The Knapsack Problem

- Problem:

  $S = \{item_1, item_2, \ldots, item_n\}$,

  $w_i = item_i$의 무게

  $p_i = item_i$의 가치

  W = 배낭에 넣을 수 있는 최대 무게

  라고 할 때, $\sum_{item_i \in A} w_i \leq W$ 를 만족하면서

  $\sum_{item_i \in A} p_i$ 가 최대가 되도록

  $A \subseteq S$가 되는 A를 결정하는 문제이다.

The Fractional Knapsack Problem

- 물건의 일부분을 잘라서 담을 수 있는 경우

품목	무게	값	값어치
$item_1$	5 lb	\$50	\$10/lb
$item_2$	10 lb	\$60	\$6/lb
$item_3$	20 lb	\$140	\$7/lb

- 탐욕적인 접근방법으로 최적해를 구할 수 있다.
- 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다!!
- $item_1 + item_3 + (5/10) * item_2 = \$50 + \$140 + (5/10)*\$60$

  $\Rightarrow \$220 \quad (30\text{ lb})$
- Optimal!

The 0-1 Knapsack Problem

- 물건의 일부분을 잘라서 담을 수 없는 경우

- The 0-1 Knapsack Problem (1)

  - 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다?
  - 항상 최적의 해를 주지는 않는다!

  - 왜 아닌지 보기: W = 30 lb

품목	무게	값	값어치
$item_1$	5 lb	\$50	\$10/lb
$item_2$	10 lb	\$60	\$6/lb
$item_3$	20 lb	\$140	\$7/lb

    - 탐욕적인 방법: $item_1 + item_3 \Rightarrow$ 25 lb $\Rightarrow$ \$190
    - 최적인 해답: $item_2 + item_3 \Rightarrow$ 30 lb $\Rightarrow$ \$200

The 0-1 Knapsack Problem

- The 0-1 Knapsack Problem (2)
  - 가장 비싼 물건부터 우선적으로 채운다?
  - 이 방법도 항상 최적의 해를 주는 건 아니다!
  - 왜 아닌지 보기: W = 30 lb

품목	무게	값
$item_1$	25 lb	\$10
$item_2$	10 lb	\$9
$item_3$	10 lb	\$9

    - 탐욕적인 방법: $item_1 \Rightarrow$ 25 lb $\Rightarrow$ \$10
    - 최적인 해답: $item_2 + item_3 \Rightarrow$ 20 lb $\Rightarrow$ \$18

The 0-1 Knapsack Problem

- The 0-1 Knapsack Problem (3)

- 무작정 알고리즘
  - n개의 물건에 대해서 모든 부분 집합을 다 고려한다.
  - 크기가 n인 집합의 부분집합의 수는 2^n개이다!!
  - 따라서 적어도 $\Omega(2^n)$ 의 시간복잡도가 필요하다.

품목	무게	값
$item_1$	25 lb	\$10
$item_2$	10 lb	\$9
$item_3$	10 lb	\$9

The 0-1 Knapsack Problem

- Dynamic Programming Approach (0-1 Knapsack Problem)
  - $i > 0$ 이고 $w > 0$일 때, 전체 무게가 w가 넘지 않도록 i번째까지의 항목 중에서 얻어진 최고의 이익(optimal profit)을 $P[i][w]$라고 하면,

$$P[i][w] = \begin{cases} maximum(P[i-1][w], p_i + P[i-1][w-w_i]) & (\text{if } w_i \le w) \\ P[i-1][w] & (\text{if } w_i > w) \end{cases}$$

    여기서 $P[i\ \text{-}1][w]$는 i번째 항목을 포함시키지 않는 경우의 최고 이익이고, $p_i + P[i\text{-}1][w - w_i]$는 i번째 항목을 포함시키는 경우의 최고 이익이다. 위의 재귀 관계식이 최적화 원칙을 만족하는지는 쉽게 알 수 있다.

  - 그러면 어떻게 최대 이익 $P[n][W]$값을 구할 수 있을까?
    - int $P[0..n][0..W]$의 2차원 배열을 만든 후, 각 항을 계산하여 넣는다
    - 여기서 $P[0][w] = 0$, $P[i][0] = 0$으로 놓으면 되므로, 계산해야 할 항목의 수는 $nW \in \Theta(nW)$

The 0-1 Knapsack Problem

- Refinement of Dynamic Programming
  - 여기서 n과 W와는 아무런 상관관계가 없다.
    만일 (임의적으로) $W = n!$이라고 한다면, 수행시간은 $\Theta(n \times n!)$이 된다.
    그렇게 되면 이 알고리즘은 앞에서 얘기한 무작정 알고리즘보다도
    나을게 하나도 없다.

  - 그럼 이 알고리즘을 최악의 경우에 $\Theta(2^n)$ 시간에 수행될 수 있도록,
    즉 무작정 알고리즘 보다 느리지 않고, 때로는 훨씬 빠르게 수행될 수
    있도록 개량할 수 있을까?
    - 착안점은 $P[n][W]$를 계산하기 위해서 $(n\text{-}1)$번째 행을 모두
      계산할 필요가 없다는데 있다.

