Chap 15 – Dynamic Programming

15.1 Rod cutting

15.2 Matrix-chain multiplication

15.3 Elements of dynamic programming

15.4 Longest common subsequence

15.5 Optimal binary search trees

Prelude

- Dynamic programming
  - Developed back in the day when "programming" meant "tabular method" (like linear programming, Chap 29)
  - Used for optimization problems
    - Find *a* solution with *the* optimal value
    - Minimization or maximization
  - Four-step method
    1. Characterize the structure of an optimal solution
    2. Recursively define the value of an optimal solution
    3. Compute the value of an optimal solution, typically in a bottom-up fashion
    4. Construct an optimal solution from computed information

15.1 Rod cutting

- Rod cutting
  - Given a rod of length n and prices $p_i, i = 1,2, \ldots, n$

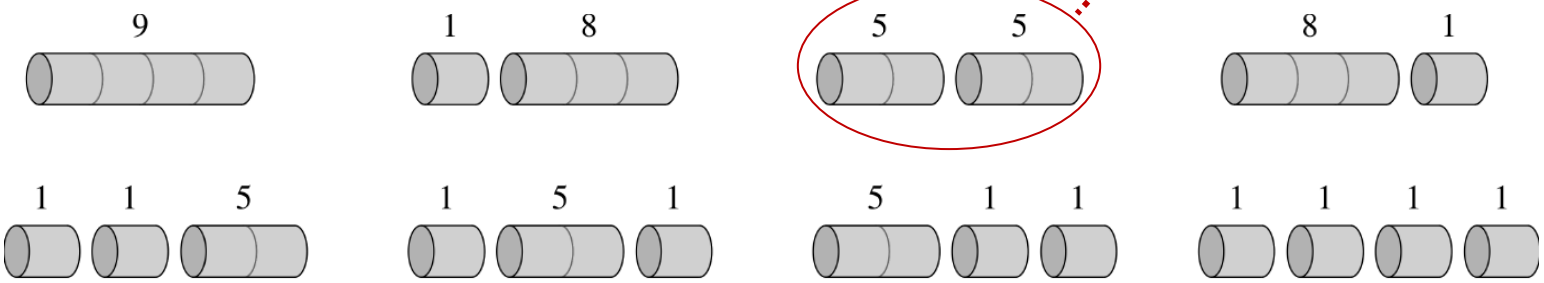
    Note: p_i = the price of a rod of length i

    How to cut the rod into pieces that have maximum prices?
  - Example

    $n = 4$

Length i	1	2	3	4
Price p_i	1	5	8	9

There are 8 ways to cut a rod of length 4: maximum



15.1 Rod cutting

- Brute-force approach
  - A rod of length n can be cut at length $1,2,\ldots,$ or $n-1$. Thus, there are 2^{n-1} different ways to cut the rod. This is equivalent to the number of ***compositions*** of the integer n, i.e. ***order-dependent*** ways to express n as a sum of positive integers, e.g.

$$\begin{aligned} 4 &= 4 \\ &= 2+2 \\ &= 1+3=3+1 \\ &= 1+1+2=1+2+1=2+1+1 \\ &= 1+1+1+1 \end{aligned}$$

15.1 Rod cutting

- Brute-force approach
  - The complexity is still exponential, even if we consider only the number of ***partitions*** of n, i.e. ***order-independent*** ways to express n as a sum of positive integers.

    Let $p(n,k) =$ # of partitions of n using only integers $\geq k$

    Then, $p(n,k) = 0, \text{if } n < k$

    $p(n,k) = 1, \text{if } n = k$

    $p(n,k) = p(n,k+1) + p(n-k,k), \text{otherwise}$

    Let the ***partition function*** $p(n) =$ # of partitions of n

    Then, $p(n) = p(n,1)$

    It is known that $p(n) = \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{2n/3}}$ as $n \to \infty$

15.1 Rod cutting

- Step 1 – Optimal substructure

  An optimal solution to a problem contains within it an optimal solution to subproblems.

  Suppose optimal solution x is obtained by cutting at length i

  Then, $x = y + z$

  where y and z are optimal solutions to the subproblems of cutting the rods of length i and $n - i$, respectively.

  Cut-and-paste proof

  If not, let y' be a better solution to cutting the rod of length i

  Then, $x' = y' + z$ is better than x. A contradiction.

15.1 Rod cutting

- Step 2 – Recursive solution: overlapping subproblem

  Let r_i be the maximum revenue for a rod of length i

  Then,

  $r_1 = p_1$

  $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1), n > 1$

  A simpler recurrence

  Since any optimal solution has a leftmost cut, we may let the cut at length i be the leftmost cut. Then,

  $r_0 = 0$

  $r_n = \max_{1 \le i \le n} (p_i + r_{n-i}), n \ge 1$

  Observe that there is only one subproblem left.

15.1 Rod cutting

- Step 2 – Recursive solution: overlapping subproblem

  Recursive top-down solution

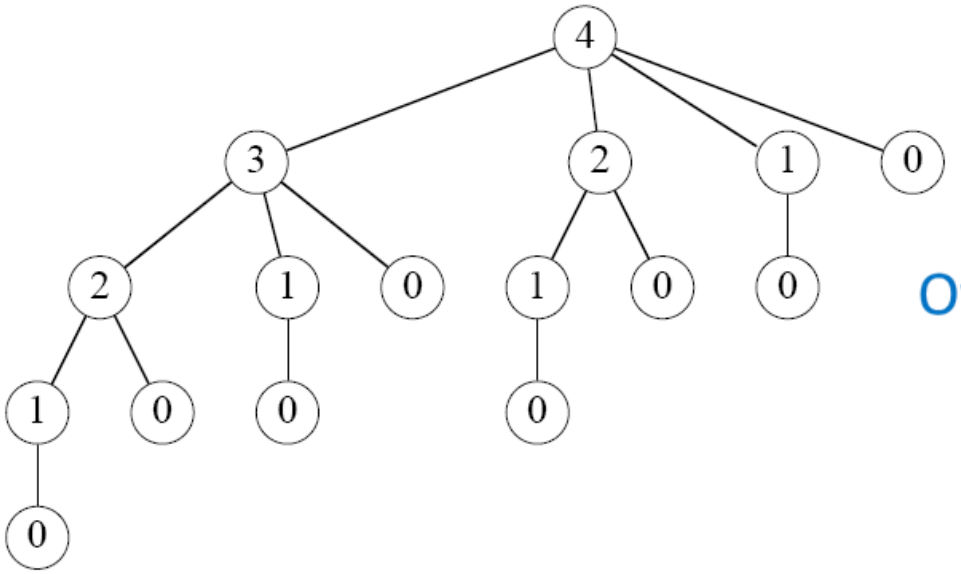
  A direct Implementation of the simpler recurrence for r_n

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

15.1 Rod cutting

- Step 2 – Recursive solution: overlapping subproblem

  Recursion tree for $n = 4$



Overlapping subproblems

$n = 0$ 8 times

$n = 1$ 4 times

$n = 2$ 2 times

Observe that there are only $n + 1$ distinct subproblems.

