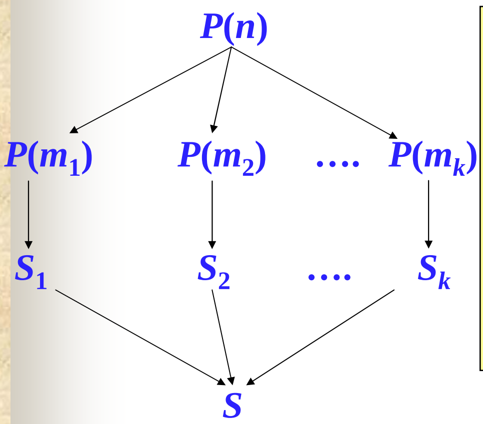
Unit 5
Dynamic Programming

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 15.

Dynamic Programming

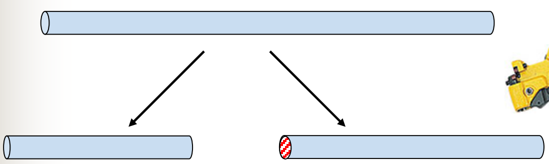
- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find ***a*** solution with ***the*** optimal value.
  - Minimization or maximization. (We'll see both.)

Dynamic Programming



- 與 divide-and-conquer 法類似, 是依遞迴方式設計的演算法.
- 與 divide-and-conquer 的最大差別在於子問題間不是獨立的, 而是重疊的.

Rod Cutting (定義)



- How to cut a steel rod of length n into pieces in order to maximize the revenue r_n?
- Each cut is free. Rod lengths are always an integer. Input: A length n and table of prices

長度 i	1	2	3	4	5	6	7	8	9	10
價格 p_i	1	5	8	9	10	17	17	20	24	30

An Example of Rod Cutting

長度 i	1	2	3	4	5	6	7	8	9	10
價格 p_i	1	5	8	9	10	17	17	20	24	30

- Consider the case of $n = 4$.
-

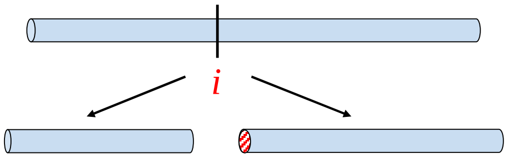
# pieces	possible ways of cutting	revenue	
1	4	9	
2	1 + 3	9	
2	2 + 2	10	max
3	1 + 1 + 2	7	
4	1 + 1 + 1 + 1	4	

Rod Cutting (觀察)

1 2 ··· $n-1$

- There are $n-1$ positions to cut the rod. Hence, there are ______ different ways to cut up the rod.
- We can use a binary string $c_1c_2 \ldots c_{n-1}$ to represent a possible way of cutting up the rod where $c_i = 1$ means there is a cut at position i; otherwise there is no cut at the position.

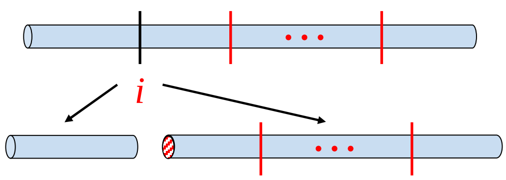
Rod Cutting (觀察 $\Rightarrow$ a recursive formula)



- If $c_1c_2\ldots c_{n-1}$ is an optimal way of cutting up the rod and assume $c_i = 1$,
- then $c_1c_2\ldots c_{i-1}$ and $c_{i+1}c_{i+2}\ldots c_{n-1}$ are respectively optimal ways of cutting up the left and right parts of the rod, i.e. $r_n = r_i + r_{n-i}$.
- It is possible that $i = 1, 2, \ldots, n-1$ or no cuts. We have:

$$\boxed{\begin{array}{l} r_n = \max\{p_n, r_1+r_{n-1}, r_2+r_{n-2}, \ldots, r_{n-1}+r_1\} \\ r_1 = p_1 \end{array}}$$

A Simpler Recursive Formula



- If $c_1c_2\ldots c_{n-1}$ is an optimal way of cutting up the rod and assume i is the least index s.t. $c_i = 1$,
- then $c_{i+1}c_{i+2}\ldots c_{n-1}$ is a optimal way of cutting up the remaining part of the rod, i.e. $r_n = p_i + r_{n-i}$.
- It is possible that $i = 1, 2, \ldots, n$ ($i = n \Rightarrow$ no cuts) We have:

$$\boxed{\begin{array}{l} r_n = \max_{1\le i\le n}\{ p_i + r_{n-i}\} \\ r_0 = 0 \end{array}}$$

It can be simplified further.

