

CSE 5311 Notes 0: Review of Dynamic Programming

(Last updated 6/9/15 2:12 PM, extracted from CSE 2320 Notes 7)

DYNAMIC PROGRAMMING APPROACH

1. Describe problem input.
2. Determine cost function and base case.
3. Determine general case for cost function. THE HARD PART!!!
4. Appropriate ordering for enumerating subproblems.
 - a. Simple bottom-up approach - from small problems towards the entire big problem.
 - b. Top-down approach with “memoization” - to attack large problems.
5. Backtrace for solution. *Most of the effort in dynamic programming is ignored at the end.*
 - a. Predecessor/back pointers to get to the subproblems whose results are in the solution.
 - b. Top-down recomputation of cost function (to reach the same subproblems as 5.a)
(Providing all solutions is an extra cost feature . . .)

7.B. WEIGHTED INTERVAL SCHEDULING

Input: A set of n intervals numbered 1 through n with each interval i having start time s_i , finish time f_i , and positive weight v_i ,

Output: A set of non-overlapping intervals to *maximize* the sum of their weights. (Two intervals i and j overlap if either $s_i < s_j < f_i$ or $s_i < f_j < f_i$.)

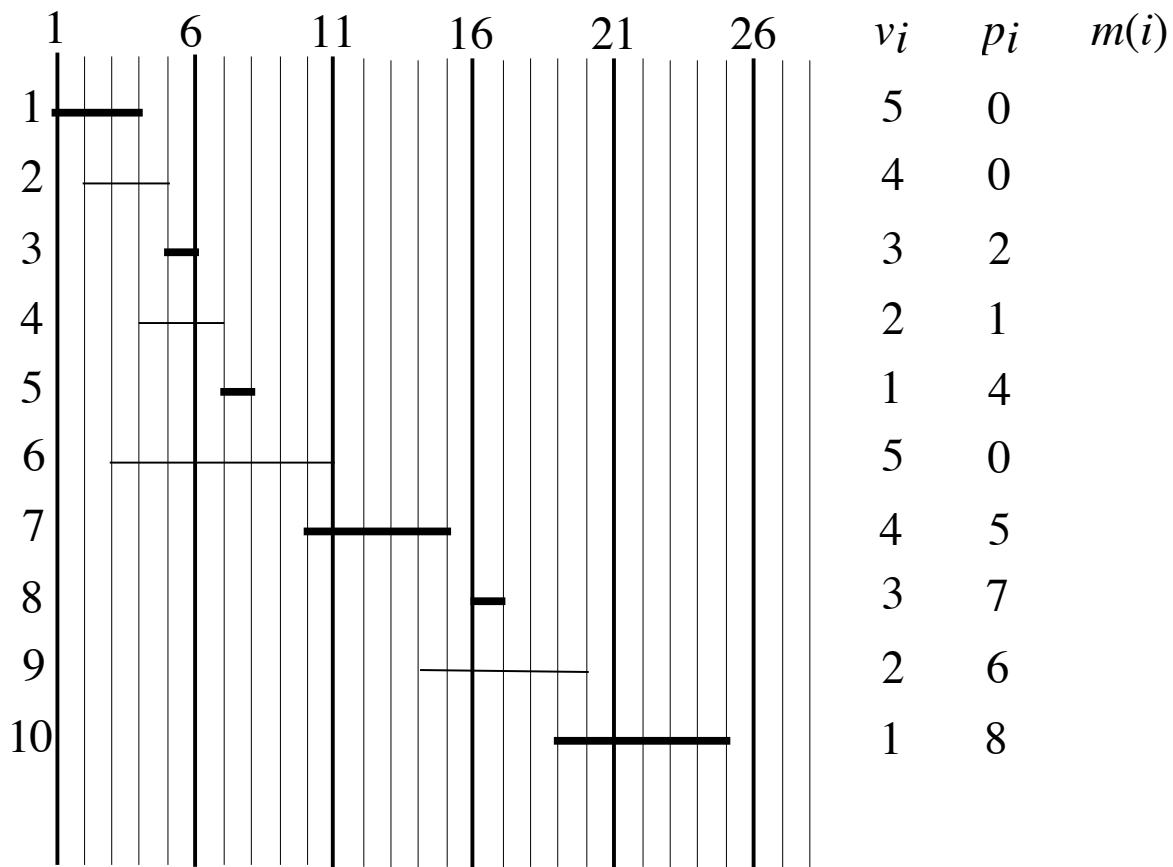
Brute-force solution: Enumerate the powerset of the input intervals, discard those cases with overlapping intervals, and compute the sum of the weights for each.