15.1 Rod cutting

- Step 2 – Recursive solution: overlapping subproblem

  Let $T(n) =$ # of times CUT-ROD is called for a rod of length n

  Then,

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \end{cases}$$

  Thus, $T(n)$ is a full-history recurrence.

  Eliminating the history, we have

$$T(n) - T(n-1) = T(n-1)$$
$$\Rightarrow T(n) = 2T(n-1)$$
$$\Rightarrow T(n) = 2^n$$

15.1 Rod cutting

- Step 2 – Recursive solution: memoization

MEMOIZED-CUT-ROD(p, n)
let $r[0..n]$ be a new array
for $i = 0$ **to** n **do** $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
if $r[n] == -\infty$
  if $n == 0$ **then** $r[n] = 0$
  else for $i = 1$ **to** n
    $r[n] = \max(r[n], p[i] +$
      MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
return $r[n]$

15.1 Rod cutting

- Step 2 – Recursive solution: memoization

  This is a $\Theta(n^2)$ algorithm – memoization turns $\Theta(2^n)$ to $\Theta(n^2)$

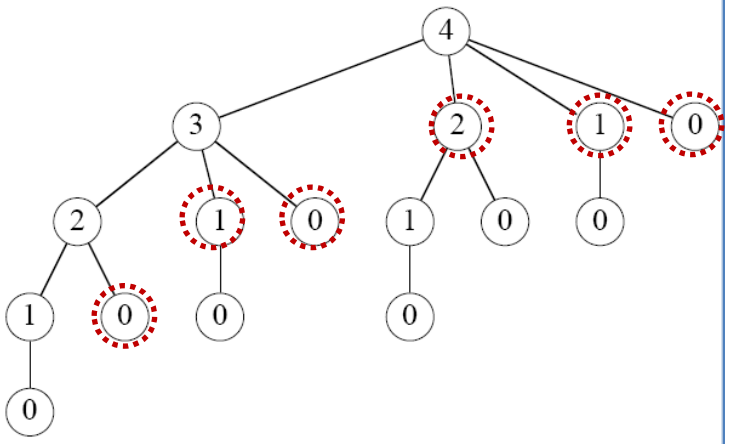
  For $O(n^2)$, observe that there are two types of calls:

  Type 1: Calls in which $r[n] = -\infty$

  $\Theta(n)$ such calls;

  each takes $O(n)$ time and

  makes $O(n)$ type 1 or type 2 calls

  Type 2: Calls in which $r[n] \neq -\infty$

  Hence, at most $O(n^2)$ type-2 calls; each takes $O(1)$ time.

  In total, $\Theta(n) \times O(n) + O(n^2) \times O(1) = O(n^2)$

15.1 Rod cutting

- Step 2 – Recursive solution: memoization

  For $\Omega(n^2)$, observe that

  running time $\geq$ time spent for type-1 calls

$$= 1 + \sum_{i=1}^{n} \Omega(i) = \Omega(n^2) \quad \because \text{add 1 for } n = 0$$

  The computation of the memoized recursive algorithm is top-down, whereas the table $r[n]$ is filled up in bottom-up order. In dynamical programming, we also make the computation bottom-up, as shown in Step 3.

15.1 Rod cutting

- Step 3 – Bottom-up tabulation

BOTTOM-UP-CUT-ROD(p, n)
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ **to** n
    $q = -\infty$
    for $i = 1$ **to** j
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$

Clearly, it takes a time in $\sum_{j=1}^{n} \Theta(j) = \Theta\left(\sum_{j=1}^{n} j\right) = \Theta(n^2)$

15.1 Rod cutting

- Step 4 – Constructing an optimal solution

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

let $r[0..n]$ and $s[0..n]$ be new arrays

$r[0] = 0$

for $j = 1$ **to** n

  $q = -\infty$

  for $i = 1$ **to** j

    if $q < p[i] + r[j - i]$

      $q = p[i] + r[j - i]$

      $s[j] = i$

  $r[j] = q$

return r and s

15.1 Rod cutting

- Step 4 – Constructing an optimal solution

PRINT-CUT-ROD-SOLUTION(p, n)

$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

while $n > 0$

print $s[n]$

$n = n - s[n]$

Example

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
s[i]	0	1	2	3	2	2	6	1	2

Running time: $O(n)$ and $\Omega(1)$

Worst-case running time: $\Theta(n)$

15.2 Matrix-chain multiplication

- Matrix-chain multiplication problem
  - Compute $A_1 A_2 \cdots A_n$, where A_i is a $p_{i-1} \times p_i$ matrix with the minimal number of scalar multiplications
  - Let A and B be $m \times n$ and $n \times p$ matrices, respectively. The classic matrix multiplication algorithm takes $m \times n \times p$ scalar multiplications to compute AB.

    N.B. Strassen's algorithm is for square matrices.
  - Example

    $A: 2 \times 3, B: 3 \times 4, C: 4 \times 5$

    $(AB)C \quad 2 \times 3 \times 4 + 2 \times 4 \times 5 = 64$

    $A(BC) \quad 2 \times 3 \times 5 + 3 \times 4 \times 5 = 90$

15.2 Matrix-chain multiplication

- Brute-force approach

  Enumerate all the possible parenthesizations

  Let $p(n) =$ the # of ways to parenthesize $A_1 A_2 \cdots A_n$, then

$$p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if } n > 1 \end{cases}$$

  By Problem 12-4, $p(n) = c_{n-1}$, where the n^{th} Catalan number

$$c_n = \frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{\sqrt{\pi} n^{3/2}}\left(1 + O\left(\frac{1}{n}\right)\right) = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

  Or, by Exercise 15.2-3

$$p(n) = \Omega(2^n)$$

15.2 Matrix-chain multiplication

- Step 1 – Optimal substructure

  If the optimal solution x for $A_1 A_2 \cdots A_n$ splits it at A_k

  $$(A_1 A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_n)$$

  Then, $x = y + z$

  where y and z are optimal solutions to the chains of k matrices and $n - k$ matrices, respectively.

