Dynamic Programming: Assembly-line scheduling

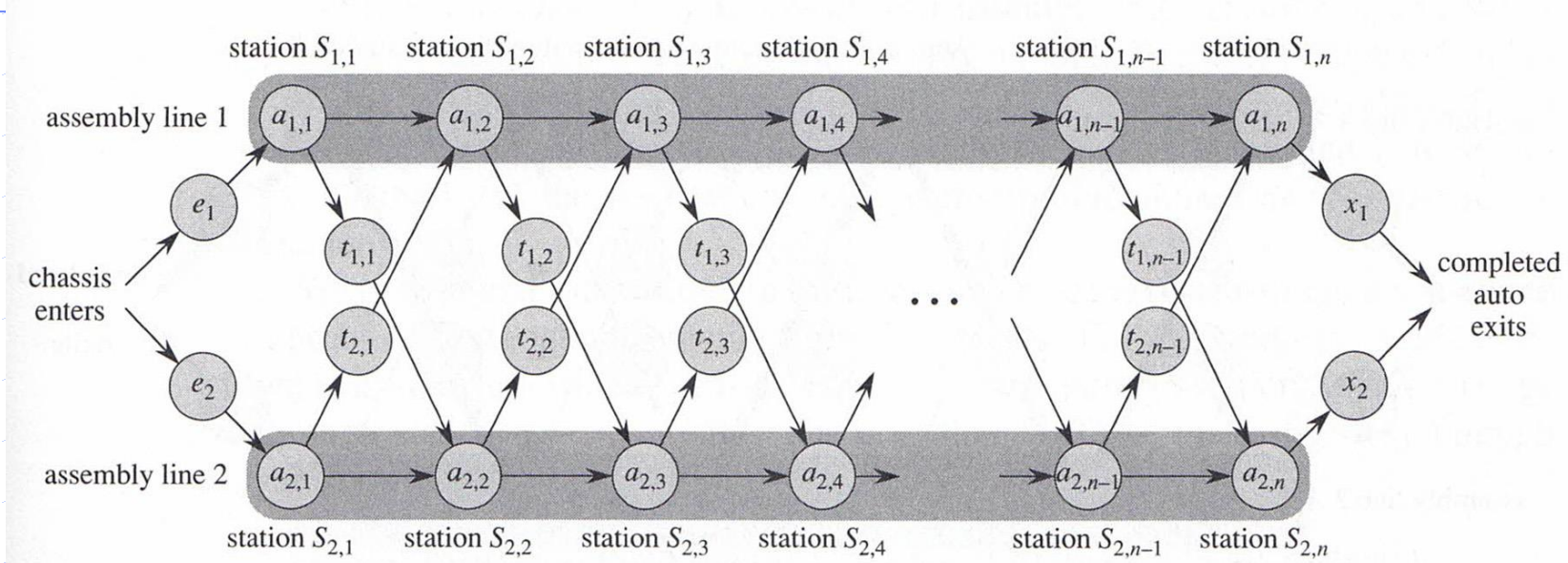
Chapter 15.1

Dynamic Programming

- A method for solving optimization problems: problems that ask for a minimum or maximum value.
- Developing a dynamic programming solution:
  1. Characterize structure of an optimal solution
     a. Optimal substructure
     b. Overlapping subproblems
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution, avoiding overlap
  4. Construct an optimal solution from computed values.