A Formal Description of the Problem

- The rod cutting problem is equivalent to the following problem:
  Given n numbers p_1, p_2, … , p_n, find a way to partition $n = i_1 + i_2 + \cdots + i_k$, such that the sum $p_{i_1} + p_{i_2} + \cdots + p_{i_k}$ is maximized.
- Let t_n denote the number of ways to partition the integer n. It can be shown that $t_n = o(2^n)$. (See the footnote in p.361.)
- However, t_n is still of super-polynomial order.

Recursive Top-Down Implementation

```
Cut-Rod(n)  // p[ ] is global
  if n == 0
      return 0
  q = –∞
  for i = 1 to n
      q = max (q, p[i] + Cut-Rod(n–i))
  return  q
```

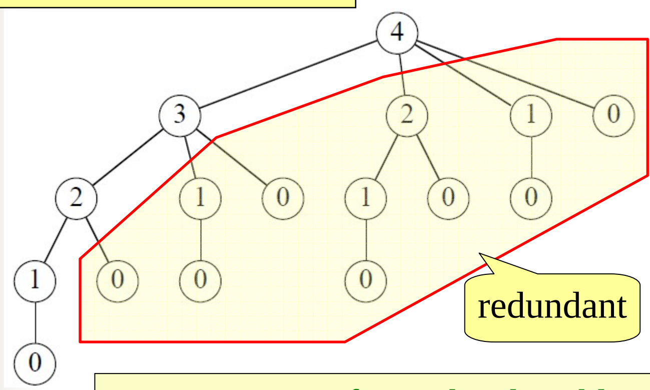
$r_n = \max_{1 \le i \le n}\{ p_i + r_{n-i}\}$
$r_0 = 0$

- Time complexity $T(n)$: # calls
  $T(n) = 1 + \Sigma_{0 \le j \le n-1} T(j)$, $T(0) = 1$.
- $T(n) = \Theta(\_\_)$

See Unit 2 p.23

The Recursion Tree of Cut-Rod(n)

$r_n = \max_{1 \le i \le n}\{ p_i + r_{n-i} \}$



How to arrange for each subproblem to be solved only once? Ans.______________.

Top-Down with Memoization

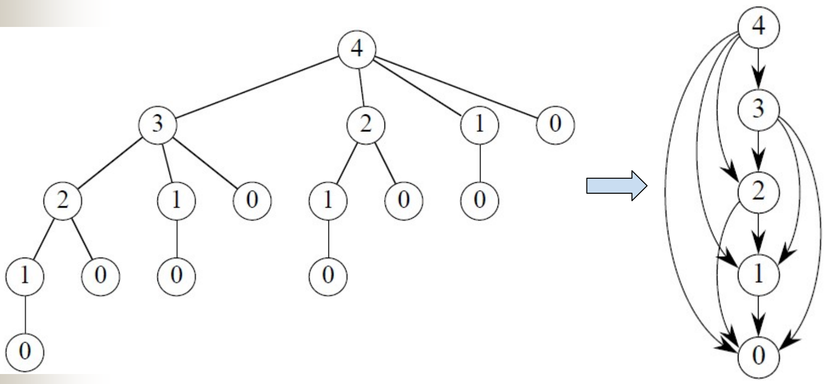
In Main( ), set $r[0] = 0$, $r[1] = r[2] = \cdots = r[n] = -\infty$ and call M_Cut-Rod(n).