  Cut-and-paste proof

  If not, let y' be a better solution to the chain of k matrices

  Then, $x' = y' + z$ is better than x.

  A contradiction.

  The same is true for z.

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

  Let $m[i,j]$ = the minimal # of scalar multiplications needed to compute $A_i A_{i+1} \cdots A_j$. Then, by optimal substructure,

  $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) & \text{if } i < j \end{cases}$$

  Wanted: $m[1,n]$

  There are totally $\binom{n}{2} + n = n(n+1)/2$ distinct subproblems

  But, many of them are overlapped in the recursive solution.

  $A_i \cdots A_k \mid A_{k+1} A_{k+2} \cdots A_j$

  $A_i \cdots A_k A_{k+1} \mid A_{k+2} \cdots A_j$

  $A_i \cdots A_k A_{k+1} A_{k+2} \mid A_{k+3} \cdots A_j$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

RECURSIVE-MATRIX-CHAIN(p, i, j)

if $i == j$ **then** $m[i, j] = 0$

else

$m[i, j] = \infty$

for $k = i$ **to** $j - 1$

$q =$ RECURSIVE-MATRIX-CHAIN(p, i, k)

$+$ RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$

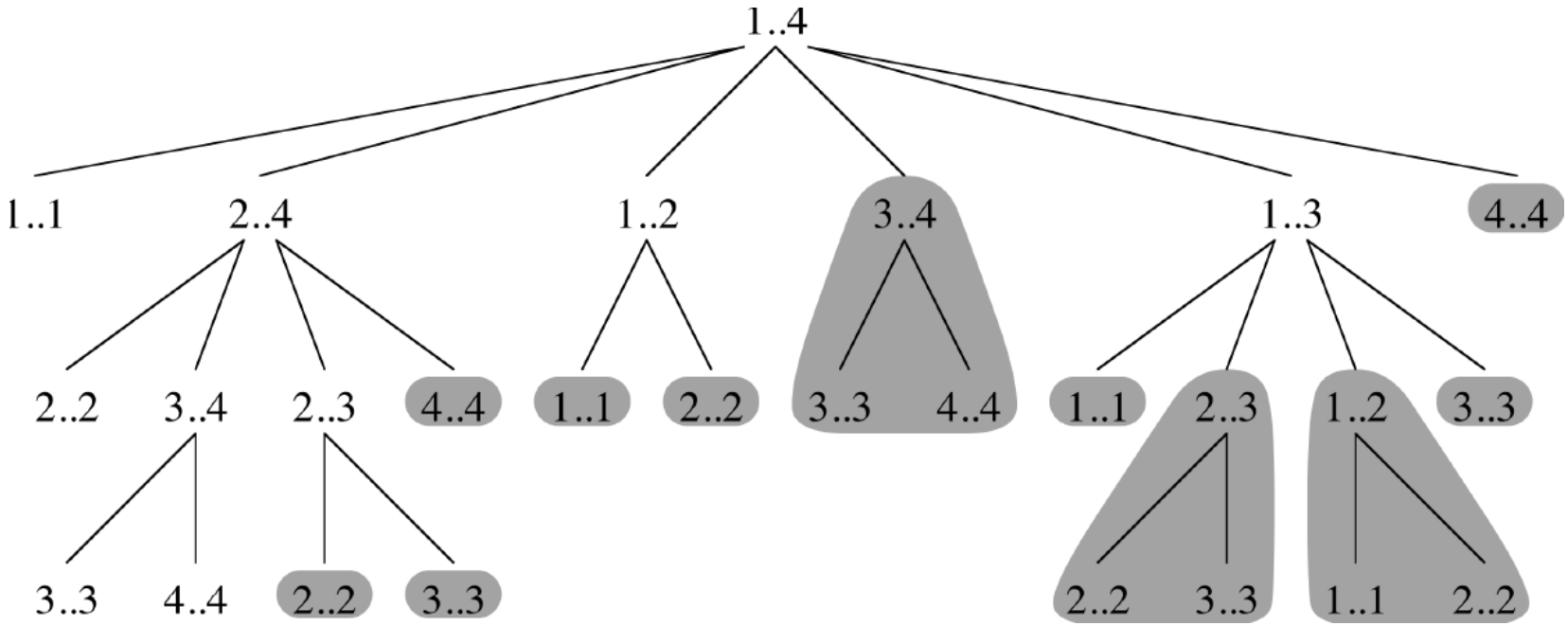
$+ p_{i-1} p_k p_j$

if $q < m[i, j]$ **then** $m[i, j] = q$

return $m[i, j]$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem



Only $\binom{4}{2} + 4 = 10$ distinct problems; but 27 computed.

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

  Let $T(n)$ = the time taken by RECURSIVE-MATRIX-CHAIN on n matrices

  Book's analysis

  $T(1) \geq 1$ $\quad \because$ at least one unit of time

  $$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$$

  $$= 2 \sum_{k=1}^{n-1} T(k) + n \qquad \text{for } n > 1$$

  Book then shows $T(n) \geq 2^{n-1}$ for $n \geq 1$ by usual induction (rather than constructive induction, as is claimed).

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

  Basis: $n = 1$

  $T(1) \geq 1 = 2^0$

  Induction step

  For $n \geq 2$

$$T(n) \geq 2\sum_{k=1}^{n-1} T(k) + n \geq 2\sum_{k=1}^{n-1} 2^{k-1} + n$$
$$= 2(2^{n-1} - 1) + n$$
$$= 2^{n-1} + (2^{n-1} + n - 2)$$
$$\geq 2^{n-1} \qquad \because 2^{n-1} + n - 2 \geq 0 \quad \forall n \geq 2$$

  It follows that $T(n) = \Omega(2^n)$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

  Alternative analysis

$$T(n) = \Theta(1) + \sum_{k=1}^{n-1} \left(T(k) + T(n-k) + \Theta(1)\right)$$

$$= 2\sum_{k=1}^{n-1} T(k) + \Theta(n)$$

  For lower bound, rewrite it as

$$T(n) \geq 2\sum_{k=1}^{n-1} T(k) + cn$$

  We may then show $T(n) = \Omega(2^n)$ by constructive induction.

