

Lecture 7, Oct 24

Dynamic Programming
Continued

Dynamic programming

- Step 1: Describe an array of values you want to compute.
- Step 2: Give a recurrence for computing later values from earlier (bottom-up).
- Step 3: Give a high-level program.
- Step 4: Show how to use values in the array to compute an optimal solution.

Example 3: Scheduling Jobs with Deadlines, Profits and Durations

Input: information (d_i, t_i, g_i) about n activities, where
 d_i = integer deadline for job i
 t_i = integer duration of job i
 g_i = real-valued profit of job i

A schedule C is a sequence $C = \{C(1), C(2), \dots, C(n)\}$ such that:
 $C(i)$ = scheduled start time of job i
 $C(i) = -1$ if job i is not scheduled

A schedule C is feasible if each scheduled job finishes by its deadline and no two scheduled jobs overlapped.

Output: A feasible schedule C with maximum profit: $P(C) = \sum_{i \in C} g_i$

Dynamic Programming Solution

Precomputation:

Sort activities according to deadline: $d_1 \leq d_2 \leq \dots \leq d_n$ ($O(n \log n)$)

Step 1. Define an array of values to compute

$\forall i \in \{0, \dots, n\}, \forall t \in \{0, \dots, d\}, A(i, t) =$ largest profit attainable from the (feasible) scheduling of a subset of jobs from $\{1, 2, \dots, i\}$, all of which finish by time t

Ultimately, we are interested in $A(n, d)$

Step 2. Provide a Recurrent Solution

$$A(0,t) = 0 \forall t \in \{0, \dots, d\}$$

For $i \in \{1, \dots, n\}$, $t \in \{0, \dots, d\}$, define $t' = \min\{t, d_i\} - t_i$
 (t' is the latest time that we can schedule job i so that it ends both by its deadline and by time t .)

Then:

$$t' < 0 \rightarrow A(i,t) = A(i-1,t)$$

$$t' \geq 0 \rightarrow A(i,t) = \max\{A(i-1,t), g_i + A(i-1,t')\}$$

Decide not to
schedule job i

Profit from job i

Profit from scheduling activities that
end before job i begins

Step 2. (cntd...) Proving the Recurrent Solution

For $i \in \{1, \dots, n\}$, $t \in \{0, \dots, d\}$, define $t' = \min\{t, d_i\} - t_i$
 (t' is the latest time that we can schedule job i so that it ends both by its deadline and by time t .)

Then:

$$t' < 0 \rightarrow A(i,t) = A(i-1,t)$$

$$t' \geq 0 \rightarrow A(i,t) = \max\{A(i-1,t), g_i + A(i-1,t')\}$$

We effectively schedule job i at the latest possible time.

This leaves the largest and earliest contiguous block of time for scheduling jobs with earlier deadlines.

Step 3. Provide an Algorithm

```
for every  $t \in \{0, \dots, d\}$ 
   $B[0, t] \leftarrow 0$ 
end for
for  $i : 1..n$ 
  for every  $t \in \{0, \dots, d\}$ 
     $t' \leftarrow \min\{t, d_i\} - t_i$ 
    if  $t' < 0$  then
       $B[i, t] \leftarrow B[i - 1, t]$ 
    else
       $B[i, t] \leftarrow \max\{B[i - 1, t], g_i + B[i - 1, t']\}$ 
    end if
  end for
end for
```

Running time? $O(nd)$

Step 4. Compute Optimal Solution

```
procedure PRINTOPT( $i, t$ )
  if  $i = 0$  then return end if
  if  $B[i, t] = B[i - 1, t]$  then
    PRINTOPT( $i - 1, t$ )
  else
     $t' \leftarrow \min\{t, d_i\} - t_i$ 
    PRINTOPT( $i - 1, t'$ )
    put "Schedule job",  $i$ , "at time",  $t'$ 
  end if
end PRINTOPT  Running time?  $O(n)$ 

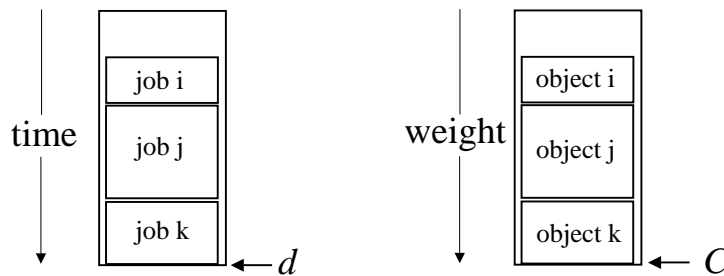
PRINTOPT( $n, d$ )
```

The Knapsack Problem: A Special Case of Job Scheduling

The general knapsack problem can be treated
and solved as a special case of the job scheduling problem

Identify time (job scheduling) with weight (knapsack):
Job deadlines \leftrightarrow knapsack capacity: $d_i = C \forall i \in \{1, \dots, n\}$
Job durations \leftrightarrow object weights: $t_i = w_i$

Running time? $O(nC)$



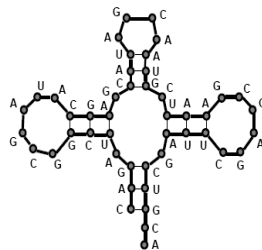
Is this a polynomial time algorithm?