1. Describe problem input.

Assume the n intervals are in ascending finish time order, i.e. $f_i \leq f_{i+1}$.

Let p_i be the *rightmost preceding interval* for interval i , i.e. the largest value $j < i$ such that intervals i and j do not overlap. If no such interval j exists, $p_i = 0$. (These values may be computed in $\Theta(n \log n)$ time using `binSearchLast()` from Notes 1.

<http://ranger.uta.edu/~weems/NOTES2320/wis.bs.c>)



- Determine cost function and base case.

$M(i)$ = Cost for optimal non-overlapping subset for the first i input intervals.

$$M(0) = 0$$

- Determine general case.

For $M(i)$, the main issue is: *Does the optimal subset include interval i ?*

If *yes*: optimal subset cannot include any overlapping intervals, so $M(i) = M(p_i) + v_i$.

If *no*: optimal subset is the same as for $M(i-1)$, so $M(i) = M(i-1)$.

This observation tells us to compute cost **both** ways and keep the maximum.

- Appropriate ordering of subproblems. Simply compute $M(i)$ in ascending i order.

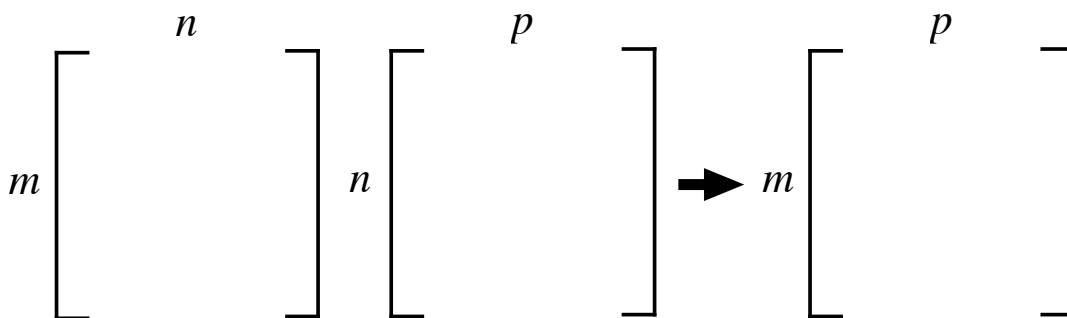
5. Backtrace for solution (with recomputation). This is the subset of intervals for $M(n)$.

```

i=n;
while (i>0)
  if (v[i]+M[p[i]]>=M[i-1])
  {
    // Interval i is in solution
    i=p[i];
  }
  else
    i--;

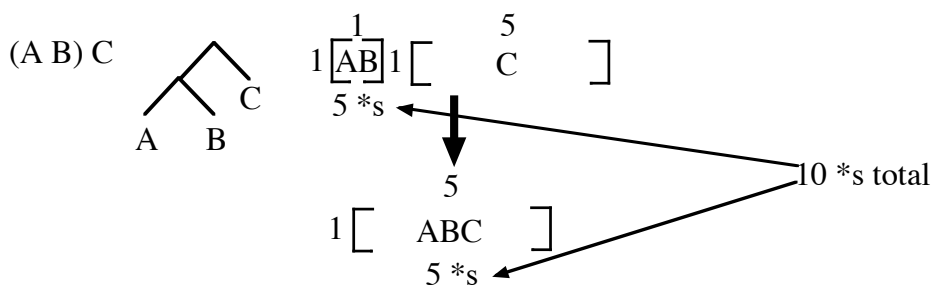
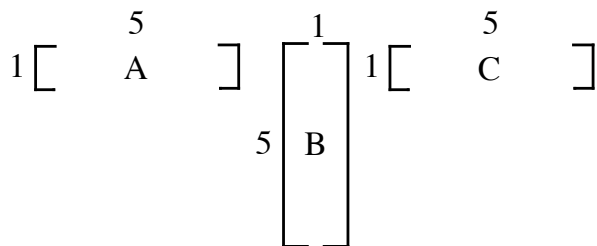
```