$r_n = \max_{1 \le i \le n}\{ p_i + r_{n-i} \}$

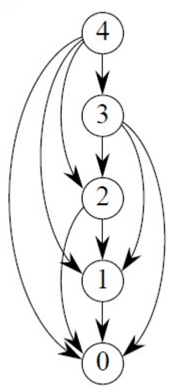
$r_0 = 0$

```
M_Cut-Rod(n)
if r[n] ≥ 0  return r[n]
q = –∞
for i = 1 to n
    q = max (q, p[i] + M_Cut-Rod(n–i))
r[n] = q
return  q
```

Subproblem graphs



An Example & Time Complexity



r[ ]
10
8
5
1
0

i	1	2	3	4
p[ ]	1	5	8	9

$r_n = \max_{1 \le i \le n}\{ p_i + r_{n-i}\}$

$r_0 = 0$

Time complexity

$T(n)$: # calls = __________.

$\therefore T(n) = \Theta($__$)$

Bottom-Up Method

```
Bottom_Up_Cut-Rod(n)  // Input: p[ ], Aux: r[ ]
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max (q, p[i] + r[j–i])
    r[j] = q
return  r[n]
```

$r_j = \max_{1 \le i \le j}\{ p_i + r_{j-i}\}$

$r_0 = 0$

Time: $\Theta(n^2)$

Space: $\Theta(n)$

Reconstructing a Solution

$r_j = \max_{1 \le i \le j}\{ p_i + r_{j-i}\}$

$s_j = \arg\max_{1 \le i \le j}\{ p_i + r_{j-i}\}$

i	1	2	3	4	5	6	7	8	9	10
p[ ]	1	5	8	9	10	17	17	20	24	30
r[ ]	1	5	8	10	13	17	18	22	25	30
s[ ]	1	2	3	2	2	6	1	2	3	10

$r[9] = p[3] + r[6] = p[3] + p[6]$

Matrix-Chain Multiplication (定義)

Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

例： $A_1 \times A_2 \times A_3 \times A_4$

p_i :13 5 89 3 34

There are 5 ways to fully parenthesize the product :

$(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3)A_4)$.

Matrix-Chain Multiplication (例)

$(A_1(A_2(A_3A_4))) \to A_1 \times (A_2A_3A_4) \to A_2 \times (A_3A_4) \to A_3 \times A_4$

cost = 13*5*34 + 5*89*34 + 89*3*34
= 2210 + 15130 + 9078
= 26418

$A_1 \times A_2 \times A_3 \times A_4$

13 5 89 3 34

$(A_1(A_2(A_3A_4)))$, costs = 26418

$(A_1((A_2A_3)A_4))$, costs = 4055

$((A_1A_2)(A_3A_4))$, costs = 54201

min $((A_1(A_2A_3))A_4)$, costs = 2856

$(((A_1A_2)A_3)A_4)$, costs = 10582

Catalan Number

For any n, # ways to fully parenthesize the product of a chain of n+1 matrices

= # binary trees with n nodes.

= # permutations generated from 1 2 … n through a stack.

= # n pairs of fully matched parentheses.

= n-th ***Catalan Number*** = $C(2n, n)/(n+1) = \Omega(4^n/n^{3/2})$

Catalan Number (例)

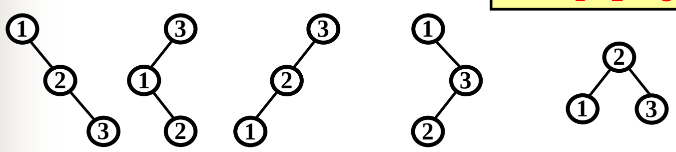
For example, when n = 3,

$C(2n, n)/(n+1) = C(6, 3)/4 = 20/4 = 5$

1	2	3
3	2	1
1	3	2
2	1	3
2	3	1

( ) ( ) ( )
((( )))
( ) (( ))
(( )) ( )
(( ) ( ))

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$



Multiplication Trees

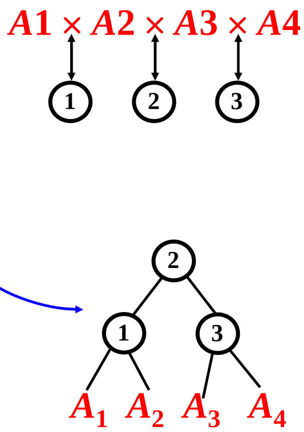
$(A_1(A_2(A_3A_4)))$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

