Analysis of Algorithms CECS 528

Topic 6. Dynamic programming

Dynamic Programming – general characteristic

- *N* stage optimization problem
- Development of a *recursive optimization procedure*
- First solving a one-stage problem and sequentially including one stage at a time
- Solving one-stage problems until the overall optimum has been found
- Based on a *backward induction process:*
  - the first stage to be analyzed is the final stage of the problem and
  - problems are solved moving back one stage at a time until all stages are included

Dynamic Programming – general characteristic

- Alternatively, the recursive procedure can be based on a *forward induction process:*
  - the first stage to be solved is the initial stage of the problem
  - problems are solved moving forward one stage at a time, until all stages are included.
- In certain problem settings, only one of these induction processes can be applied (e.g., only backward induction is allowed in most problems involving uncertainties).

Principle of optimality

An optimal policy has the following property:

Whatever the current state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the current decision.

Proposed by Richard Bellman in 1957.

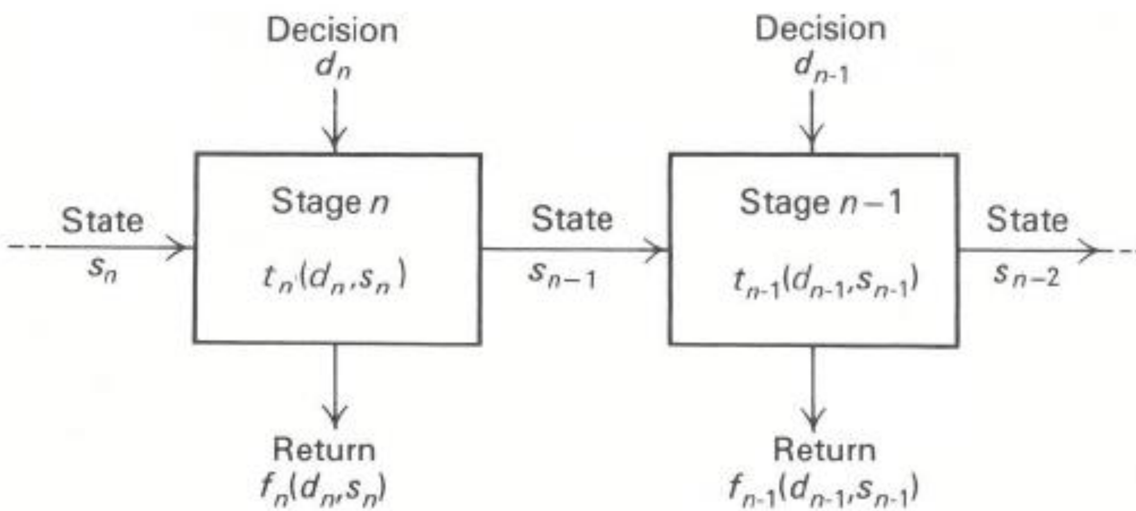
Applied to control theory, mathematical economics, machine learning, and others.

Optimization problems

- Objective function – what should be optimized
- State variables – characteristics of the current situation
- Control variables – the parameters used to be changed
- Optimal policy – decision making at each stage to achieve the optimum

Multistage decision process

For a particular stage n, there is **return** $f_n(d_n, s_n)$ where d_n is a permissible decision that may be chosen from the set D_n and s_n is the *state* of the process with n stages to go.



* - Assumption: the state s_n of the system with n stages to go is a full description of the system for decision-making purposes; the knowledge of prior states is unnecessary. The next state of the process depends entirely on the current state of the process and the current decision taken.

Transition function

The *transition* function t_n is defined such that, given s_n, the state of the process with n stages to go, the subsequent state of the process with $(n - 1)$ stages to go is given by

$$s_{n-1} = t_n(d_n, s_n),$$

where d_n is the decision chosen for the current stage and state.

Optimization

- Objective is to maximize the sum of the return functions (or minimize the sum of cost functions) over all stages of the decision process.
- Constraints are that the decision chosen for each stage belong to some set D_n of permissible decisions.
- *G*iven that the process is in state s_n with n stages to go, the optimization problem is to choose the decision variables d_n, $d_{n-1}, \ldots, d_0$ to solve the following problems:

$$v_n(s_n) = \text{Max}\left[f_n(d_n, s_n) + f_{n-1}(d_{n-1}, s_{n-1}) + \cdots + f_0(d_0, s_0)\right],$$

$$s_{m-1} = t_m(d_m, s_m) \qquad (m = 1, 2, \ldots, n),$$

$$d_m \in D_m \qquad (m = 0, 1, \ldots, n).$$

Optimization

Since $f_n(d_n, s_n)$ involves only the decision variable d_n and not the decision variables $d_{n-1}, \ldots, d_0$, it is possible to first maximize over this latter group for every possible d_n and then choose d_n so as to maximize the entire expression.