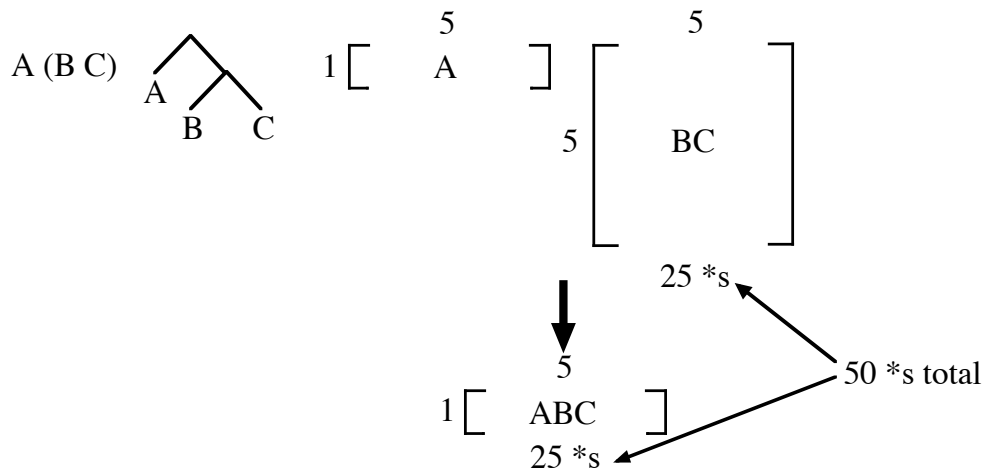
7.C. OPTIMAL MATRIX MULTIPLICATION ORDERING (very simplified version of query optimization)



Only one strategy for multiplying two matrices – requires mnp scalar multiplications (and $m(n - 1)p$ additions).

There are two strategies for multiplying three matrices:





Aside: Ways to parenthesize n matrices? (Catalan numbers)

$$C_0 = 1 \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \text{ for } n \geq 0 \quad C_n = \frac{1}{n+1} \binom{2n}{n}$$

(http://en.wikipedia.org/wiki/Catalan_number)

Observation: Final tree cannot be optimal if any subtree is not.

1. Describe problem input.

n matrices $\Rightarrow n + 1$ sizes

$$P_0 \left[\begin{array}{c} P_1 \\ M_1 \end{array} \right] P_1 \left[\begin{array}{c} P_2 \\ M_2 \end{array} \right] \dots P_{n-1} \left[\begin{array}{c} P_n \\ M_n \end{array} \right]$$

2. Determine cost function and base case.

$C(i, j) = \text{Cost for optimally multiplying } M_i \dots M_j$

$C(i, i) = 0$

3. Determine general case.

Consider a specific case $C(5, 9)$. The optimal way to multiply $M_5 \dots M_9$ could be any of the following:

$$C(5, 5) + C(6, 9) + P_4 P_5 P_9$$

$$C(5, 6) + C(7, 9) + P_4 P_6 P_9$$

$$C(5, 7) + C(8, 9) + P_4 P_7 P_9$$

$$C(5, 8) + C(9, 9) + P_4 P_8 P_9$$

Compute all four and keep the smallest one.

Abstractly: Trying to find $C(i, j)$

$$P_{i-1} \begin{bmatrix} P_k \\ C(i, k) \end{bmatrix} P_k \begin{bmatrix} P_j \\ C(k+1, j) \end{bmatrix}$$

$$C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k+1, j) + P_{i-1} P_k P_j\}$$

4. Appropriate ordering of subproblems.

Since smaller subproblems are needed to solve larger problems, run value for $j - i$ for $C(i, j)$ from 0 to $n - 1$. Suppose $n = 5$:

| 0 | 1 | 2 | 3 | 4 |
|----------|----------|----------|----------|----------|
| $C(1,1)$ | $C(1,2)$ | $C(1,3)$ | $C(1,4)$ | $C(1,5)$ |
| $C(2,2)$ | $C(2,3)$ | $C(2,4)$ | $C(2,5)$ | |
| $C(3,3)$ | $C(3,4)$ | $C(3,5)$ | | |
| $C(4,4)$ | $C(4,5)$ | | | |
| $C(5,5)$ | | | | |

