CST207
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 6: Dynamic Programming

Lecturer: Dr. Yang Lu

Email: luyang@xmu.edu.my

Office: A1-432

Office hour: 2pm-4pm Mon & Thur

Recall Calculation of the nth Fibonacci Term

- The time complexity of this algorithm is $\Theta(2^n)$.
- A lot of time is wasted on *recomputing* the same term.

```c
int fib(int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

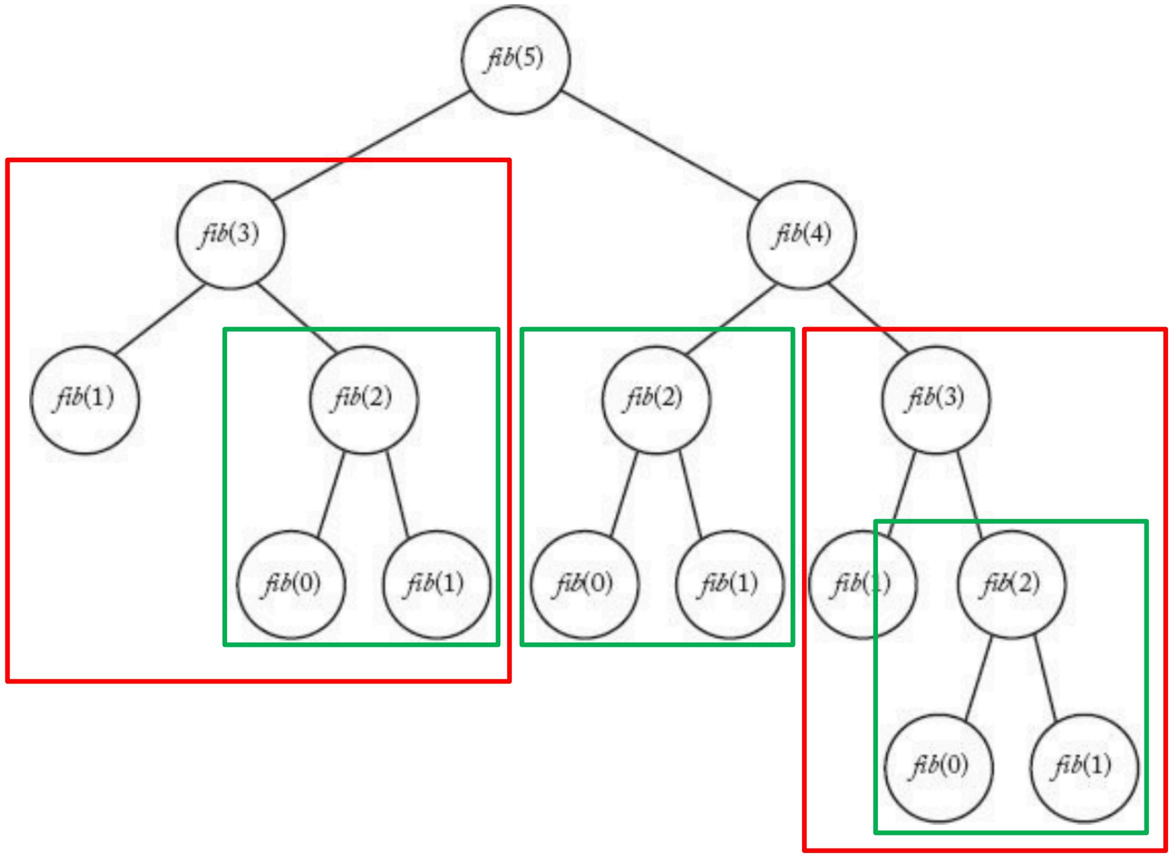


Image source: Figure 1.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Recall Calculation of the nth Fibonacci Term

- A straightforward solution: store the values in an array to avoid recomputing.
- The time complexity reduces from $\Theta(2^n)$ to $\Theta(n)$.

```
int fib2 (int n)
{
    index i;
    int f[0...n];

    f[0] = 0;
    if (n > 0){
        f[1] = 1;
        for (i = 2; i <= n; i++)
            f[i] = f[i - 1] + f[i - 2];
    }
    return f[n];
}
```

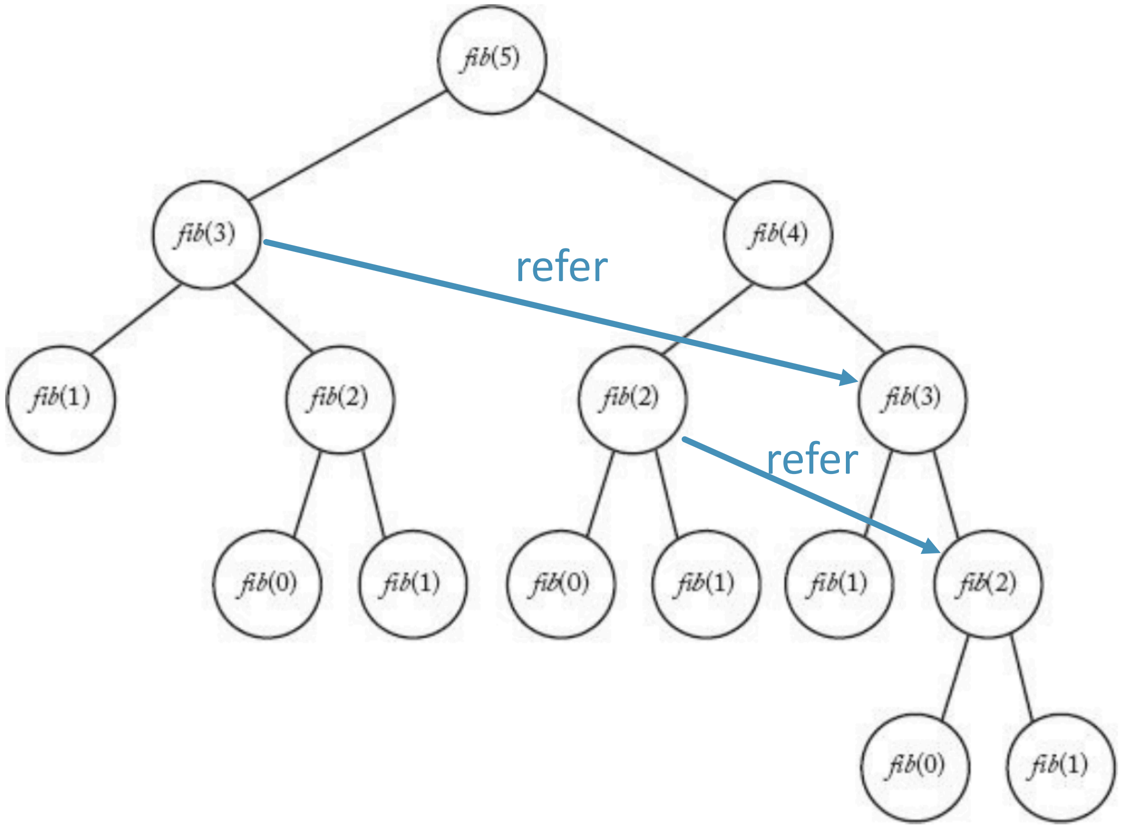


Image source: Figure 1.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Dynamic Programming

- Dynamic programming is similar to divide-and-cnoquer.
  - An instance of a problem is divided into smaller instances.
- However, the difference is:
  - Divide-and-cnoquer is a top-down approach.
  - Dynamic programming is a bottom-up approach.
- The steps in the development of a dynamic programming algorithm are:
  - Establish a *recursive property* that gives the solution to an instance of the problem.
  - Solve an instance of the problem in a *bottom-up* fashion by solving smaller instances first.

Outline

We discuss dynamic programming with six problems:

- The binomial coefficient
- Chained matrix multiplication
- Optimal binary search trees
- Knapsack problem
- Floyd’s algorithm for shortest paths
- Sequence alignment

THE BINOMIAL COEFFICIENT

The Binomial Coefficient

- The binomial coefficient is calculated by:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{for } 0 \le k \le n.$$

- We cannot compute the binomial coefficient directly by the definition because $n!$ is very large even for moderate values of n.

The Binomial Coefficient

- By representing binomial coefficients as the Pascal's traiangle, we can establish the recursive property:

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

- Each entry is the sum of the two above.
- The computation of $n!$ and $k!$ is eliminated.

k

n

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

The Pascal's triangle

Image source: https://en.wikipedia.org/wiki/Binomial_coefficient

The Binomial Coefficient Solved By Recursion

- Like the recursive version of the nth Fibonacci term calculation algorithm, using recursion to calculate binomial coefficient is very inefficient.
- A great number of terms are recomputed.
  - bin_coef_recursion(n-1,k-1) and bin_coef_recursion(n-1,k) both need the result of bin_coef_recursion(n-2,k-1).
- The divide-and-conquer approach is always inefficient when an instance is divided into two smaller instances that are almost as large as the original instance.

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

The Binomial Coefficient Solved By Dynamic Programming

- Store the computation result of $\binom{i}{j}$ in $B[i][j]$ with an array B.
- Recomputing can be avoided by directly indexing the array.
- The steps for constructing a dynamic programming algorithm for this problem:
  - Establish a recursive property:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ or } j = i. \end{cases}$$

  - Solve an instance of the problem in a *bottom-up* fashion by computing from the first row of B.
- The optimal solution is $B[n][k]$.

The Binomial Coefficient Solved By Dynamic Programming

- We only need to calculate up to the kth column for each row.
- Actually, the calculation only needs the previous row. Therefore, all the rows before the previous row can be discarded.
  - The algorithm can be further improved by just using a single 1-d array.