The 0-1 Knapsack Problem

- $P[n][W]$는 아래 식으로 표현할 수 있다

$$P[n][W] = \begin{cases} maximum(P[n-1][W], p_n + P[n-1][W - w_n]) & (\text{if } w_n \le W) \\ P[n-1][W] & (\text{if } w_n > W) \end{cases}$$

  - 따라서 (n-1)번째 행에서는 $P[n\text{-}1][W]$와 $P[n\text{-}1][W\text{-}w_n]$ 항만 필요
  - i-번째 행에 어떤 항목이 필요한지를 결정한 후에, 다시 (i-1)번째 행에 필요한 항목을 결정
    - $P[\mathrm{i}][\mathrm{w}]$는 $P[i\text{-}1][w]$와 $P[i\text{-}1][w\text{-}w_i]$로 계산
  - 이런 식으로 $n = 1$이나 $w \le 0$일 때까지 계속해 나가면 된다.

The 0-1 Knapsack Problem

- Ex 4.7
  - W=30 lb

품목	무게	값
$item_1$	5 lb	$50
$item_2$	10 lb	$60
$item_3$	20 lb	$140

  - We need P[3][W] = P[3][30]
    - To compute P[3][30] -- max(P[3-1][30], p_3 + P[3-1][30-w_3])
      = max(P[2][30], p_3 + P[2][10])
    - To compute P[2][30] -- max(P[2-1][30], p_2 + P[2-1][30-w_2])
      = max(P[1][30], p_2 + P[1][20])
    - To compute P[2][10] -- max(P[2-1][10], p_2 + P[2-1][10-w_2])
      = max(P[1][10], p_2 + P[1][0])

The 0-1 Knapsack Problem

- Compute row 1
  - $P[1][w] = \begin{cases} \max(P[0][w], \$50+P[0][w-5]) & (\text{if } w_1 = 5 \le w) \\ P[0][w] & (\text{if } w_1 = 5 > w) \end{cases}$

    $= \begin{cases} \$50 & (\text{if } w_1 = 5 \le w) \\ \$0 & (\text{if } w_1 = 5 > w) \end{cases}$
  - Therefore

    $P[1][0] = \$0$; $P[1][10] = \$50$; $P[1][20] = \$50$; $P[1][30] = \$50$

- Compute row 2
  - $P[2][10] = \begin{cases} \max(P[1][10], \$60+P[1][0]) & (\text{if } w_2 = 10 \le 10) \\ P[1][10] & (\text{if } w_2 = 10 > 10) \end{cases}$

    $= \$60$
  - $P[2][30] = \begin{cases} \max(P[1][30], \$60+P[1][20]) & (\text{if } w_2 = 10 \le 30) \\ P[1][30] & (\text{if } w_2 = 10 > 30) \end{cases}$

    $= \$60 + \$30 = \$110$

The 0-1 Knapsack Problem

- Compute row 3
  - $P[3][30] = \begin{cases} \max(P[2][30], \$140+P[2][10]) & (\text{if } w_3 = 20 \leq 30) \\ P[1][10] & (\text{if } w_3 = 20 > 30) \end{cases}$

    $= \$140 + \$60 = \$200$

- The modified algorithm – compute only 7 entries
- The original algorithm - compute $3 \times 30 = 90$ entries

The 0-1 Knapsack Problem

- Efficiency in the worst case
  - Compute at most 2^i entries in the $(n - i)$-th row
    - The total number is $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$.
    - 최악의 경우 수행시간 $\Theta(2^n)$
  - The number of entries computed is in $O(nW)$
    - What about the number of the modified algorithm ?
    - If $n = W+1$, and $w_i = 1$ for all i, then the total number of entries is about

      $1 + 2 + 3 + \ldots + n = n(n+1) / 2 = (W+1)(n+1) / 2$
    - For arbitrary large values of n and W, $\Theta(nW)$
  - Combining these 2 results, the worst case is in $O(min(2^n, nW))$

The 0-1 Knapsack Problem

- 아직 아무도 최악의 경우 수행시간이 지수(exponential)보다 나은 알고리즘을 발견하지 못했고,
- 아직 아무도 그러한 알고리즘은 없다라고 증명한 사람도 없다.

  -> **NP문제**

Chained Matrix Multiplication (연쇄 행렬곱셈)

- $i \times j$ 행렬과 $j \times k$행렬을 곱하기 위해서는 일반적으로 $i \times j \times k$번 만큼의 기본적인 곱셈이 필요하다.

- 연쇄적으로 행렬을 곱할 때, 어떤 행렬곱셈을 먼저 수행하느냐에 따라서 필요한 기본적인 곱셈의 횟수가 달라지게 된다.