5. Backtrace for solution – explicitly save the k value that gave each $C(i, j)$.

<http://ranger.uta.edu/~weems/NOTES2320/optMM.c>

```
// Optimal matrix multiplication order using dynamic programming
#include <stdio.h>
main()
{
int p[20];
int n;
int c[20][20];
int trace[20][20];

int i,j,k;
int work;

scanf("%d",&n);
for (i=0;i<=n;i++)
    scanf("%d",&p[i]);
for (i=1;i<=n;i++)
    c[i][i]=trace[i][i]=0;
for (i=1;i<n;i++)
    for (j=1;j<=n-i;j++)
    {
        printf("Compute c[%d][%d]\n",j,j+i);
        c[j][j+i]=999999;
        for (k=j;k<j+i;k++)
        {
            work=c[j][k]+c[k+1][j+i]+p[j-1]*p[k]*p[j+i];
            printf(" k=%d gives cost %3d=c[%d][%d]+c[%d][%d]+p[%d]*p[%d]*p[%d]\n",
                k,work,j,k,k+1,j+i,j-1,k,j+i);
            if (c[j][j+i]>work)
            {
                c[j][j+i]=work;
                trace[j][j+i]=k;
            }
        }
        printf(" c[%d][%d]==%d,trace[%d][%d]==%d\n",j,j+i,
            c[j][j+i],j,j+i,trace[j][j+i]);
    }

printf(" ");
for (i=1;i<=n;i++)
    printf(" %3d ",i);
printf("\n");
for (i=1;i<=n;i++)
{
    printf("%2d ",i);
    for (j=1;j<=n;j++)
        if (i>j)
            printf(" ----- ");
        else
            printf(" %3d %3d ",c[i][j],trace[i][j]);
    printf("\n");
    printf("\n");
}
}
```

It is straightforward to use integration to determine that the k loop body executes about $\frac{n^3}{6}$ times.

```

4
2 4 3 5 2
Compute c[1][2]
k=1 gives cost 24=c[1][1]+c[2][2]+p[0]*p[1]*p[2]
c[1][2]==24, trace[1][2]==1
Compute c[2][3]
k=2 gives cost 60=c[2][2]+c[3][3]+p[1]*p[2]*p[3]
c[2][3]==60, trace[2][3]==2
Compute c[3][4]
k=3 gives cost 30=c[3][3]+c[4][4]+p[2]*p[3]*p[4]
c[3][4]==30, trace[3][4]==3
Compute c[1][3]
k=1 gives cost 100=c[1][1]+c[2][3]+p[0]*p[1]*p[3]
k=2 gives cost 54=c[1][2]+c[3][3]+p[0]*p[2]*p[3]
c[1][3]==54, trace[1][3]==2
Compute c[2][4]
k=2 gives cost 54=c[2][2]+c[3][4]+p[1]*p[2]*p[4]
k=3 gives cost 100=c[2][3]+c[4][4]+p[1]*p[3]*p[4]
c[2][4]==54, trace[2][4]==2

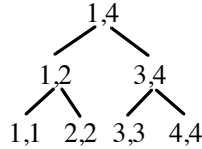
```

```

Compute c[1][4]
k=1 gives cost 70=c[1][1]+c[2][4]+p[0]*p[1]*p[4]
k=2 gives cost 66=c[1][2]+c[3][4]+p[0]*p[2]*p[4]
k=3 gives cost 74=c[1][3]+c[4][4]+p[0]*p[3]*p[4]
c[1][4]==66, trace[1][4]==2