```
int bin_coef_dp (int n, int k)
{
    index i, j;
    int B[0...n][0...k];

    for (i = 0; i <= n; i++)
        for (j = 0; j <= min(i, k); j++)
            if (j == 0 || j == i)
                B[i][j] = 1;
            else
                B[i][j] = B[i - 1][j - 1] + B[i - 1][j];
    return B[n][k];
}
```

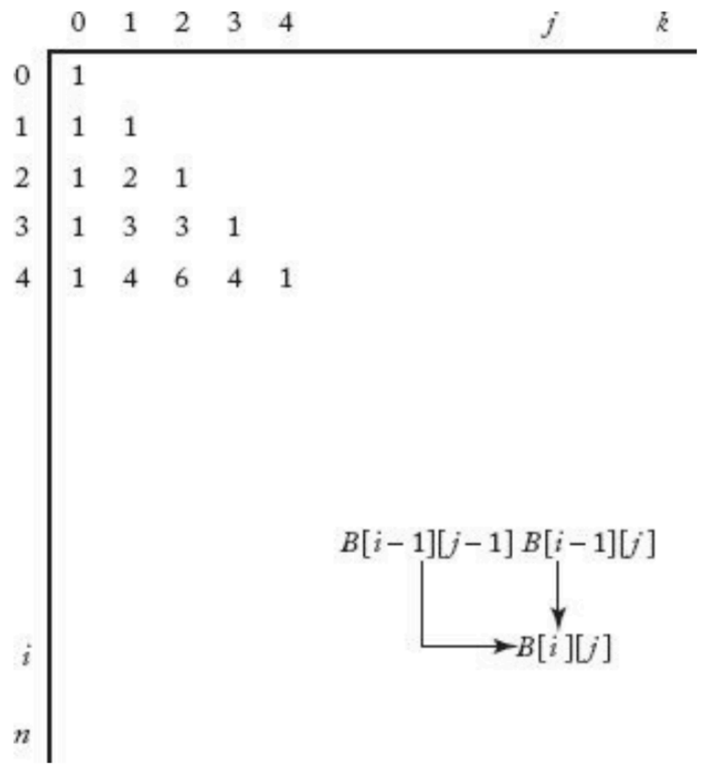


Image source: Figure 3.1, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

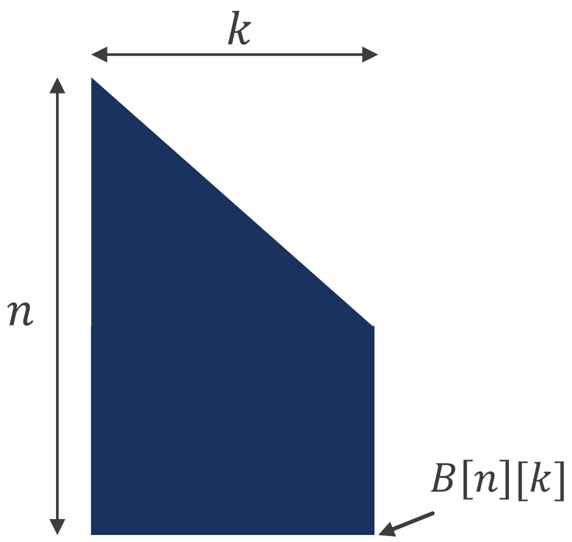
The Binomial Coefficient Solved By Dynamic Programming

- Every-case time complexity of this algorithm is determined by:

$$1 + 2 + 3 + 4 + \cdots + k + \underbrace{(k+1) + (k+1) + \cdots + (k+1)}_{n-k+1 \text{ times}}.$$

- It equals

$$\frac{k(k+1)}{2} + (n-k+1)(k+1) = \frac{(2n-k+2)(k+1)}{2} \in \Theta(nk).$$



Shape of the array

CHAINED MATRIX MULTIPLICATION

Chained Matrix Multiplication

- To multiply an $i \times j$ matrix with a $j \times k$ matrix using the standard method, it is necessary to do $i \times j \times k$ elementary multiplications.
- Consider the chained matrix multiplication:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

- The total number of elementary multiplications depends on the multiplication order.

$$\begin{array}{ll} A(B(CD)) & 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680 \\ (AB)(CD) & 20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880 \\ A((BC)D) & 2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232 \\ ((AB)C)D & 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320 \\ (A(BC))D & 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120 \end{array}$$

Chained Matrix Multiplication

- Our goal is to develop an algorithm that determines the optimal order for multiplying n matrices.
  - The input of the algorithm is the dimensions of these matrices.
- Let d_0 be the number of rows in A_1 and d_k be the number of columns in A_k for $1 \le k \le n$, the dimension of A_k is $d_{k-1} \times d_k$.
  - We have $n + 1$ dimensions for multiplying n matrices.
- We can decompose the matrices, such that the optimal solution with n matrices can be constructed in *bottom-up* fashion.
- Then, we can define for $1 \le i \le j \le n$, $M[i][j]$ is the minimum number of multiplications needed to multiply A_i through A_j, if $i < j$, and $M[i][i] = 0$.
- The optimal solution is $M[1][n]$.

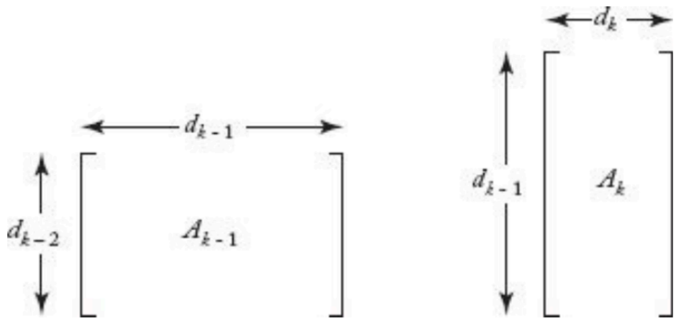


Image source: Figure 3.7, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Chained Matrix Multiplication

- Assume we have six matrices, the optimal order must have one of the following factorizations:
  - $A_1(A_2A_3A_4A_5A_6)$
  - $(A_1A_2)(A_3A_4A_5A_6)$
  - $(A_1A_2A_3)(A_4A_5A_6)$
  - $(A_1A_2A_3A_4)(A_5A_6)$
  - $(A_1A_2A_3A_4A_5)A_6$
- Generally, the optimal order must be with some k, for $1 \leq k \leq n-1$:

$$(A_1 \ldots A_k)(A_{k+1}A_n)$$

Chained Matrix Multiplication

- We can obtain the following recursive property for $1 \leq i \leq j \leq n$:

$$M[i][j] = \min_{i \leq k \leq j-1} \left(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j \right), \qquad \text{if } i < j.$$

$$M[i][i] = 0.$$

- Different from the binomial coefficient problem that each term is calculated by the top left and top terms, $M[i][j]$ needs the term on its left and its bottom.

	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$
$i = 1$	$M[1][1]$	$M[1][2]$	$M[1][3]$	$M[1][4]$	$M[1][5]$	$M[1][6]$
$i = 2$		$M[2][2]$	$M[2][3]$	$M[2][4]$	$M[2][5]$	$M[2][6]$
$i = 3$			$M[3][3]$	$M[3][4]$	$M[3][5]$	$M[3][6]$
$i = 4$				$M[4][4]$	$M[4][5]$	$M[4][6]$
$i = 5$					$M[5][5]$	$M[5][6]$
$i = 6$						$M[6][6]$

	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$	$j = 6$	
$i = 1$	$M[1][1]$	$M[1][2]$	$M[1][3]$	$M[1][4]$	$M[1][5]$	$M[1][6]$	Diagnal 5
$i = 2$		$M[2][2]$	$M[2][3]$	$M[2][4]$	$M[2][5]$	$M[2][6]$	Diagnal 4
$i = 3$			$M[3][3]$	$M[3][4]$	$M[3][5]$	$M[3][6]$	Diagnal 3
$i = 4$				$M[4][4]$	$M[4][5]$	$M[4][6]$	Diagnal 2
$i = 5$					$M[5][5]$	$M[5][6]$	Diagnal 1
$i = 6$						$M[6][6]$	Diagnal 0

Pseudocode of Chained Matrix Multiplication

- Except the loop over $diagonal$ and the loop over i, find the minimum value is also a loop over k.
- For given values of $diagonal$ and i, for $i \leq k \leq j-1$, the number of passes through k is

  $j-1-i+1 = i + diagonal - 1 - i + 1 = diagonal$
- The number of passes through i is $n - diagonal$.
- The number of passes through $diagonal$ is $n-1$.
- Totally, the every-case time complexity is:

$$\sum_{diagonal=1}^{n-1} (n - diagonal) \times diagonal \in \Theta(n^3).$$