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: overlapping subproblem

  Indeed, $T(n) = \Theta(3^n)$

  Since $T(n)$ is a full-history recurrence, by the elimination of History, we have

  $T(n) - T(n-1) = 2T(n-1) + \Theta(1)$

  Thus,

  $T(n) = 3T(n-1) + \Theta(1)$

  It is easily shown by substitution method that $T(n) = \Theta(3^n)$

  Hint

  Prove that $T(n) \leq a3^n - b$ and $T(n) \geq c3^n - d$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: memoization

MEMOIZED-MATRIX-CHAIN(p) // $n = p.length - 1$

let $m[i,j] = \infty, 1 \le i \le n, 1 \le j \le n$

return LOOKUP-CHAIN$(m, p, 1, n)$

LOOKUP-CHAIN(m, p, i, j)

if $m[i,j] = \infty$

  if $i == j$ **then** $m[i,j] = 0$

  else for $k = i$ to $j - 1$ $+p_{i-1}p_kp_j$

    $q =$ LOOKUP-CHAIN(m, p, i, k) + LOOKUP-CHAIN$(m, p, k+1, j)$

    if $q < m[i,j]$ **then** $m[i,j] = q$

return $m[i,j]$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: memoization

  This is a $\Theta(n^3)$ algorithm – memoization turns $\Theta(3^n)$ to $\Theta(n^3)$

  For $O(n^3)$, observe that there are two types of calls:

  Type 1: Calls in which $m[i,j] = \infty$

  $\Theta(n^2)$ such calls.

  Each takes $O(n)$ time and makes $O(n)$ type 1 or type 2 calls.

  Type 2: Calls in which $m[i,j] \neq \infty$

  Hence, at most $O(n^3)$ type-2 calls, each taking O(1) time.

  In total,

  $\Theta(n^2) \times O(n) + O(n^3) \times O(1) = O(n^3)$

15.2 Matrix-chain multiplication

- Step 2 – Recursive solution: memoization

  For $\Omega(n^3)$, observe that

  running time $\geq$ time spent for type-1 calls

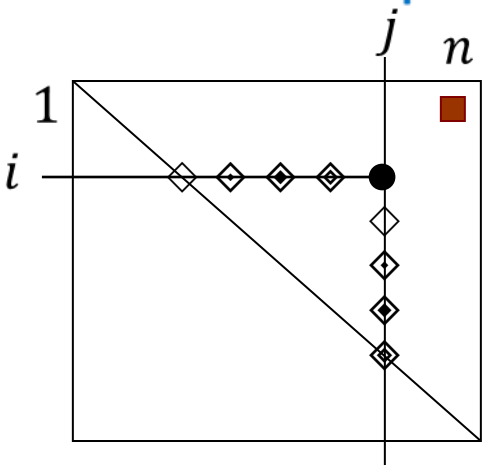
$$= \sum_{i=1}^{n} \left(\Omega(1) + \sum_{j=i+1}^{n} \Omega\big(2(j-i)\big) \right) \quad \because \ \Omega(1) \text{ for } i = j$$

$$= \Omega(n) + \sum_{i=1}^{n} \sum_{k=1}^{n-i} \Omega(2k) = \Omega(n^3)$$

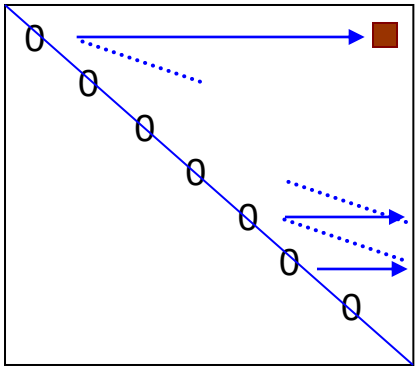
  Again, the computation of the memoized recursive algorithm is top-down, whereas the table $m[i, j]$ is filled up in bottom-up order.

15.2 Matrix-chain multiplication

- Step 3 – Bottom-up tabulation

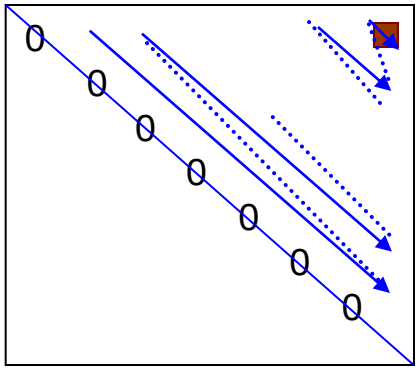


diagonal-by-diagonal



row-by-row

$d = j - i = 0\ \ 1\ \ 2\ \ \cdots\ n-1$



$\text{Space} = \Theta(n^2)$

$$\text{Time} = \Theta(n) + \sum_{d=1}^{n-1} \Theta\big((n-d)d\big)$$

$$= \Theta(n) + \Theta\left(\sum_{d=1}^{n-1} (n-d)d\right) = \Theta(n^3)$$

15.2 Matrix-chain multiplication

- Step 4 – Constructing an optimal solution

  Maintain a table $s[i,j], 1 \le i \le n-1, 2 \le j \le n$, such that

  $s[i,j]$ = the value of k, $i \le k < j$, that contributes to $m[i,j]$

  PRINT-OPTIMAL-PARENS(s,i,j)

  if $i == j$ **then** print "A"$_i$

  else print "("

  PRINT-OPTIMAL-PARENS$(s,i,s[i,j])$

  PRINT-OPTIMAL-PARENS$(s,s[i,j]+1,j)$

  print ")"