```

| | 1 | 2 | 3 | 4 |
|---|-------|-------|-------|----|
| 1 | 0 | 0 | 24 | 1 |
| 2 | ----- | 0 | 0 | 60 |
| 3 | ----- | ----- | 0 | 0 |
| 4 | ----- | ----- | ----- | 0 |



```

7
1 7 9 5 1 5 10 3
Compute c[1][2]
k=1 gives cost 63=c[1][1]+c[2][2]+p[0]*p[1]*p[2]
c[1][2]==63, trace[1][2]==1
Compute c[2][3]
k=2 gives cost 315=c[2][2]+c[3][3]+p[1]*p[2]*p[3]
c[2][3]==315, trace[2][3]==2
Compute c[3][4]
k=3 gives cost 45=c[3][3]+c[4][4]+p[2]*p[3]*p[4]
c[3][4]==45, trace[3][4]==3
Compute c[4][5]
k=4 gives cost 25=c[4][4]+c[5][5]+p[3]*p[4]*p[5]
c[4][5]==25, trace[4][5]==4
Compute c[5][6]
k=5 gives cost 50=c[5][5]+c[6][6]+p[4]*p[5]*p[6]
c[5][6]==50, trace[5][6]==5
Compute c[6][7]
k=6 gives cost 150=c[6][6]+c[7][7]+p[5]*p[6]*p[7]
c[6][7]==150, trace[6][7]==6
Compute c[1][3]
k=1 gives cost 350=c[1][1]+c[2][3]+p[0]*p[1]*p[3]
k=2 gives cost 108=c[1][2]+c[3][3]+p[0]*p[2]*p[3]
c[1][3]==108, trace[1][3]==2
Compute c[2][4]
k=2 gives cost 108=c[2][2]+c[3][4]+p[1]*p[2]*p[4]
k=3 gives cost 350=c[2][3]+c[4][4]+p[1]*p[3]*p[4]
c[2][4]==108, trace[2][4]==2
Compute c[3][5]
k=3 gives cost 250=c[3][3]+c[4][5]+p[2]*p[3]*p[5]
k=4 gives cost 90=c[3][4]+c[5][5]+p[2]*p[4]*p[5]
c[3][5]==90, trace[3][5]==4
Compute c[4][6]
k=4 gives cost 100=c[4][4]+c[5][6]+p[3]*p[4]*p[6]
k=5 gives cost 275=c[4][5]+c[6][6]+p[3]*p[5]*p[6]
c[4][6]==100, trace[4][6]==4
Compute c[5][7]
k=5 gives cost 165=c[5][5]+c[6][7]+p[4]*p[5]*p[7]
k=6 gives cost 80=c[5][6]+c[7][7]+p[4]*p[6]*p[7]
c[5][7]==80, trace[5][7]==6
Compute c[1][4]
k=1 gives cost 115=c[1][1]+c[2][4]+p[0]*p[1]*p[4]
k=2 gives cost 117=c[1][2]+c[3][4]+p[0]*p[2]*p[4]
k=3 gives cost 113=c[1][3]+c[4][4]+p[0]*p[3]*p[4]
c[1][4]==113, trace[1][4]==3
Compute c[2][5]
k=2 gives cost 405=c[2][2]+c[3][5]+p[1]*p[2]*p[5]
k=3 gives cost 515=c[2][3]+c[4][5]+p[1]*p[3]*p[5]
k=4 gives cost 143=c[2][4]+c[5][5]+p[1]*p[4]*p[5]
c[2][5]==143, trace[2][5]==4

```

```

Compute c[3][6]
k=3 gives cost 550=c[3][3]+c[4][6]+p[2]*p[3]*p[6]
k=4 gives cost 185=c[3][4]+c[5][6]+p[2]*p[4]*p[6]
k=5 gives cost 540=c[3][5]+c[6][6]+p[2]*p[5]*p[6]
c[3][6]==185, trace[3][6]==4

```

```

Compute c[4][7]
k=4 gives cost 95=c[4][4]+c[5][7]+p[3]*p[4]*p[7]
k=5 gives cost 250=c[4][5]+c[6][7]+p[3]*p[5]*p[7]
k=6 gives cost 250=c[4][6]+c[7][7]+p[3]*p[6]*p[7]
c[4][7]==95, trace[4][7]==4

```

```

Compute c[1][5]
k=1 gives cost 178=c[1][1]+c[2][5]+p[0]*p[1]*p[5]
k=2 gives cost 198=c[1][2]+c[3][5]+p[0]*p[2]*p[5]
k=3 gives cost 158=c[1][3]+c[4][5]+p[0]*p[3]*p[5]
k=4 gives cost 118=c[1][4]+c[5][5]+p[0]*p[4]*p[5]
c[1][5]==118, trace[1][5]==4

