be known.

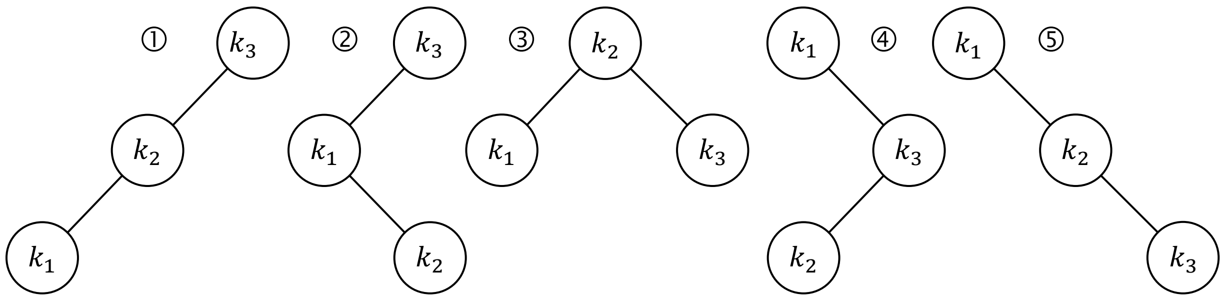
Note: For simplicity, we limit our discussion to minimizing the average number of comparisons in a successful search.

Conventions:

- Let $k_1, k_2, \dots, k_n$ be keys of a binary search tree. Assuming that:
$$k_1 < k_2 < \cdots < k_n$$
- $\forall i \in [1, n]$: Let p_i be the probability of searching for k_i.
- $\forall i \in [1, n]$: Let c_i be the number of times the comparison is executed to find out k_i:
$$c_i = level(k_i) + 1$$
- The average number of comparisons in a successful search in the tree is
$$Cost = \sum_{i=1}^{n} (c_i \times p_i)$$

The tree must be designed and constructed in such a way that the value of $Cost$ is minimized.

Example: Consider 5 possibilities of constructing a binary search tree that contains three keys: $k_1 < k_2 < k_3$. The probabilities of searching for them are $p_1 = 0.7, p_2 = 0.2, p_3 = 0.1$.



1. 3 (0.7) + 2 (0.2) + 1 (0.1) = 2.6
2. 2 (0.7) + 3 (0.2) + 1 (0.1) = 2.1
3. 2 (0.7) + 1 (0.2) + 2 (0.1) = 1.8
4. 1 (0.7) + 3 (0.2) + 2 (0.1) = 1.5
5. 1 (0.7) + 2 (0.2) + 3 (0.1) = **1.4**

As you will see, the total number of binary search trees with n keys is equal to the n^{th} Catalan number,

$$c(n) = \begin{cases} \frac{1}{n+1}\binom{2n}{n} & n > 0 \\ 1 & n = 0 \end{cases}$$

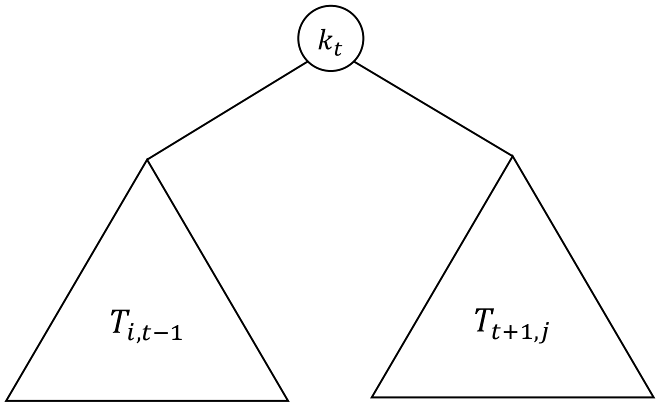
which grows to infinity as fast as $\frac{4^n}{n^{1.5}}$. So, an exhaustive-search approach is unrealistic.

Let denote $C(i,j)$ the smallest average number of comparisons made in a successful search in an optimal binary search tree $T_{i,j}$ made up of keys $k_i, \dots, k_j$, where i, j are some integer indices, $1 \le i \le j \le n$.