$$
\begin{aligned}
v_n(s_n) = \text{Max}\Big\{ & f_n(d_n, s_n) \quad + \quad \text{Max}\left[f_{n-1}(d_{n-1}, s_{n-1}) + \cdots + f_0(d_0, s_0) \right] \Big\}, \\
& \text{subject to:} \qquad\qquad\quad \text{subject to:} \\
& s_{n-1} = t_n(d_n, s_n) \qquad s_{m-1} = t_m(d_m, s_m) \qquad (m = 1, 2, \ldots, n-1), \\
& d_n \in D_n, \qquad\qquad\qquad d_m \in D_m, \qquad\qquad\quad (m = 0, 1, \ldots, n-1).
\end{aligned}
$$

Recursive formulation

$$v_n(s_n) = \text{Max}\,\{f_n(d_n, s_n) + v_{n-1}[t_n(d_n, s_n)]\},$$

$$d_n \in D_n.$$

This is a formal statement of the *principal of optimality*:

"..an optimal sequence of decisions for a multistage problem has the property that, regardless of the current decision d_n and current state s_n, all subsequent decisions must be optimal, given the state s_{n-1} resulting from the current decision. "

Stage-zero problem

It is necessary to initiate the computation by solving the "stage-zero" problem. The stage-zero problem is not defined recursively, since there are no more stages after the final stage of the decision process. The stage-zero problem is then the following:

$$v_0(s_0) = \text{Max } f_0(d_0, s_0),$$

$$d_0 \in D_0.$$

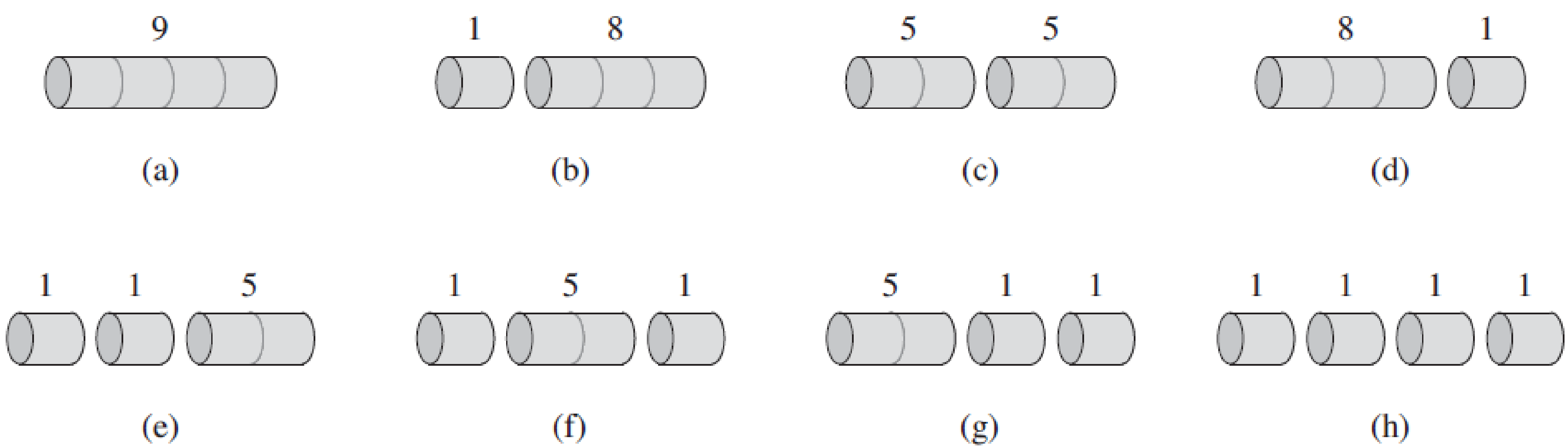
Often there *is no stage-zero problem,* as $v_0(s_0)$ is identically zero for all final stages.

Rod-cutting problem

Given a rod of length n inches and a table of prices p_i for $i=1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Example

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



The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive top-down implemenation

