COMP 6651
Algorithm Design Techniques

Lecturer: Thomas Fevens

Department of Computer Science and Software Engineering, Concordia U
thomas.fevens@concordia.ca

Table of Contents

Dynamic programming

The term *programming* does not mean “computer coding”, it is related to the original meaning of programming as making plans of events, i.e. making program of an evening.

Terms *Dynamic programming, linear programming* were used back in the 1940s for designing optimized plans of management of large systems using tables.

Dynamic programming is used in finding the optimal value of a solution where several solutions exists.

Examples:
Classroom assignment, computer job scheduling, spell-checking,...

Usually there are more than one solution to the problem.

Motivating Example

Calculate Fibonacci Sequence F_n

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

The first few terms of the sequence are

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

The associated computational problem:

Input: $n \geq 0$
Output: F_n

Simple recursive solution

F(n)
  if n ≤ 1 **then**
    return 1
  else
    return Fib($n-1$) + **Fib**($n-2$)

Let $T(n)$ = number of addition operations

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \text{ if } n \geq 2 \\ T(0) &= T(1) = 0 \end{aligned}$$

How large is $T(n)$?

Notice that $T(n)$ is monotone, therefore

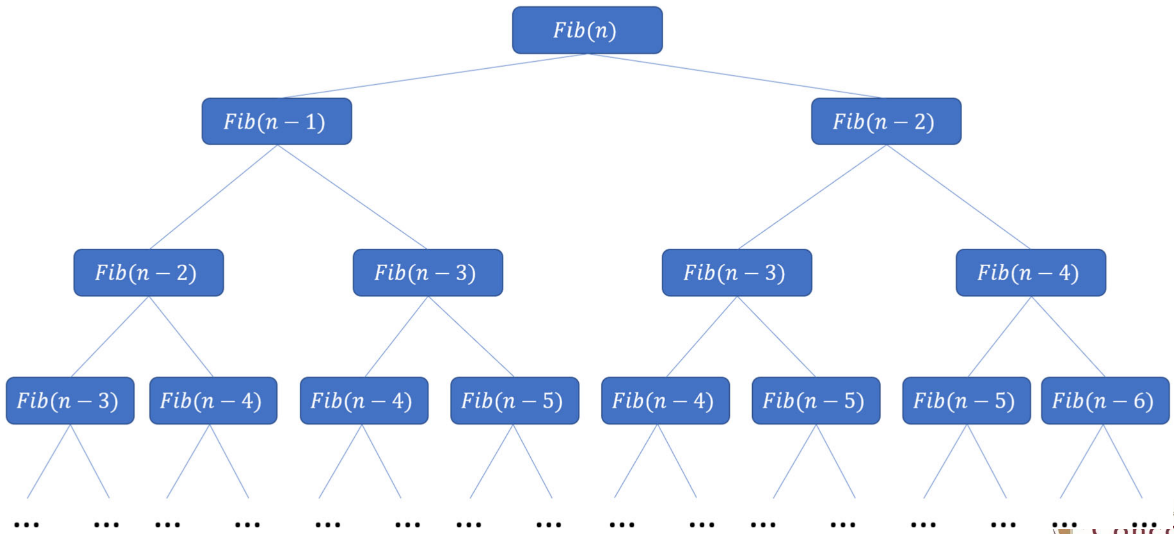
$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq T(n-2) + T(n-2) + 1 \\ &= 2T(n-2) + 1 \end{aligned}$$

Every two steps the value of $T(n)$ doubles, therefore

$$T(n) = \Omega\left(2^{n/2}\right)$$

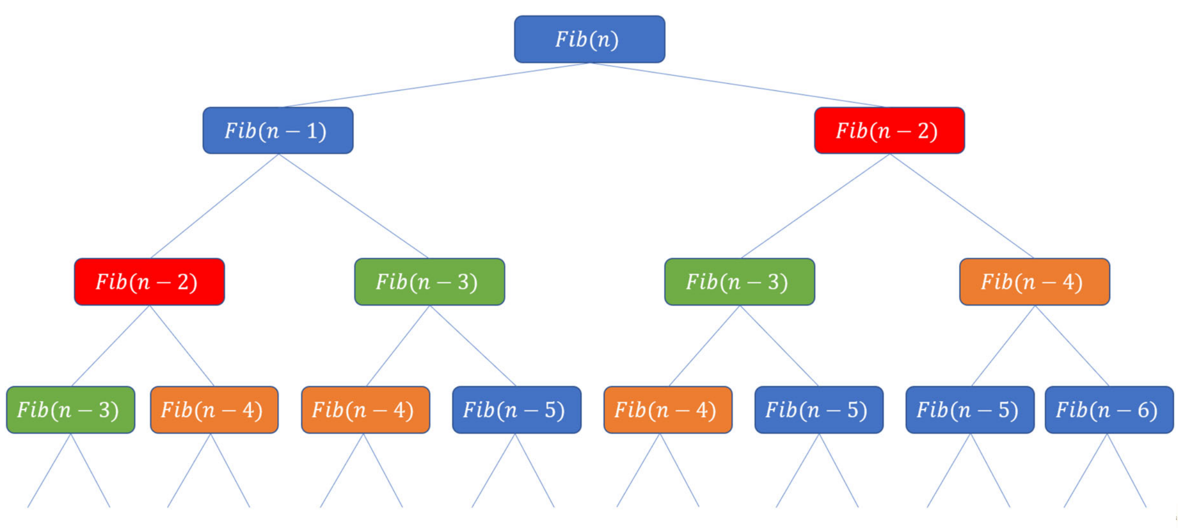
This is exponential! We should be able to do better

Below is the execution function calls.



Dynamic Programming Fibonacci Sequence Assembly-line Sched. Rod Cutting Matrices Multiplication Longest Common Subseq. Optimal BSTs

There is a lot of redundant calls calculating the same values.



Obvious solution: remember Fibonacci numbers that you computed before and look them up when you need them!