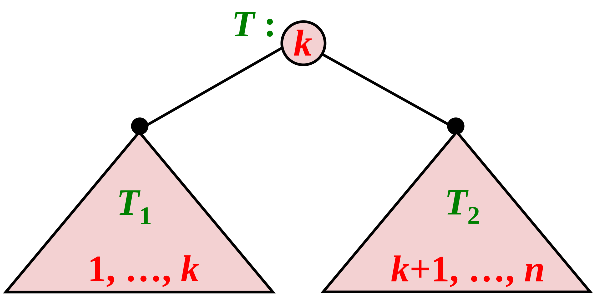
$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$



Matrix-Chain Multiplication (觀察)

☛ If T is an optimal solution for $A_1, A_2, \ldots, A_n$



☛ then, T_1 (resp. T_2) is an optimal solution for $A_1, A_2, \ldots, A_k$ (resp. $A_{k+1}, A_{k+2}, \ldots, A_n$).

Matrix-Chain Multiplication (設計)

- Let $m[i, j]$ be the minmum number of scalar multiplications needed to compute the product $A_i \ldots A_j$, for $1 \le i \le j \le n$.
- If the optimal solution splits the product $A_i \ldots A_j = (A_i \ldots A_k) \times (A_{k+1} \ldots A_j)$, for some k, $i \le k < j$, then $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j$, we have :

$$m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}$$
$$= 0 \text{ if } i = j$$

Matrix-Chain Multiplication (實例)

- Consider an example with sequence of dimensions <5,2,3,4,6,7,8>

$$m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}$$

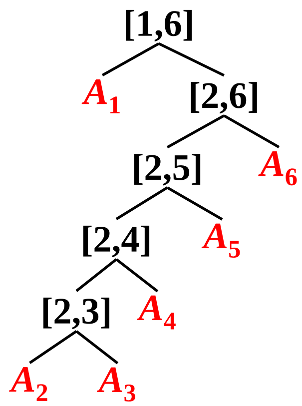
	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Matrix-Chain Multiplication (找解)

$m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}$
$s[i, j]$ = a value of k that gives the minimum

s	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

$\boxed{A_1((((A_2A_3)A_4)A_5)A_6)}$



Matrix-Chain Multiplication (分析)

$m[i, j] = \min_{i \le k < j}\{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}$

- To fill the entry $m[i, j]$, it needs $\Theta(j-i)$ operations. Hence the execution time of the algorithm is

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)=\sum_{j=1}^{n}\sum_{i=1}^{j}(j-i)=\sum_{j=1}^{n}\left[j^2-\frac{j(j+1)}{2}\right]$$
$$=\sum_{j=1}^{n}\Theta(j^2)=\Theta(n^3)$$

Time: $\Theta(n^3)$
Space: $\Theta(n^2)$

Steps for Developing a DP Algorithm

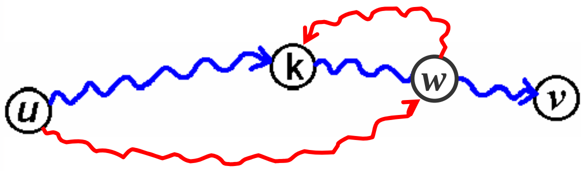
- ***Characterize*** the structure of an optimal solution.)
- ***Derive*** a ***recursive formula*** for computing the values of optimal solutions.
- ***Compute the value*** of an optimal solution typically in a bottom-up fashion (top-down is also applicable).
- ***Construct*** an optimal solution from computed information in a top-down fashion.

Elements of Dynamic Programming

- **Optimal substructure** (a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems)
- **Overlapping subproblems**
- **Memorization**
- **Reconstructing an Optimal Solution**

Subtleties of Optimal Substructure

- Consider the following problems on unweighted di-graphs, and examine if they exhibit optimal substructures:
  - ○ • Find a shortest path between two given nodes.
  - ✗ • Find a longest simple path between two given nodes.