  Let $T(n)$ = # of pairs of parentheses printed for n matrices

  $T(1) = 0; T(n) = T(k) + T(n-k) + 1$, for some $1 \le k < n$

  A simple induction shows that $T(n) = n - 1$

15.2 Matrix-chain multiplication

- Step 4 – Constructing an optimal solution

MATRIX-CHAIN-ORDER(p) // $n = p.length - 1$

for $i = 1$ **to** n **do** $m[i, i] = 0$

for $l = 2$ **to** n // diagonal $d = l - 1$

  for $i = 1$ **to** $n - l + 1$

    $j = i + l - 1$

    $m[i, j] = \infty$

    for $k = i$ **to** $j - 1$

      $q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

      if $q < m[i, j]$

        $m[i, j] = q$

        $s[i, j] = k$ // could store $s[i, j]$ in $m[j, i]$

15.4 Longest common subsequence

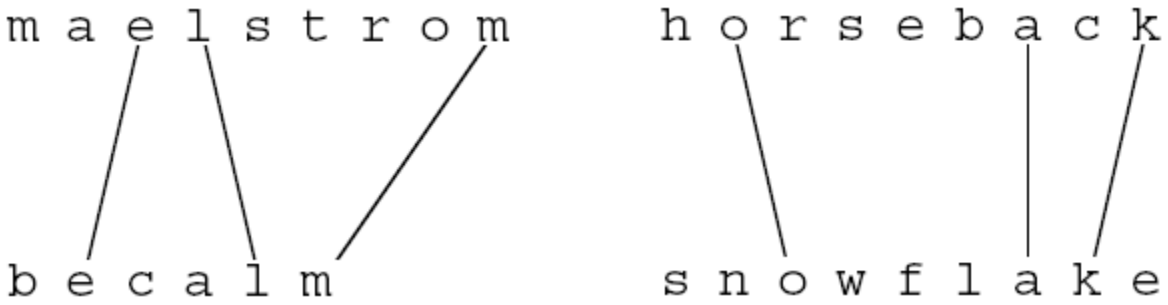
- Longest common subsequence

  Given 2 sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  Find a subsequence common to both whose length is longest.

  A subsequence doesn't have to be consecutive, but it has to be in order.



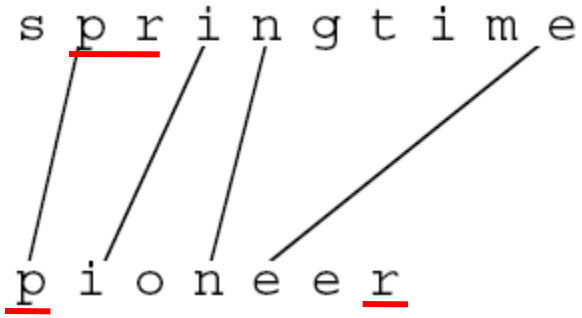
15.4 Longest common subsequence

- Brute force algorithm

  For every subsequence of X, check if it's a subsequence of Y.

  Time: $\Omega(2^m), O(n2^m)$

  - 2^m subsequences of X to check.
  - Each subsequence takes $\Omega(1)$ and $O(n)$ time to check: scan Y for first letter, from there scan for second, and so on.



15.4 Longest common subsequence

- Step 1 – Optimal substructure

THEOREM

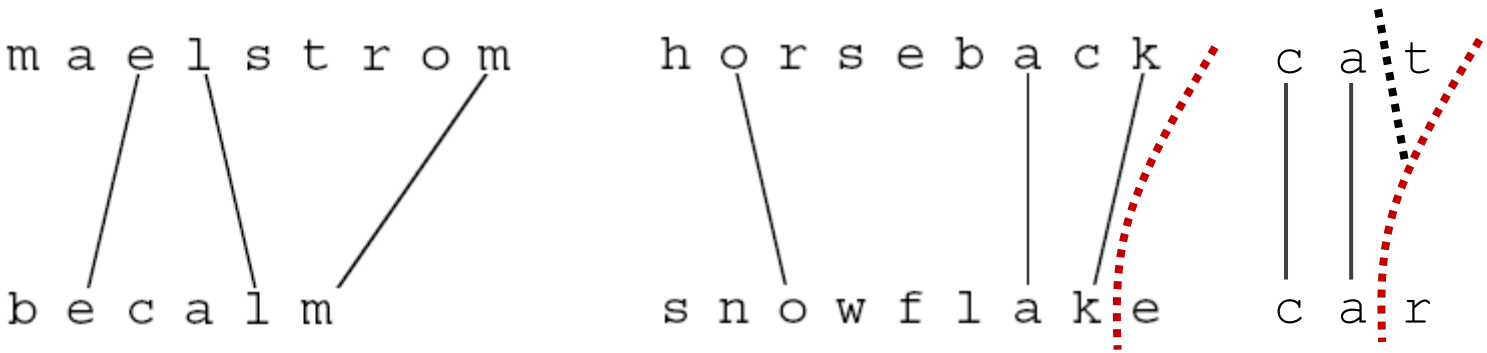
$X_i = \langle x_1, x_2, \ldots, x_i \rangle$

Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be an LCS of X and Y

1 If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

2 If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y

3 If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}



15.4 Longest common subsequence

- Step 1 – Optimal substructure

  ***Proof of case* 1**

  First show that $z_k = x_m = y_n$

  Suppose not. Let $Z' = \langle z_1, z_2, \ldots, z_k, x_m \rangle$

  Then, Z' is a CS of X and Y that is longer than Z.

  A contradiction.

  Next show that Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

  Clearly, Z_{k-1} is a CS of X_{m-1} and Y_{n-1}

  Let W be another CS of X_{m-1} and Y_{n-1} that is longer than Z_{k-1}

  Then, $W' = W + x_m$ is a CS of X and Y that is longer than Z

  A contradiction.

15.4 Longest common subsequence

- Step 1 – Optimal substructure

  Proof of case 2

  If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y

  Suppose not

  Let W be a CS of X_{m-1} and Y that is longer than Z

  Then, W is a CS of X and Y that is longer than Z

  A contradiction.