- If $i = j$: The tree $T_{i,j}$ contains only one node.

$$C(i,i) = c_i \times p_i = 1 \times p_i = p_i$$

- If $i > j$: The tree $T_{i,j}$ is considered as empty one and $C(i,j) = 0$.
- If $i < j$: We will consider all possible ways to construct $T_{i,j}$. Let $T_{i,j}^t$ be a $T_{i,j}$ whose the root contains key k_t, where $i \le t \le j$. The left subtree $T_{i,t-1}$ contains keys $k_i, \dots, k_{t-1}$ optimally arranged, and the right subtree $T_{t+1,j}$ contains keys $k_{t+1}, \dots, k_j$ also optimally arranged.



Since $T_{i,t-1}$ and $T_{t+1,j}$ are two optimal binary search trees so $C(i,t-1)$ and $C(t+1,j)$ are the smallest average numbers of comparisons made in a successful search in $T_{i,t-1}$ and $T_{t+1,j}$, respectively.

$$C(i,t-1) = \sum_{s=i}^{t-1} c_s \times p_s$$

$$C(t+1,j) = \sum_{s=t+1}^{j} c_s \times p_s$$

Obviously, the smallest average number of comparisons made in a successful search in $T_{i,j}^{t}$ is shown as follows:

$$\sum_{s=i}^{t-1}\big((c_s+1)\times p_s\big)+\sum_{s=t+1}^{j}\big((c_s+1)\times p_s\big)+p_t$$

$$=C(i,t-1)+\sum_{s=i}^{t-1}p_s+C(t+1,j)+\sum_{s=t+1}^{j}p_s+p_t$$

$$=C(i,t-1)+C(t+1,j)+\sum_{s=i}^{j}p_s$$

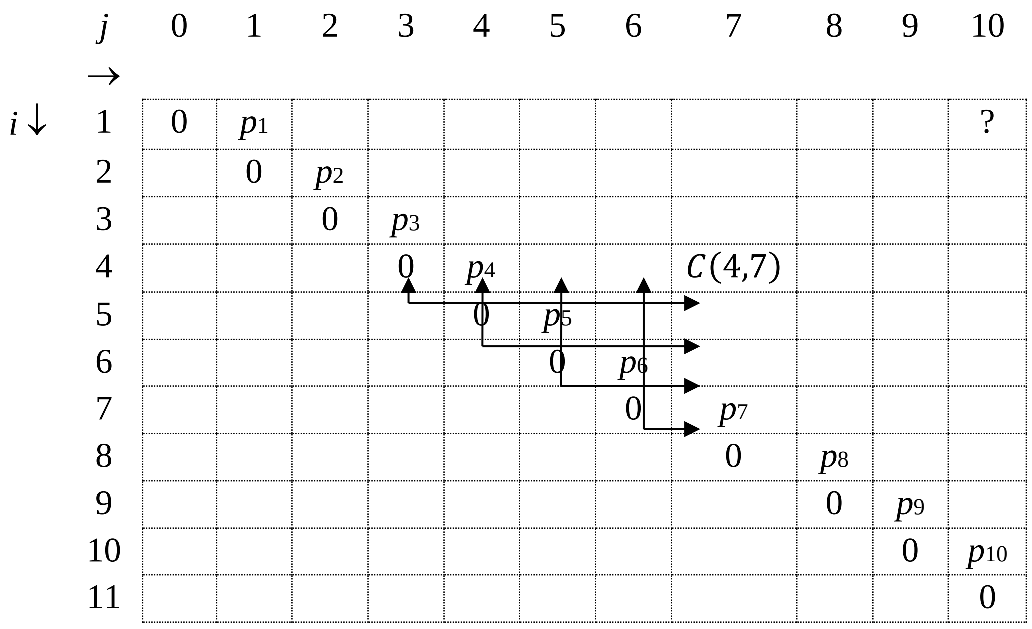
Thus, we have the recurrence:

$$C(i,j)=\min_{i\le t\le j}\left\{C(i,t-1)+C(t+1,j)+\sum_{s=i}^{j}p_s\right\}$$

$$=\min_{i\le t\le j}\{C(i,t-1)+C(t+1,j)\}+\sum_{s=i}^{j}p_s$$

where $1\le i\le j\le n$.