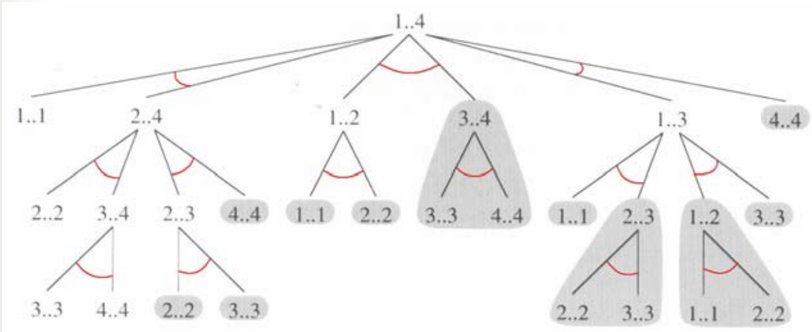
Overlapping Subproblems

- When we solve a problem by DP strategy, we usually derive a recursive formula after characterizing the structure of optimal solutions.
- However, if you implement a DP algorithm by a recursive program, the program will revisit some subproblems over and over again.
- A standard DP implementation ***solves each sub-problem once and stores the solution in a table*** where it can be looked up when needed.

Overlapping Subproblems
(例:Matrix-Chain Multiplication)

- The recursion tree for computing

$$m[i, j] = \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}$$



Overlapping Subproblems & Memorization

- Hence, in general, ***don't implement a DP algorithm by a recursive program***,
- unless you use a memory mechanism to memorize those subproblems that have been solved during the execution.
- The efficiency of a DP algorithm depends heavily on the total number of distinct subproblems; e.g. rod cutting has **Θ(__)** subproblems & matrix-chain multiplication has **Θ(__)** subproblems.

Reconstructing an Optimal Solution

- As a practical matter, we often store which choice we made in each subproblem in a table, and then using this information we can construct an optimal solution efficiently (usually in linear-time.)
- For instance, arrays $s[j]$ and $s[i, j]$ are used to store such information in the previous 2 examples.

Longest Common Subsequence (定義)

Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ find a maximum-length common subsequence of X and Y.

例1：Input: **ABCBDAB** **BDCABA**

C.S.'s: **AB**, **ABA**, **BCB**, **BCAB**, **BCBA** …

Longest: **BCAB**, **BCBA**, … Length = 4

A B C B D A B

B D C A B A

例 2 :

`vintner`

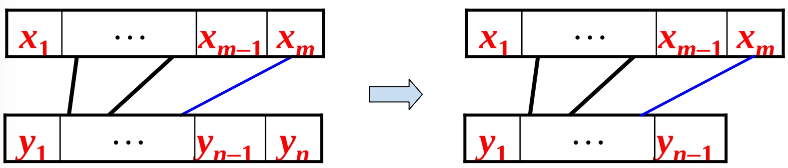
`writers`

Longest Common Subsequence (觀察)

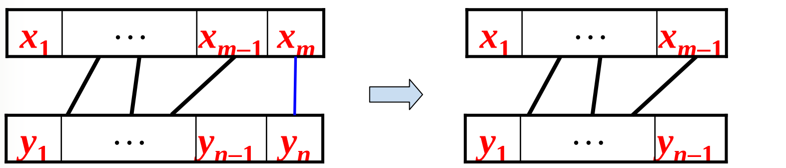
- Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be a LCS of $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$.
- If $z_k \neq x_m$, then Z is a LCS of $\langle x_1, x_2, \dots, x_{m-1} \rangle$ and Y.
- If $z_k \neq y_n$, then Z is a LCS of X and $\langle y_1, y_2, \dots, y_{n-1} \rangle$.
- If $z_k = x_m = y_n$, then $\langle z_1, z_2, \dots, z_{k-1} \rangle$ is a LCS of $\langle x_1, x_2, \dots, x_{m-1} \rangle$ and $\langle y_1, y_2, \dots, y_{n-1} \rangle$.
- Hence, LCS exhibits optimal substructure.

Longest Common Subsequence (觀察)

- Case 1-1: $x_m \neq y_n$, and y_n is not matched.



- Case 2-1: $x_m = y_n$, and they form a match.