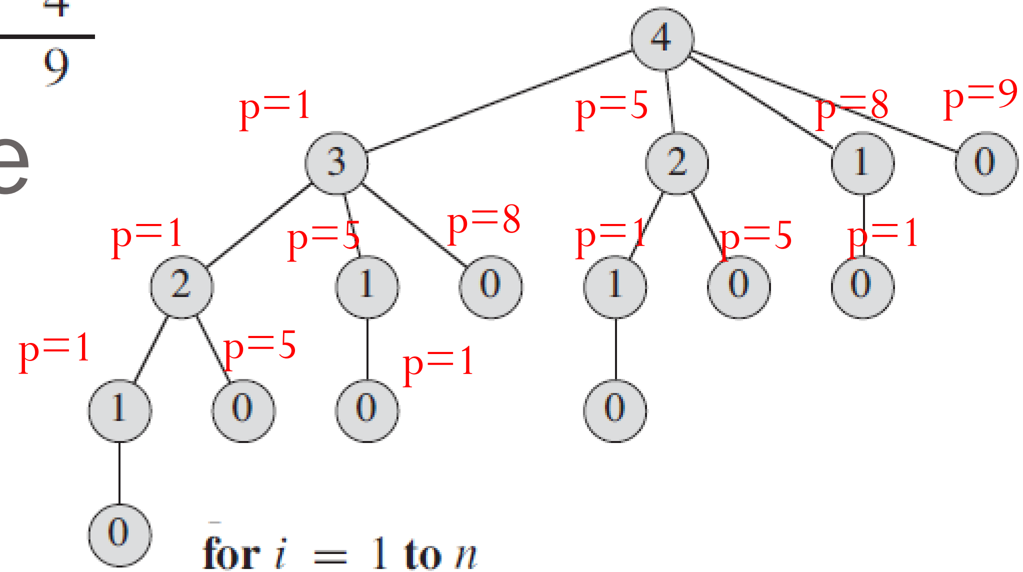
```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Procedure CUT-ROD takes as input an array $p[1..n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n - i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using

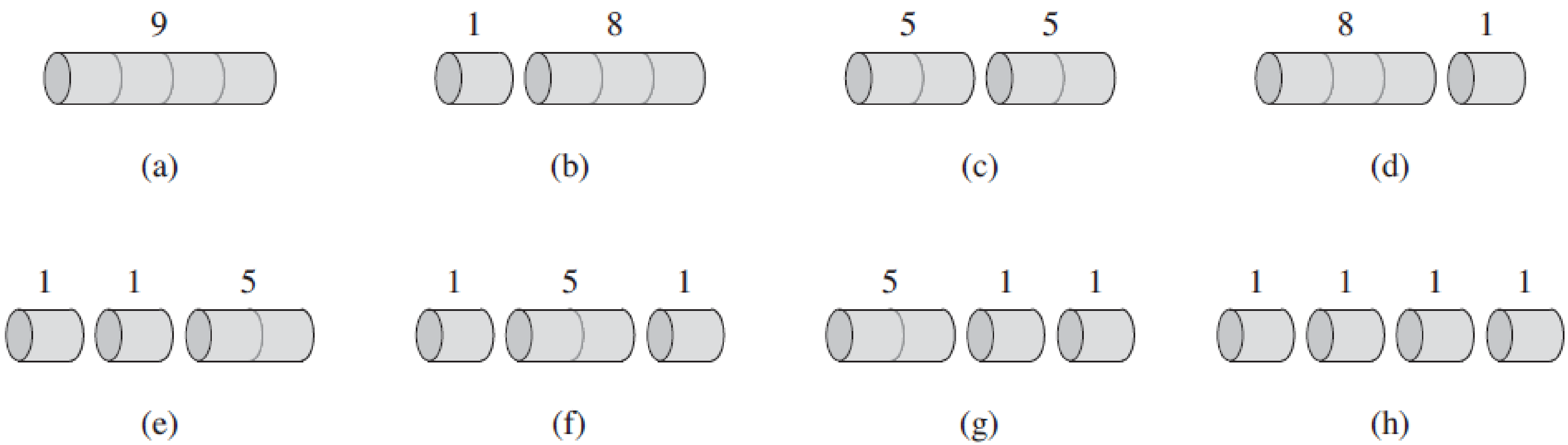
$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \ .$$

length i	1	2	3	4
price p_i	1	5	8	9

Recursion tree

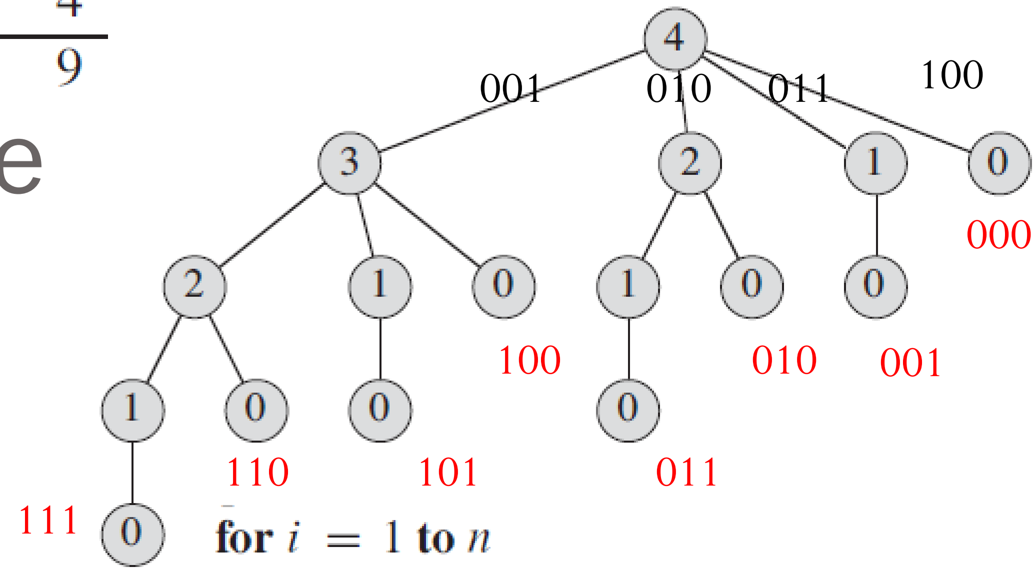


for $i = 1$ **to** n
$q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$



length i	1	2	3	4
price p_i	1	5	8	9

Recursion tree



for $i = 1$ **to** n
$q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$

The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for n=4. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size s - t and leaving a remaining subproblem of size t . A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n.

In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

CUT-ROD is inefficient

Once the input size becomes moderately large, the program would take a long time to run.

For n = 40, the program can take at least several minutes, and most likely more than an hour.

