Dynamic Programming

CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material based on *Algorithm Design* by Jon Kleinberg and Éva Tardos (Chapter 6)

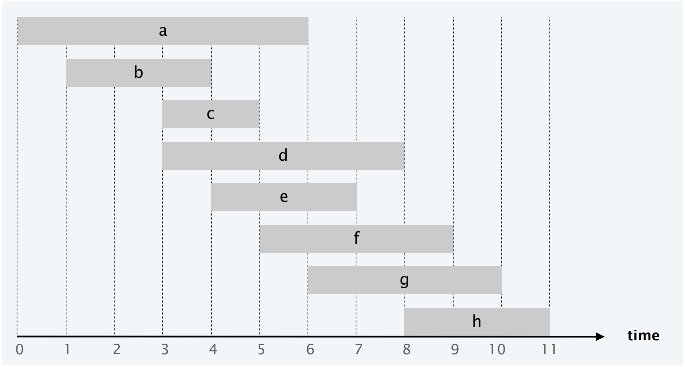
Algorithms paradigms

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.
- *Dynamic programming* is a fancy name for caching away intermediate results in a table for later reuse.

Weighted interval scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



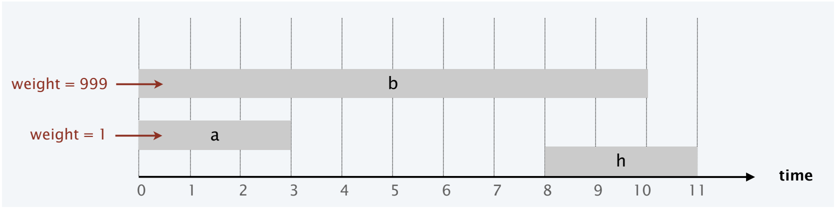
$v_1 = 3, v_2 = 1, v_3 = 8, v_4 = 2, v_5 = 7, v_6 = 123, v_7 = 5, v_8 = 12$

Earliest-finish-time first algorithm (Greedy)

Earliest finish-time first.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Recall. Greedy algorithm is correct if all weights are 1.
Observation. Greedy algorithm fails spectacularly for weighted version.

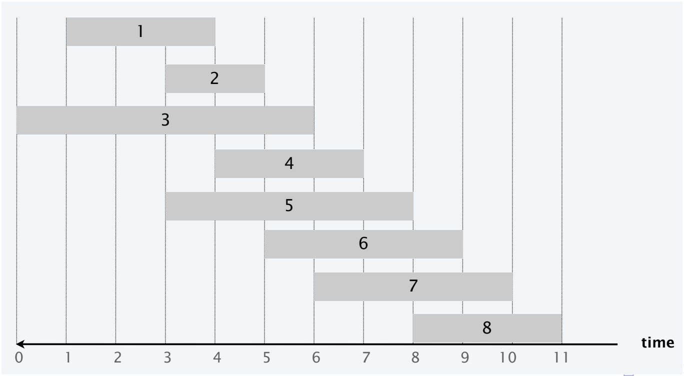


Notation. Label jobs by finishing time; $f_1 \leq f_2 \leq \ldots \leq f_n$. (*sorting required!*).

Definition

$p(j)$ =largest index $i < j$ such that job i is compatible with j.

Note that $p(j) = 0$ if no request $i < j$ is disjoint from j.
Example. For the case below: $p(1) = p(2) = p(3) = 0$, $p(4) = 1$, $p(5) = 0$, $p(6) = 2$, $p(7) = 3$, $p(8) = 5$.



Dynamic programming: binary choice

Suppose we have an optimal solution $\mathcal{O}$, and consider a job n.

- Case 1. $n \in \mathcal{O}$
- Case 1. $n \notin \mathcal{O}$

Note that one of the above two case always happens!

Case1 $n \in \mathcal{O} \implies$ no interval $p(n) < i < n$ belongs to $\mathcal{O}$. Moreover $\mathcal{O}$ must include an **optimal solution** to the problem of requests $\{1, \ldots, p(n)\}$

Case 2 $n \notin \mathcal{O} \implies \mathcal{O}$ is optimal solution to the problem of requests $\{1, \ldots, n-1\}$

Notation.

- $OPT(j) =$ value of optimal solution to the problem consisting of job requests $1, 2, \ldots, j$.
- $\mathcal{O}_j =$ optimal solution to $\{1, \ldots, j\}$.