```

```

Compute c[2][6]
k=2 gives cost 815=c[2][2]+c[3][6]+p[1]*p[2]*p[6]
k=3 gives cost 765=c[2][3]+c[4][6]+p[1]*p[3]*p[6]
k=4 gives cost 228=c[2][4]+c[5][6]+p[1]*p[4]*p[6]
k=5 gives cost 493=c[2][5]+c[6][6]+p[1]*p[5]*p[6]
c[2][6]==228, trace[2][6]==4

```

```

Compute c[3][7]
k=3 gives cost 230=c[3][3]+c[4][7]+p[2]*p[3]*p[7]
k=4 gives cost 152=c[3][4]+c[5][7]+p[2]*p[4]*p[7]
k=5 gives cost 375=c[3][5]+c[6][7]+p[2]*p[5]*p[7]
k=6 gives cost 455=c[3][6]+c[7][7]+p[2]*p[6]*p[7]
c[3][7]==152, trace[3][7]==4

```

```

Compute c[1][6]
k=1 gives cost 298=c[1][1]+c[2][6]+p[0]*p[1]*p[6]
k=2 gives cost 338=c[1][2]+c[3][6]+p[0]*p[2]*p[6]
k=3 gives cost 258=c[1][3]+c[4][6]+p[0]*p[3]*p[6]
k=4 gives cost 173=c[1][4]+c[5][6]+p[0]*p[4]*p[6]
k=5 gives cost 168=c[1][5]+c[6][6]+p[0]*p[5]*p[6]
c[1][6]==168, trace[1][6]==5

```

```

Compute c[2][7]
k=2 gives cost 341=c[2][2]+c[3][7]+p[1]*p[2]*p[7]
k=3 gives cost 515=c[2][3]+c[4][7]+p[1]*p[3]*p[7]
k=4 gives cost 209=c[2][4]+c[5][7]+p[1]*p[4]*p[7]
k=5 gives cost 398=c[2][5]+c[6][7]+p[1]*p[5]*p[7]
k=6 gives cost 438=c[2][6]+c[7][7]+p[1]*p[6]*p[7]
c[2][7]==209, trace[2][7]==4

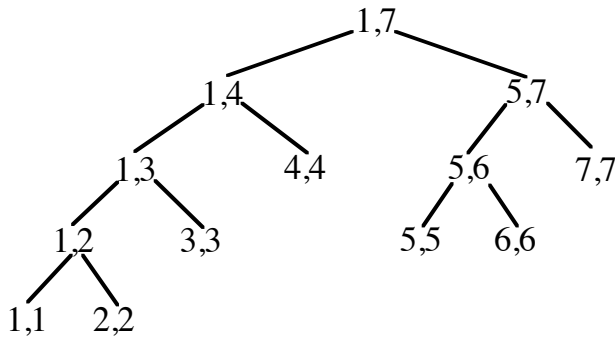
```

```

Compute c[1][7]
k=1 gives cost 230=c[1][1]+c[2][7]+p[0]*p[1]*p[7]
k=2 gives cost 242=c[1][2]+c[3][7]+p[0]*p[2]*p[7]
k=3 gives cost 218=c[1][3]+c[4][7]+p[0]*p[3]*p[7]
k=4 gives cost 196=c[1][4]+c[5][7]+p[0]*p[4]*p[7]
k=5 gives cost 283=c[1][5]+c[6][7]+p[0]*p[5]*p[7]
k=6 gives cost 198=c[1][6]+c[7][7]+p[0]*p[6]*p[7]
c[1][7]==196, trace[1][7]==4

```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|-------|-------|-------|-------|-------|-------|---|
| 1 | 0 | 63 | 108 | 113 | 118 | 168 | 196 |
| 2 | ----- | 0 | 315 | 108 | 143 | 228 | 209 |
| 3 | ----- | ----- | 0 | 45 | 90 | 185 | 152 |
| 4 | ----- | ----- | ----- | 0 | 25 | 100 | 95 |
| 5 | ----- | ----- | ----- | ----- | 0 | 50 | 80 |
| 6 | ----- | ----- | ----- | ----- | ----- | 0 | 150 |
| 7 | ----- | ----- | ----- | ----- | ----- | ----- | 0 |



7.F. SUBSET SUM (<http://ranger.uta.edu/~weems/NOTES2320/subsetSum.c>)

Given a “set” of n positive integer values, find a subset whose sum adds to a value m .