Assembly-line scheduling

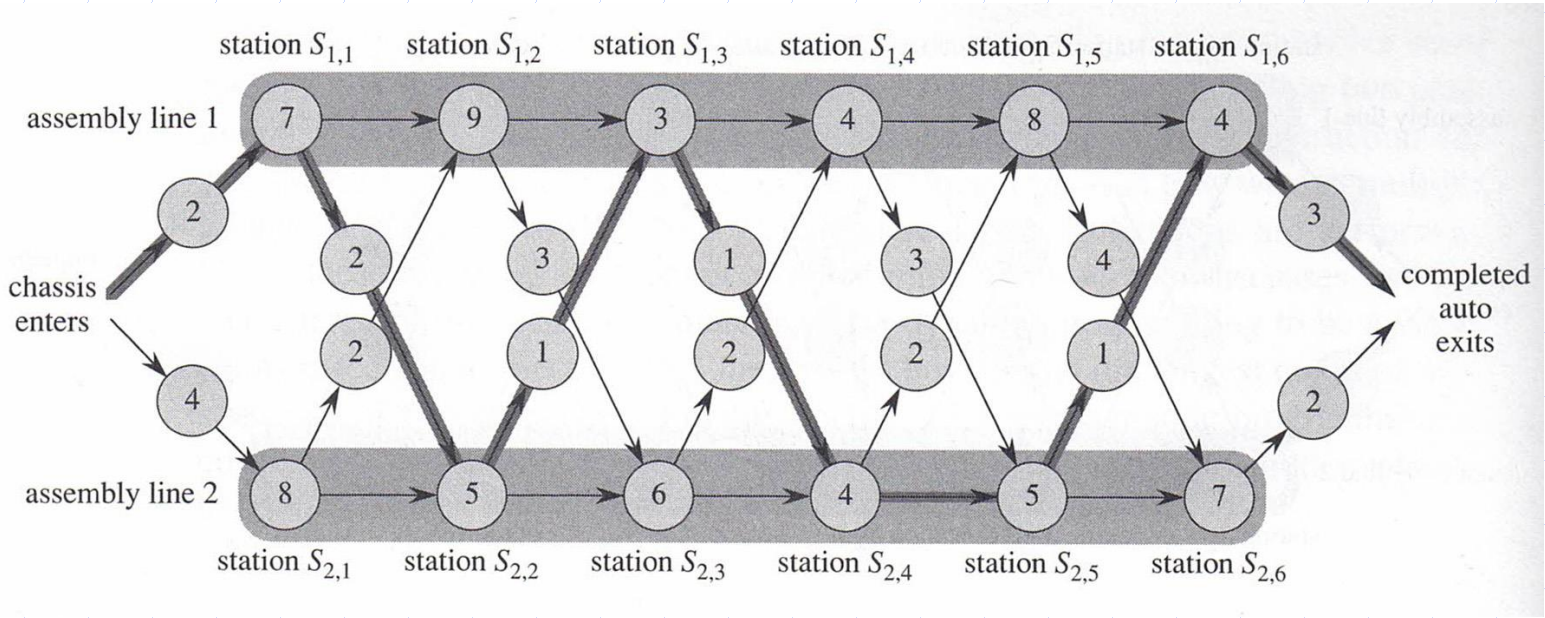
(2nd ed. §15.1)



Two assembly lines, with processing and transfer times. Stations $S_{1,j}$ and $S_{2,j}$ do the same job.

What is the fastest way from start to finish?

Assembly-line example



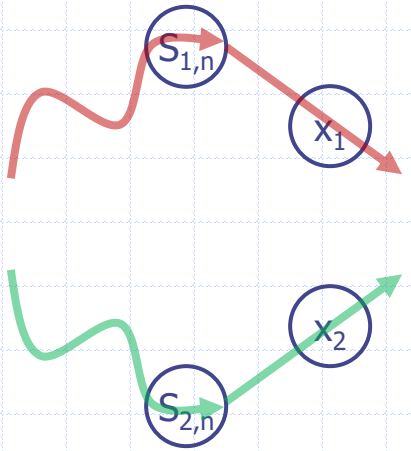
In this example, the fastest way through uses $S_{1,1}$, $S_{2,2}$, $S_{1,3}$, $S_{2,4}$, $S_{2,5}$, and $S_{1,6}$

Assembly-line scheduling

1. Characterize structure of an optimal solution
    a. Optimal substructure
    b. Overlapping subproblems

The best solution is the quickest one of

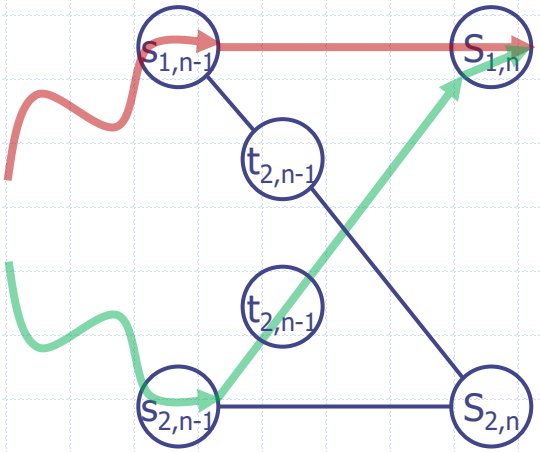
- getting through $S_{1,n}$ as quickly as possible, followed by going through the line one exit.
- getting through $S_{2,n}$ as quickly as possible, followed by going through the line two exit.