RNA secondary structure – cont'd

- Denote the RNA molecule by $B = b_1 b_2 \dots b_n$, where each b_i is from $\{A, C, G, U\}$.
- A matching is a set S of pairs (b_i, b_j)

Conditions on S

- (*No sharp turns.*) The ends of each pair in S are separated by at least four intervening bases; that is, if $(b_i, b_j) \in S$, then $i < j - 4$.
- The elements of any pair in S consist of either $\{A, U\}$ or $\{C, G\}$ (in either order).
- S is a matching: no base appears in more than one pair.
- (*The non-crossing condition.*) If (b_i, b_j) and (b_k, b_ℓ) are two pairs in S , then we cannot have $i < k < j < \ell$.



RNA secondary structure – cont'd

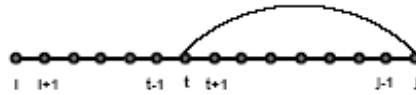
- To find the structure, we are trying to optimize the free energy.
- In the first approximation, we are trying to *maximize* the number of pairs in S .
- Now, this is a combinatorial problem!

Step 1: The array

- $\text{OPT}(i,j)$ = the biggest possible number of matchings realizable on the segment $b_i b_{i+1} \dots b_j$ of B .
- Initialize $\text{OPT}(i,j)=0$ if $i \geq j-4$.
- Want to know $\text{OPT}(1,n)$.

Step 2: The recurrence

(b)



- $OPT(i,j)$:
- We have two options:
 - not to include b_j :
 - get $OPT(i,j-1)$
 - include $b_j - b_t$ for some allowable base pair
 - get $1 + OPT(1,t-1) + OPT(t+1,j-1)$

Step 2: The recurrence – cont'd

We have two options:

- not to include b_j :
- get $OPT(i,j-1)$
- include $b_j - b_t$ for some allowable base pair
- get $1 + OPT(1,t-1) + OPT(t+1,j-1)$

(5.13) $OPT(i, j) = \max(OPT(i, j-1), \max(1 + OPT(i, t-1) + OPT(t+1, j-1)))$,
where the max is taken over t such that b_t and b_j are an allowable base pair (under the Watson-Crick condition (ii)).

Step 3: The Program

To find $OPT(i,j)$ only need to know $OPT(k,l)$ for $l-k < j-i$.

Fill the array in increasing order of $j-i$.

```

Initialize  $OPT(i,j) = 0$  whenever  $i \geq j - 4$ 
For  $i = 1, 2, \dots, n$ 
  For  $j = i + 5, i + 6, \dots, n$ 
    Compute  $OPT(i,j)$  using the recurrence in (5.13)
  Endfor
Endfor
Return  $OPT(1, n)$ 

```

Example



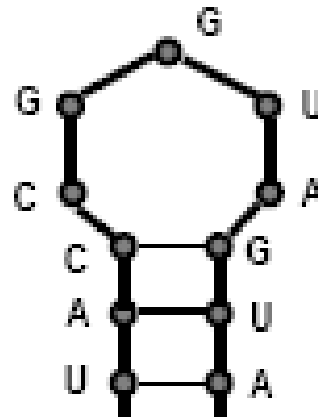
	$i \downarrow$	G	G	C	C	A	U
$j \rightarrow$		6	7	8	9	10	11
G	6	0	0	0	0	0	0
U	5	0	0	0	0	1	1
A	4	0	0	0	0	1	2
G	3	0	0	1	1	1	2
U	2	0	0	1	1	2	2
A	1	0	0	1	1	2	3

Running time

- Building the table: $O(n)$ per entry – total of $O(n^3)$.
- Computing the optimal solution $O(n^2)$.

Example - Solution

	i↓	G	G	C	C	A	U
j→		6	7	8	9	10	11
G	6	0	0	0	0	0	0
U	5	0	0	0	0	1	1
A	4	0	0	0	0	1	2
G	3	0	0	1	1	1	2
U	2	0	0	1	1	2	2
A	1	0	0	1	1	2	3



Example 5

- Given an n by n table of 0/1 pixels, find the biggest all-0 square.