Algorithm Design and Implementation

Principle of Algorithms VIII

Dynamic Programming I

Guoqiang Li

School of Software, Shanghai Jiao Tong University

Algorithmic 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; combine solutions to subproblems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

- fancy name for caching intermediate results in a table for later reuse

Dynamic programming applications

Application areas.

- Computer science: AI, compilers, systems, graphics, theory, . . .
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

Some famous dynamic programming algorithms

- Avidan-Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman-Ford-Moore for shortest path.
- Knuth-Plass for word wrapping text in
- Cocke-Kasami-Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman for sequence alignment.

Dynamic programming books

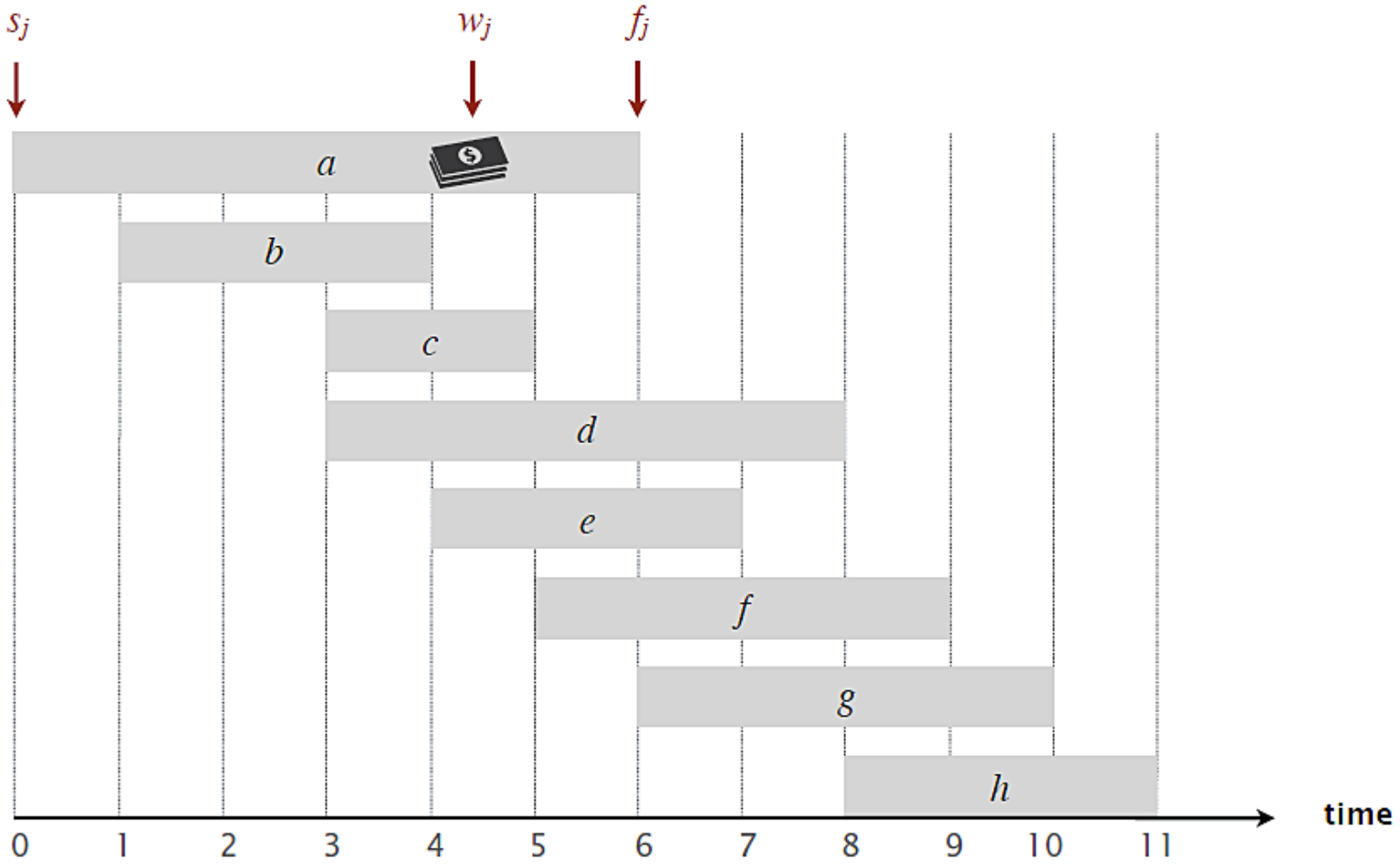
Weighted Interval Scheduling

Weighted interval scheduling

Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.