We are looking for $\mathcal{O}_n$ and $OPT(n)$.

Case1 $n \in \mathcal{O} \implies$ no interval $p(n) < i < n$ belongs to $\mathcal{O}$. Moreover $\mathcal{O}$ must include an **optimal solution** to the problem of requests $\{1, \ldots, p(n)\}$

Case 2 $n \notin \mathcal{O} \implies \mathcal{O}$ is optimal solution to the problem of requests $\{1, \ldots, n-1\}$

$$OPT(j) = \begin{cases} v_j + OPT(p(j)) & j \in \mathcal{O}_j \\ OPT(j-1) & j \in \mathcal{O}_j \end{cases}$$

which implies

$$OPT(j) = \begin{cases} 0 & j \in \text{if } j = 0 \\ \max(v_j + OPT(p(j)), OPT(j-1)) & \text{otherwise} \end{cases}$$

Fact

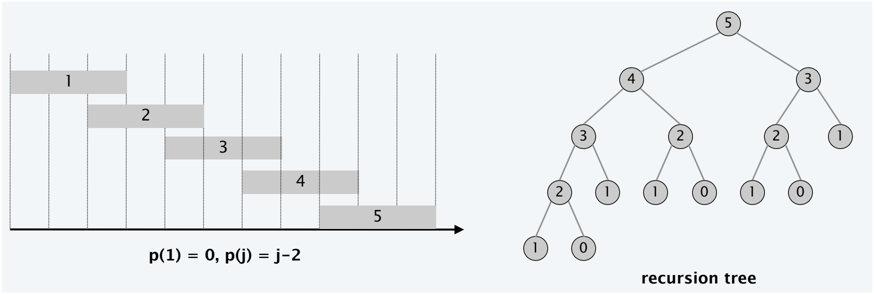
$j \in \mathcal{O}_j \iff v_j + OPT(p(j)) \geq OPT(j-1)$

Weighted interval scheduling: brute force

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

Compute-Opt(j)
if j = 0
   return 0.
else
   return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j-1)).
```

Observation. Recursive algorithm fails spectacularly because of redundant subproblems $\Longrightarrow$ exponential algorithms.
Example. Number of recursive calls for family of "layered" instances grows exponentially.



Proof of exponential complexity

Proposition

Time complexity of brute force algorithm if $\Omega(2^n)$ for the worst case.

Proof.

Consider the case from example above, i.e. $p(j) = j - 2$ for $j \geq 2$. Then we have $T(1) = c$ and

$$T(n) = T(n-1) + T(n-2) + c.$$

$T(n) \geq 2T(n-2) + c \geq 2(2T(n-2) + c) = 2^2 T(n-4) + 2c \geq$
$2^2(2T(n-6) + c) + 2c = 2^{3} T(n - 2 \cdot 3) + (2^2 + 2)c >$
$2^{3} T(n - 2 \cdot 3) + 2^{3-1}c > \ldots > 2^{n/2} \cdot c + 2 \cdot 2^{n/2}$
Hence $T(n) = \Omega(2^n)$. □

Memoization

Idea. Do not solve anything twice, store and use the first solution!
Memoization. Cache results of each subproblem; lookup as needed.

- We need some memory for storage subsolutions.
- Dynamic Programming always use some arrays.
- In this case: $M[0..n]$, $M[j]$ is initially “empty”, but later contains Compute-Opt(j).

Solution with memoization

```
Input:  n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].
Compute p[1], p[2], …, p[n].

for j = 1 to  n
   M[j] ← empty.
M[0] ← 0.

M-Compute-Opt(j)
_________________
if M[j] is empty
   M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j - 1)).
return M[j].
```

Running time

Proposition

If the input interval are sorted by their finish time, the running time of M-Compute-Opt(n) *is* $O(n)$*.*

Proof.

Let $NotEmpty$ is the number of $M[j] \neq \emptyset$. Initially $NotEmpty = 0$, but in each iteration $NotEmpty \leftarrow NotEmpty + 1$. But M has only $n + 1$ elements, so $O(n)$. □

Weighted interval scheduling: finding a solution

Question. Dynamic Programming algorithm computes optimal value. How to find solution itself?
Answer. Make a second pass.