```
int chained_mat_mult (int n,
                      const int d[],
                      index P[][])
{
    index i, j, k, diagonal;
    int M[1...n][1...n]

    for (i = 1; i <= n; i++)
        M[i][i] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)
        for (i = 1; i <= n - diagonal; i++){
            j = i + diagonal;
            M[i][j] = min(M[i][k] + M[k + 1][j] + d[i - 1] * d[k] * d[j]);
            P[i][j] = the value of k that gives the minimum;
        }
    return M[1][n];
}
```

XIAMEN UNIVERSITY MALAYSIA 廈門大學馬來西亞分校

厦门大学信息学院 SCHOOL OF INFORMATICS XIAMEN UNIVERSITY

厦门大学计算机科学系 Computer Science Department of Xiamen University

Determine the Optimal Order

- The optimal order is determined by recursively examining the array P.

Optimal order:
$A_1((((A_2A_3)A_4)A_5)A_6)$

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

P

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

OPTIMAL BINARY SEARCH TREES

Optimal Binary Search Trees

- A *binary search tree* is a binary tree of keys that come from an ordered set, such that
  - Each node contains one key.
  - The keys in the left subtree of a given node are less than or equal to the key in that node.
  - The keys in the right subtree of a given node are greater than or equal to the key in that node.

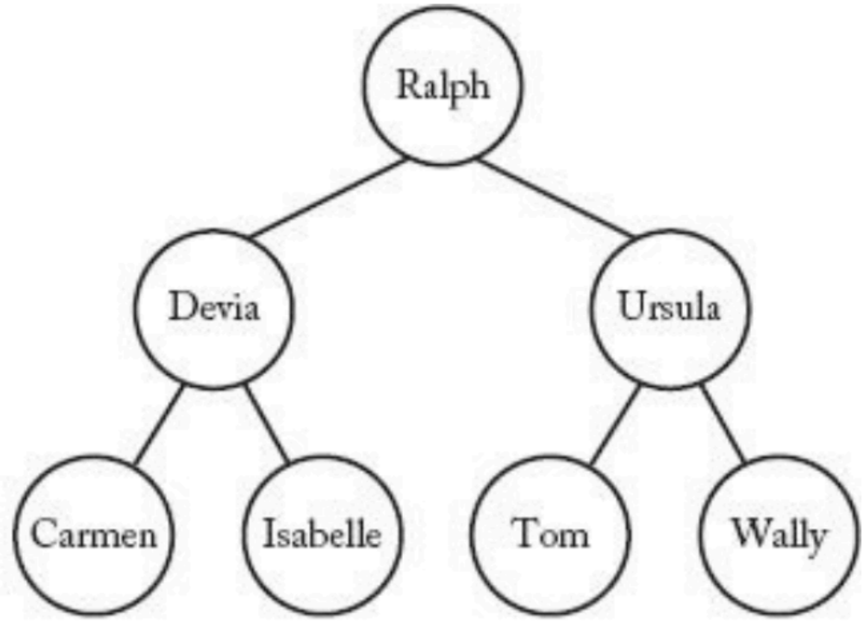


Image source: Figure 3.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees

- The number of comparisons done by search to locate a key is called the *search time*.
- We want to know the average search time of a binary search tree while the keys do not have the same probability.
  - E.g. Tom is a common name is the United States. It has higher probability to be a search key.
  - Thus, put the node whose key has high probability to lower depth will decrease the average search time.

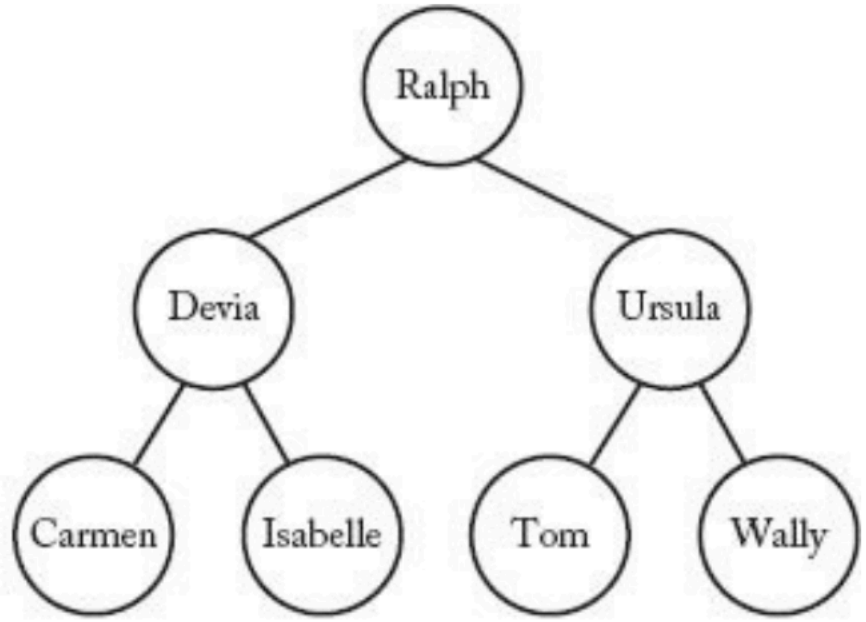


Image source: Figure 3.10, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees

- An *optimal binary search tree* minimizes the average time it takes to locate a key.
- Assume the search key is always in the tree. Let $Key_1, Key_2, \ldots, Key_n$ be the n keys in order, and let p_i be the probability that Key_i is the search key.
  - The actual values of the keys are not important.