Fib(n)

$F[0..n] \leftarrow$ initialize all entries of a global array to -1 (indicating "not computed yet")

return Memoized-RecFib(n)

Memoized-RecFib(n)

\\\\ has access to global array $F[0..n]$

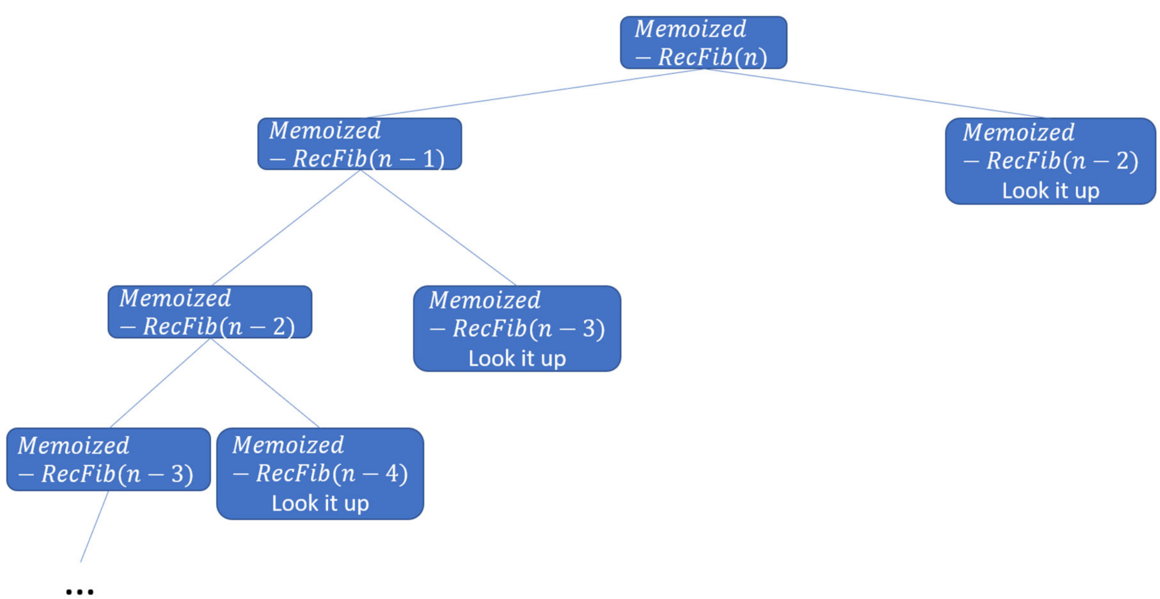
if F[n] $\neq$ -1 **then**

  return $F[n]$ \\\\ look up the value

else

  return $F[n] \leftarrow$ **Memoized-RecFib**$(n-1)$ + **Memoized-RecFib**$(n-2)$

A lot of redundancy is now gone and we only need $O(n)$ additions.



This is the essence of dynamic programming!

There are two approaches:

1. Recursion with overlapping subproblems + a table or a map to store solutions to subproblems
2. Similar to (1), but often you can get rid of recursion altogether and populate the table iteratively

Approach (1) is called (top down with) memoization (NOT memoRization) in the context of dynamic programming.

Approach (2) is sometimes referred to as iterative dynamic programming (or the bottom-up method).

Using approach (2) to computing the Fibonacci sequence:

IterativeFib(n)
  $F[0..n] \leftarrow$ initialize all entries of a global array to -1 (indicating "not computed yet")
  $F[0] \leftarrow F[1] \leftarrow 1$
  for $i \leftarrow 2, n$ **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
  return $F[n]$

Iterative approach vs memoization

- Iterative approach has a benefit of avoiding recursive function calls and function calls may be expensive in real-life programming
- Memoization is easier because sometimes the pattern of recursive calls is not easy to understand
- Memoization may use less memory if not all entries in the table need to be filled in

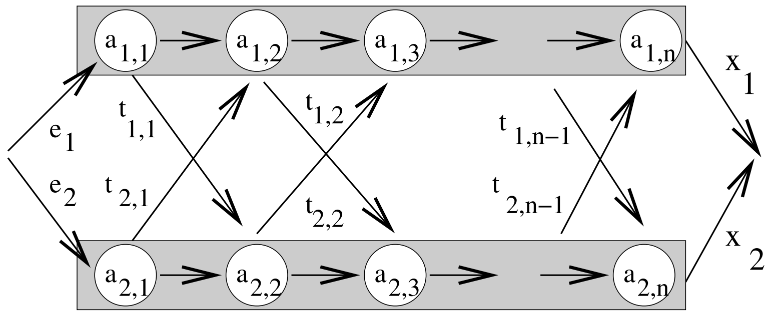
Structure of subproblems

1. Find a structure of subproblems parameterized by one or more variables

   Example: $F_i(i < n)$ is a subproblem of F_n with the variable – index i
2. Optimal solution to a problem should be reconstructable from optimal solutions to subproblems (**optimal substructure property**)
3. Since problems rely on subproblems, which rely on sub-subproblems, and so on, many of the sub-sub-...-subproblems must be *shared* in order to achieve savings (**overlapping subproblems property**)

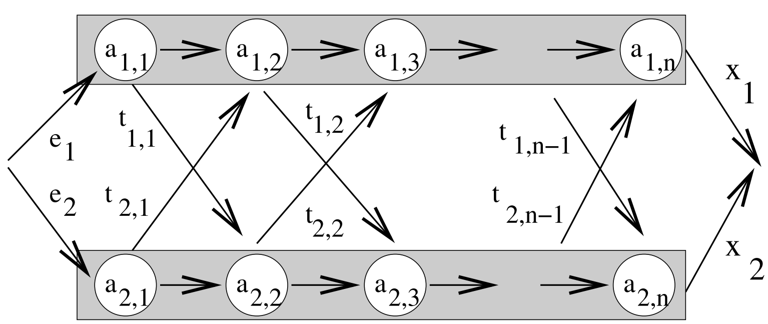
Usually (3) follows from (1) and bounds on values that variables can achieve

1. Assembly-line scheduling



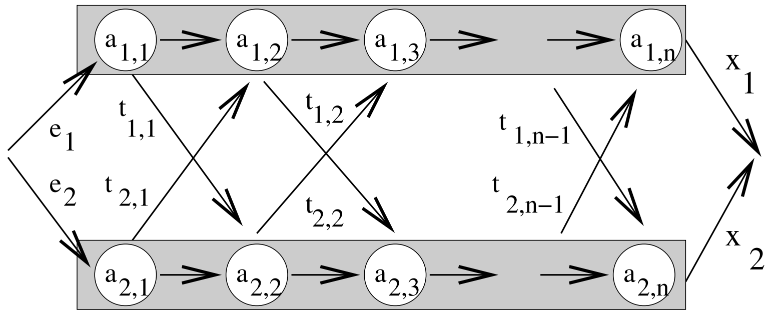
A product can be made on line 1 or line 2, each line consisting of several stations.

e_1, x_1 the time needed to enter, exit line 1; e_2, x_2 the time needed to enter, exit line 2,
$a_{1,i}$ the time needed in station i on line 1,
$a_{2,i}$ the time needed in station i on line 2,