In fact, each time n increased 1, the program's running time would approximately double.

CUT-ROD(p, n) calls CUT-ROD(p, n-i) for i = 1,2, …,n.

Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each j=0,1,…, n - 1.

When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

Total number of calls $T(n)$

$T(0) = 1$: the initial call at its root.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD(p, n - i), where $j = n - i$.

$$T(n) = 2^n$$

So, the running time of CUT-ROD is exponential in n.

Make it work!

- If we can *store* the solutions to the smaller problems in a *bottom-up* manner rather than recompute them, the run time can be drastically improved (at the cost of additional memory usage).
- To implement this approach we simply solve the problems starting for smaller lengths and *store* these optimal revenues in an *array* (of size $n+1$).
- Then when evaluating longer lengths we simply *look-up* these values to determine the optimal revenue for the larger piece.

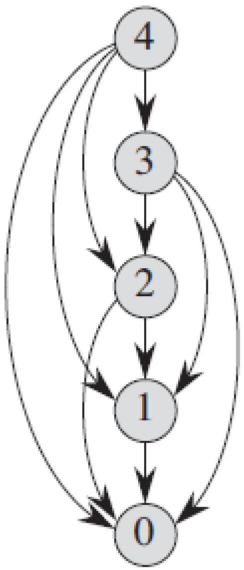
The bottom-up version

length i	1	2	3	4
price p_i	1	5	8	9

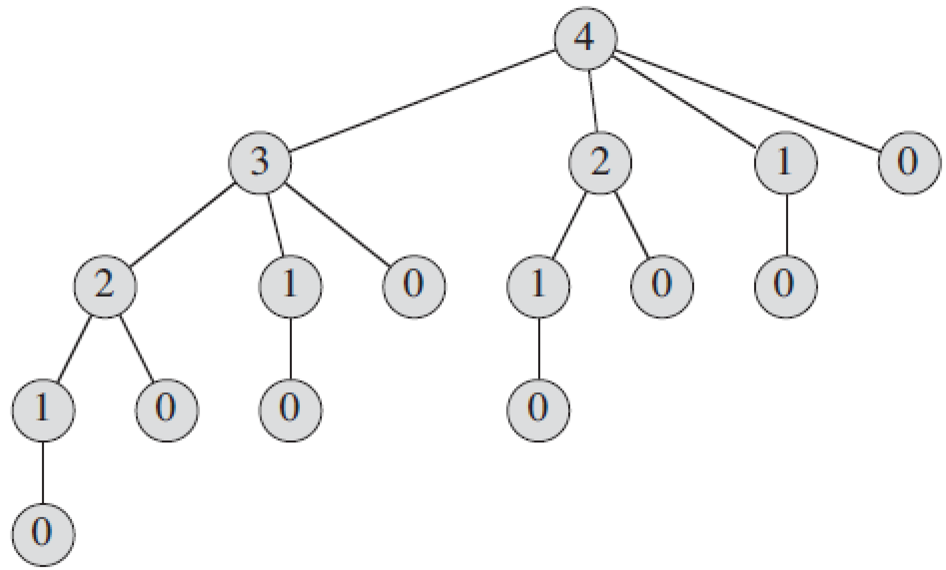
```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Collapse the nodes