Example: Consider table C of the size $(1..n+1)\times(0..n)$ where $n=10$.



	$j \rightarrow$	0	1	2	3	4	5	6	7	8	9	10
$i\downarrow$	1	0	p_1									?
	2		0	p_2								
	3			0	p_3							
	4				0	p_4			$C(4,7)$			
	5					0	p_5					
	6						0	p_6				
	7							0	p_7			
	8								0	p_8		
	9									0	p_9	
	10										0	p_{10}
	11											0

Algorithm

```
OptimalBST(p[1..n]) {
  int C[1 .. n + 1, 0 .. n], R[1 .. n + 1, 0 .. n];

  for (i = 0; i ≤ n; i++)   C[i + 1, i] = R[i + 1, i] = 0;
  for (i = 1; i ≤ n; i++) {
    C[i, i] = p[i];
    R[i, i] = i;
  }

  for (diag = 1; diag < n; diag++)
    for (i = 1; i ≤ n – diag; i++) {
      j = i + diag;
      val = $\min_{i \le t \le j}(C[i, t-1] + C[t+1, j])$;
      R[i, j] = The value of $t$ that minimizes val;
      C[i, j] = val + $\sum_{s=i}^{j} p[s]$;
    }

  return   <C[1, n], R>;
}
```

Analysis of the algorithm: $\Theta(n^3)$

Algorithm (for constructing the tree)

```
tree(i, j) {
  t = R[i, j];
  if (t == 0)     return NULL;
  p      = new node;
  p->key = key[t];
  p->left  = tree(i, t - 1);
  p->right = tree(t + 1, j);
  return p;
}
root = tree(1, n);
```

Subset-Sum Problem

Find a subset of a given set $A = \{a_1, a_2, \dots, a_n\}$ of n positive integers whose sum is equal to a given positive integer k.

Let's assume that `SubsetSums(A, k)` is the function that finds a subset of a set A whose sum is equal to a given positive integer k. The function returns a boolean value depending on the existence of such subset. In fact, this is a recursive function.

Algorithm (recursive version)

```
SubsetSums(a[], n, k) {
  if (k == 0)
    return   true;
  if (n == 0)
    return   false;
  if (a[n] > k)
    return   SubsetSums(a, n - 1, k);

  return SubsetSums(a, n - 1, k - a[n]) ||
         SubsetSums(a, n - 1, k);
}
SubsetSums(a, n, k);
```

Analysis of the algorithm: $O(2^n)$

Now, let's design a dynamic programming algorithm for this problem. A table $V[0..n, 0..k]$ is used to hold the results of subproblems:

- If there is a subset of a set $\{a_1, a_2, \dots, a_i\}$ whose sum is j $(1 \le j \le k)$: $V[i,j] = 1$
- Otherwise: $V[i,j] = 0$

We have the recurrence:

$$V[i,j] = \begin{cases} 1 & V[i-1,j] = 1 \lor V[i-1,j-a_i] = 1 \; where \; j \ge a_i \\ 0 & otherwise \end{cases}$$

where $V[0,1..k] = 0, V[0..n,0] = 1$.

Algorithm

```
SubsetSumsDP(a[1 .. n], n, k) {
  int V[0 .. n, 0 .. k];

  V[0 .. n, 0] = 1;
  V[0, 1 .. k] = 0;
  for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ k; j++) {
      tmp = 0;
      if (j ≥ a[i])
        tmp = V[i - 1, j - a[i]];
      V[i, j] = V[i - 1, j] || tmp;
    }
}

SubsetSumsDP(a, n, k);
```

Analysis of the algorithm: $\Theta(nk)$.

How to print the subset itself?

Algorithm

```
if (V[n, k]) {
  while (k) {
    if (V[n - 1, k - a[n]] == 1 && V[n - 1, k] == 0) {
      cout << a[n] << " ";
      k -= a[n];
    }
    n--;
  }
}
```

Knapsack Problem

Given n items of known weights $w_1, w_2, \dots, w_n (\in \mathbb{Z}^+)$ and values $v_1, v_2, \dots, v_n (\in \mathbb{R}^+)$ and a knapsack of capacity $C (\in \mathbb{Z}^+)$, find the most valuable subset of the items that fit into the knapsack.