Longest Common Subsequence (設計)

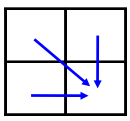
- Let $c[i, j]$ be the length of an LCS of the prefixes $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$ for $1 \le i \le m$ and $1 \le j \le n$. We have :

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \ne y_j \end{cases}$$

Longest Common Subsequence (例+分析)

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \ne y_j \end{cases}$$

c[ ]	A	B	C	B	D	A	B
B	0	1	1	1	1	1	1
D	0	1	1	1	2	2	2
C	0	1	2	2	2	2	2
A	1	1	2	2	2	3	3
B	1	2	2	3	3	3	4
A	1	2	2	3	3	4	4

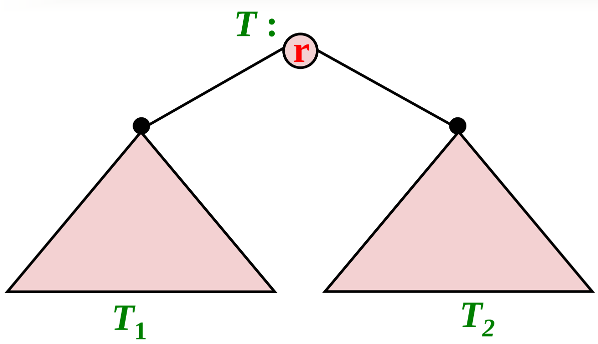


Time: $\Theta(mn)$

Space: $\Theta(mn)$

可得LCS's: **BCBA**, **BDAB**, & **BCAB**.

Binary Search Trees

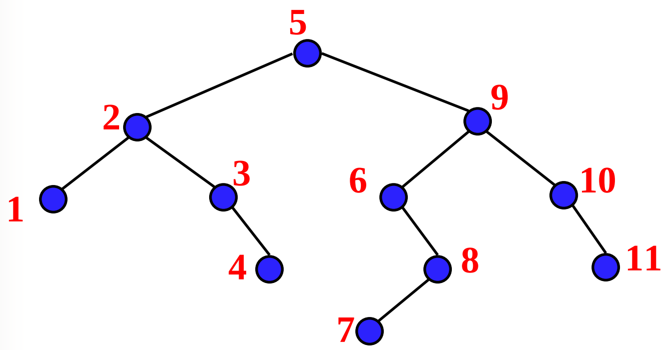


1. T_1 and T_2 are binary search trees
2. **key(T_1) < key(r) & key(T_2) > key(r)**

Assumption : No two nodes may have equal keys.

An Example of Binary Search Trees

Assume that the key sequence: $k_1 < k_2 < \ldots < k_{11}$

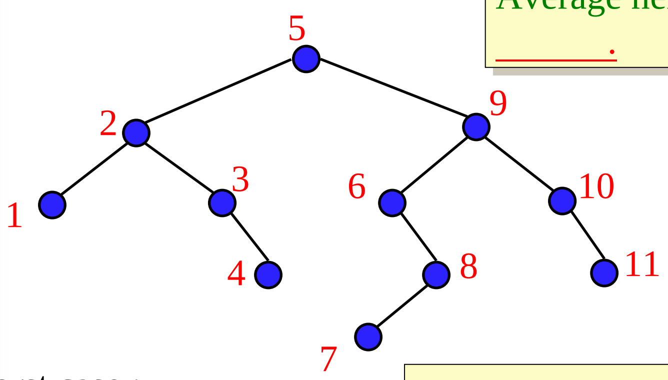


Searching time = O(the ______ of the tree)

Construct Binary Trees Thru Insertions

Insertion sequence of nodes : 5, 2, 1, 9, 6, 3, 4 , 8 , 10 , 11 , 7

Average height: ______.

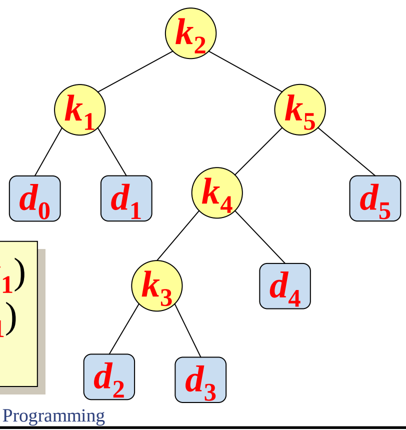


A worst case : ____________________.

How to construct a balanced BS tree?