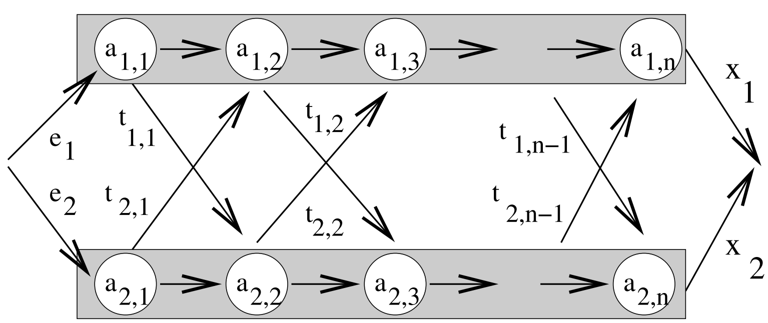
$t_{1,i}$ the time needed to transfer from line 1 to 2.
$t_{2,i}$ the time needed to transfer from line 2 to 1.

Find the shortest time to complete the product.



$f_1[i]$ ... the fastest way through station i on line 1
$f_2[i]$ ... the fastest way through station i on line 2

$f_1[1] = e_1 + a_{1,1}$
$f_2[1] = e_2 + a_{2,1}$

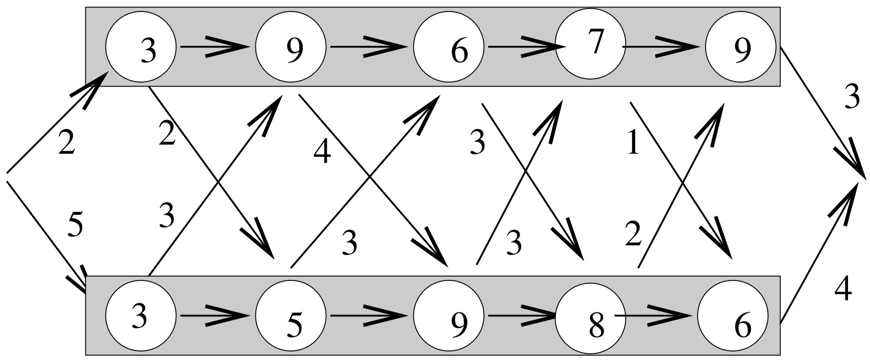


$f_1[j] = \min\{f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\}$
$f_2[j] = \min\{f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\}$

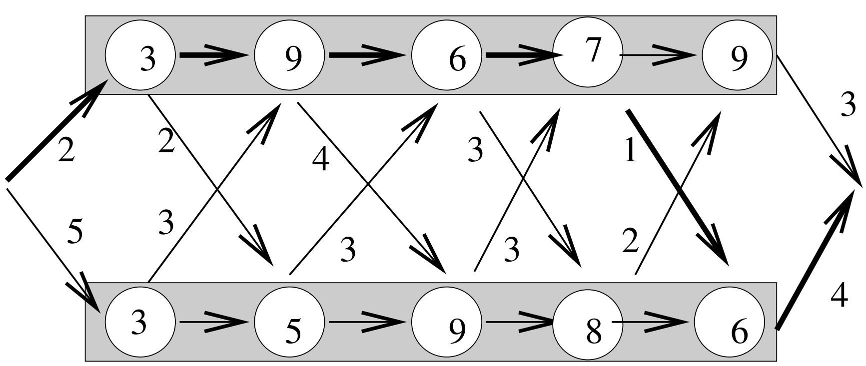
Solution $= \min\{f_1[n] + x_1, f_2[n] + x_2\}$

Recursive calculation of f_1, f_2 is very inefficient.

We can do it iteratively (bottom-up).



	f_1	f_2
1	5	8
2	14	12
3	20	21
4	27	29
5	36	34
	39	38



	f_1	f_2
1	5	8
2	14	12
3	20	21
4	27	29
5	36	34
	39	38

FastestWay(a, t, e, x, n)

$f[1,1] \leftarrow e[1] + a[1,1]$; $f[2,1] \leftarrow e[2] + a[2,1]$
for $j \leftarrow 2, n$ **do**
  if $(f[1,j-1] + a[1,j] \leq f[2,j-1] + t[2,j-1] + a[1,j])$ **then**
    $f[1,j] \leftarrow f[1,j-1] + a[1,j]$; $l[1,j] \leftarrow 1$
  else
    $f[1,j] \leftarrow f[2,j-1] + t[2,j-1] + a[1,j]$; $l[1,j] \leftarrow 2$
  if $(f[2,j-1] + a[2,j] \leq f[1,j-1] + t[1,j-1] + a[2,j])$ **then**
    $f[2,j] \leftarrow f[2,j-1] + a[2,j]$; $l[2,j] \leftarrow 2$
  else
    $f[2,j] \leftarrow f[1,j-1] + t[1,j-1] + a[2,j]$; $l[1,j] \leftarrow 1$
if $(f[1,n] + x[1] \leq f[2,n] + x[2])$ **then**
  $f\_fin \leftarrow f[1,n] + x[1]$; $l\_fin \leftarrow 1$
else
  $f\_fin \leftarrow f[2,n] + x[2]$; $l\_fin \leftarrow 2$

Dynamic programming algorithm usually consists of four steps:

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 in bottom-up fashion or by using memoization.
4. Construct an optimal solution from computed information.

(an optimal solution $\neq$ value of an optimal solution)

2. Rod Cutting (§14.1)

- Rods (metal sticks) are cut and sold.
- Rods of length $n \in \mathbb{N}$ are available. A cut is done at no cost.
- For each length $i \in \mathbb{N}$, $i \leq n$, of rod has a given price $p_i \in \mathbb{R}^+$
- Goal: cut the rods such (into $k \in \mathbb{N}$ pieces) that

$$r_n = \sum_{j=1}^{k} p_{i_j} \text{ is maximized subject to } \sum_{j=1}^{k} i_j = n$$