Assembly-line scheduling

Getting through $S_{1,n}$ as quickly as possible is accomplished by the quickest one of:

- getting through $S_{1,n-1}$ as quickly as possible, followed by going through $S_{1,n}$
- getting through $S_{2,n-1}$ as quickly as possible, followed by going through the transfer from line two to line one, followed by going through $S_{1,n}$

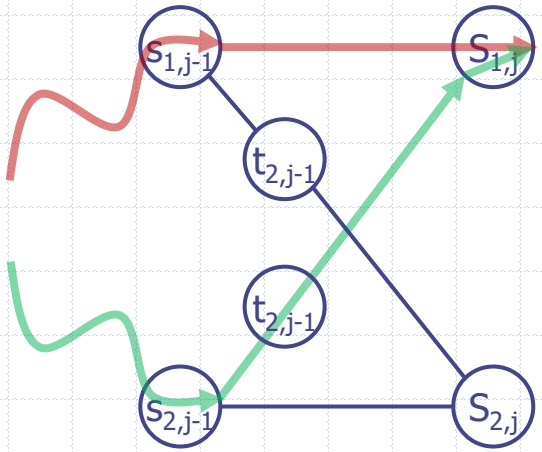


Getting through $S_{2,n}$ as quickly as possible is symmetric.

Assembly-line scheduling

Getting through $S_{1,j}$ (where $j > 1$) as quickly as possible is accomplished by the quickest one of:

- getting through $S_{1,j-1}$ as quickly as possible, followed by going through $S_{1,j}$
- getting through $S_{2,j-1}$ as quickly as possible, followed by going through the transfer from line two to line one, followed by going through $S_{1,j}$



Getting through $S_{2,j}$ as quickly as possible is symmetric.

Optimal substructure

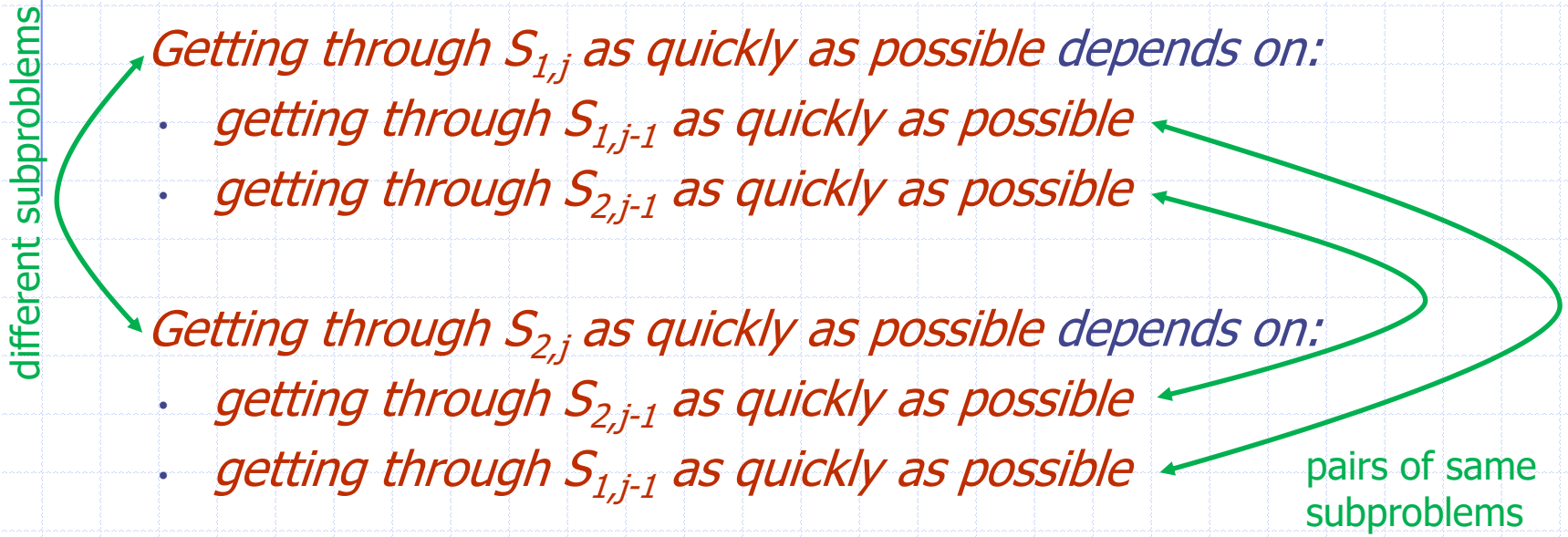
This problem has optimal substructure: the subproblems solved are of the same type and must be solved optimally.