```
Find-Solution(j)
if j = 0
    return $\emptyset$.
else if (v[j] + M[p[j]] > M[j-1])
    return $\{j\}$ $\cup$ Find-Solution(p[j]).
else
    return Find-Solution(j-1).
```

Running time: $T(n) = T(f(n)) + c$, where $f(n)$ is either $p(n)$ or $n-1$ and $p(n) \leq n-1$. Hence
$T(n) \leq T(n-1) + c \leq T(n-2) + 2c \leq \ldots \leq n \cdot c + T(0) = O(n)$.

Instability of recurrences

$T(n) = T(n-1) + c = O(n)$
$T(n) = 2T(n-1) + c = O(2^n)$
$T(n) = 2T(n-2) + c = O(2^n)$
$T(n) = 2T(n-k) + c = O(2^n)$ for any value of k, for instance:
$T(n) = 2T(n-10^{10^{10}}) + c = O(2^n)$, etc.

Weighted interval scheduling: bottom-up

Bottom-up dynamic programming. Unwind recursion.

```
BOTTOM-UP ($n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$)

Sort jobs by finish time so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
Compute $p(1), p(2), \ldots, p(n)$.
$M[0] \leftarrow 0$.
FOR j = 1 TO $n$
    $M[j] \leftarrow \max\{ v_j + M[p(j)], M[j-1] \}$.
```

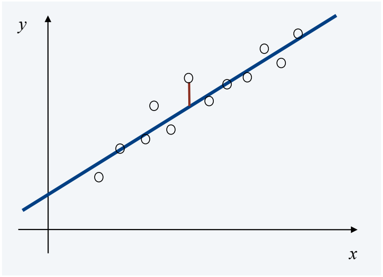
Weighted interval scheduling: Total running time

- Total running time is $O(n \log n)$ because of initial sorting!

Least Squares

Least squares. Foundational problem in statistics.

- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error: $SSE = \displaystyle\sum_{i=1}^{n} (y_i - ax_i - b)^2$



Solution:

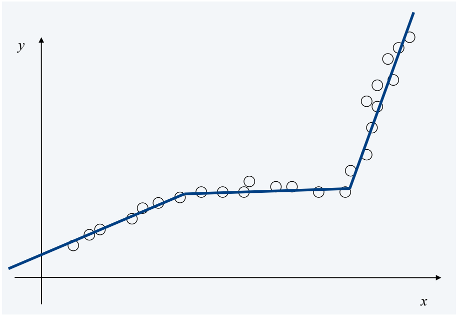
$$a = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - n\sum_{i=1}^{n} x_i)^2}, \quad b = \frac{\sum_{i=1}^{n} y_i - a\sum_{i=1}^{n} x_i}{n}$$

Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

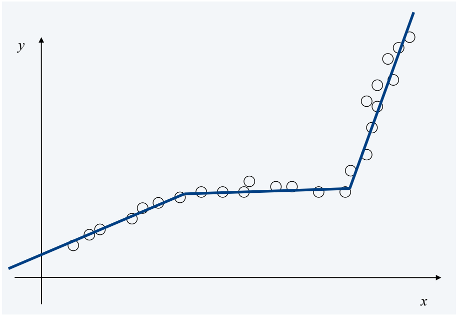
Question. What is a reasonable choice for $f(x)$ to balance accuracy (*goodness of fit*) and parsimony (*number of lines*)?



Segmented Least Squares

Given n points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$ and a constant $c > 0$, find a sequence of lines that minimizes $f(x) = E + cL$:

- $E =$ the sum of the sums of the squared errors in each segment.
- $L =$ the number of lines.



Dynamic programming: multiway choice

Notation.

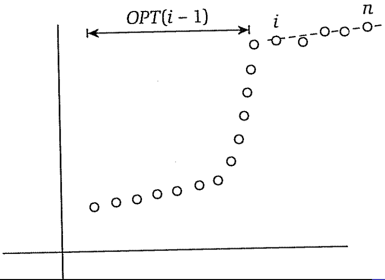
- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- $e(i,j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$:

- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some$i$.
- $Cost = e(i,j) + c + OPT(i–1)$.

which leads to:

$$OPT(j) = \begin{cases} 0 & j \in \text{if } j = 0 \\ \min_{1 \leq i \leq j}(e(i,j) + c + OPT(i-1)) & \text{otherwise} \end{cases}$$



Segmented least squares algorithm

```
SEGMENTED-LEAST-SQUARES ($n, p_1, \ldots, p_n, c$)

FOR j = 1 TO $n$
    FOR i = 1 TO $j$
        Compute the least squares $e(i, j)$ for the segment $p_i, p_{i+1}, \ldots, p_j$.