Two jobs are compatible if they don't overlap.

Goal: find max-weight subset of mutually compatible jobs.



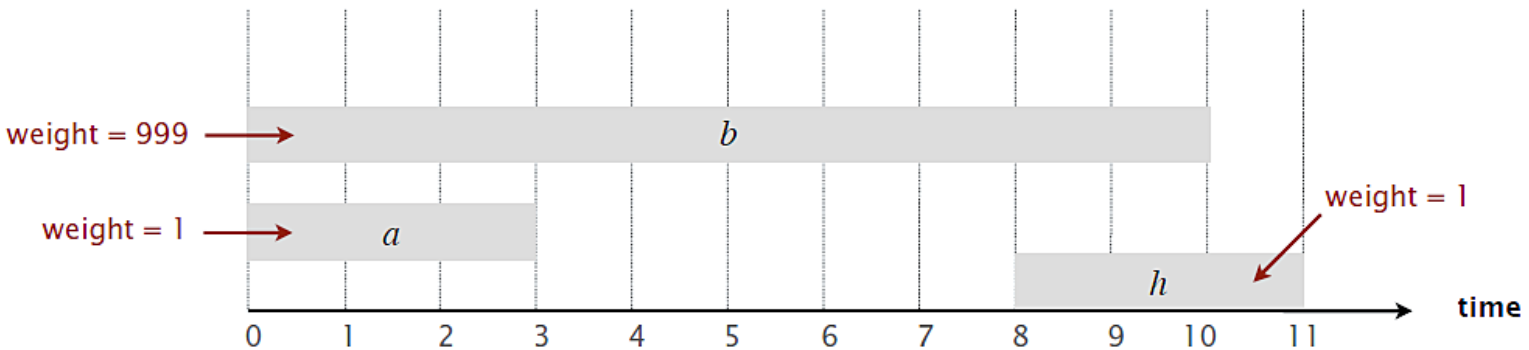
Earliest-finish-time first algorithm

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.