- 예를 들어서, 다음 연쇄행렬곱셈을 생각해 보자:
  - $A_1 \times A_2 \times A_3$.
  - A_1: 10×100, A_2; 100×5, A_3: 5×50
  - $(A_1 \times A_2) \times A_3$: 기본적인 곱셈의 총 횟수는 **7,500**회
  - $A_1 \times (A_2 \times A_3)$: 기본적인 곱셈의 총 횟수는 **75,000**회
  - 따라서, 연쇄적으로 행렬을 곱할 때 기본적인 곱셈의 횟수가 가장 적게 되는 최적의 순서를 결정하는 알고리즘을 개발하는 것이 목표

Chained Matrix Multiplication

A	$\times$	B	$\times$	C	$\times$	D
20×2		2×30		30×12		12×8

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

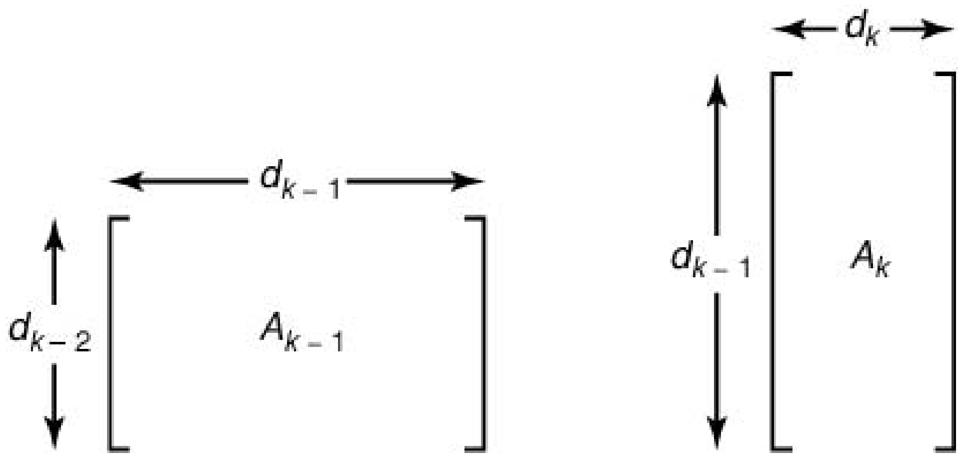
Chained Matrix Multiplication

- 무작정알고리즘: 가능한 모든 순서를 모두 고려해 보고, 그 가운데에서 가장 최소를 택한다.

- 시간복잡도 분석: 최소한 지수(exponential-time) 시간

- 증명:
  - n개의 행렬($A_1, A_2, \ldots, A_n$)을 곱할 수 있는 모든 순서의 가지 수를 t_n이라고 하자.
  - 만약 A_1이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_2, \ldots, A_n$을 곱하는 데는 t_{n-1}개의 가지수가 있을 것이다.
  - A_n이 마지막으로 곱하는 행렬이라고 하면, 행렬 $A_1, \ldots, A_{n-1}$을 곱하는 데는 또한 t_{n-1}개의 가지수가 있을 것이다.
  - 그러면, $t_n \geq t_{n-1} + t_{n-1} = 2\, t_{n-1}$이고 $t_2 = 1$이라는 사실은 쉽게 알 수 있다.
  - 따라서 $t_n \geq 2t_{n-1} \geq 2^2 t_{n-2} \geq \ldots \geq 2^{n-2} t_2 = 2^{n-2} = \Theta(2^n)$

Chained Matrix Multiplication



- d_k를 행렬 A_k의 열(column)의 수라고 하자
  - 자연히 A_k의 행(row)의 수는 d_{k-1}; A_1의 행의 수는 d_0라고 하자.
  - For $1 \leq i \leq j \leq n$, let

    M[i][j] = i < j일 때 A_i부터 A_j까지의 행렬을 곱하는데 필요한 곱셈의 최소 횟수

    $$= minimum_{i \leq k \leq j-1}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j)$$

    M[i][j] = 0 if i = j

Chained Matrix Multiplication

- Ex 3.5

$$\begin{array}{cccccc} A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\ 5\times 2 & 2\times 3 & 3\times 4 & 4\times 6 & 6\times 7 & 7\times 8 \end{array}$$