Getting through $S_{1,j}$ as quickly as possible is accomplished by the quickest one of:

- *getting through $S_{1,j-1}$ as quickly as possible, followed by going through $S_{1,j}$*
- *getting through $S_{2,j-1}$ as quickly as possible, followed by going through the transfer from line two to line one, followed by going through $S_{1,j}$*

Overlapping subproblems

This problem has overlapping subproblems: different subproblems require the same subproblem(s) in their solution.



Recursive definition

2. Recursively define the value of an optimal solution

Assume we are given matrices s[1..2,1..n], t[1..2,1..n-1], e[1..2] and x[1..2] defining the problem.

Let f[i, j] denote the time taken from the start in the quickest way of getting through $S_{i,j}$

$$f[i, j] = \min(f[i, j-1], f[3-i, j-1] + t[3-i, j-1]) + s[i, j] \quad \text{(for } j > 1\text{)}$$

$$f[i, 1] = e[i] + s[i, 1]$$

Recursive definition

The value of an optimal solution is

min(f[1, n] + x[1], f[2, n] + x[2]).

By implementing f as a function call, we now have a recursive algorithm for our problem:

```
solution() { return min(f(1,n) + x[1], f(2,n) + x[2]);}
f(i,j) {
    if (j=1)
            return e[i] + s[i,1];
    else
            return min(f(i,j-1), f(3-i, j-1) + t[3-i, j-1]) + s[i, j]
```

Analysis of straight recursion

Let T(j) denote the time taken for function f(i, j).
Then the time for the entire solution is O(1) + T(n).

$T(j) = c$ if $j = 1$

$T(j) = 2T(j-1) + c$ if $j > 1$

$T(j) = (2^j-1)c$ or $\Theta(2^j)$

So the time for the entire solution is $\Theta(2^n)$. Exponential is bad. But this does not take into account the overlapping subproblems.

Memoization

3. Compute the value of an optimal solution, avoiding overlap.

```
solution() {
        allocate matrix m[1..2, 1..n] = 0                    // memos
        return min(f(1,n) + x[1], f(2,n) + x[2]);
}
f(i,j) {
        if( m[i,j] ≠ 0 ) return m[i, j];
        if (j=1)
                m[i,j] = e[i] + s[i,1];
        else
                m[i,j] = min(f(i,j-1), f(3-i, j-1) + t[3-i, j-1]) + s[i, j];
        return m[i, j];
}
```

Analysis of memoization

Allocating m[] takes O(n) or O(1) time depending on model.

Consider all calls to f(i, j). Let k be the number of such calls.

Then k – 2n of them return inside the first **if**, taking O(1) time each. (Because m[] holds 2n values and each time through the rest of the function fills in 1 previously unfilled value.)

For the remaining 2n calls, there is O(1) nonrecursive work apiece. (The recursive work is counted in the "consider all calls".)

We conclude that the total work over all calls to f(i,j) is

$$(k-2n) \cdot O(1) + 2n \cdot O(1)$$

$$= O(k) + O(n).$$

Analysis of memoization

So what is k?

solution() calls f(i, j) twice.

f(i, j) passes the first **if** 2n times, and each time this happens it has the potential to call f(i, j) twice.

Thus the total number of calls, k, is at most 4n+2.

The total work of the algorithm is therefore the total work for solution() plus the total work in f(i,j), or

$$O(n) + (O(k) + O(n))$$
$$= O(n) + (O(n) + O(n))$$
$$= O(n).$$

That's a far sight better than $\Theta(2^n)$.