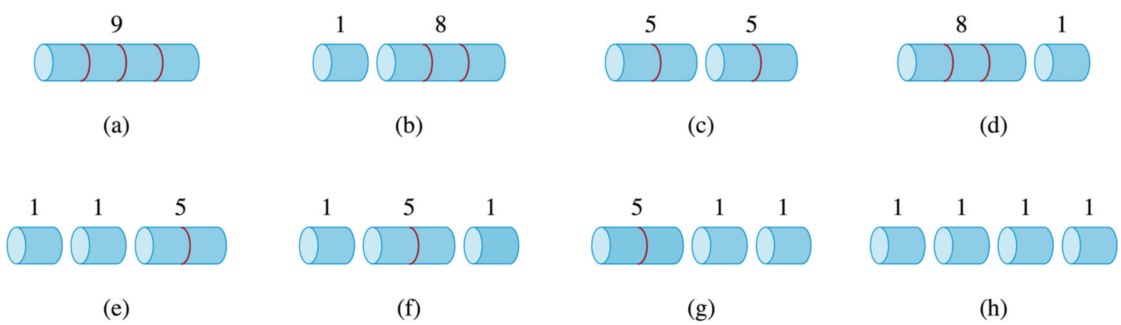
Note that it is possible that $i_j = i_l$ for $j \neq l$ (i.e., two or more pieces of the same length)

Example

Sample price table for rods:

Length i	0	1	2	3	4	5	6	7	8	9	10
Price p_i	0	1	5	8	9	10	17	17	20	24	30

The 8 possible ways of cutting up a rod of length 4:



Optimal decomposition: $i_1 + i_2 = 2 + 2 = 4$
with price $r_4 = p_{i_1} + p_{i_2} = p_2 + p_2 = 5 + 5 = 10$.

For the above example, you can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

$r_1 = 1$	from solution 1 = 1	(no cuts),
$r_2 = 5$	from solution 2 = 2	(no cuts),
$r_3 = 8$	from solution 3 = 3	(no cuts),
$r_4 = 10$	from solution 4 = 2 + 2,	
$r_5 = 13$	from solution 5 = 2 + 3,	
$r_6 = 17$	from solution 6 = 6	(no cuts),
$r_7 = 18$	from solution 7 = 1 + 6 or 7 = 2 + 2 + 3,	
$r_8 = 22$	from solution 8 = 2 + 6,	
$r_9 = 25$	from solution 9 = 3 + 6,	
$r_{10} = 30$	from solution 10 = 10	(no cuts).

View a decomposition as consisting of a first undivided piece of length i cut off the left-hand end, and then some decomposition of a right-hand remainder of length $n - i$.

The solution with no cuts has the first piece with size $i = n$ and revenue p_n and the remainder has size 0 with corresponding revenue $r_0 = 0$.

We say that the rod-cutting problem exhibits **optimal substructure**: optimal solutions to a problem incorporate optimal solutions to related subproblems, which you may solve independently.

Then we can express the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}.$$

Structure of the solution

Wanted: r_n = maximal value of rod (cut or as a whole) with length n.

Sub-problems: maximal value r_k for each $0 \leq k < n$

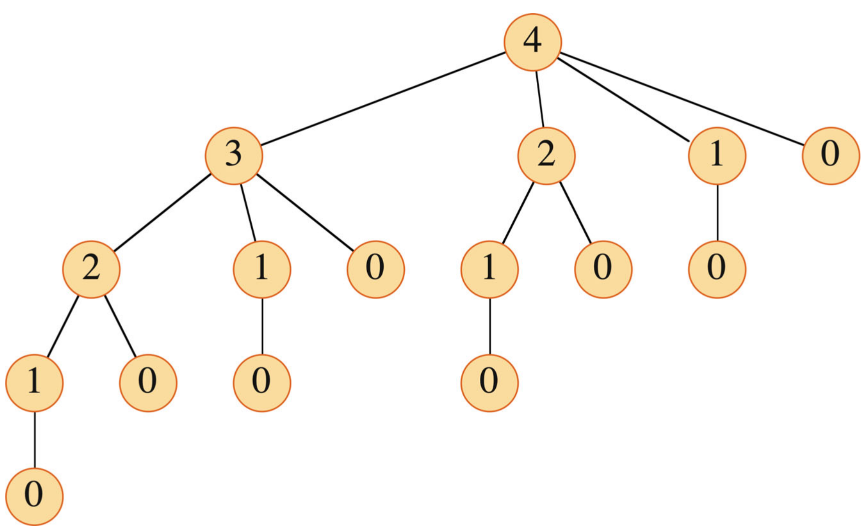
Recursion

$$\begin{aligned} r_k &= \max\{p_i + r_{k-i} : 1 \leq i \leq n\}, k > 0 \\ r_0 &= 0 \end{aligned}$$

Dependency: r_k depends (only) on values $p_i, 1 \leq i \leq k$ and the optimal cuts $r_i, i < k$

Solution in r_n

Recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$.



Use dynamic-programming. Instead of solving the same subproblems repeatedly, as above, arrange for each subproblem to be solved only once.

MEMOIZED-CUT-ROD(p, n)
Input: $n \geq 0$, Prices p
Output: best value

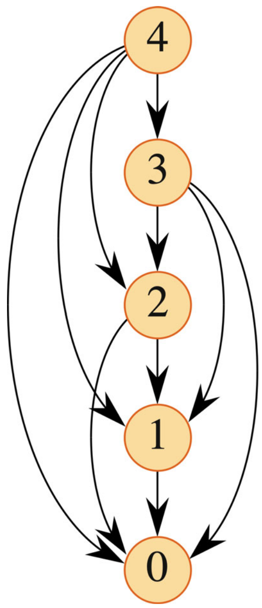
```
Let r[0..n] be a new array   ▷ Will remember solution values in memoization table r
for i ← 0, n do                                              ▷ Initialization
    r[i] ← −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```


MEMOIZED-CUT-ROD-AUX(p, n, r)
Input: $n \geq 0$, Prices p, Memoization Table r
Output: best value