Optimal Binary Search Trees (定義1)

- Given n keys $K = \langle k_1, k_2, \ldots, k_n \rangle$ in sorted order, we wish to build a binary search tree T from K.
- Let p_i be the probability that k_i is the search key.
- Some searches may be for values not in K, so we also have ***dummy keys*** $d_0, d_1, \ldots, d_n$ representing values not in K.



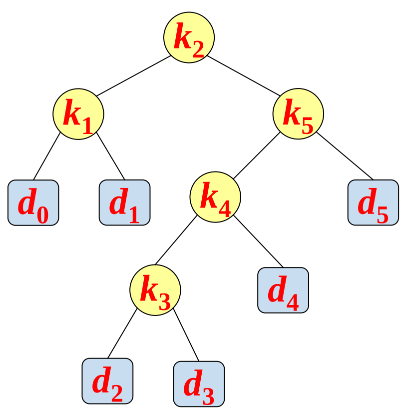
d_0: $(-\infty, k_1)$
d_i : (k_i, k_{i+1})
d_n: (k_n, ∞)

Optimal Binary Search Trees (定義2)

- Let q_i be the probability that a search will be for values in the range corresponding to d_i.
- We have $\Sigma_{1\le i\le n}\, p_i + \Sigma_{0\le i\le n}\, q_i = 1$.
- E[search cost in T]
  $= \Sigma_{1\le i\le n}(\text{depth}_T(k_i)+1) \cdot p_i + \Sigma_{0\le i\le n}(\text{depth}_T(d_i)+1) \cdot q_i$
  $= 1 + \Sigma_{1\le i\le n}\text{depth}_T(k_i) \cdot p_i + \Sigma_{0\le i\le n}\text{depth}_T(d_i) \cdot q_i$
- The goal is to construct a binary search tree whose expected search cost is smallest. Such a tree is called an ***optimal binary search tree***.

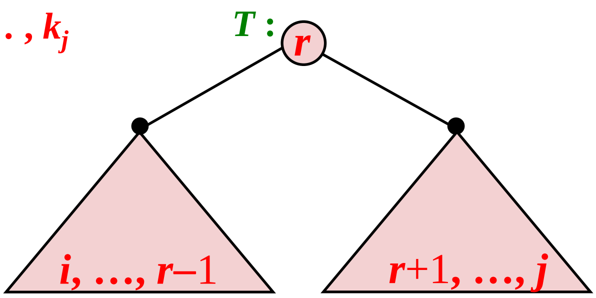
Optimal Binary Search Trees (例子)

node	depth	prob.	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75



Optimal Binary Search Trees (觀察)

- Let $e[i, j]$ = the value of optimal solution for k_i $k_{i+1}, \ldots, k_j$



$$e[i, j] = \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\}$$
$$= q_{i-1} \text{ if } j = i-1$$

$$w(i, j) = \Sigma_{i \le l \le j}\, p_l + \Sigma_{i-1 \le l \le j}\, q_l$$

Optimal Binary Search Trees (實例)

$$e[i, j] = \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\}$$
$$= q_{i-1} \text{ if } j = i-1$$

e[ ]	0	1	2	3	4	5
1	5	45	90	125	175	275
2		10	40	70	120	200
3			5	25	60	130
4				5	30	90
5					5	50
6						10

node	freq.
k_1	15
k_2	10
k_3	5
k_4	10
k_5	20
d_0	5
d_1	10
d_2	5
d_3	5
d_4	5
d_5	10

Optimal Binary Search Trees (找解)

$e[i, j] = \min_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\}$

$R[i, j]$ = a value of r that gives the minimum

R	1	2	3	4	5
1	1	1	2	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Time = $O(n^3)$

Space = $O(n^2)$

$e[1,5]$

$e[1,1]$ $e[3,5]$

$e[3,4]$

$e[3,3]$

Notes

- Once we have developed a straightforward DP algorithm, we will often find that we can improve on the time or space it uses.
- In the literature, the time complexities of matrix-chain multiplication and optimal binary search tree have been improved from $O(n^3)$ to $O(n^2)$, and then to $O(n \log n)$.
- The space complexity of LCS can be reduced from $O(mn)$ to $O(m+n)$ and still keep enough information to retrace a solution.