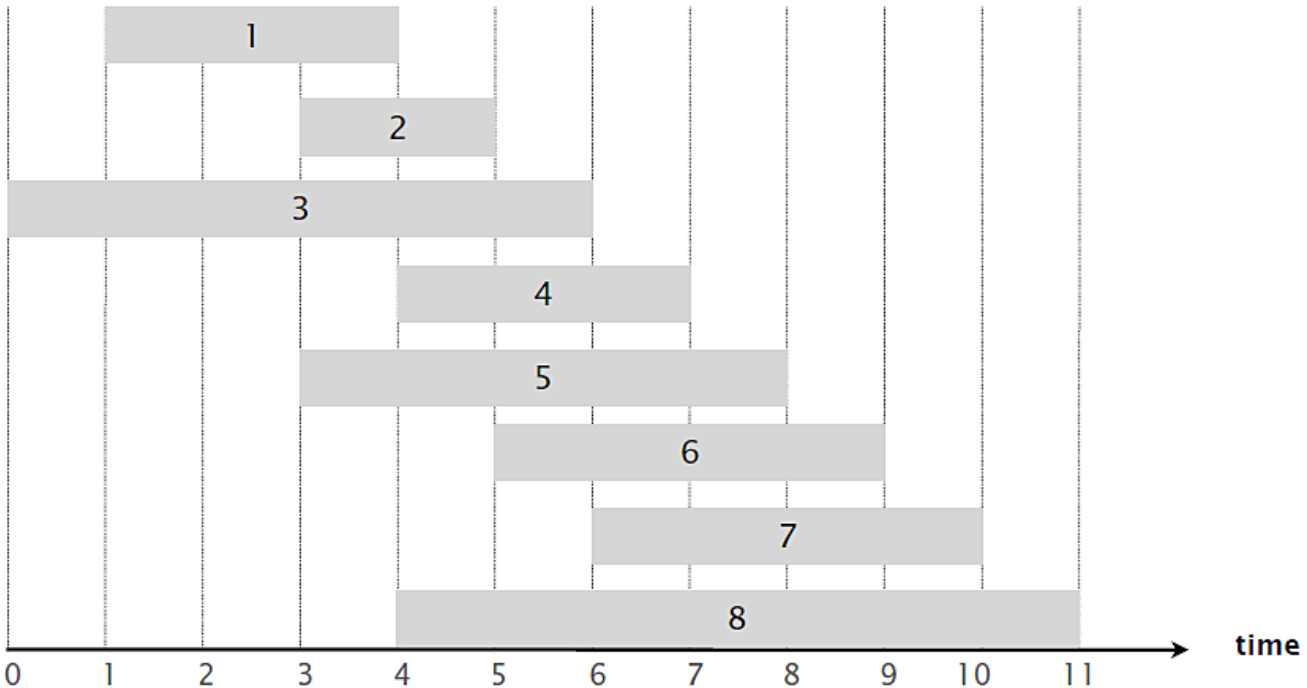


Weighted interval scheduling

Convention. Jobs are in ascending order of finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

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

Example. $p(8) = 1, p(7) = 3, p(2) = 0$



Dynamic programming: binary choice

$OPT(j)$: max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \ldots, j$.

Goal. $OPT(n)$: max weight of any subset of mutually compatible jobs.

Case 1. $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j-1$.

Case 2. $OPT(j)$ select job j.

- Collect profit w_j.
- Can't use incompatible jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

Dynamic programming: binary choice

Bellman equation.

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

Weighted interval scheduling: brute force

BruteForce($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$ via binary search;
Return ComputeOPT(n);

ComputeOPT(j)

if $(j = 0)$ **then**
  Return 0
end
else
  Return **max**{ComputeOPT(*j-1*), w_j + ComputeOPT(*p[j]*)}
end

Quiz 1

What is running time of ComputeOPT(n) in the worst case?

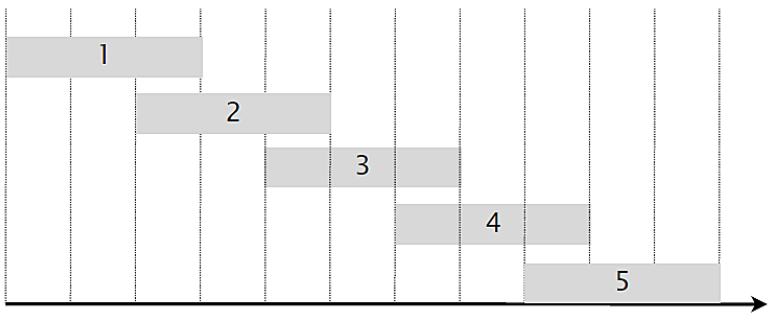
A. $\Theta(n \log n)$
B. $\Theta\left(n^2\right)$
C. $\Theta\left(1.618^n\right)$
D. $\Theta\left(2^n\right)$

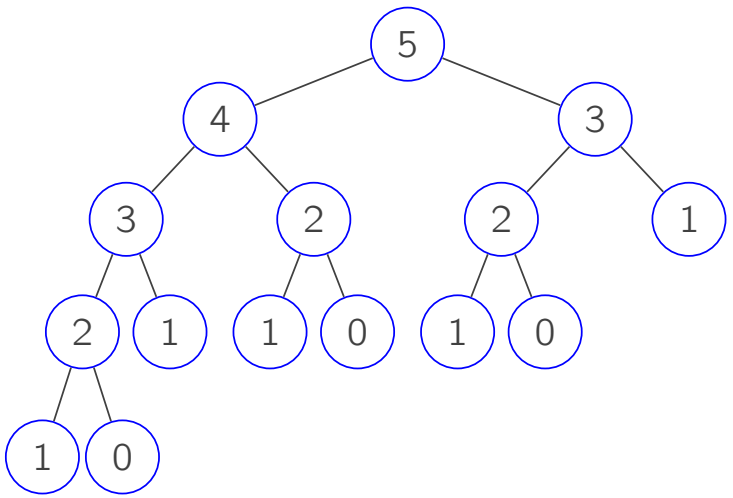
```
ComputeOPT(j)

if (j = 0) then
 |  Return 0
end
else
 |  Return max{ComputeOPT(j-1), w_j + ComputeOPT(p[j])}
end
```

Weighted interval scheduling: brute force

Observation. Recursive algorithm is spectacularly slow because of overlapping subproblems $\Longrightarrow$ exponential-time algorithm.

Example. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.





Weighted interval scheduling: memorization

TopDown dynamic programming (memorization).

- Cache result of subproblem j in $M[j]$.
- Use $M[j]$ to avoid solving subproblem j more than once.