```
if r[n] ≥ 0 then                          ▷ Already have a solution for length n?
    return r[n]
if n = 0 then
    q ← 0
else
    q = −∞
    for i ← 1, n do                          ▷ i is the position of the first cut
        q ← max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
    r[n] ← q                          ▷ Remember the solution value for length n
return q
```

Running time

$T(n) = \sum_{i=1}^{n} i = \Theta(n^2)$



Subproblem-Graph

Describes the mutual dependencies of the subproblems

4 3 2 1 0

and must not contain cycles

Construction of the Optimal Cut

During the (recursive) computation of the optimal solution for each $k \leq n$ the recursive algorithm determines the optimal length of the first rod

Store the length of the first rod in a separate table $s[1..n]$. We modify **MEMOIZED-CUT-ROD-AUX**(p, n, r) to compute s of optimal first-piece sizes. Then can print out the complete list of piece sizes in an optimal decomposition of a rod of length n.

For our example:

i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]		1	2	3	2	2	6	1	2	3	10

3. Optimal sequence of matrix multiplications of matrices of different sizes

Example: Consider multiplying 4 matrices:

$$M_1 \times M_2 \times M_3 \times M_4$$

There are several possible orders of evaluation (matrix multiplication is associative), but some orders save on the number of scalar multiplications.

M_1 is 10×20
M_2 is 20×5
M_3 is 5×10
M_4 is 10×5

Assume that $m \times n$ matrix with $n \times p$ matrix takes mnp multiplications

Order 1 $= M_1 \times ((M_2 \times M_3) \times M_4)$ takes $1000 + 1000 + 1000 = 3000$ mult.
Order 2 $= (M_1 \times M_2) \times (M_3 \times M_4)$ takes $1000 + 250 + 250 = 1500$ mult.

Matrix Chain Multiplication (§14.2)

More generally, determine an optimal order to multiply matrices

$$A_1 \times A_2 \times \cdots \times A_n$$

with respective dimensions

$$p_0 \times p_1,\ p_1 \times p_2,\ \ldots, p_{n-1} \times p_n$$

Formally:

Input: $P[0..n]$ - array of $n+1$ positive integers, representing dimensions of matrices as above
Output: optimal parenthesization to minimize the total cost of multiplying

Naïve solution

One solution would be to exhaustively check all possible parenthesizations.

For example, if the chain of matrices is $<A_1, A_2, A_3, A_4>$, then you can fully parenthesize the product $A_1 \times A_2 \times A_3 \times A_4$ in 5 distinct ways:

$$(A_1 \times (A_2 \times (A_3 \times A_4))),$$

$$(A_1 \times ((A_2 \times A_3) \times A_4)),$$
$$((A_1 \times A_2) \times (A_3 \times A_4)),$$
$$((A_1 \times (A_2 \times A_3)) \times A_4),$$
$$(((A_1 \times A_2) \times A_3) \times A_4).$$

Let $P(n)$ denote the number alternative parenthesizations of sequence of n matrices.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to this recurrence is $\Omega(2^n)$. So the number of solutions to check is exponential in n.

Let's try a dynamic programming method instead for this problem.

Subproblems

Consider an optimal parenthesization

$$\Big(\ (A_1 \times (A_2 \times \cdots \times A_i))\boxed{\times}(((A_{i+1} \times (A_{i+2} \times A_{i+3})) \times \cdots \times A_n)\ \Big)$$

Some multiplication is going to be performed last according to this parenthesization
In the above example $\rightarrow$ the last multiplication is placed in a box

This naturally partitions the original problem into two subproblems, either side of the last multiplication

More generally, subproblems are defined by two indices i and j (i.e., the parameters of the subproblems)

Find minimum cost parenthesization of

$$A_i \times A_{i+1} \times \cdots \times A_j$$

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

Optimal substructure property

Let $OPT[i,j]$ denote the minimum cost of parenthesization of

$$A_i \times A_{i+1} \times \cdots \times A_j$$

(the result has dimensions $p_{i-1} \times p_j$)

Then $OPT[i,j]$ consists of performing kth multiplication last for some $k \in \{i, i+1, \ldots, j-1\}$ (assuming $j > i$) and optimally parenthesizing

$$A_i \times A_{i+1} \times \cdots \times A_k \text{ and } A_{k+1} \times \cdots \times A_j$$

Therefore:

$$OPT[i,j] = OPT[i,k] + OPT[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

Computing optimal value

We want do define an array $m[i,j]$ that stores the minimum cost of multiplication of the subproblem $A_i \times \cdots \times A_j$