- The search time c_i for a given key is

$$c_i = depth(Key_i) + 1,$$

  - Recall that $depth(root) = 0$.
- The average search time we want to minimize is

$$\sum_{i=1}^{n} c_i p_i$$

Example

- This figure shows the five different trees when $n = 3$.
- The probabilities are:

$$p_1 = 0.7 \quad p_2 = 0.2 \quad p_3 = 0.1$$

- The average search times are:
  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
- Tree 5 is optimal.

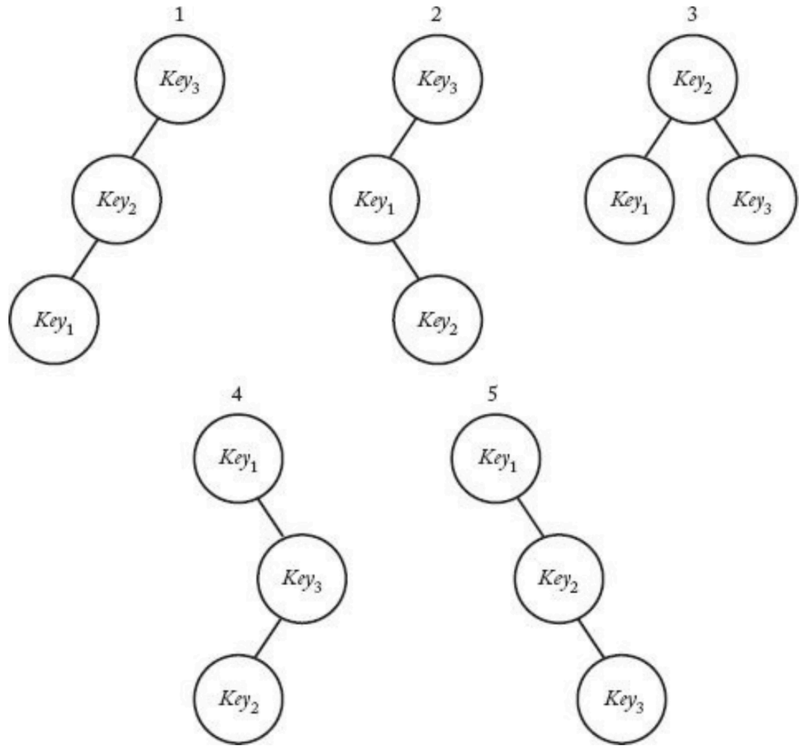


Image source: Figure 3.11, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees by Dynamic Programming

- As usual, enumerating and calculating all cases is impossible, which is again exponential.
- We can decompose the tree with subtrees, such that the optimal binary search tree can be constructed in *bottom-up* fashion.
- We use $A[i][j]$ to represent the optimal search time of the binary search tree constructed from Key_i to Key_j.
- The optimal solution is $A(1, n)$.
- For $1 \leq k \leq n$, there must exist an optimal binary search tree whose root has Key_k.
  - Its subtrees must also be optimal.

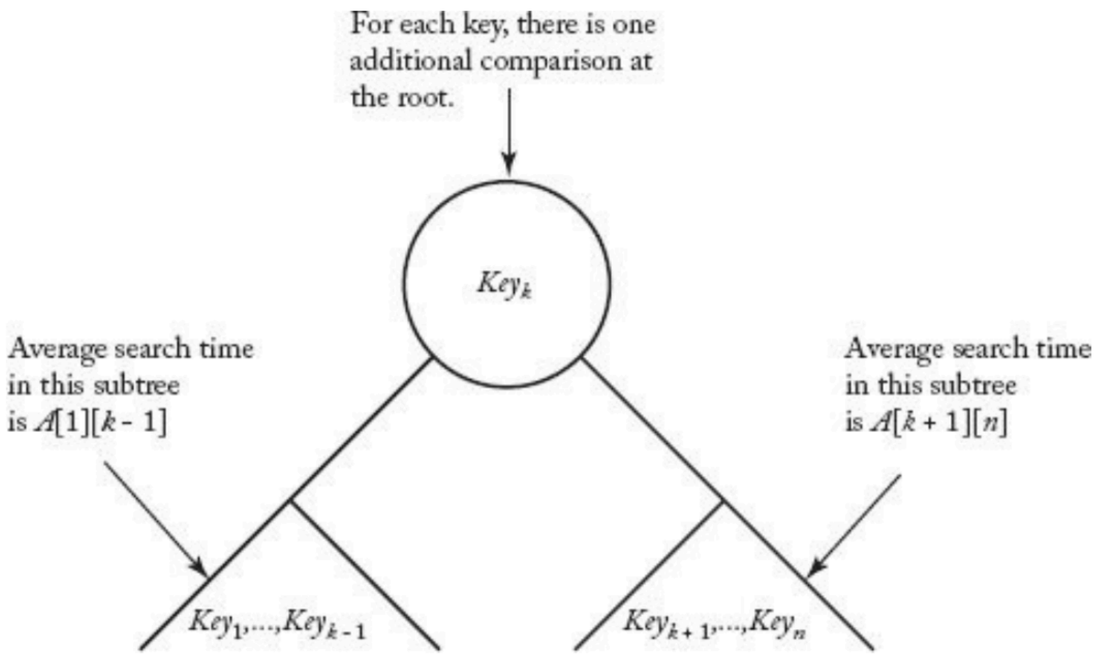


Image source: Figure 3.13, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees by Dynamic Programming

- Because the subtrees have one more depth, we should add the probabilities of all their keys.
- The average time in left subtree is:

$$A[1][k-1] + p_1 + \cdots + p_{k-1}$$

- The average time in right subtree is:

$$A[k+1][n] + p_{k+1} + \cdots + p_n$$

- The average time searching for root: p_k
- Totally:

$$A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$

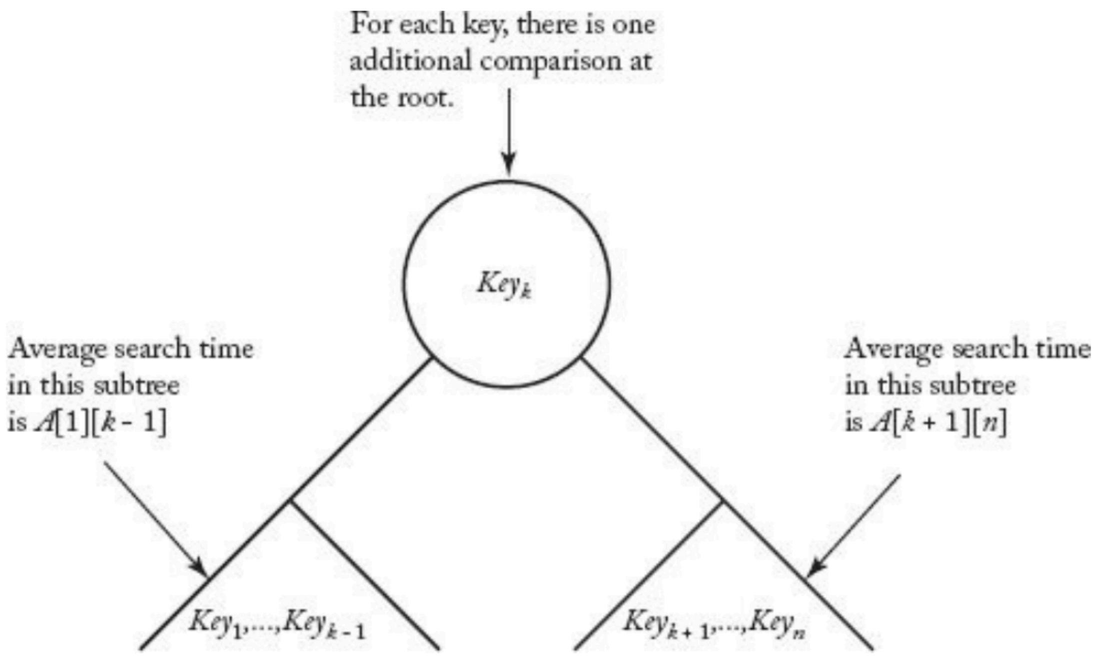


Image source: Figure 3.13, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Optimal Binary Search Trees by Dynamic Programming

- We can derive the following recursive property:

$$A[i][j] = \min_{i \le k \le j}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \qquad \text{for } i < j$$

$$A[i][i] = p_i$$

$A[i][i-1]$ and $A[j+1][j]$ are defined to be 0.

- The bottom-up strategy for solving this recursive property is similar to the chained matrix multiplication problem.
  - Use the diagonal trick.

Pseudocode of Optimal Binary Search Trees

- The every-case time complexity is $\Theta(n^3)$.