$M[0] \leftarrow 0$.
FOR $j = 1$ TO $n$
    $M[j] \leftarrow \min_{1 \le i \le j} \{ e_{ij} + c + M[i-1] \}$.

RETURN $M[n]$.
```

Segmented least squares analysis

Theorem

The dynamic programming algorithm solves the segmented least squares problem in $O(n^3)$ time and $O(n^2)$ space.

Proof.

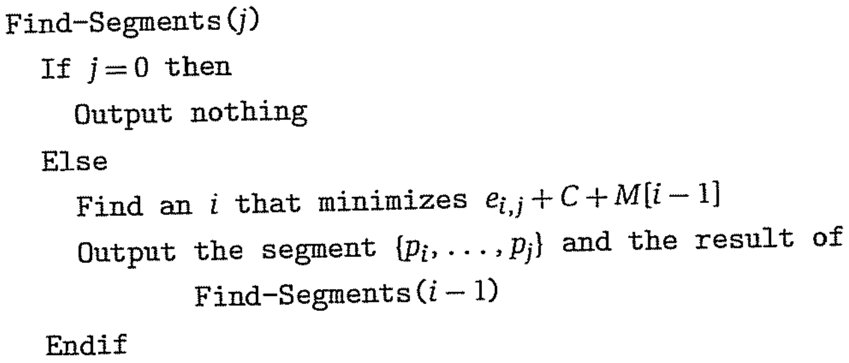
- Bottleneck $=$ computing $e(i,j)$ for $O(n^2)$ pairs.
- $O(n)$ per pair using formula:

$$a = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - n\sum_{i=1}^{n} x_i)^2}, \quad b = \frac{\sum_{i=1}^{n} y_i - a\sum_{i=1}^{n} x_i}{n}$$

$\square$

Remark. It can be improved to $O(n^2)$ time and $O(n)$ space by precomputing various statistics. See Footnote 1 on page 266 of textbook.

Finding Segments



```
Find-Segments($j$)
  If $j = 0$ then
    Output nothing
  Else
    Find an $i$ that minimizes $e_{i,j} + C + M[i-1]$
    Output the segment $\{p_i, \ldots, p_j\}$ and the result of
          Find-Segments($i-1$)
  Endif
```

Knapsack problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.

Ex. $\{1, 2, 5\}$ has value 35.

Ex. $\{3, 4\}$ has value 40.

Ex. $\{3, 5\}$ has value 46 (but exceeds weight limit).

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

Greedy by value. Repeatedly add item with maximum v_i.

Greedy by weight. Repeatedly add item with minimum w_i.

Greedy by ratio. Repeatedly add item with maximum ratio v_i / w_i.

Observation. None of greedy algorithms is optimal.

Dynamic programming: false start

Def. $OPT(i)$ = max profit subset of items $1, \ldots, i$.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- Selecting item i does not immediately imply that we will have to reject other items.
- Without knowing what other items were selected before i, we don't even know if we have enough room for i.

Conclusion. Need more subproblems!

Dynamic programming: adding a new variable

Def. $OPT(i, w)$ = max profit subset of items $1, \ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using weight limit w.

optimal substructure property
(proof via exchange argument)

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $\{ 1, 2, \ldots, i-1 \}$ using this new weight limit.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack problem: bottom-up

```
KNAPSACK ($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)