Then the solution to the whole problem is $m[1,n]$

The computational array:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k,j] + p_{i-1} \cdot p_k \cdot p_j\} & i < j \end{cases}$$

A proof of correctness would show that the computational array actually implements the meaning of the array definition.

Computing optimal value

Subproblem $A_i \times \cdots \times A_j$ has size $l = j - i + 1$, the number of matrices in the matrix chain.

Smallest subproblem size $l = 1$, i.e., $i = j$ or a matrix chain of a single matrix, in which case there is nothing to multiply

$$m[i,i] = 0$$

Working on a subproblem of size l we rely on having solved subproblems of size $< l$

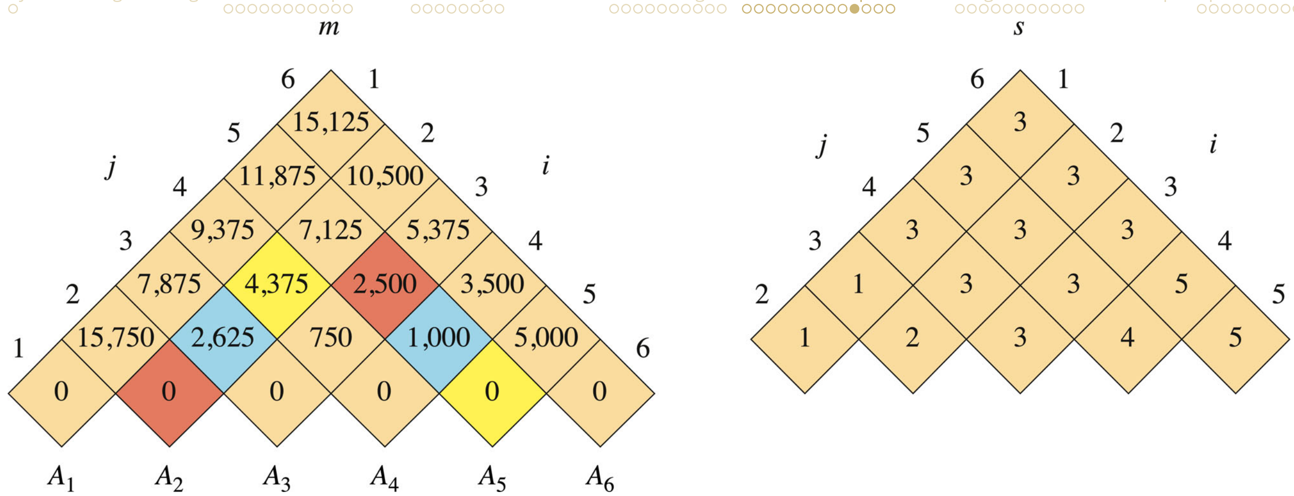
Starting with $m[i,i]$, using an iterative dynamic programming approach, we solve increasing larger subproblems $m[i, i+l-1]$, with increasing lengths of matrix chains l until we solve the final subproblem $m[1,n]$

Pseudocode

MATRIX-CHAIN-ORDER($p[0..n]$)

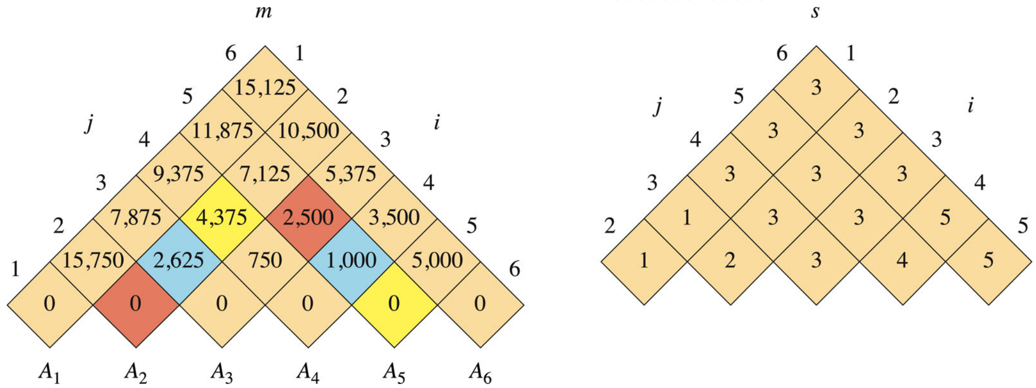
```
initialize tables m[1..n, 1..n] and s[1..n − 1, 2..n]
for i ← 1, n do                                   ▷ chain length 1
    m[i, i] ← 0
for l ← 2, n do                                   ▷ l is the chain length
    for i ← 1, n − l + 1 do                       ▷ chain begins at A_i
        j = i + l − 1                             ▷ chain end at A_j
        m[i, j] ← ∞
        for k ← i, j − 1 do                       ▷ try A_{i..k} A_{k+1..j}
            q ← m[i, k] + m[k + 1, j] + p[i − 1]p[k]p[j]
            if q < m[i, j] then
                m[i, j] ← q                       ▷ remember this cost
                s[i, j] ← k                       ▷ remember this index
return m[1, n]
```


Dynamic Programming Fibonacci Sequence Assembly-line Sched. Rod Cutting Matrices Multiplication Longest Common Subseq. Optimal BSTs



The m and s tables computed by **MATRIX-CHAIN-ORDER** for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



Of the entries that are not tan, the pairs that have the same color are taken together in line 9 of **MATRIX-CHAIN-ORDER** when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13,000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11,375 \end{cases}$$
$$= 7125.$$

Time Complexity

A simple inspection of the **MATRIX-CHAIN-ORDER** pseudocode yields a worst-case running time of $O(n^3)$.

There are three nested **for** loops, where each loop index can take on at most $n-1$ values.

It can be shown that worst case running time for **MATRIX-CHAIN-ORDER** is $\Omega(n^3)$.

The space requirements are $\Theta(n^2)$.

Question: Is the algorithm given by the **MATRIX-CHAIN-ORDER** pseudocode an example of an iterative approach or memoization?

Computing actual parenthesization

As usual, to compute an actual parenthesization remember the choice of k resulting in the min value of $m[i,j]$. These choices are recorded in $s[i,j]$ as on line 12 in the **MATRIX-CHAIN-ORDER** pseudocode).

Using s to print actual parenthesization

The following recursive function prints the actual parenthesization
PRINT-OPTIMAL-PARENS$(s[1..n-1, 2..n], i, j)$

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 ")"

Initial call is to **PRINT-OPTIMAL-PARENS**$(s, 1, n)$

For our example, the call **PRINT-OPTIMAL-PARENS**(s,1,6) prints

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

4. Longest common subsequence (§14.4)

Given two strings of characters,

$x_1x_2x_3\cdots x_n$
$y_1y_2y_3\cdots y_n$

find the longest substring that is in both strings. The substring can be obtained by omitting some characters.

Example: 100100111
010101101

<u>common subsequences</u>: the empty string 0
0000 11111
01011 001001
0010111 1010111

There are 2^n common subsequences (that is, n characters, in LCS or not, for each), so exhaustive search of all subsequences is not feasible.

For $X = x_1x_2x_3 \cdots x_m$ we define $X_i = x_1x_2x_3 \cdots x_i$, the prefix of X of length i.

Theorem

Let $X = x_1x_2x_3 \cdots x_m$ and $Y = y_1y_2y_3 \cdots y_n$ be sequences, and let $Z = z_1z_2 \cdots z_k$ be any 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 $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

By the theorem, the LCS problem has an optimal-substructure property which suggests a recursive solution.