```
struct nodetype
{
    keytype key;
    nodetype* left;
    nodetype* right;
};

typedef nodetype* node_pointer;
```

```
node_pointer construct_opt_search_tree (index i, j)
{
    index k;
    node_pointer p;

    k = P[i][j];
    if (k == 0)
        return NULL;
    else{
        p = new nodetype;
        p -> key = Key[k];
        p -> left = construct_opt_search_tree(i, k - 1);
        p -> right = construct_opt_search_tree(k + 1, j);
        return p;
    }
}
```

```
void opt_search_tree (int n,
                      const float p[],
                      float& minavg,
                      index P[][])
{
    index i, j, k, diagonal;
    float A[1...n+1][0...n];

    for (i = 1; i <= n; i++){
        A[i][i - 1] = 0;
        A[i][i] = p[i];
        P[i][i] = i;
        P[i][i - 1] = 0;
    }
    A[n + 1][n] = 0;
    P[n + 1][n] = 0;
    for (diagonal = 1; diagonal <= n - 1; diagonal++)
        for (i = 1; i <= n - diagonal; i++){
            j = i + diagonal;
            A[i][j] = min(A[i][k - 1] + A[k + 1][j]) + sum(p[i...j]);
            P[i][j] = the value of k that gives the minimum;
        }
    minavg = A[1][n];
}
```

KNAPSACK PROBLEM

Knapsack Problem

- Problem description:
  - Given n items and a "knapsack.”
  - Item i has weight $w_i > 0$ and has value $v_i > 0$.
  - Knapsack has capacity of W.
  - Goal: Fill knapsack so as to maximize total value.
- Mathematical description:
  - Given two n-tuples of positive numbers $< v_1, v_2, \dots, v_n >$ and $< w_1, w_2, \dots, w_n >$, and $W > 0$, we wish to determine the subset $T \subseteq \{1,2,\dots,n\}$ that

$$\text{maximize} \sum_{i \in T} v_i \qquad \text{subject to} \sum_{i \in T} w_i \leq W$$

Example

- Weight capacity $W = 5\text{kg}$.
- The possible ways to fill the knapsack:
  - {1, 2, 3} has value $37 with weight 4kg.
  - {3, 4} has value $35 with weight 5kg.
  - {1, 2, 4} has value $42 with weight 5kg. (optimal)

i	v_i	w_i
1	$10	1kg
2	$12	1kg
3	$15	2kg
4	$20	3kg

Knapsack Problem by Dynamic Programming

- We can decompose the item set and the maximum weight, such that the optimal solution with n items and W capcity can be constructed in *bottom-up* fashion.
- We define $V(i,j)$ as the optimal solution of items subset $\{1, \dots, i\}$ with capacity j.
- The optimal solution is $V(n, W)$.
- There are two cases for $V(i,j)$:
  - $V(i,j)$ does not include item i, because of out of capacity or not worthy.
    - $V(i,j) = V(i-1, j)$.
  - $V(i,j)$ includes item i.
    - $V(i,j) = V(i-1, j-w_i) + v_i$.

Knapsack Problem by Dynamic Programming

- We can establish the recursive property:

$$V(i,j) = \begin{cases} V(i-1,j) & \text{if } j - w_i < 0 \\ \max(V(i-1,j), V(i-1,j-w_i) + v_i) & \text{if } j - w_i \geq 0 \end{cases}$$

$$V(0,j) = 0 \ \text{ for } j \geq 0$$

$$V(i,0) = 0 \ \text{ for } i \geq 0$$

- The bottom-up construction is easy, just loop over i and j for calculating array V.

Example

$V(1,1) = \max(V(0,1), V(1,1-w_1)+v_1)$
$V(0,1) = 0$
$V(1,0) + 10 = 10$

$V(2,2) = \max(V(1,2), V(1,2-w_2)+v_2)$
$V(1,2) = 10$
$V(1,1) + 12 = 22$

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$i=0$	0	0	0	0	0	0
$i=1$	0	10	10	10	10	10
$i=2$	0	12	22	22	22	22
$i=3$	0	12	22	27	37	37
$i=4$	0	12	22	27	37	42

i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg
4	\$20	3kg

$V(3,2) = \max(V(2,2), V(2,2-w_3)+15)$
$V(2,2) = 22$
$V(2,0) = 0$

$V(4,5) = \max(V(3,5), V(3,5-w_5)+v_5)$
$V(3,5) = 37$
$V(3,2) + 20 = 42$

FLOYD’S ALGORITHM FOR SHORTEST PATHS

The Shortest Path Problem

- A common problem encounted by air travelers is the determination of the shortest way to fly from one city to another without a direct fight.
- We represent this kind of problem by using a graph.

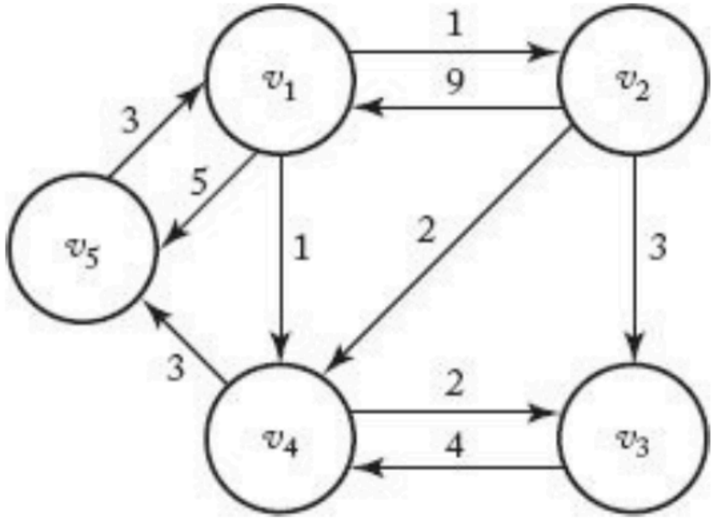


Image source: Figure 3.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Review of Graph Theory

- In graph theory, the *edges* are linked between *vertices*.
- If each edge has a direction, the graph is called a *directed graph*.
- If the edges have values associated with them, the values are called *weights* and the graph is called a *weighted graph*.
  - Weights are usually assumed to be nonnegative.
  - In many applications weights are used to represent distances.

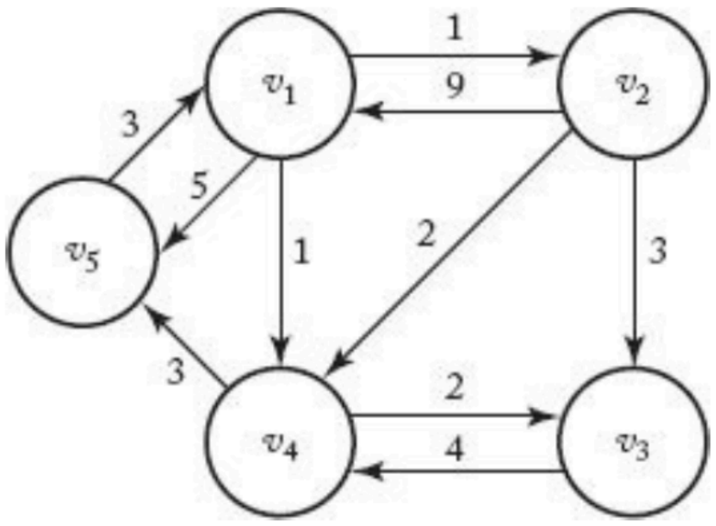


Image source: Figure 3.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Review of Graph Theory

- In a directed graph, a *path* is a sequence of vertices such that there is an edge from each vertex to its successor.
  - In the figure, $[v_1, v_4, v_3]$ is a path and $[v_3, v_4, v_1]$ is not a path.
- A path is called *simple* if it never passes through the same vertex twice.
- The *length* of a path in a weighted graph is the sum of the weights on the path.