Optimization?

Enumerating subsets (combinations) would take exponential time.

1. Describe problem input. Array $S = S_1, S_2, \dots, S_n$ and m .
2. Determine cost function and base case.

$C(i)$ = Smallest index j such that there is some combination of S_1, S_2, \dots, S_j , that includes S_j and sums to i .

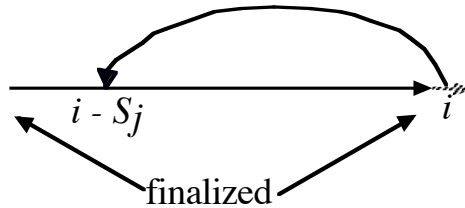
$C(0) = 0$ (Will assume that $S_0 = 0$)

3. Determine general case for cost function.

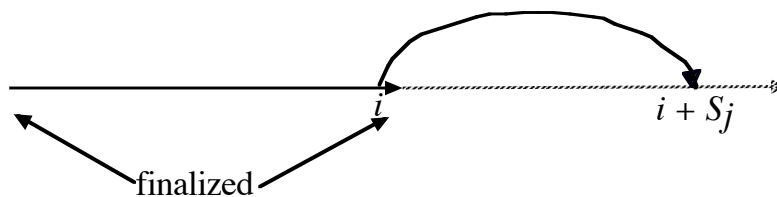
$$C(i) = \min_{\substack{\{j\} \\ \text{s.t. } C(i-S_j) \text{ is defined} \\ \text{and } C(i-S_j) < j}} \{j\}$$

4. Appropriate ordering of subproblems:

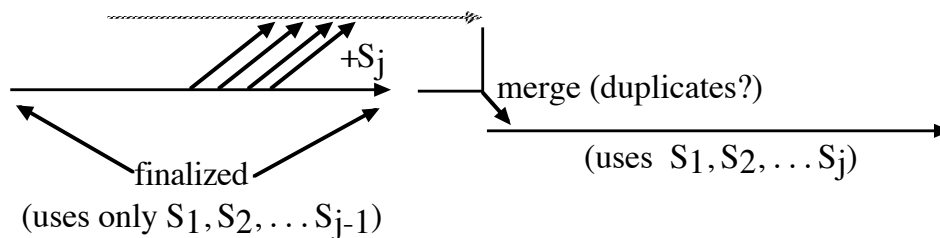
- a. Iterate over i looking backwards (like the cost function) to previous “finalized” solutions.



- b. (Aside, Dijkstra’s algorithm-like) Iterate over finalized $C(i)$ to compute $i + S_j$ for each $j > C(i)$ and attempt update forward. After updates, $C(i+1)$ has final value.



- c. (Aside) Maintain ordered list of finalized solutions from using S_1, S_2, \dots, S_{j-1} and generate new ordered list that also uses S_j to reach some new values.



5. Backtrace for solution - if $C(m)$ is defined, then backtrace using C values to subtract out each value in subset. (Indices will appear in strictly decreasing order during backtrace.)

```
// Initialize table for DP
C[0]=0; // DP base case
// For each potential sum, determine the smallest index such
// that its input value is in a subset to achieve that sum.
for (potentialSum=1; potentialSum<=m; potentialSum++)
{
  for (j=1; j<=n; j++)
  {
    leftover=potentialSum-S[j]; // To be achieved with other values
    if (leftover<0) // Too much thrown away
      continue;
    if (C[leftover]==(-1)) // No way to achieve leftover
      continue;
    if (C[leftover]<j) // Indices are included in
      break; // ascending order.
  }
  C[potentialSum]=(j<=n) ? j : (-1);
}
}
```

```

if (C[m]==(-1))
    printf("No solution\n");
else
{
    printf("Solution\n");
    printf("  i  S\n");
    printf("-----\n");
    for (i=m;i>0;i-=S[C[i]])
        printf("%3d %3d\n",C[i],S[C[i]]);
}

```

Example: $m = 12, n = 4$

| | | | | | | |
|-------|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | |
| S_i | 0 | 3 | 6 | 7 | 9 | [The S_i values do not require ordering.] |

| | | | | | | | | | | | | | |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| C_i | | | | | | | | | | | | | |

Time is $\Theta(mn)$. Space is $\Theta(m)$. [What happens if m and each S_i are multiplied by the same constant?]