  Proof of case 3

  Symmetric to case 2

15.4 Longest common subsequence

- Step 2 – Recursive Solution

Define $c[i,j]$ = the length of an LCS of X_i and Y_j

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

Wanted: $c[m,n]$

Best case: $\Theta(\min(m,n))$

$c[m,n] \rightarrow c[m-1,n-1] \rightarrow c[m-2,n-2] \dashrightarrow c[m-n,0]$

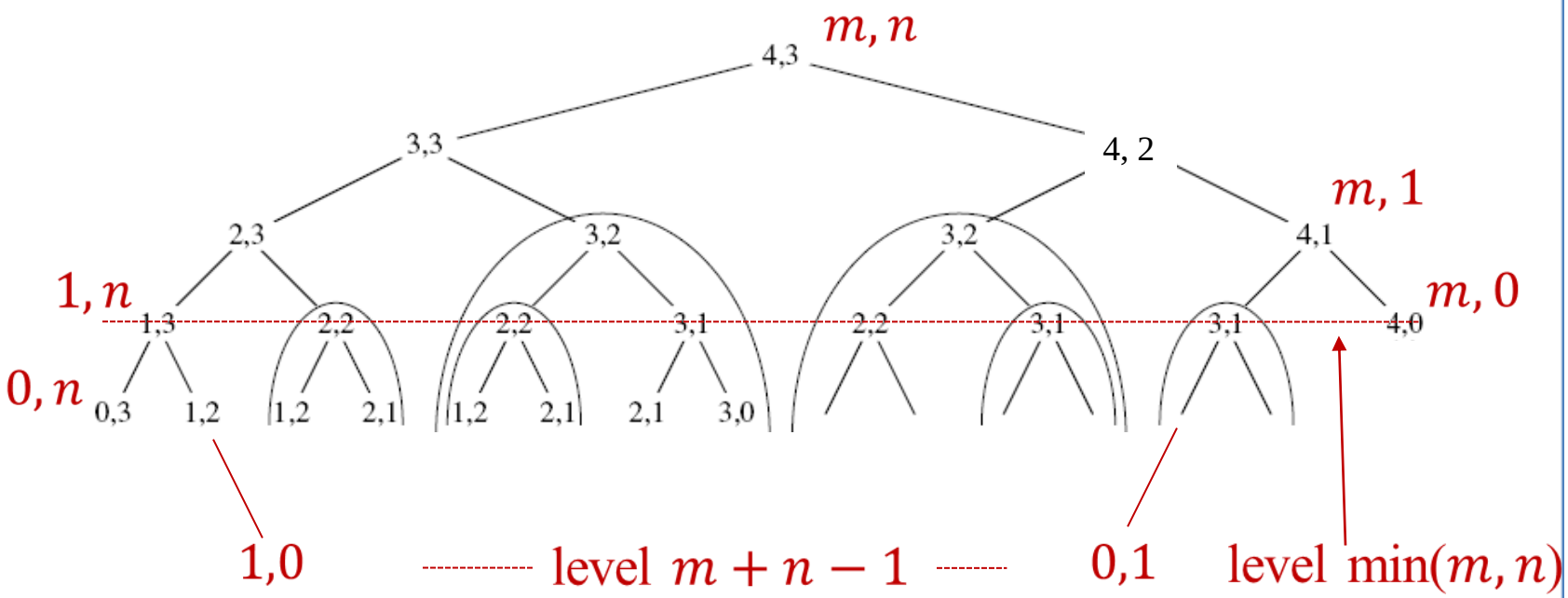
assuming that $m \geq n$, e.g.

X = c r a z y s n o o p y

Y = s n o o p y

15.4 Longest common subsequence

- Step 2 – Recursive Solution: overlapping subproblems



$X = \text{d o g s}$

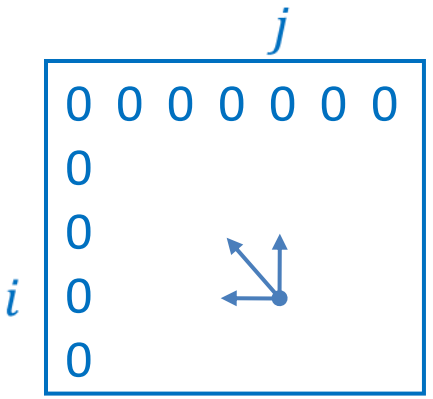
Worst case: $\Omega\left(2^{\min(m,n)}\right)$, $O(2^{m+n})$, e.g. $Y = \text{c a t}$

But, there are only $(m+1)(n+1) = \Theta(mn)$ subproblems.

15.4 Longest common subsequence

- Step 3 – Bottom-up tabulation

Table c



row-by-row or column-by-column

space $\Theta(mn)$; time $\Theta(mn)$

Table b (not really necessary)

$b[i,j] = \nwarrow, \leftarrow, \text{or } \uparrow$, pointing to the table entry whose subproblem we used in solving LCS of X_i and Y_j

Table c can be used to construct an optimal solution.

If we need only the length of an LCS, $\min(m,n)$ entries plus $O(1)$ additional space suffice (Ex. 15.4-4).