$$A_4(A_5A_6) \qquad\qquad (A_4A_5)A_6$$

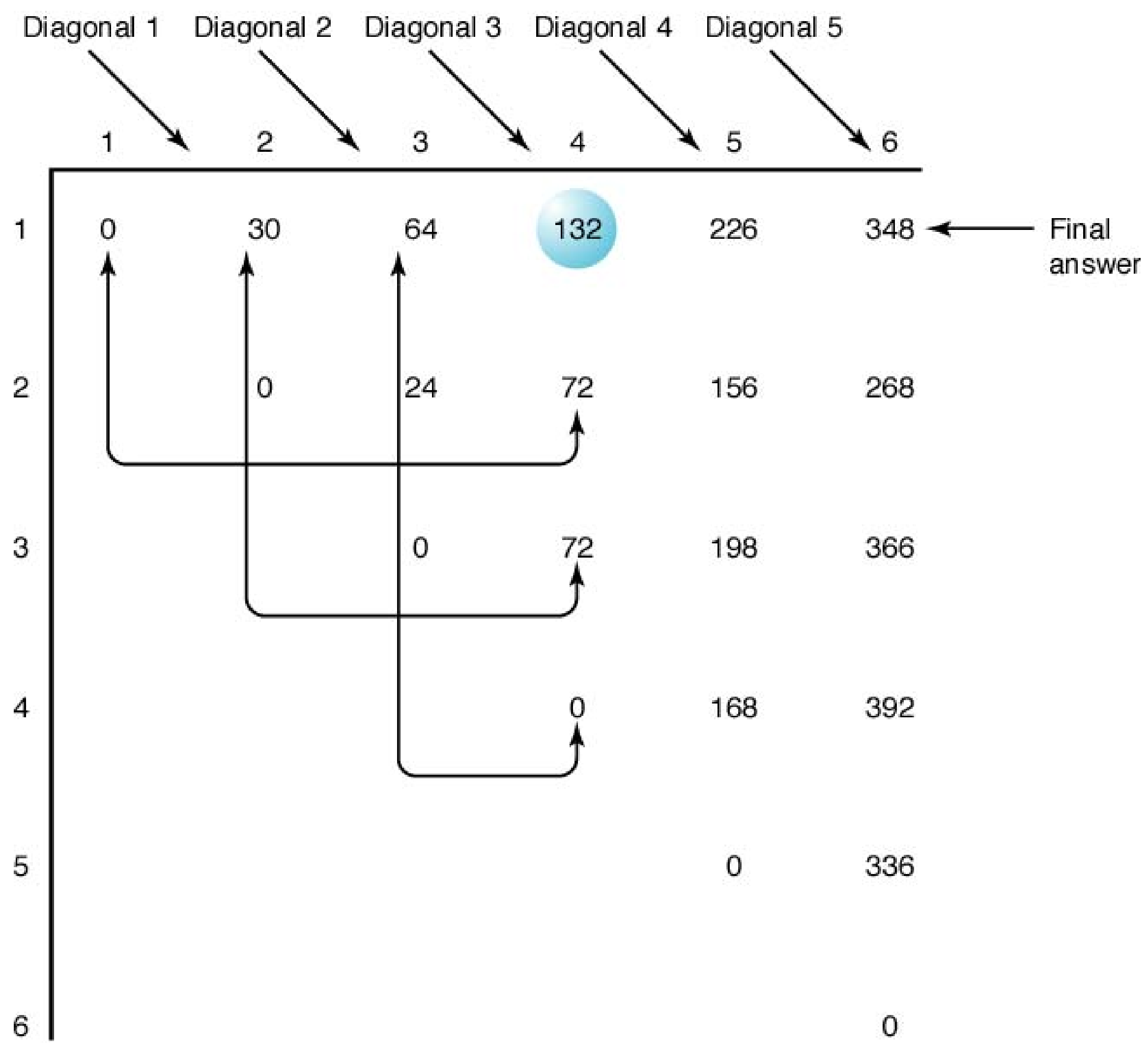
$$\begin{aligned} M[4][6] &= minimum(M[4][4]+M[5][6]+4\times 6\times 8, M[4][5]+M[6][6]+4\times 7\times 8) \\ &= minimum(0+6\times 7\times 8+4\times 6\times 8, 4\times 6\times 7+0+4\times 7\times 8) \\ &= minimum(528,392)=392 \end{aligned}$$

$M[i][j]$	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Chained Matrix Multiplication

- Ex 3.6
  - For diagonal 0: M[i][i] = 0 for $1 \leq i \leq 6$
  - For diagonal 1: $M[1][2] = \min(M[1][k] + M[k+1][2] + d_0 d_k d_2)$
    $= M[1][1] + M[2][2] + d_0 d_1 d_2 = 30$
    Compute M[2][3], M[3][4], M[4][5], M[5][6]
  - For diagonal 2: $M[1][3] = \min(M[1][k] + M[k+1][3] + d_0 d_k d_3)$
    $= \min(M[1][1] + M[2][3] + d_0 d_1 d_3,$
    $M[1][2] + M[3][3] + d_0 d_2 d_3)$
    $= \min(0+24+5\text{x}2\text{x}4, 30+0+5\text{x}3\text{x}4) = 64$
    Compute M[2][4], M[3][5], M[4][6]
  - For diagonal 3: $M[1][4] = \min(M[1][k] + M[k+1][4] + d_0 d_k d_4)$
    $= \min(M[1][1] + M[2][4] + d_0 d_1 d_4,$
    $M[1][2] + M[3][4] + d_0 d_2 d_4,$
    $M[1][3] + M[4][4] + d_0 d_3 d_4)$
    $= \min(0+72+5\text{x}2\text{x}6, 30+72+5\text{x}3\text{x}6, 64+0+5\text{x}4\text{x}6) = 132$
    Compute M[2][5], M[3][6]

Chained Matrix Multiplication



Chained Matrix Multiplication

- 최소곱셈(Minimum Multiplication) 알고리즘
  - 문제
    - n개의 행렬을 곱하는데 필요한 기본적인 곱셈의 횟수의 최소치를 결정하고, 그 최소치를 구하는 순서를 결정하라.
  - 입력
    - 행렬의 수 n와 배열 $d[0..n]$,
      $d[i\text{-}1] \times d[i]$는 i번째 행렬의 규모를 나타낸다.
  - 출력
    - 기본적인 곱셈의 횟수의 최소치를 나타내는 *minmult*;
      최적의 순서를 얻을 수 있는 배열 P,
      여기서 $P[i][j]$는 행렬 i부터 j까지가 최적의 순서로 갈라지는 기점