FOR $w = 0$ TO $W$
    $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO $n$
    FOR $w = 1$ TO $W$
    IF $(w_i > w)$  $M[i, w] \leftarrow M[i-1, w]$.
    ELSE            $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.
```

Knapsack problem: bottom-up demo

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

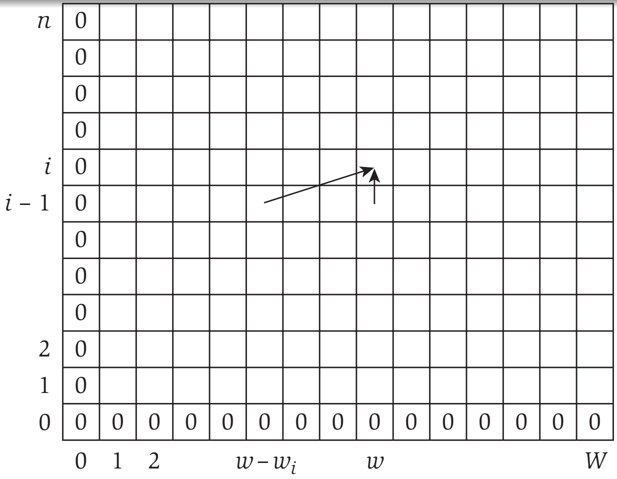
$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

The two-dimensional table of OPT values.



- The leftmost column and bottom row is always 0.
- The entry for $OPT(i, w)$ is computed from the two other entries $OPT(i-1, w)$ and $OPT(i-1, w-w_i)$, as indicated by the arrows.

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
①	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
②	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

③	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

Knapsack problem: running time

Theorem. There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(n\ W)$ time and $\Theta(n\ W)$ space.

weights are integers between 1 and W

Pf.

- Takes $O(1)$ time per table entry.
- There are $\Theta(n\ W)$ table entries.
- After computing optimal values, can trace back to find solution: take item i in $OPT(i, w)$ iff $M[i, w] > M[i-1, w]$. ▪

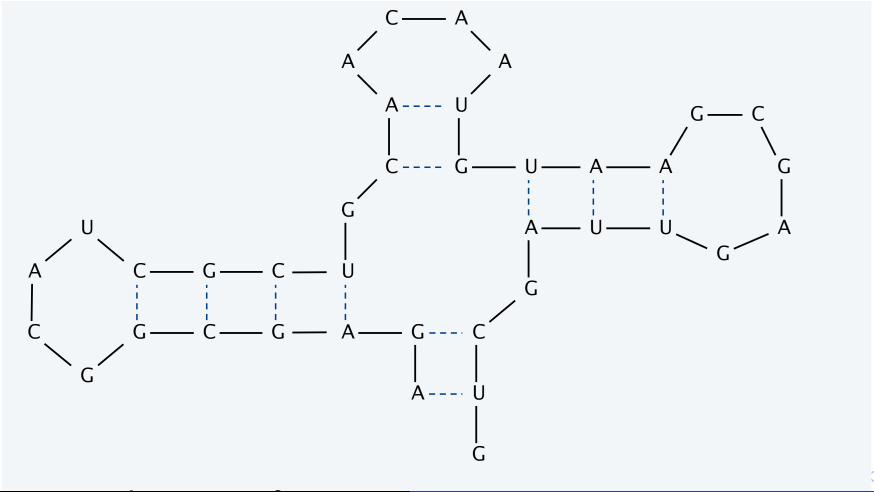
Remarks.

- Not polynomial in input size! $\longleftarrow$ "pseudo-polynomial"
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

RNA secondary structure

RNA. String $B = b_1 b_2 \dots b_n$ over alphabet $\{A, C, G, U\}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.



RNA secondary structure

Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

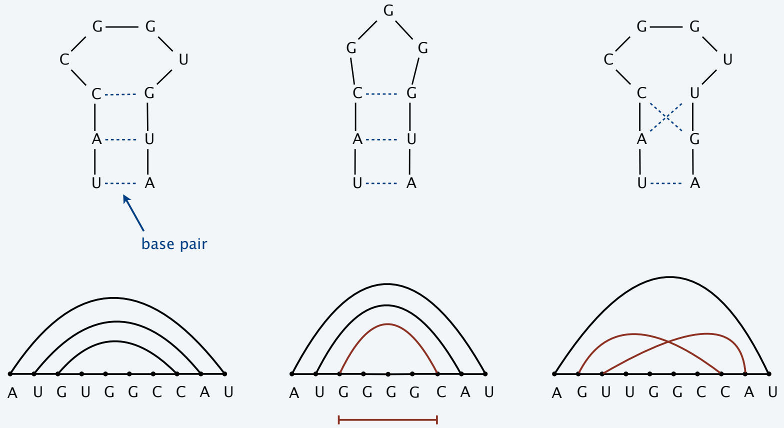
- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick complement: $A - U$, $U - A$, $C - G$, or $G - C$.
- [No sharp turns] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the minimum total free energy (which is approximated by number of base pairs).

Goal. Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

RNA secondary structure

Examples.



ok

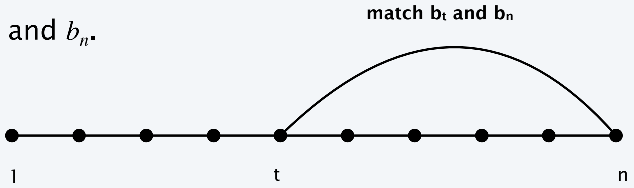
sharp turn
(≤4 intervening bases)

crossing

RNA secondary structure: subproblems

First attempt. $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \dots b_j$.

Choice. Match b_t and b_n.



Difficulty. Results in two subproblems but one of wrong form.

- Find secondary structure in $b_1 b_2 \dots b_{t-1}$. $\leftarrow$ $OPT(t-1)$
- Find secondary structure in $b_{t+1} b_{t+2} \dots b_{n-1}$. $\leftarrow$ need more subproblems

Dynamic programming over intervals

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} b_j$.

Case 1. If $i \geq j - 4$.

- $OPT(i, j) = 0$ by no-sharp turns condition.

Case 2. Base b_j is not involved in a pair.

- $OPT(i, j) = OPT(i, j - 1)$.

Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.

- Noncrossing constraint decouples resulting subproblems.
- $OPT(i, j) = 1 + \max_t(OPT(i, t - 1) + OPT(t + 1, j– - 1))$.

$\uparrow$

take max over t such that $i \leq t < j - 4$ and b_t and b_j are Watson-Crick complements

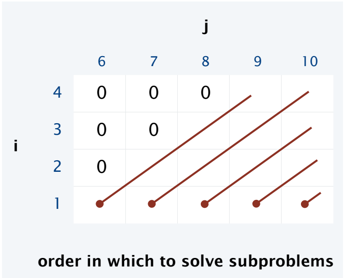
Bottom-up dynamic programming over intervals

Question. In which order to solve the subproblems?

Answer. Do shortest intervals first.