7.H. THE PARKING PROBLEM (aside, <http://ranger.uta.edu/~weems/NOTES2320/parkingOff.oneBased.c>)

Requirements:

Implement a $\Theta(kn)$ -time, $\Theta(k + n)$ -space dynamic programming algorithm to determine a **minimum cost sequence** of parking permits to cover n not-necessarily-adjacent days you need to drive downtown. Each element of a solution sequence will be one of k available permit types, each covering a different number of **consecutive** days at some cost.

The first line of the input will be positive integers for k and n . $k \leq 10$ and $n \leq 100$. The next k lines will be pairs of positive integers for the permit types. The two values of a pair will be the number of days and cost, respectively. Note that the k pairs will appear in strictly increasing order for the number of days and likewise for the costs. (For example, nobody will spend \$20 for a three-day permit if a four-day permit is just \$15.) Each of the remaining n lines will contain an integer corresponding to a day you must park. These values appear in strictly increasing order.

The output is 1) the table of subproblems, i.e. their cost and backtrace information, 2) the cost of the final solution, and 3) the sequence of permits needed and the range of days covered by each. The sequence may be output in reverse order. The last day for the last permit should be the last day you need to park. The start date for the first permit may be earlier than the first day you need to park.

Getting Started:

1. The left column gives an input instance and the right column gives a solution

```

3 10
1 5
2 7
3 9
1
2
3
4
7
8
9
12
13
15

```

DP table:

```

prefixCost[0]=0 permitTypeUsed[0]=0
prefixCost[1]=5 permitTypeUsed[1]=1
prefixCost[2]=7 permitTypeUsed[2]=2
prefixCost[3]=9 permitTypeUsed[3]=3
prefixCost[4]=14 permitTypeUsed[4]=1
prefixCost[5]=19 permitTypeUsed[5]=1
prefixCost[6]=21 permitTypeUsed[6]=2
prefixCost[7]=23 permitTypeUsed[7]=3
prefixCost[8]=28 permitTypeUsed[8]=1
prefixCost[9]=30 permitTypeUsed[9]=2
prefixCost[10]=35 permitTypeUsed[10]=1
Cost is 35

```

Permits used are:

```

Permit type 1 cost 5 begin 15 end 15
Permit type 2 cost 7 begin 12 end 13
Permit type 3 cost 9 begin 7 end 9
Permit type 1 cost 5 begin 4 end 4
Permit type 3 cost 9 begin 1 end 3

```

2. This problem is slightly similar to the weighted interval scheduling problem. In particular, an optimal solution is determined for *every* prefix of the input sequence of n days.
3. Achieving $\Theta(kn)$ time requires maintaining a table with k entries whose entry i indicates the *subscript* of the *latest* input day such that a permit of type i cannot cover *both* this day and the day whose subproblem is under consideration.

1. Describe problem input.

The k permit types are in arrays $permitDays_1 \dots permitDays_k$ and $permitAmount_1 \dots permitAmount_k$.

The n days to be covered by permits are in the array $day_1 \dots day_n$.

2. Determine cost function and base case.

$C(i)$ = Minimum cost for permits to cover the first i input days.

$C(0) = 0$

3. Determine general case.

$$C(i) = \min_{1 \leq j \leq k} \left\{ permitAmount_j + C\left(\max\{lag \mid 0 \leq lag < i \wedge day_{lag} < day_i - permitDays_j + 1\}\right) \right\}$$

Translated: For each day, try every permit. To the cost of a candidate permit, add the cost of previous permits to optimally cover all driving days before the candidate permit.

4. Appropriate ordering of subproblems. Compute $C(i)$ in ascending i order.

Using the cost function directly will lead to $\Theta(kn^2)$ time due to table scanning for lag .

$\Theta(kn)$ approach - have an array for the k lag values and attempt to increment before trying the corresponding permit for i .

5. Backtrace for solution - easily done if the j giving the minimum is saved for each i . Not difficult to use recomputation (see code).

For much more about the parking permit problem and its role in online computation, see:

Adam Meyerson. "The Parking Permit Problem", *IEEE Symposium on Foundations of Computer Science (FOCS)* 2005.