Chained Matrix Multiplication

- 알고리즘:

```
int minmult(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] = minimum(M[i][k]+M[k+1][j]+d[i-1]*d[k]*d[j]);
                      i <= k <= j-1
            P[i][j] = 최소치를 주는 k의 값
        }
    return M[1][n];
}
```

Chained Matrix Multiplication

- 최소곱셈 알고리즘의 모든 경우 분석
  - 단위연산: 각 k값에 대하여 실행된 명령문 (instruction), 여기서 최소값인 지를 알아보는 비교문도 포함한다.
  - 입력크기: 곱할 행렬의 수 n
  - 분석: $j = i + diagonal$이므로,
    - i-루프를 수행하는 횟수 = $n - diagonal$
    - k-루프를 수행하는 횟수 =

$$(j-1)-i+1=((i+diagonal)-1)-i+1=diagonal$$

    - 따라서

$$\sum_{diagonal=1}^{n-1}[(n-diagonal)\times diagonal]=\frac{n(n-1)(n+1)}{6}\in\Theta(n^3)$$

Chained Matrix Multiplication

- 최적 순서의 구축
  - 최적 순서를 얻기 위해서는 $M[i][j]$를 계산할 때 최소값을 주는 k값을 $P[i][j]$에 기억한다.
  - 예: $P[2][5] = 4$인 경우의 최적 순서는 $(A_2 A_3 A_4) A_5$이다.

$P[i][j]$

	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

- $P[1][6] = 1;\quad A_1(A_2 A_3 A_4 A_5 A_6)$
- $P[2][6] =5;\quad A_1((A_2 A_3 A_4 A_5) A_6)$
- 따라서 최적 분해는 $(A_1((((A_2 A_3) A_4) A_5) A_6))$.

Chained Matrix Multiplication

- 최적의 해를 주는 순서의 출력
  - 문제: n개의 행렬을 곱하는 최적의 순서를 출력하시오
    - 입력: n과 P
    - 출력: 최적의 순서
  - 알고리즘:

```
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 << “)”;
    }
}
```

Chained Matrix Multiplication

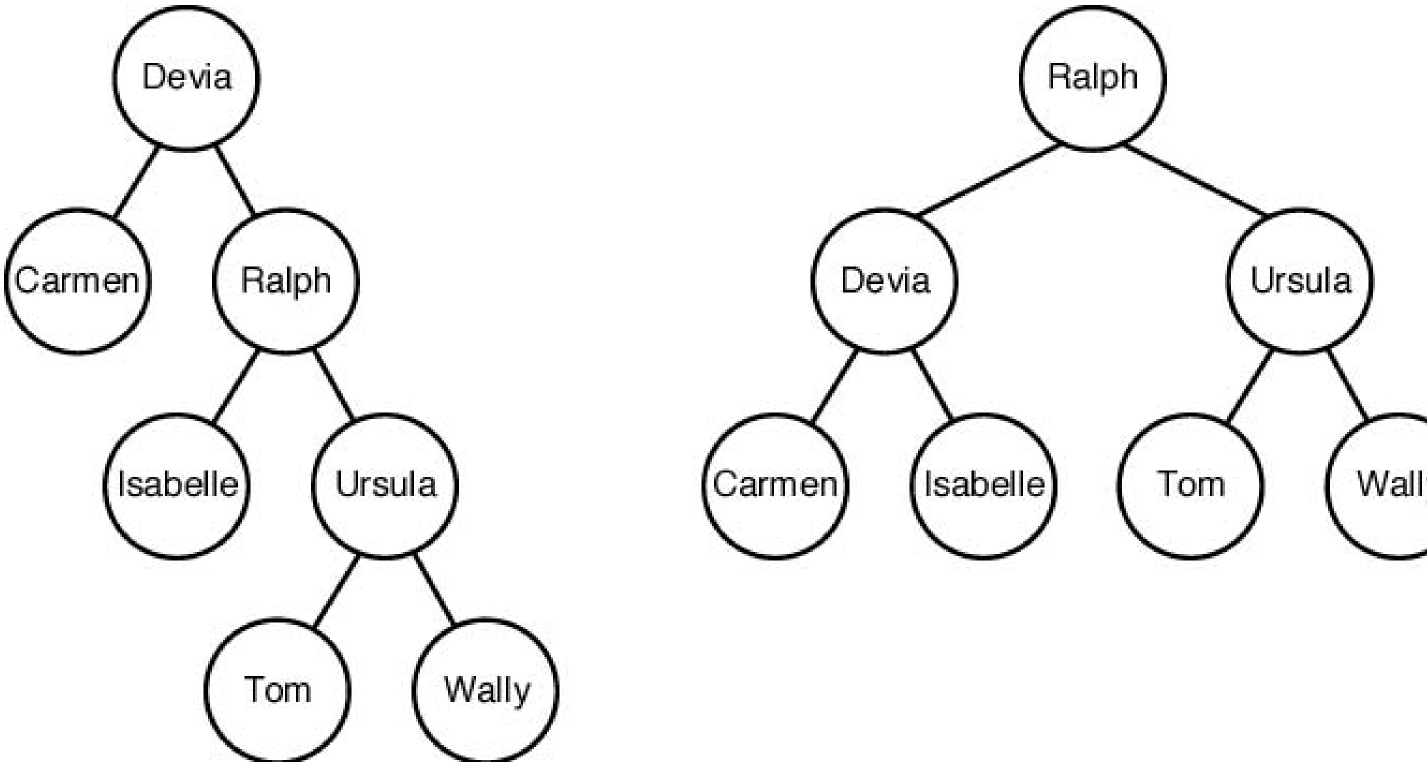
- 최적의 해를 주는 순서의 출력
  - order(i,j)의 의미: $A_i \times \ldots \times A_j$ 의 계산을 수행하는데 기본적인 곱셈의 수가 가장 적게 드는 순서대로 괄호를 쳐서 출력하시오.
  - 분석: $T(n) \in \Theta(n)$. 어떻게?