The recursive solution also has the overlapping-subproblems property. I.e., a naïve recursive solution based on the theorem recomputes many sub-instances twice.

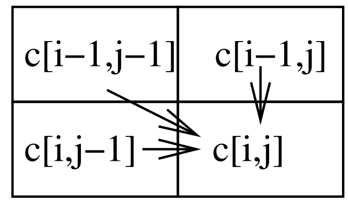
For example, while finding LCS of X and Y, the solution needs to determine both the LCS of X and Y_{n-1} and the LCS of X_{m-1} and Y. But both of these need to find the LCS of X_{m-1} and Y_{n-1}.

This results in an exponential algorithm.

We build a dynamic programming solution bottom-up (iteratively) for an efficient algorithm.

$c[i,j]$ stores the length of the LCS of X_i and Y_j.

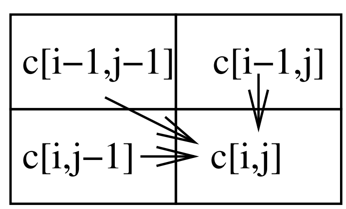
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$



The matrix c is computed in the row-major order.

We keep array $b[i,j]$ to point to the table entry used in an optimal solution to the $c[i,j]$ entry.

LCS-Length($X[1..m]$, $Y[1..n]$)



Define $c[0..m, 0..n]$, $b[1..m, 1..n]$
for $i \leftarrow 1, m$ **do**
    $c[i,0] \leftarrow 0$
for $j \leftarrow 1, n$ **do**
    $c[0,j] \leftarrow 0$
for $i \leftarrow 1, m$ **do** $\triangleright$ compute table entries in row-major order
    for $j \leftarrow 1, n$ **do**
        if $x[i] = y[j]$ **then**
            $c[i,j] = c[i-1,j-1]+1$; $b[i,j] =$ "$\nwarrow$"
        else if $c[i-1,j] \geq c[i,j-1]$ **then**
            $c[i,j] = c[i-1,j]$; $b[i,j] =$ "$\uparrow$"
        else
            $c[i,j] = c[i,j-1]$; $b[i,j] =$ "$\leftarrow$"
return c, b

Example: $X = <A, B, C, B, D, A, B>$
$Y = <B, D, C, A, B, A>$

The c and b tables computed by **LCS-LENGTH**:

i \ j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Values in the rows, columns are non-decreasing.

$T(D, m, n) = \Theta(mn)$ or $\Theta(n^2)$ if m, n similar

$S(m, n) = \Theta(mn)$

The code for the algorithm can be improved.

Constructing an LCS

With the b table returned by **LCS-LENGTH**, you can quickly construct an LCS of $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$.

Begin at $b[m, n]$ and trace through the table by following the arrows. Each "↖" encountered in an entry $b[i, j]$ implies that $x_i = y_j$ is an element of the LCS that **LCS-LENGTH** found.

This method gives you the elements of this LCS in reverse order.

The best known algorithm for this problem is of Masek and Pateson:

$\Theta(n^2/\log n)$

Paper: “How to compute string-edit distances quickly.”, in ”Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparison.”, pp. ”337–349”, ”Addison Wesley”,1983.

However it is faster than the previous algorithm only when $n > 200000$.

Many problems of subsequence problem type

E.g., given a sequence of numbers $(a_1, a_2, a_3, \cdots, a_n)$ find the *longest increasing subsequence (LIS)* in it.

Example:
sequence $(10, 3, 12, 18, 30, 4, 6, 21, 7, 20)$

LIS = $(3, 4, 6, 7, 20)$

Cannot be done exhaustively; there are 2^n subsequences of a string of length n.

Construct a table
$[l_1, l_2, \ldots, l_n]$,
where l_i is the length of LIS ending with a_i.

$$l_i = 1 + \max\{l_j, 1 \le j < i, \text{ and } a_j < a_i\}$$

Here we use $\max\{\emptyset\} = 0$ if no possible l_j

For example,

input (a_i)	(	10	3	12	18	30	4	6	21	7	20	)
i		1	2	3	4	5	6	7	8	9	10	
l_i	[	1	1	2	3	4	2	3	4	4	5	]

$$l_4 = 1 + \max\{l_1, l_2, l_3\} = 1 + \max\{1, 1, 2\} = 3$$

and

$$l_{10} = 1 + \max\{l_1, l_2, l_3, l_4, l_6, l_7, l_9\} = 1 + \max\{1, 1, 2, 3, 2, 3, 4\} = 5$$

To find l_i we need to find maximum among at most $i - 1$ items.

Run-time $= \sum_{i=1}^{n} i - 1 = \frac{n(n-1)}{2}$

We can keep a table that shows how the subsequence was obtained.

Dynamic programming gives $O(n^2)$ algorithm.

There exists an $O(n \log n)$ algorithm, see

M. L. Fredman. On computing the length of longest increasing subsequences. Discrete Math., 11:29-35, 1975.

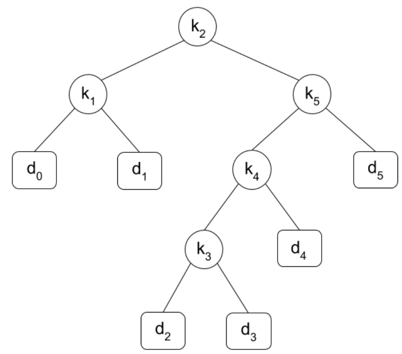
5. Optimal binary search trees (§14.5)

We have nodes and the probabilities with which the nodes are searched.

n nodes $k_1, k_2, \ldots, k_n$ such that $k_1 < k_2 < \ldots < k_n$ (distinct keys)

probabilities $p_1, p_2, \ldots, p_n$ of searching for these nodes,

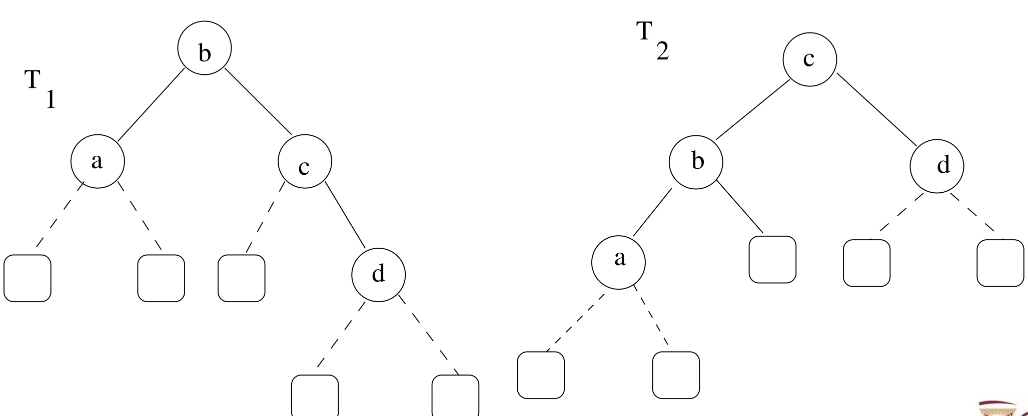
probabilities $q_0, q_1, q_2, \ldots, q_n$ of searching for elements "between" nodes (corresponds to a failed search) — we'll place these probabilities at dummy nodes $d_0, d_1, d_2, \ldots, d_n$ (leaves of the tree).



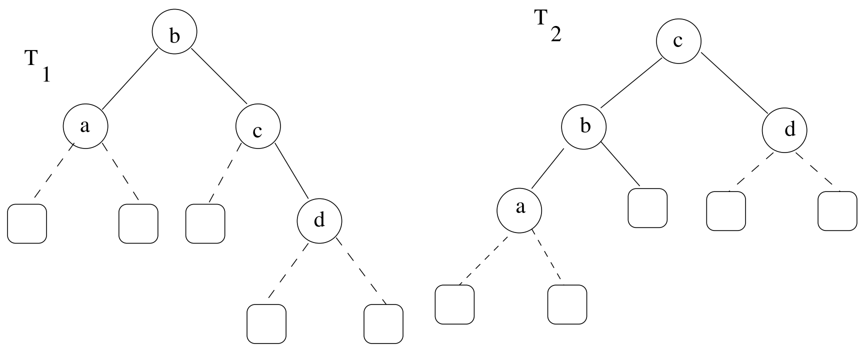
For example,

node		k_1	k_2	k_3	k_4	k_5