VS.



The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure.

Extended bottom-up cut

```
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 = -∞
    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
```

Print solution

The following procedure takes a price table p and a rod size n, and it calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1 \,..\, n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

```
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]
```

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
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Difficulties with recursiveness

- When there is a recursive algorithm based on subproblems but the total number of subproblems is not too great, it is possible to cache (*memoize*) all solutions in a table.
  - Fibonacci numbers.
  - Binomial coefficients.
- (In both cases, the algorithm is by far not as good as computing the known formulas.)

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

```
int fib(int n) {
  int f[n+1];
  f[1] = f[2] = 1;
  for (int i = 3; i <= n; i++)
   f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

Longest Common Subsequence Problem

- The diff program in Unix: what does it mean to say that we find the places where two files differ (including insertions and deletions)? Or, what does it mean to keep the "common" parts?
- Let it mean the longest subsequence present in both:

$$
\begin{array}{lccccccccc}
X = & a & b & & c & & b & & d\,a\,b \\
Y = & & b & d & c & a & b & a & b\,a \\
 & & b & & c & & b & & \,a
\end{array}
$$

- Running through all subsequences would take exponential time. There is a faster solution, recognizing that we only want to find some longest subsequence.
- Let $c[i,j]$ be the length of the longest common subsequence of the prefix of $X[1..i]$ and $Y[1..j]$. Recursion:

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

We are not computing a function $C(X,Y,i,j)$ by naive recursion, but collect the values $c[i,j]$ as they are computed, in array c: $C(X,Y,i,j,c)$ checks whether $c[i,j]$ is defined. If yes, it just returns $c[i,j]$; else it uses the above recursion, and assigns $c[i,j]$ before returning the value.

```
X = a b   c   b   d a b
Y =   b d c a b a b a
      b   c   b     a
```

Non-recursive implementation

- Compute the table $c[i, j]$ "bottom-up".
- Also, store the value $b[i, j] = \{\leftarrow\}, \{\uparrow\}$ or $\{\nwarrow\}$ depending on whether the optimum is $c[i, j-1]$, $c[i-1, j]$ or $c[i-1, j-1]+1$. (We also use the value $\{\leftarrow, \uparrow\}$ in case $c[i, j] = c[i-1, j] = c[i, j-1]$, though this is not important.)
- Find the longest common subsequence walking backwards on the arrows.

	j	1	2	3	4	5	6
i		b	d	c	a	b	a
1	a	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	b	↖ 1	← 1	← 1	↑ ← 1	↖ 2	← 2
3	c	↑ 1	↑ ← 1	↖ 2	← 2	↑ ← 2	↑ ← 2
4	b	↖ 1	↑ ← 1	↑ 2	↑ ← 2	↖ 3	← 3
5	d	↑ 1	↖ 2	↑ ← 2	↑ ← 2	↑ 3	↑ ← 3
6	a	↑ 1	↑ 2	↑ ← 2	↖ 3	↑ ← 3	↖ 4
7	b	↖ 1	↑ 2	↑ ← 2	↑ 3	↖ 4	↑ ← 4

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

- Compute the table $c[i,j]$ "bottom-up".
- Also, store the value $b[i,j] = \{\leftarrow\}, \{\uparrow\}$ or $\{\nwarrow\}$ depending on whether the optimum is $c[i,j-1]$, $c[i-1,j]$ or $c[i-1,j-1]+1$. (We also use the value $\{\leftarrow, \uparrow\}$ in case $c[i,j] = c[i-1,j] = c[i,j-1]$, though this is not important.)
- Find the longest common subsequence walking backwards on the arrows.

LCS-length(X,Y)

```
m ← X.length; n ← Y.length
b[1..m, 1..n], c[0..m, 0..n] ← new tables
for i = 1 to m do c[i,0] ← 0
for j = 1 to n do c[0,j] ← 0
for i = 1 to m do
    for j = 1 to n do
        if x_i = y_j then
            c[i,j] ← c[i−1, j−1] + 1
            b[i,j] ← {↖}
        else
            c[i,j] ← max(c[i, j−1], c[i−1, j]); b[i,j] ← ∅
            if c[i−1, j] = c[i,j] then b[i,j] ← {↑}
            if c[i, j−1] = c[i,j] then b[i,j] ← b[i,j] ∪ {←}
return c, b
```

LCS-Print(b, X, i, j)

Algorithm 11.2: LCS-PRINT(b, X, i, j)

Print the longest common subsequence of $X[1..i]$ and $Y[1..j]$ using the table b computed above.

if $i = 0$ **or** $j = 0$ **then return**
if $\nwarrow \in b[i,j]$ **then** LCS-PRINT$(b, X, i-1, j-1)$; print x_i
else if $\uparrow \in b[i,j]$ **then** LCS-PRINT$(b, X, i-1, j)$
else LCS-PRINT$(b, X, i, j-1)$

Longest Common Subsequence Problem

The recursive relation for the Longest Common Subsequence Problem (for two strings $x_1 \ldots x_n$ and $y_1 \ldots y_m$) is

$$lcs(x_1 \ldots x_i, y_1 \ldots y_j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ lcs(x_1 \ldots x_{i-1}, y_1 \ldots y_{j-1}) + 1, & \text{if } x_i = y_j \\ \max\{lcs(x_1 \ldots x_{i-1}, y_1 \ldots y_j), & \\ lcs(x_1 \ldots x_i, y_1 \ldots y_{j-1})\}, & \text{otherwise} \end{cases}$$

Use the example XMJYAUZ and MZJAWXU. Obviously, this algorithm can be used to find the longest common subsequence in two DNA strings as well (AGCT).

Matrix chain multiplication

1. Let's assume that we have n matrices that need to be multiplied together: $A_1A_2A_3\ldots A_n$. (They are not all necessary the same sizes.) As an example, let's take three matrices that have sizes A_1:(10,100), A_2:(100,5), A_3:(5,50). When we multiply all three together we are guaranteed to get a matrix of size A:(10,50). The question is: how many single-register multiplications does it take to get A? Note that if we multiply matrices of size $m_1 \times m_2$ by $m_2 \times m_3$ (without using a fancy algorithm like Strassen's), we are required to do $m_1m_2m_3$ total single-register multiplications.

Matrix chain multiplication

2. If we multiply the first two together, we get (10)(5)=50 entries at 100 multiplications per entry for a total of 5000 multiplications. If we then take this (10,5) matrix and multiply it by the (5,50) matrix, we will get (10)(50) entries at 5 multiplications per entry for a total of 2500 multiplications. So it took a total of 7500 multiplications total to multiply the three. If we multiply them in the other order, it should take 10 times as long. The order is called a *parenthesization* (and it must uniquely determine the order).

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

A1:(10,100); A2:(100,5); A3(5, 50)

B1=A1*A2:(10,5) - 5000 mult.; B2=B1*A3:(10,50) - 2500 mult. Total: 7500

B1=A2*A3:(100,50) - 25000 mult.; B2=B1*A3:(10,50) - 50000 mult. Total: 75,000

Matrix chain multiplication

3. Given n matrices to be multiplied together in order, what order should we use to perform the multiplication? We are going to define $m[i,j]$ to be the number of multiplications necessary to compute the product $A_i \ldots A_j$ in the optimal case. Then we get the following recursive relation for $m[i,j]$ for $i \leq j$:

$$m[i,j] = \begin{cases} 0, & \text{if } i = j \\ \min_{\{i \leq k < j\}} \{m[i,k] + m[k+1,j] + \text{cost}\}, & \text{otherwise} \end{cases}$$

where cost represents the cost of multiplying together the results of $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$.

Matrix chain multiplication

Consider an example:

$A_1 : (3, 5), A_2 : (5, 4), A_3 : (4, 1), A_4 : (1, 9)$

MATRIX-CHAIN-ORDER(p)