- Chained matrix multiplication
  - $\Theta(n^3)$ – Godbole (1973)
  - $\Theta(n^2)$ – Yao (1982)
  - $\Theta(n \lg n)$ – Hu and Shing (1982, 1984)

Optimal Binary Search Trees

- Definition
  - *binary search tree*
    - A binary tree of items (keys), that come from an ordered set
    - Each node contain one key
    - The keys in the left subtree of a given node are less than or equal to the key in that tree
    - The keys in the right subtree of a given node are greater than or equal to the key in that tree
  - *depth* – number of edges from the root (level of the node)
  - *balanced* – if the depth of the 2 subtrees of every node never differ by more than 1
  - *optimal* –the average time it takes to locate a key is minimized

Optimal Binary Search Trees



Two binary search trees

Optimal Binary Search Trees

- Data types

```
Struct nodetype {
      keytype key
      nodetype* left
      nodetype* right
}
Typedef nodetype* node_pointer;
```

Optimal Binary Search Trees

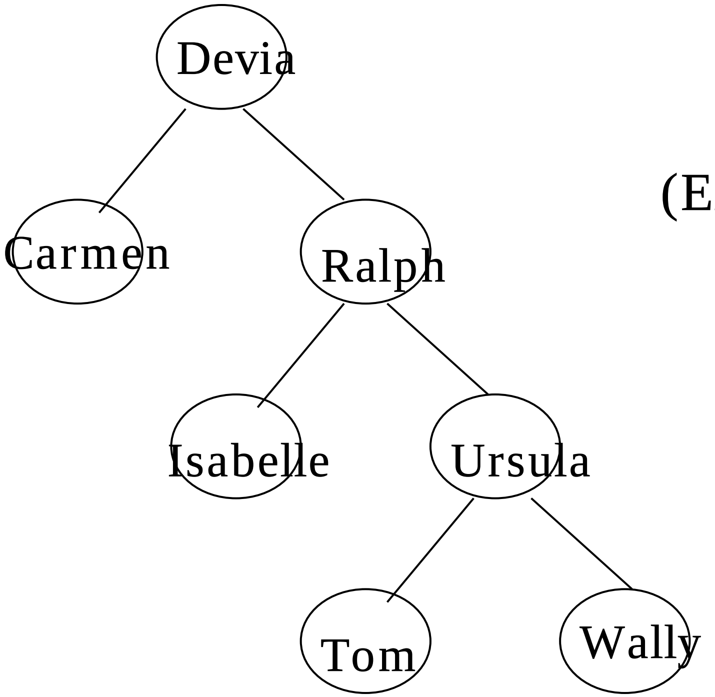
- Problem: determine the node containing a key in a binary search tree (assume that key is in the tree)
  - Inputs: a pointer tree & a key keyin
  - Outputs: a pointer p to the node containing the key

```
void search (node_pointer tree, keytype keyin, node_pointer& p) {
      bool found;
      p = tree;
      found = false;
      while (!found)
            if (p->key == keyin)
                  found = true;
            else if (keyin < p->key)
                  p = p->left;                // Advance to left child
            else
                  p = p->right;               // Advance to right child
}
```