Let's denote $T(i,j)$ the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

$$T(i,j) = \begin{cases} \max\{T(i-1,j), v_i + T(i-1, j-w_i)\} & j \geq w_i \\ T(i-1,j) & j < w_i \end{cases}$$

$$\begin{cases} T(0,j) = 0 & j \geq 0 \\ T(i,0) = 0 & i \geq 0 \end{cases}$$

Example: Consider a set of 4 items:

$$\{w_1 = 2, v_1 = 12\}$$

$$\{w_2 = 1, v_2 = 10\}$$

$$\{w_3 = 3, v_3 = 20\}$$

$$\{w_4 = 2, v_4 = 15\}$$

and $C = 5$.

i↓j→	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
4	0					

i↓j→	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
4	0					

i↓j→	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0					

i↓j→	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Algorithm

```
Knapsack(w[1 .. n], v[1 .. n], C) {
  int T[0 .. n, 0 .. C];

  T[0 .. n, 0] = T[0, 1 .. C] = 0;
  for (i = 1; i ≤ n; i++)
    for (j = 1; j ≤ C; j++)
      if (j ≥ w[i])
        T[i, j] = max{ T[i – 1, j],
                       v[i] + T[i – 1, j – w[i]]};
      else
        T[i, j] = T[i – 1, j];
  return T[n, C];
}
```

How to print the subset itself?

Traveling Salesman Problem

A *tour* (also called a *Hamiltonian circuit*) in a directed graph is a path from a vertex to itself that passes through each of the other vertices exactly once.

An *optimal tour* in a weighted, directed graph is such a path of minimum length.

The *traveling salesperson problem* is to find an *optimal tour* in a weighted, directed graph when at least one tour exists.

Note: Given graph $G = (V, E)$ where $V = \{v_1, v_2, \dots, v_n\}$. The starting vertex is irrelevant to the length of an optimal tour, we will consider v_1 to be the starting vertex.

Remark:

"If v_i is the first vertex after v_1 on an optimal tour, the subpath of that tour from v_i to v_1 must be a shortest path from v_i to v_1 that passes through each of the other vertices exactly once."

Let's denote

– W: An adjacency matrix representation of the graph G. The element $W[i, j] = k$ if there is an edge connecting a vertex v_i with a vertex v_j, $W[i, j] = \infty$ if there is no edge connecting two vertices v_i and v_j, or $W[i, j] = 0$ if $i = j$.

– $A(\subseteq V)$: A subset of V.

– $D[v_i, A]$: Length of a shortest path from v_i to v_1 passing through each vertex in A exactly once.

In general, $\forall i \in [2, n], v_i \notin A$:

$$D[v_i, A] = \begin{cases} \min\limits_{j: v_j \in A} \left\{ W[i, j] + D\left[v_j, A \backslash \{v_j\}\right] \right\} & A \neq \emptyset \\ W[i, 1] & A = \emptyset \end{cases}$$

Algorithm

```
TSP(n, W[1 .. n, 1 .. n], P[1 .. n, 1 .. n]) {
  D[1 .. n, subset of V \ {v₁}];
  P[1 .. n, subset of V \ {v₁}];

  D[2 .. n, ∅] = W[2 .. n, 1];
  for (k = 1; k ≤ n - 2; k++)
    for (all subsets A ⊆ V \ {v₁} containing k vertices)
      for (i such that i ≠ 1 and vᵢ ∉ A) {
        D[i, A] = $\min_{j:v_j\in A}$ {W[i, j] + D[j, A \ {vⱼ}]};
        P[i, A] = value of j that gave the minimum;
      }

  D[1, V \ {v₁}] = $\min_{2\le j\le n}$ {W[1][j] + D[j][V \ {v₁, vⱼ}]};
  P[1, V \ {v₁}] = value of j that gave the minimum;

  return { D[1, V \ {v₁}], P };
}
```

Analysis of the algorithm: $T(n) \in \Theta(n^2 2^n), M(n) \in \Theta(n 2^n)$

How to print the optimal tour?