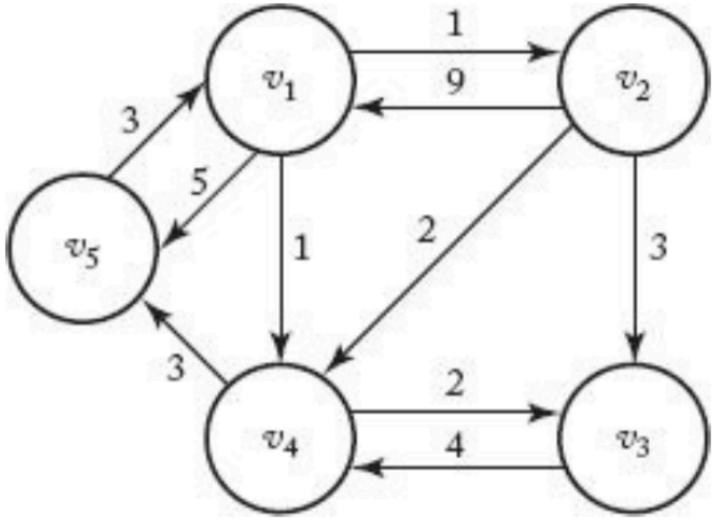


Image source: Figure 3.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

The Shortest Path Problem

- A shortest path must be a simple path.
- There are three simple paths from v_1 to v_3, and their lengths are:

$$\begin{aligned} length[v_1, v_2, v_3] &= 1 + 3 = 4 \\ length[v_1, v_4, v_3] &= 1 + 2 = 3 \\ length[v_1, v_2, v_4, v_3] &= 1 + 2 + 2 = 5. \end{aligned}$$

- Obviously, $[v_1, v_4, v_3]$ is the shortest path from v_1 to v_3.

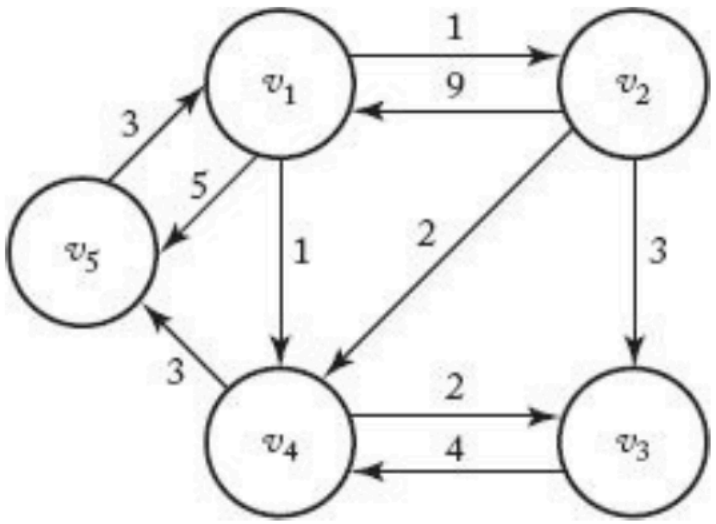


Image source: Figure 3.2, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

The Shortest Path Problem

- An obvious algorithm would be to determine all the paths from the starting vertex to the ending vertex, and select the ones with the minimum length.
- Suppose all vertices are connected
  - The second vertex in the path can be any of the $n-1$ vertices.
  - The third vertex in the path can be any of the $n-2$ vertices.
  - …
- The total number of paths:

$$(n-1)(n-2)\ldots 1 = (n-1)!,$$

which has factorial complexity.

Shortest Path Problem by Dynamic Programming

- We create an array W called *adjacency matrix* to represent the graph.

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge from } v_i \text{ to } v_j \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \\ 0 & \text{if } i = j. \end{cases}$$

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

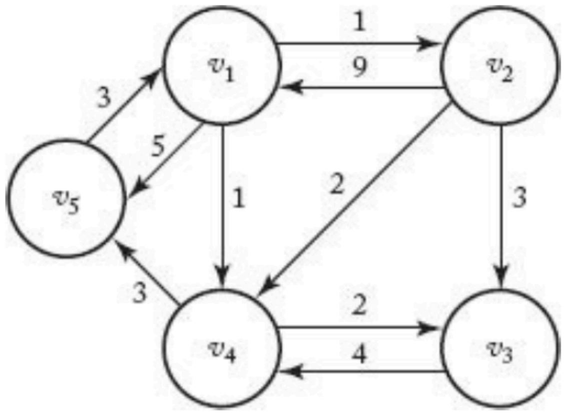


Image source: Figure 3.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Shortest Path Problem by Dynamic Programming

- We can decompose the vertices, such that the optimal solution with n vertices can be constructed in *bottom-up* fashion with its subsets.
- We create an array D that contains the lengths of the shortest paths in the graph.
  - $D[i][j]$ is the shortest path from v_i to v_j.
- To calculate D, we create a sequence of $n+1$ arrays $D^{(k)}$, where $0 \leq k \leq n$.
  - $D^{(k)}[i][j]$ is the length of a shortest path from v_i to v_j using only vertices in the set $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices.
- Thus, we have $D^{(0)} = W$ and $D^{(n)} = D$.

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Image source: Figure 3.3, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Shortest Path Problem by Dynamic Programming

- Therefore, to determine D from W we need only find a way to obtain $D^{(n)}$ from $D^{(0)}$.
- The steps for using dynamic programming:
  - Establish a recursive property with which we can compute $D^{(k)}$ from $D^{(k-1)}$.
  - Solve an instance of the problem in a bottom-up fashion by repeating the process for $k = 1$ to n. This creates the sequence $D^{(0)}, D^{(1)}, D^{(2)}, \dots, D^{(n)}$.

Shortest Path Problem by Dynamic Programming

- We accomlish Step 1 by considering the shortest path, using only vertices in $\{v_1, v_2, \dots, v_k\}$ as intermediate vertices with two cases:
  - Case 1. It does not use v_k. Then
$$D^{(k)}[i][j] = D^{(k-1)}[i][j].$$
  - Case 2. It uses v_k. Then
$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j]$$

Shortest Path Problem by Dynamic Programming

- Because we calculate $D^{(k)}$ in bottom-up fashion, we know all the values in $D^{(k-1)}$.
- Thus, $D^{(k)}$ could be determined by

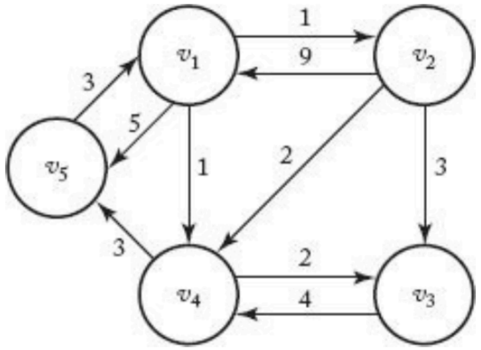
$$D^{(k)}[i][j] = \min\Big(\underbrace{D^{(k-1)}[i][j]}_{\text{Case 1}},\ \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{Case 2}}\Big).$$

- After D is calculated, we have actually calculated the shortest path from v_i to v_j for any i and j.

Example

We calculate $D[5][4]$ as an example.

- $D^{(0)}[5][4] = W[5][4] = \infty$.
- $D^{(1)}[5][4] = \min(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4]) = \min(\infty, 3+1) = 4$.
- $D^{(2)}[5][4] = \min(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4]) = \min(4, 4+2) = 4$.
  - $D^{(1)}[5][2] = \min D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2] = \min(\infty, 3+1) = 4$.
  - $D^{(1)}[2][4] = \min D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4] = \min(2{,}9+1) = 2$.
- $D^{(3)}[5][4] = \min(D^{(2)}[5][4], D^{(2)}[5][3] + D^{(2)}[3][4]) = \cdots$
- $D^{(4)}[5][4] = \min(D^{(3)}[5][4], D^{(3)}[5][2] + D^{(3)}[4][4]) = \cdots$
- $D^{(5)}[5][4] = \min(D^{(4)}[5][4], D^{(4)}[5][5] + D^{(4)}[5][4]) = \cdots$



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

Pseudocode of Floyd's Algorithm

- The every-case time complexity is obviously $\Theta(n^3)$.
- We can use an array P to record the index of intermediate vertex.
  - If at least one intermediate vertex exists, $P[i][j]$ is the highest index of an intermediate vertex on the shortest path from v_i to v_j; otherwise $P[i][j]$ is 0.