```
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        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
return m and s
```

The knapsack problem

Given: volumes $b \geqslant a_1, \ldots, a_n > 0$, and integer values $w_1 \geqslant \cdots \geqslant w_n > 0$.

$$\begin{array}{ll} \text{maximize} & w_1x_1 + \cdots + w_nx_n \\ \text{subject to} & a_1x_1 + \cdots + a_nx_n \leqslant \quad b, \\ & \qquad\qquad\qquad x_i = 0, 1,\ i = 1, \ldots, n. \end{array}$$

In other words, find a subset $i_1 < \cdots < i_k$ of the set of items $1, \ldots, n$ (by choosing which $x_i = 1$) such that

- the sum of their volumes $a_{i_1} + \cdots + a_{i_k}$ is less than the volume b of our knapsack,
- the sum of their values $w_{i_1} + \cdots + w_{i_k}$ is maximal.

Special cases

Subset sum problem find $i_1, \ldots, i_k$ with $a_{i_1} + \cdots + a_{i_k} = b$.
Obtained by setting $w_i = a_i$. Now if there is a solution with value b, we are done.

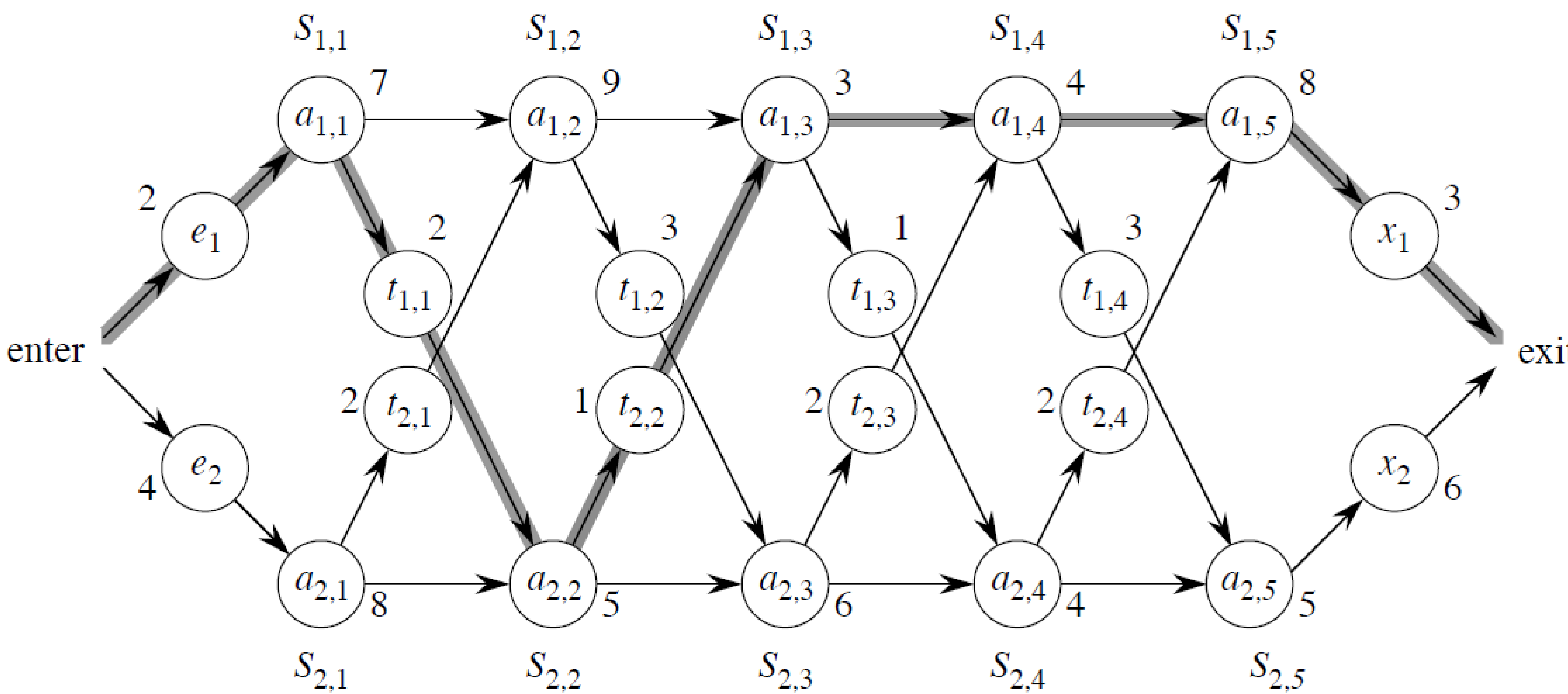
Partition problem Given numbers $a_1, \ldots, a_n$, find $i_1, \ldots, i_k$ such that $a_{i_1} + \cdots + a_{i_k}$ is as close as possible to $(a_1 + \cdots + a_n)/2$.

Dynamic programming - summary

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 in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Assembly-line scheduling



Assembly-line scheduling

Automobile factory with two assembly lines.

- Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$.
- Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$.
- Entry times e_1 and e_2.
- Exit times x_1 and x_2.
- After going through a station, can either
  - stay on same line; no cost, or
  - transfer to other line; cost after $S_{i,j}$ is $t_{i,j}$. ($j = 1, \ldots, n-1$. No $t_{i,n}$, because the assembly line is done after $S_{i,n}$.)

Problem: Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has n stations.
- 2^n subsets.
- Infeasible when n is large.

Structure of an optimal solution

Think about fastest way from entry through $S_{1,j}$.

- If $j = 1$, easy: just determine how long it takes to get through $S_{1,1}$.
- If $j \geq 2$, have two choices of how to get to $S_{1,j}$:
  - Through $S_{1,j-1}$, then directly to $S_{1,j}$.
  - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$.

Suppose fastest way is through $S_{1,j-1}$.

Key observation

What is the fastest way from entry through $S_{1,j-1}$ in this solution?

If there were a faster way through $S_{1,j-1}$, we would use it instead to come up with a faster way through $S_{1,j}$.

Now suppose a fastest way is through $S_{2,j-1}$. Again, we must have taken a fastest way through $S_{2,j-1}$. Otherwise use some faster way through $S_{2,j-1}$ to give a faster way through $S_{1,j}$

Optimal structure

Generally: An optimal solution to a problem (fastest way through $S_{1,j}$) contains within it an optimal solution to subproblems (fastest way through $S_{1,j-1}$ or $S_{2,j-1}$).

This is optimal substructure.

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

Fastest way through $S_{1,j-1}$ is either

- fastest way through $S_{1,j-1}$ then directly through $S_{1,j}$, or
- fastest way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$.

Optimal structure

Symmetrically:

Fastest way through $S_{2,j}$ is either

- fastest way through $S_{2,j-1}$ then directly through $S_{2,j}$, or
- fastest way through $S_{1,j-1}$, transfer from line 1 to line 2, then through $S_{2,j}$.

Therefore, to solve problems of finding a fastest way through S1, j and S2, j , solve subproblems of finding a fastest way through $S_{1,j-1}$ and $S_{2,j-1}$.

Recursive solution

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \ldots, n$.

Goal: fastest time to get all the way through $= f^*$.

$$
\begin{aligned}
f^* &= \min(f_1[n] + x_1,\ f_2[n] + x_2) \\
f_1[1] &= e_1 + a_{1,1} \\
f_2[1] &= e_2 + a_{2,1}
\end{aligned}
$$

For $j = 2, \ldots, n$:

$$
\begin{aligned}
f_1[j] &= \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) \\
f_2[j] &= \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})
\end{aligned}
$$

$f_i[j]$ gives the *value* of an optimal solution. What if we want to *construct* an optimal solution?

Recursive solution

- $l_i[j]$ = line # (1 or 2) whose station $j-1$ is used in fastest way through $S_{i,j}$.
- In other words $S_{l_i[j],j-1}$ precedes $S_{i,j}$.
- Defined for $i = 1, 2$ and $j = 2, \ldots, n$.
- l^* = line # whose station n is used.

For example:

j	1	2	3	4	5