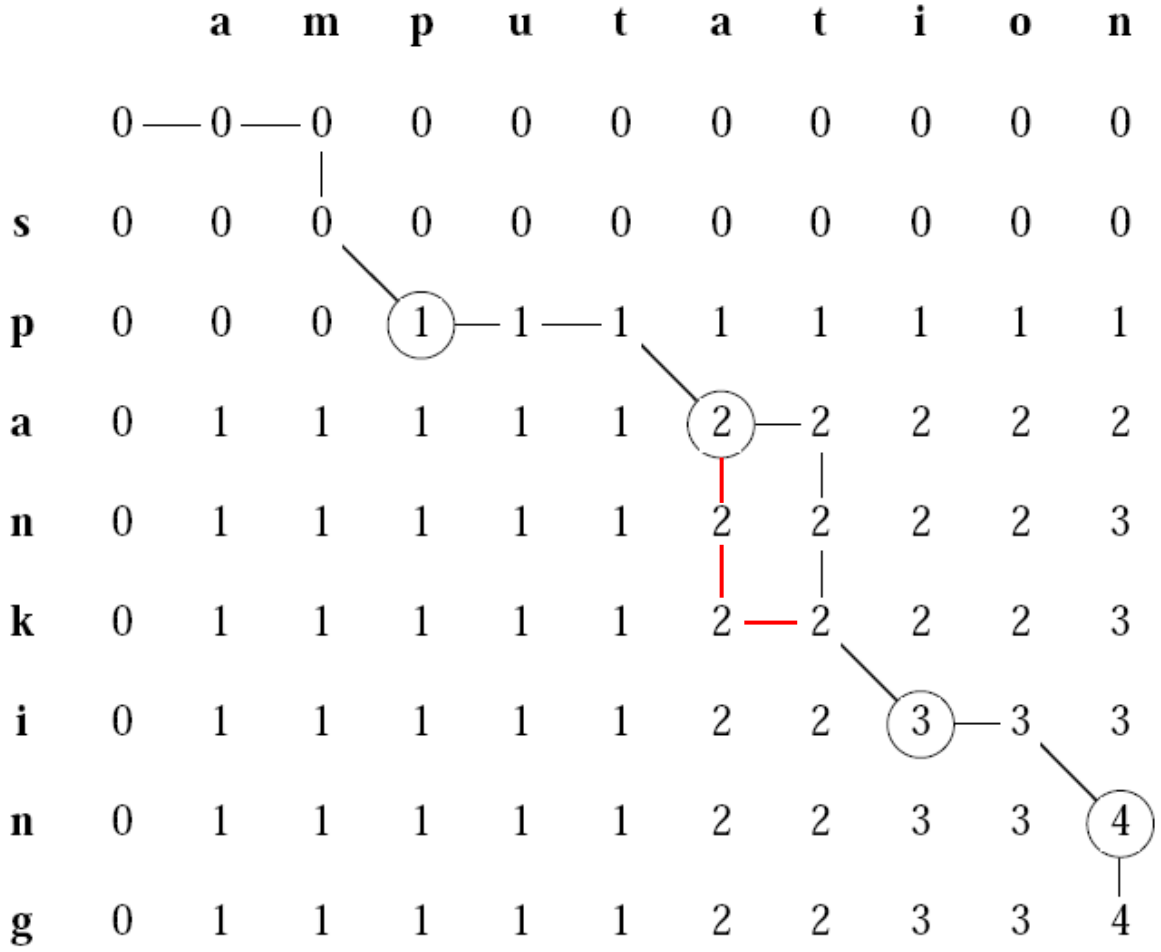
15.4 Longest common subsequence

- Step 4 – Constructing an optimal solution

PRINT-LCS(b, X, i, j)
if $i \neq 0$ and $j \neq 0$ Time: $O(m+n)$
  if $b[i,j] ==$ " $\nwarrow$ "
    PRINT-LCS$(b, X, i-1, j-1)$
    print x_i
  else if $b[i,j] ==$ "$\uparrow$"
    PRINT-LCS$(b, X, i-1, j)$
  else
    PRINT-LCS$(b, X, i, j-1)$

15.4 Longest common subsequence

- An example: LCS of spanking and amputation



Answer: pain

15.5 Optimal binary search trees

- Optimal binary search trees
  - Given n distinct keys in sorted order $k_1 < k_2 < \cdots < k_n$
  - Want to build a BST from the keys
  - For each key, there is a probability p_i that a search is for k_i
  - Want a BST with minimum expected search cost
  - Similar to book, but simplified here.

    We assume that all searches are successful, i.e.

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

    Book has probabilities q_i of unsuccessful searches between keys k_i and k_{i+1}, i.e. $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$

15.5 Optimal binary search trees

- Optimal binary search trees

  Let $\text{depth}_T(k_i) =$ the depth of k_i in BST T

  For key k_i, search cost $= \text{depth}_T(k_i) + 1$

  Thus,

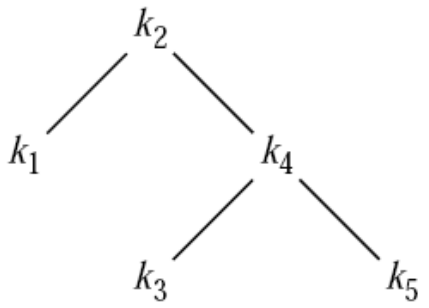
$$\begin{aligned}
\text{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i \\
&= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i
\end{aligned}$$

15.5 Optimal binary search trees

- Optimal binary search trees

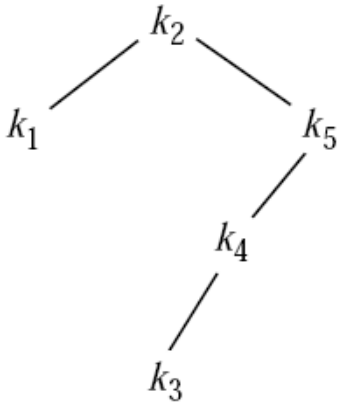
Example

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

E[search cost] = 2.15



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

E[search cost] = 2.10

15.5 Optimal binary search trees

- Optimal binary search trees

  Observations

  - An optimal BST might not have smallest height.
  - An optimal BST might not have the highest-probability key at root.

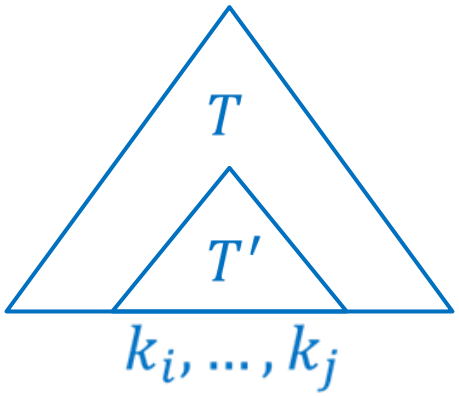
  Brute force algorithm

  - Construct each n-node BST.
  - For each BST, put in keys (there is only one way to do so)
  - Then compute the expected search cost for the BST.
  - But, there are $\Omega\left(\frac{4^n}{n^{3/2}}\right)$ different BSTs with n nodes.

15.5 Optimal binary search trees

- Step 1 – Optimal substructure

  Any subtree of a BST contains keys in contiguous range.



If T is an optimal BST and T' is a subtree with keys $k_i, \ldots, k_j$ then T' must be an optimal BST for keys $k_i, \ldots, k_j$

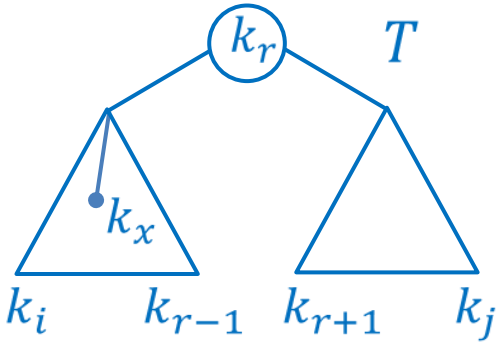
Proof: Cut-and-paste

15.5 Optimal binary search trees

- Step 2 – Recursive Solution

  Let

  $$w(i,j) = \sum_{k=i}^{j} p_k$$



  and suppose k_r is the root of a BST T for $k_i, \ldots, k_j$

  Since the depth of a node k_x in a subtree goes up by 1 in T, the expected search cost of k_x in T increases p_x.

  Thus, expected search cost in T

  $$= p_r + e[i, r-1] + w(i, r-1) + e[r+1, j] + w(r+1, j)$$
  $$= e[i, r-1] + e[r+1, j] + w(i, j)$$

15.5 Optimal binary search trees

- Step 2 – Recursive Solution

  Define

  $e[i,j]$ = expected search cost of an optimal BST for $k_i, \ldots, k_j$

  Then,

  $$e[i,j] = \begin{cases} 0 & i-1 = j \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & i \le j \end{cases}$$

  Wanted: $e[1,n]$

  W shall use an $(n+1) \times (n+1)$ table e, indexed by

  $e[1..n+1, 0..n]$

  so that the diagonal entries $e[i,i-1]$ are all 0's.