```
void floyd ( int n,
             const number W[][],
                   number D[][],
                   index  P[][])
{
    index i, j, k;

    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            P[i][j] = 0;
    D = W;
    for (k = 1; k <= n; k++)
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (D[i][k] + D[k][j] < D[i][j]){
                    P[i][j] = k;
                    D[i][j] = D[i][k] + D[k][j];
                }
}
```

path(5,3) calls path(5,4) and path(4,3).
path(5,4) calls path(5,1) and path(1,4).
Output: v5 v1 v4 v3

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

P

```
void path (index q, r)
{
    if (P[q][r] != 0){
        path(q, P[q][r]);
        cout << "v" << P[q][r];
        path(P[q][r], r);
    }
}
```


Image source: Figure 3.5, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Dynamic Programming and Optimization Problems

- For an optimization problem like the shortest path problem, it has an optimal solution (e.g. path) with an optimal value (e.g. length).
- The steps of developing a dynamic programming algorithm for an opimization problem can be generalized as:
  - Establish a recursive property that gives the optimal solution to an instance of the problem/
  - Compute the value of an optimal solution in a bottom-up fashion.
  - Construct an optimal solution in a bottom-up fashion.

Definition

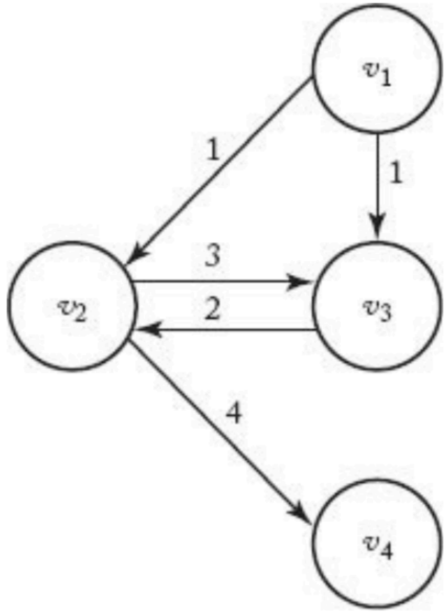
The principle of optimality is said to apply in a problem if an optimal solution to an instance of a problem always contains optimal solution to all subinstances.

Example

- We show the following example shows that dynamic programming does not apply in every optimization problem.
- Consider the longest path problem with single path.
- The optimal longest simple path from v_1 to v_4 is $[v_1, v_3, v_2, v_4]$.
- However, the subpath $[v_1, v_3]$ is not an optimal longest path from v_1 to v_3 because

$$length[v_1,v_3] = 1 \quad \text{and} \quad length[v_1,v_2,v_3] = 4.$$

- The reason is that the optimal path from v_1 to v_3 ($[v_1,v_2,v_3]$) and from v_3 to v_4 ($[v_3,v_2,v_4]$) cannot be put together to construct a simple path.



SEQUENCE ALIGNMENT

Sequence Alignment

- Sequence alignment finds the optimal way to align two sequences.
- Use DNA sequence as an example:

A A C A G T T A C C

T A A G G T C A

- The following shows two possible alignments:

_ A A C A G T T A C C	A A C A G T T A C C
T A A _ G G T _ _ C A	T A _ A G G T _ C A

- There are two possible way to make alignments:
  - Insert a gap as represented by “_”.
  - Find a mismatch.

Cost of Sequence Alignment

- The cost of these two alignments are different:

_ A A C A G T T A C C	A A C A G T T A C C
T A A _ G G T _ _ C A	T A _ A G G T _ C A

- By assignment the gap with cost 2 and mismatch with cost 1,
  - The left one has a cost of 10.
  - The right one has a cost of 7.
- The optimal sequence alignment is with the minimum cost.

Sequence Alignment by Dynamic Programming

- Use $x[0 \ldots m]$ and $y[0 \ldots n]$ to represent the two sequences.
- We can decompose each sequence, such that the optimal solution for sequences with length m and n can be constructed in *bottom-up* fashion.