Optimal Binary Search Trees

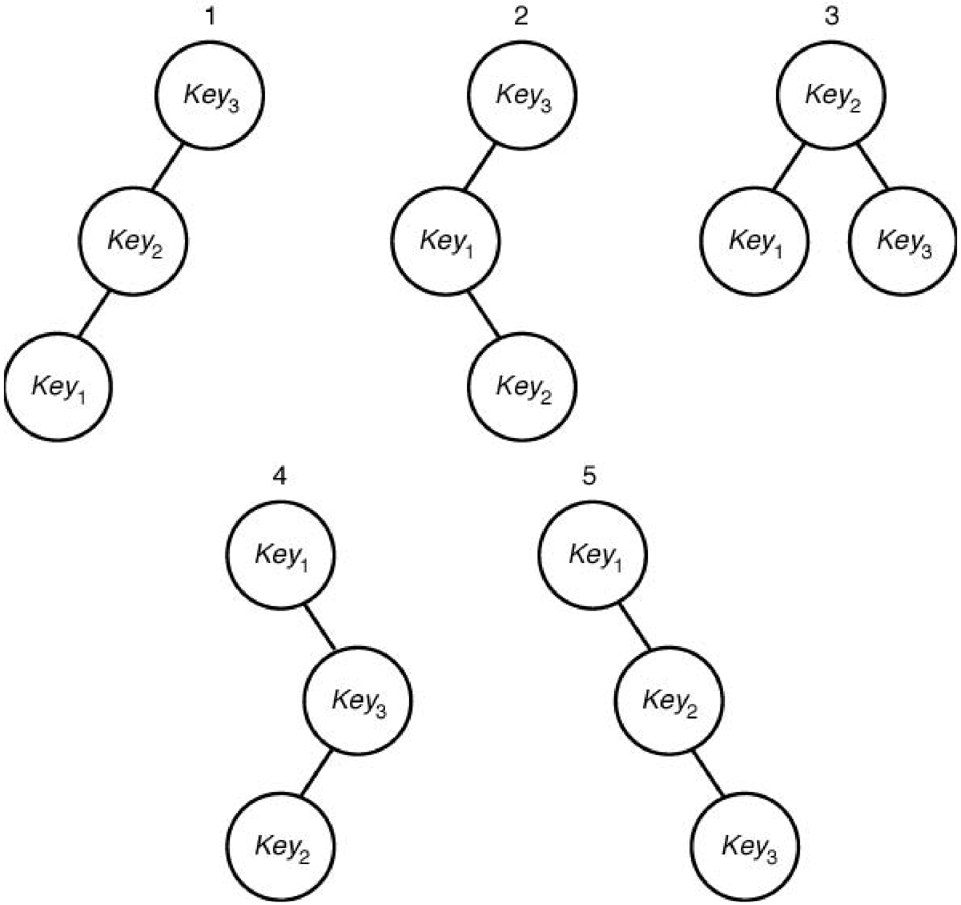
- Analysis
  - Search time – the number of comparisons to locate a key
    - Search time a given key   depth(key) + 1

      where depth(key) is the depth of the node containing the key



(Ex) Search time for "Ursula"
depth(Ursula) + 1 = 2 + 1 = 3

Optimal Binary Search Trees



p_i – the probability that Key_i is the search key

$p_1 = 0.7$, $p_2 = 0.2$, $p_3 = 0.1$

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) $2(0.7) + 1(0.2) + 2(0.1) = 1.8$