```
RNA($n, B_1, \ldots, b_n$)
FOR $k = 5$ TO $n - 1$
   FOR $i = 1$ TO $n - k$
      $j \leftarrow i + k$
$M[i,j] = \max(M[i,j-1], 1 + \max_t(M[i,t-1] + M[t+1,j-1]))$
RETURN $M[1,n]$.
```

$\uparrow$ take max over t such that $i \le t < j - 4$ and b_t and b_j are Watson-Crick complements



order in which to solve subproblems

RNA sequence *ACCGGUAGU*

	$j = 6$	7	8	9
4	0	0	0	
3	0	0		
2	0			
$i = 1$				

Initial values

	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			

Filling in the values for $k = 5$

	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		

Filling in the values for $k = 6$

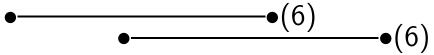
	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	

Filling in the values for $k = 7$

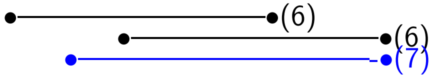
	$j = 6$	7	8	9
4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2

Filling in the values for $k = 8$

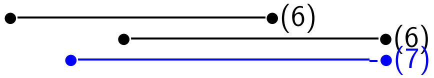
i						1	2	3	4	
j	1	2	3	4	5	**6**	7	8	9	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=4$



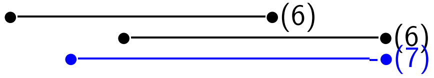
i						1	2	3	4	
j	1	2	3	4	5	6	**7**	8	9	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=5>4$



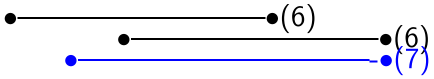
i						1	2	3	4	
j	1	2	3	4	5	6	7	**8**	9	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=6>4$



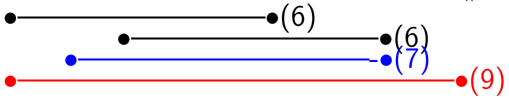
i						1	2	3	4	
j	1	2	3	4	5	6	7	**8**	9	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=6>4$



i						1	2	3	4	
j	1	2	3	4	5	6	7	8	**9**	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=7>4$



i						1	2	3	4	
j	1	2	3	4	5	6	7	8	**9**	
	A	**C**	**C**	**G**	**G**	**U**	**A**	**G**	**U**	$k-1=\mathbf{8}>4$



Bottom-up dynamic programming over intervals

Theorem

The dynamic programming algorithm solves the RNA secondary substructure problem in $O(n^3)$ time and $O(n^2)$ space.

Dynamic programming summary

Outline.

- Polynomial number of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from smallest to largest, with an easyto- compute recurrence that allows one to determine the solution to a subproblem from the solution to smaller subproblems.

Techniques.

- Binary choice: weighted interval scheduling.
- Multiway choice: segmented least squares.
- Adding a new variable: knapsack problem.
- Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up. Different people have different intuitions.