- Let $opt(i, j)$ be the cost of the optimal alignment of the subsequences $x[i \ldots m]$ and $y[j \ldots n]$.
- The optimal alignment is $opt(0,0)$.
- Now, how can we build the recursive property?

Sequence Alignment by Dynamic Programming

The optimal alignment must start with one of the three cases:

- $x[0]$ is aligned with $y[0]$. $x[0] = y[0]$ has no cost and $x[0] \neq y[0]$ has cost 1.
  - The optimal cost is $opt(1,1) + cost$.
- $x[0]$ is aligned with a gap and the cost is 2.
  - The optimal cost is $opt(1,0) + 2$.
- $y[0]$ is aligned with a gap and the cost is 2.
  - The optimal cost is $opt(0,1) + 2$.

Sequence Alignment by Dynamic Programming

- Thus, we can establish the recursive property:

$$opt(i,j) = \min(opt(i+1,j+1) + cost, opt(i+1,j) + 2, opt(i,j+1) + 2)$$

- Different from the previous examples where the optimal solution is at the end of the array.
  - $opt(0,0)$ is the optimal solution.
- We should determine the terminal condition:
  - If we have passed the end of sequence x, that is when $i = m$, we should insert $n - j$ gaps to make alignment.
    - $opt(m,j) = 2(n-j)$.

```
A T C _ _ _
A T C G T C
```

  - If we have passed the end of sequence y, that is when $j = n$, we should insert $m - i$ gaps to make alignment.
    - $opt(i,n) = 2(m-i)$.

```
A T C G T C
A T C _ _ _
```

Sequence Alignment by Dynamic Programming

- Again, we use the diagonal trick for the recursive property:

$$opt(i,j) = \min(opt(i+1,j+1) + cost, opt(i+1,j) + 2, opt(i,j+1) + 2)$$

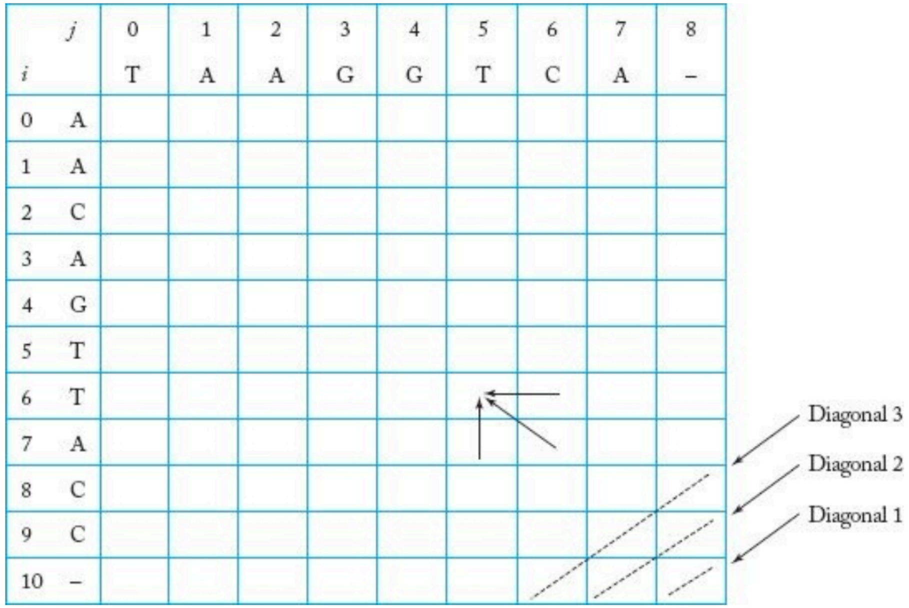


Image source: Figure 3.19, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

j	0	1	2	3	4	5	6	7	8
i	T	A	A	G	G	T	C	A	–
0 A	7	8	10	12	13	15	16	18	20
1 A	6	6	8	10	11	13	14	16	18
2 C	6	5	6	8	9	11	12	14	16
3 A	7	5	4	6	7	9	11	12	14
4 G	9	7	5	4	5	7	9	10	12
5 T	8	8	6	4	4	5	7	8	10
6 T	9	8	7	5	3	3	5	6	8
7 A	11	9	7	6	4	2	3	4	6
8 C	13	11	9	7	5	3	1	3	4
9 C	14	12	10	8	6	4	2	1	2
10 –	16	14	12	10	8	6	4	2	0

TA_AGGT_CA
AACAGTTACA
A_AGGT_CA
ACAGTTACA
_AGGT_CA
CAGTTACA
AGGT_CA
AGTTACA
GGT_CA
GTTACA
GT_CA
TTACA
T_CA
TACA
_CA
ACA
CA
CC
A
C

Image source: Figure 3.20, Richard E. Neapolitan, Foundations of Algorithms (5th Edition), Jones & Bartlett Learning, 2014

Conclusion

For an optimization problem, to determine the decomposition and the representation of array is the most difficult part for designing a dynamic programming algorithm.

- The binomial coefficient: calculate $B[n][k]$ from $B[i][j]$.
- The chained matrix multiplication: calculate $M[1][n]$ from $M[i][j]$.
- Optimal binary search tree: calculate $A[1][n]$ by $A[i][j]$.
- The knapsack problem: calculate $V(n, W)$ by $V(i, j)$.
- The shortest path problem: calculate $D[i][j]$ from $D^{(k)}[i][j]$.
- Sequence alignment: calculate $opt(0,0)$ by $opt(i, j)$.

Conclusion

After this lecture, you should know:

- The difference between divide-and-conquer and dynamic programming.
- Why is dynamic programming efficient.
- The condition to use dynamic programming.
- The steps of designing a dynamic programming algorithm.

Thank you!

- Any question?
- Don't hesitate to send email to me for asking questions and discussion. ☺

Acknowledgement: Thankfully acknowledge slide contents shared by Prof. Xuemin Hong