15.5 Optimal binary search trees

- Step 3 – Bottom-up tabulation

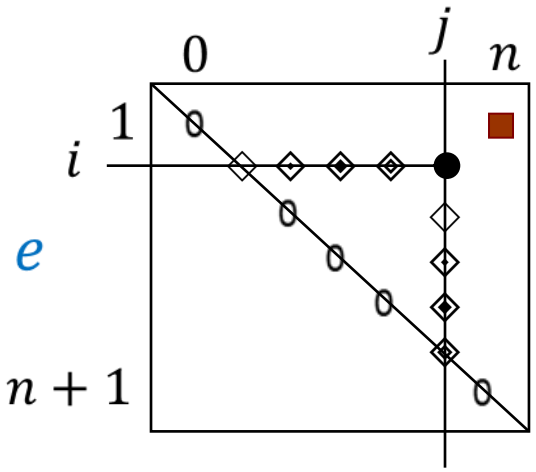


Table e

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

Just like

Matrix Chain Multiplication

Table $root[1..n, 1..n]$

$root[i, j]$ = root of subtree with keys $k_i, \ldots, k_j$

Table $w[1..n+1, 0..n]$

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

$w[i, i-1] = 0, \ 1 \le i \le n$

$w[i, j] = w[i, j-1] + p_j, \ 1 \le i \le j \le n$

15.5 Optimal binary search trees

- Step 4 – Constructing an optimal solution

CONSTRUCT-OPTIMAL-BST($root$)
$r = root[1, n]$
print "k"$_r$ "is the root"
CONSTRUCT-OPTIMAL-SUBTREE$(1, r-1, r, \text{"left"}, root)$
CONSTRUCT-OPTIMAL-SUBTREE$(r+1, n, r, \text{"right"}, root)$

CONSTRUCT-OPTIMAL-SUBTREE$(i, j, r, dir, root)$
if $i \le j$
    $t = root[i, j]$
    print "k"$_t$ "is" dir "child of k"$_r$
    CONSTRUCT-OPTIMAL-SUBTREE$(i, t-1, t, \text{"left"}, root)$
    CONSTRUCT-OPTIMAL-SUBTREE$(t+1, j, t, \text{"right"}, root)$

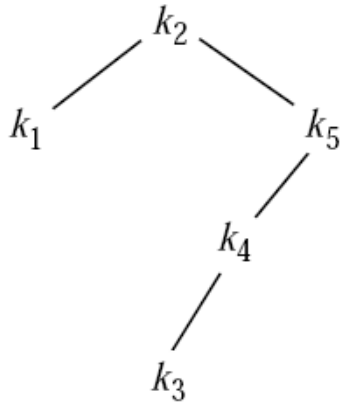
Time: $\Theta(n)$, just like Matrix Chain Multiplication.

15.5 Optimal binary search trees

- An example

e	0	1	2	3	4	5
1	0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35
3			0	.05	.3	.85
4				0	.2	.7
5					0	.3
6						0

p_i



$root$	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

w	0	1	2	3	4	5
1	0	.25	.45	.5	.7	1.0
2		0	.2	.25	.45	.75
3			0	.05	.25	.55
4				0	.2	.5
5					0	.3
6						0

15.3 Elements of dynamic programming

- Elements of dynamic programming
  - Optimal substructure

    Dynamic programming uses optimal substructure *bottom up*
    - *First* find optimal solutions to subproblems.
    - *Then* choose which to use in optimal solution to the problem

    Of course, not every optimization problem satisfies this property. (See next page)
  - Overlapping subproblems

    Divide-and-conquer algorithms do not have overlapping subproblems.

15.3 Elements of dynamic programming

- Elements of dynamic programming

  Given an unweighted, directed graph $G = (V, E)$

  Shortest path problem

  Find path $u \rightsquigarrow v$ with fewest edges.

  - Must be *simple* (no cycles), since removing a cycle from a path gives a path with fewer edges.

  Longest simple path problem

  Find *simple* path $u \rightsquigarrow v$ with most edges

  - If didn't require simple, could repeatedly traverse a cycle to make an arbitrarily long path.

15.3 Elements of dynamic programming

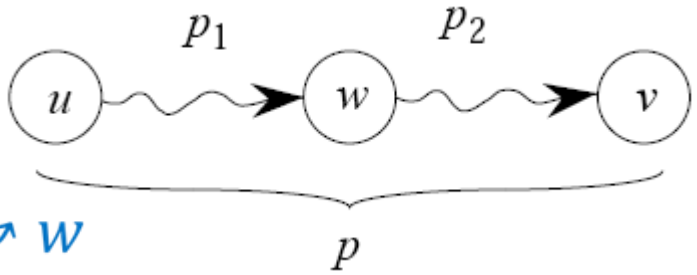
- Elements of dynamic programming

THEOREM Shortest path has optimal substructure.

Suppose p is a shortest path $u \rightsquigarrow v$

Let w be any vertex on p.

Let p_1 be the portion of p going $u \rightsquigarrow w$



Then, p_1 is a shortest path $u \rightsquigarrow w$.

Proof

Suppose there exists a shorter path p_1' going $u \rightsquigarrow w$.

Then, the path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ is shorter than p.

A contradiction.

Same argument applies to p_2

15.3 Elements of dynamic programming

- Elements of dynamic programming

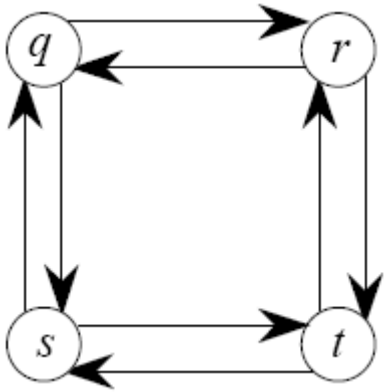
  Longest simple path doesn't have optimal substructure.

  Consider

  $q \to r \to t$ = longest simple path $q \rightsquigarrow t$

  Are its subpaths longest simple paths?

  NO!



  - Subpath $q \rightsquigarrow r$ is $q \to r$

    Longest simple path $q \rightsquigarrow r$ is $q \to s \to t \to r$
  - Subpath $r \rightsquigarrow t$ is $r \to t$

    Longest simple path $r \rightsquigarrow t$ is $r \to q \to s \to t$