p_i		0.05	0.25	0.2	0.1	0.1
q_i	0.05	0.05	0.1	0.05	0.25	0.25

node		a	b	c	d
p_i		0.05	0.25	0.2	0.2
q_i	0.05	0.05	0.1	0.05	0.05



$E[T]$ the cost of a tree T expresses the expected number of comparisons corresponding to the given probabilities of searches.



Expected number of comparisons:

$E[T_1] = 0.25 + 2(0.05 + 0.2) + 3(0.05 + 0.05 + 0.1 + 0.2) + 4(0.05 + 0.05) = 2.35$

$E[T_2] = 0.2 + 2(0.25 + 0.2) + 3(3 \cdot 0.05 + 0.1) + 4(0.05 + 0.05) = 2.25$

$E[T]$ the cost of a tree T expresses the expected number of comparisons corresponding to the given probabilities of searches.

$$
\begin{aligned}
E[T] &= \sum_{i=1}^{n} p_i \cdot (depth(k_i) + 1) + \sum_{i=0}^{n} q_i \cdot (depth(d_i) + 1) \\
&= 1 + \sum_{i=1}^{n} p_i \cdot depth(k_i) + \sum_{i=0}^{n} q_i \cdot depth(d_i)
\end{aligned}
$$

since

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Find a binary search tree T that gives lowest expected cost.

Since there are an exponential number of binary trees with n nodes, exhaustive search would not yield an efficient algorithm. Instead, we use dynamic programming.

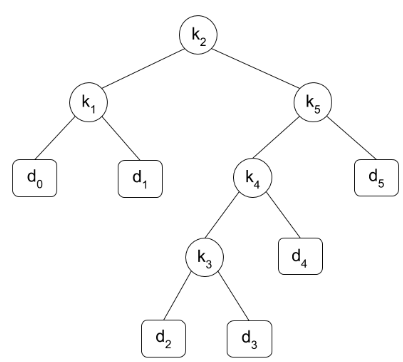
The solution is based on the **optimal substructure** of an optimal tree:

If an optimal tree T has a subtree T' with keys $k_i, k_{i+1}, \ldots, k_j$ and dummy keys $d_{i-1}, d_i, \ldots, d_j$ then T' must be also optimal for those keys.

Proof:

The usual cut-and-paste argument applies. Assume that T is an optimal tree with subtree T'. If there were a subtree T'' whose expected cost is lower than that of T', then cutting T' out of T and pasting in T'' would result in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

Let $e[1 \cdots n+1, 0 \cdots n]$ with $e[i,j]$, for $j \geq i-1$, be the expected cost of an optimal search tree for keys $k_i, k_{i+1}, \ldots, k_j$.



We wish to compute $e[1, n]$.

If k_r is the root of the **subtree** with $k_i, k_{i+1}, \ldots, k_j$,
$e[i,j] = e[i, r-1] + e[r+1, j] + w(i,j)$ where
$w(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k$, or

Let $w[1 \cdots n+1, 0 \cdots n]$ with
$w[i,j] = w[i, j-1] + p_j + q_j$, for $1 \leq i \leq j \leq n$, for sums of probabilities.

The lowest cost tree is obtained by minimizing over all possible choices of r, $i \leq r \leq j$.

$r[i,j]$ for roots of optimal tree with $k_i, k_{i+1}, \ldots, k_j$.

Optimal BST is used in cases where elements in the tree are not changing over long periods of time (static BST).

The algorithm computes three tables: $e[i,j]$, $w[i,j]$, and $r[i,j]$

Initial conditions:
$e[i,i-1] = q_{i-1}$,
$w[i,i-1] = q_{i-1}$,
for $1 \le i \le n+1$

Indexing of arrays

$$
\begin{array}{ccccccccc}
 & & & & [1,4] & & & & \\
 & & & [1,3] & & [2,4] & & & \\
 & & [1,2] & & [2,3] & & [3,4] & & \\
 & [1,1] & & [2,2] & & [3,3] & & [4,4] & \\
[1,0] & & [2,1] & & [3,2] & & [4,3] & & [5,4]
\end{array}
$$

Use the recursive relations:

$$w[i,j] = w[i,j-1] + p_j + q_j$$

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

It gives $O(n^3)$ algorithm.

It can be improved to $O(n^2)$ algorithm,
see Knuth, Acta Informatica 1:14-25, 1971

Dynamic Programming | Fibonacci Sequence | Assembly-line Sched. | Rod Cutting | Matrices Multiplication | Longest Common Subseq. | Optimal BSTs

node		*a*	*b*	*c*	*d*
i	0	1	2	3	4
p_i		0.05	0.25	0.2	0.2
q_i	0.05	0.05	0.1	0.05	0.05

$$
w:\quad
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 0.75 & & 0.9 & & & \\
 & & 0.5 & & 0.65 & & 0.6 & & \\
 & 0.15 & & 0.4 & & 0.35 & & 0.3 & \\
0.05 & & 0.05 & & 0.1 & & 0.05 & & 0.05
\end{array}
$$

e.g.,

$$
\begin{aligned}
w[1,1] &= w[1,0] + p_1 + q_1 = 0.05 + 0.05 + 0.05 = 0.15 \\
w[1,2] &= w[1,1] + p_2 + q_2 = 0.15 + 0.25 + 0.1 = 0.5 \\
&\ldots
\end{aligned}
$$

w:

1

0.75 0.9

0.5 0.65 0.6

0.15 0.4 0.35 0.3

0.05 0.05 0.1 0.05 0.05

$$
\begin{aligned}
e[1,1] &= \min\{e[1,0]+e[2,1]\} + w[1,1] = \min\{0.05+0.05\} + 0.15 = 0.25 \\
&\Rightarrow r[1,1] = 1 \\
e[2,2] &= \min\{e[2,1]+e[3,2]\} + w[2,2] = \min\{0.05+0.1\} + 0.4 = 0.55 \\
&\Rightarrow r[2,2] = 2 \\
e[1,2] &= \min\{(e[1,0]+e[2,2]), (e[1,1]+e[3,2])\} + w[1,2] \\
&= \min\{(0.05+0.55), (0.25+0.1)\} + 0.5 = 0.85 \\
&\Rightarrow r[1,2] = 2 \\
&\ldots
\end{aligned}
$$

w:

1

0.75 0.9

0.5 0.65 0.6

0.15 0.4 0.35 0.3

0.05 0.05 0.1 0.05 0.05

---

e:

2.25

1.5 1.85

0.85 1.2 1.1

0.25 0.55 0.5 0.4

0.05 0.05 0.1 0.05 0.05

---

root:

3

2 3

2 2 3

1 2 3 4