Algorithm

```
TSP_Tour(P[1..n, 1..n]) {
  cout << v₁;
  A = V \ {v₁};
  k = 1;
  while (A ≠ ∅) {
    k = P[k, A];
    cout << vₖ;
    A = A \ {vₖ};
  }
}
```

Memoization (or Memory Function)

Dynamic programming and divide-and-conquer solve problems that have a recurrence relation. The divide-and-conquer is a top-down technique. It has the disadvantage that it solves common subproblems multiple times. This leads to poor efficiency (typically, exponential or worse). The dynamic programming is a bottom-up technique, and solving all the subproblems only once. This has the disadvantage that solutions to some of subproblems are often not necessary for getting a solution to the problem given.

We would like to have the best of both worlds, i.e. all the necessary subproblems solved only once. This is possible using memory functions – a hybrid technique that combines the strengths of divide-and-conquer and dynamic programming.

The hybrid technique uses a top-down approach with table of subproblem solution. Before determining the solution recursively, the algorithm checks if the subproblem has already been solved by checking the table. If the table has a valid value then the algorithm uses the table value; otherwise it proceeds with the recursive solution.

Example: Finding the n^{th} Fibonacci number.

Algorithm

```
Fib(f[0 .. n], n) {
  if (f[n] < 0)
    f[n] = Fib(f, n - 1) + Fib(f, n - 2);
  return f[n];
}

Fib_Memo(n) {
  f[0] = 0;
  f[1] = 1;
  f[2 .. n] = -1;
  return Fib(f, n);
}
```

Example: Matrix chain multiplication

The recurrence relation is as follows:

$$C(i,j) = \begin{cases} \min\limits_{i \le k < j}\{C(i,k) + C(k+1,j) + d_{i-1} \times d_k \times d_j\} & i < j \\ 0 & i = j \end{cases}$$

Algorithm

```
ChainMatrixMult_Memo(d[0 .. n]) {
  C[.., ..] = ∞;
  return MMM(C, d, 1, n);
}

MMM(C[1 .. n, 1 .. n], d[0 .. n], i, j) {
  if (C[i, j] < ∞)
    return C[i, j];

  if (i == j)
    C[i, j] = 0;
  else
    for (k = i; k < j; k++) {
      t = MMM(C, d, i, k) + MMM(C, d, k + 1, j) + d[i - 1]
* d[k] * d[j];
      if (t < C[i, j])
        C[i, j] = t;
    }
  return C[i, j];
}
```

Example: The knapsack problem

The recurrence relation is as follows:

$$T[i,j] = \begin{cases} max\{T[i-1,j], v_i + T[i-1, j-w_i]\} & j \ge w_i \\ T[i-1,j] & j < w_i \end{cases}$$

where $\begin{cases} T[0,j] = 0 & j \ge 0 \\ T[i,0] = 0 & i \ge 0 \end{cases}$

Algorithm

```
Knapsack(T[0 .. n, 0 .. W] , i, j) {
   if (T[i, j] < 0) {
      if (j < w[i])
         tmp = Knapsack(T, i - 1, j);
      else
         tmp = max(Knapsack(T, i - 1, j), v[i] + Knapsack(T, i-
1, j - w[i]));
      T[i, j] = tmp;
   }
   return T[i, j];
}

Knapsack_Memo() {  // global variable: w[1 .. n], v[1 .. n]
   T[,] = -1;
   T[0, 0 .. W] = T[0 .. n, 0] = 0;
   return Knapsack(T, n, W);
}
```

Remark: The input data is a set of 4 items:

$$\{w_1 = 2, v_1 = 12\}$$

$$\{w_2 = 1, v_2 = 10\}$$

$$\{w_3 = 3, v_3 = 20\}$$

$$\{w_4 = 2, v_4 = 15\}$$

and $C = 5$:

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–1	12	22	–1	22
3	0	–1	–1	22	–1	32
4	0	–1	–1	–1	–1	**37**