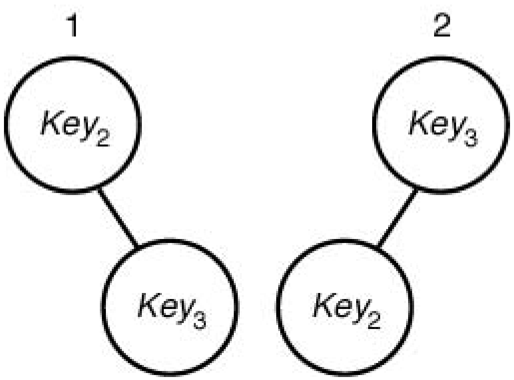
4) $1(0.7) + 3(0.2) + 2(0.1) = 1.5$

5) $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

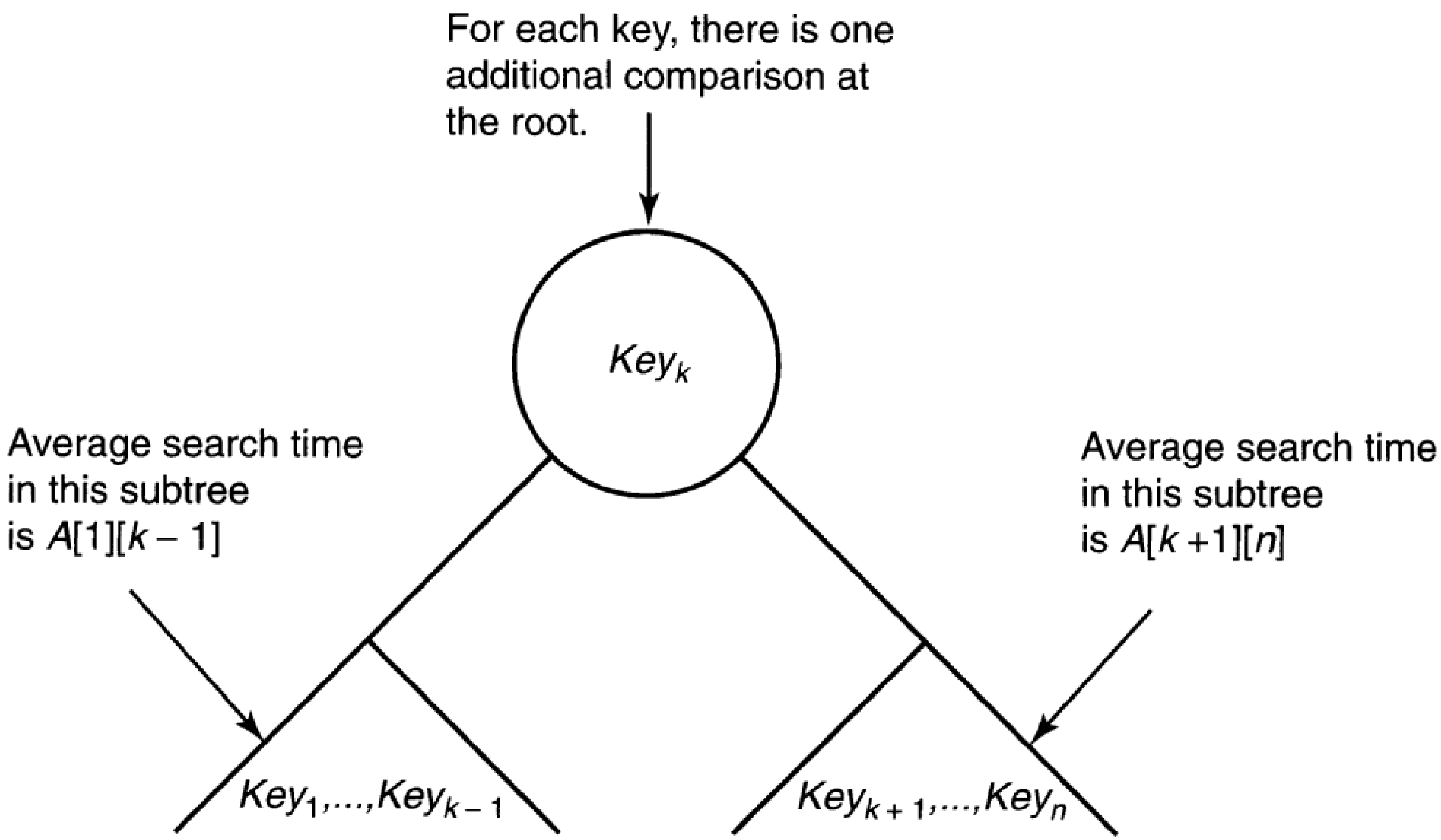
Optimal Binary Search Trees

- Dynamic programming
  - Suppose that Key_i through Key_j are arranged in a tree that minimizes $$\sum_{m=i}^{j} c_m p_m$$
    - C_m – the number of comparisons needed to locate Key_m in the tree
    - p_m – the probability that Key_m is the search key
  - $A[i][j]$ – the optimal value that minimized the tree
    - $A[i][i] = p_i$

Ex 3.8) Determine $A[2][3]$



Optimal Binary Search Trees



Average time for tree k, $A[1][n]$ =

$$A[1][k-1] + p_1 + \ldots + p_{k-1} \quad + \quad p_k \quad + \quad A[k+1][n] + p_{k+1} + \ldots + p_n$$

Average time in left subtree	Additional time comparing at root	Average time searching at root	Average time in right subtree	Additional time comparing at root

Optimal Binary Search Trees

$A[1][k-1] + p_1 + \ldots + p_{k-1} + p_k + A[k+1][n] + p_{k+1} + \ldots + p_n$

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

$$A[1][n] = \underset{1 \le k \le n}{\text{minimun}}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

$$A[i][j] = \underset{i \le k \le j}{\text{minimun}}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad (i < j)$$

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

$$A[i][i-1] = 0 \quad \text{and} \quad A[j+1][j] = 0$$

Optimal Binary Search Trees

- Optimal Binary Search Tree Algorithm

```
void optsearchtree(int n, const float p[], float& minavg, index R[][]) {
    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]; R[i][i] = i; R[i][i-1] = 0;
    }
    A[n+1][n] = 0; R[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]) + ∑_{m=i}^{j} p_m
            R[i][j] = a value of k that gave the minimum;
        }
    minavg = A[1][n];
}
```

$$A[i][j] = \min(A[i][k-1]+A[k+1][j]) + \sum_{m=i}^{j} p_m$$

Optimal Binary Search Trees

- Every-case Time Complexity Analysis
  - Basic operation: addition & comparison for each value of k
  - Input size: n, the number of keys
  - 분석: $j = i + diagonal$이므로, (Algorithm 3.6 참조)
    - i-루프를 수행하는 횟수 = $n - diagonal$
    - k-루프를 수행하는 횟수 = j-i+1= (i+diagonal)-i+1=diagonal+1
    - 따라서

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times (diagonal + 1)] = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

Optimal Binary Search Trees

- Build Binary Search Tree Algorithm

Output: a pointer tree to an optimal binary search tree containing the *n* keys

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

Optimal Binary Search Trees

- Example 3.9
  - Don Isabelle Ralph Wally

    Key[1] Key[2] Key[3] Key[4]

    $p_1=3/8$ $p_2=3/8$ $p_3=1/8$ $p_4=1/8$

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

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

R