$f_1[j]$	9	18	20	24	32
$f_2[j]$	12	16	22	25	30

$f^* = 35$

j	2	3	4	5
$l_1[j]$	1	2	1	1
$l_2[j]$	1	2	1	2

$l^* = 1$

Go through optimal way given by l values

Compute an optimal solution

Could just write a recursive algorithm based on above recurrences.

- Let $r_i(j)$ = # of references made to $f_i[j]$.
- $r_1(n) = r_2(n) = 1$.
- $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$ for $j = 1, \ldots, n-1$.

Claim

$r_i(j) = 2^{n-j}$.

Proof Induction on j, down from n.

Basis: $j = n$. $2^{n-j} = 2^0 = 1 = r_i(n)$.

Inductive step: Assume $r_i(j+1) = 2^{n-(j+1)}$.

$$\text{Then } r_i(j) \;=\; r_i(j+1) + r_2(j+1) \;=\; 2^{n-(j+1)} + 2^{n-(j+1)} \;=\; 2^{n-(j+1)+1} \;=\; 2^{n-j}\,.$$

■ (claim)

Therefore, $f_1[1]$ alone is referenced 2^{n-1} times!

So top down isn't a good way to compute $f_i[j]$.

Observation

$f_i[\,j\,]$ depends only on $f_1[\,j-1]$ and $f_2[\,j-1]$

for $j \geq 2$.

So compute in order of increasing j.

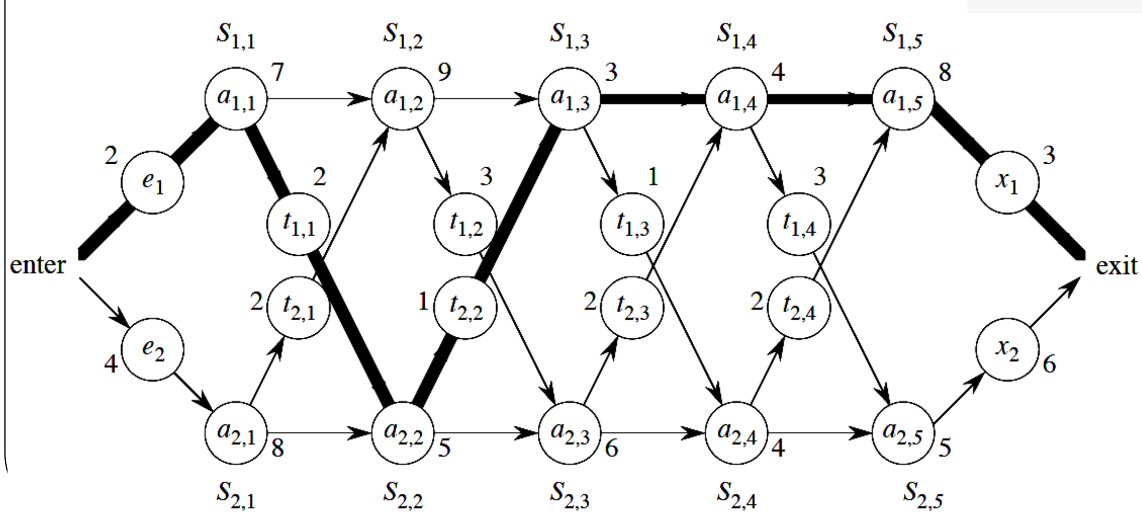
FASTEST-WAY(a, t, e, x, n)

$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ **to** 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 $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
      $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \leq f_2[n] + x_2$
  then $f^* = f_1[n] + x_1$
    $l^* = 1$
  else $f^* = f_2[n] + x_2$
    $l^* = 2$

Go through example.

FASTEST-WAY(a, t, e, x, n)

$f_1[1] \leftarrow e_1 + a_{1,1}$
$f_2[1] \leftarrow e_2 + a_{2,1}$
for $j \leftarrow 2$ **to** 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 $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
      $l_2[j] \leftarrow 1$
if $f_1[n] + x_1 \leq f_2[n] + x_2$
  then $f^* = f_1[n] + x_1$
    $l^* = 1$
  else $f^* = f_2[n] + x_2$
    $l^* = 2$



Constructing an optimal solution

PRINT-STATIONS(l, n)

$i \leftarrow l^*$

print "line " i ", station " n

for $j \leftarrow n$ **downto** 2

  do $i \leftarrow l_i[j]$

    print "line " i ", station " $j - 1$

Constructing an optimal solution

PRINT-STATIONS(l, n)

$i \leftarrow l^*$

print "line " i ", station " n

for $j \leftarrow n$ **downto** 2

  do $i \leftarrow l_i[j]$

    print "line " i ", station " $j - 1$

Go through example.

Time $= \Theta(n)$