Memoization traceback

4. Construct an optimal solution from computed values.

```
solution() {
        allocate matrix m[1..2, 1..n] = 0          // memos
        if( f(1,n) + x[1] < f(2,n) + x[2]) {       // note this fills memo table
                return traceback(1, n) and f(1, n) + x[1];
        }
        else {
                return traceback(2, n) and f(2, n) + x[2];
        }
}
```

Memoization traceback

```
traceback(i, j) {
        if (j=1) {
                return (S_{i,j})                    // by (S_{i,j}) I mean "station i,j"
        }                                           // however you encode it.

        if( m[i,j] = m[i, j-1] + s[i, j]) {
                path = traceback(i, j-1)
        }
        else {
                path = traceback(3-i, j-1)
        }
        return path + (S_{i,j})
}
```

Storing choices

If the green condition on the previous slide is slow to compute, you can alternatively store your choices along the way.

```
solution() {
        allocate matrix m[1..2, 1..n] = 0          // memos
        allocate matrix ch[1..2, 1..n] = 0         // choices
        if( f(1,n) + x[1] < f(2,n) + x[2]) {       // note this fills memo table
                return traceback(1, n) and f(1, n) + x[1];
        }
        else {
                return traceback(2, n) and f(2, n) + x[2];
        }
}
```

Storing choices

```
f(i,j) {
        if( m[i,j] ≠ 0 ) return m[i, j];
        if (j=1)
            m[i,j] = e[i] + s[i,1];
        else {
            pathOneTime = f(i, j-1) + s[i, j];
            pathTwoTime = f(3-i, j-1) + t[3-i, j-1] + s[i, j];
            if(pathOneTime < pathTwoTime) {
                    m[i, j] = pathOneTime;
                    ch[i, j] = i;
            }
            else {
                    m[i, j] = pathTwoTime;
                    ch[i, j] = 3-i;
            }
        }
        return m[i, j];
}
```

Storing choices

Now, tracing back through the stored choices is easy:

```
traceback(i, j) {
        if (j=1) {
                return (S_{i,j})                    // by (S_{i,j}) I mean "station i,j"
        }                                           // however you encode it.

        return traceback(ch[i, j], j-1) + (S_{i,j})
}
```

Dynamic Programming

Dynamic programming is computing the memos without recursion. Typically it is "bottom-up" (e.g. starting at j=1) rather than "top-down" (e.g. starting at j=n).

```
solution() {
        allocate matrix m[1..2, 1..n]
        for(j=1 to n)
            for(i = 1 to 2)
                m[i, j] = f(i, j);
        return min(m[1,n] + x[1], m[2,n] + x[2]);
}
f(i, j) {
    if(j = 1)    return e[i] + s[i, j];
    else
        return min(m[i,j-1], m[3-i, j-1] + t[3-i, j-1]) + s[i, j];
}
```

Equivalent pseudocode

TIMTOWTDI. There is more than one way to do it.

Sometimes you might see DP written without the recursively-formulated function.

```
solution() {
        allocate matrix m[1..2, 1..n]

        m[1, 1] = e[1] + s[1, 1]
        m[2, 1] = e[2] + s[2, 1]

        for(j=2 to n)
            for(i = 1 to 2)
                m[i, j] = min(m[i,j-1], m[3-i, j-1] + t[3-i, j-1]) + s[i, j];

        return min(m[1,n] + x[1], m[2,n] + x[2]);
}
```

Dynamic Programming Traceback

Traceback can be done the same way for DP as was done for memoization: the traceback() function is the same. One can also use the method of storing choices.

Exercise: write a modification of the previous version of solution() that stores the choices made.

(solve exercise before viewing next two slides)

Dynamic Programming Traceback

```
solution() {
        allocate matrix m[1..2, 1..n]
        allocate matrix ch[1..2, 1..n]

        m[1, 1] = e[1] + s[1, 1]                    // ch[1, 1] and ch[2,1] not needed.
        m[2, 1] = e[2] + s[2, 1]

        for(j=2 to n) {
            for(i = 1 to 2) {
                pathOneTime = m[i, j-1] + s[i, j]
                pathTwoTime = m[3-i, j-1] + t[3-i, j-1] + s[i, j]
                if(pathOneTime < pathTwoTime) {
                    m[i, j] = pathOneTime;
                    ch[i, j] = i;
                }
                else {
                    m[i, j] = pathTwoTime;
                    ch[i, j] = 3-i;
                }
            }
        }
```

Dynamic Programming Traceback

```
        if( m[1,n] + x[1] < m[2,n] + x[2]) {        // note this fills memo table
            return traceback(1, n) and m[1, n] + x[1];
        }
        else {
            return traceback(2, n) and m[2, n] + x[2];
        }
}
```

Software engineering note: this is a long subroutine. In object-oriented style, such long subroutines are discouraged: they should be broken into smaller subroutines for readability. Only write code in this way if you've already written the readable version and profiling (detailed timing of the running code) tells you that speed is a bottleneck in this part of the program.