Weighted interval scheduling: memorization

TopDown($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p[1], p[2], \cdots, p[n]$ via binary search;
M[0] $\leftarrow$ 0 ;
Return MemComputeOPT(n);

MemComputeOPT(j)

if $M[j]$ *is uninitialized* **then**
  $M[j] \leftarrow$
    max{MemComputeOPT(*j-1*), w_j + MemComputeOPT(*p[j]*)};
end
Return $M[j]$;

Weighted interval scheduling: running time

Claim

Memorized version of algorithm takes $O(n \log n)$ time.

Proof.

- Sort by finish time: $O(n \log n)$ via mergesort.
- Compute $p[j]$ for each j: $O(n \log n)$ via binary search.
- MemComputeOPT(j): each invocation takes $O(1)$ time and either
  - returns an initialized value $M[j]$
  - initializes $M[j]$ and makes two recursive calls
- Progress measure $\Phi = \#$ initialized entries among $M[1..n]$.
  - initially $\Phi = 0$; throughout $\Phi \leq n$.
  - increase Φ by $1 \Rightarrow \leq 2n$ recursive calls.
- Overall running time of MemComputeOPT(n) is $O(n)$.

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find optimal solution?

A. Make a second pass by calling FindSolution(n).

```
FindSolution(j)

if j = 0 then
    Return ∅;
end
else if w_j + M[p[j]] > M[j − 1] then
    Return {j} ⋃ FindSolution(p[j]);
end
else
    Return FindSolution(j − 1);
end
```

$$M[j] = \max\{M[j-1], w_j + M[p[j]]\}$$

Analysis. # of recursive calls $\leq n \Longrightarrow O(n)$.

Weighted interval scheduling: bottom-up dynamic programming

Bottom-up dynamic programming. Unwind recursion.

BottomUp($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber 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 **do**
  $M[j] \leftarrow \max\{M[j-1], w_j + M[p[j]]\}$
end

Running time. The bottom-up version takes $O(n \log n)$ time.

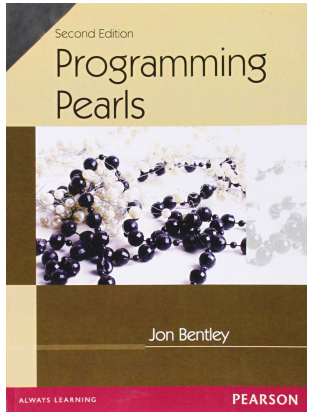
Maximum subarray problem

Goal. Given an array x of n integer (positive or negative), find a contiguous sub-array whose sum is maximum.

12	5	-1	31	-61	59	26	-53	58	97	-93	-23	84	-15	6

187

Applications. Computer vision, data mining, genomic sequence analysis, technical job interviews, . . .



Maximum rectangle problem

Goal. Given an $n \times n$ matrix A, find a rectangle whose sum is maximum.

$$A = \begin{bmatrix} -2 & 5 & 0 & -5 & -2 & 2 & -3 \\ 4 & -3 & -1 & 3 & 2 & 1 & -1 \\ -5 & 6 & 3 & -5 & -1 & -4 & -2 \\ -1 & -1 & 3 & -1 & 4 & 1 & 1 \\ 3 & -3 & 2 & 0 & 3 & -3 & -2 \\ -2 & 1 & -2 & 1 & 1 & 3 & -1 \\ 2 & -4 & 0 & 1 & 0 & -3 & -1 \end{bmatrix}$$

Applications. Databases, image processing, maximum likelihood estimation, technical job interviews, . . .

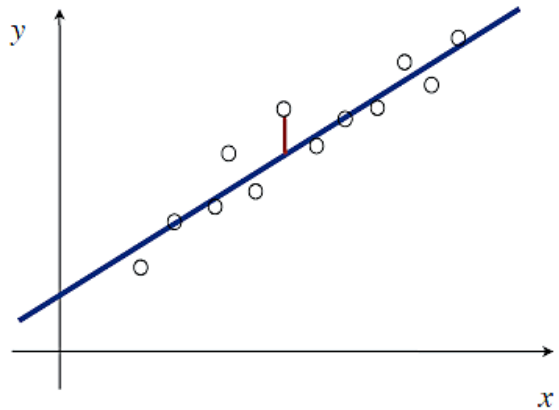
Segmented Least Squares

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:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



Solutions. Calculus $\Rightarrow$ min error is achieved when

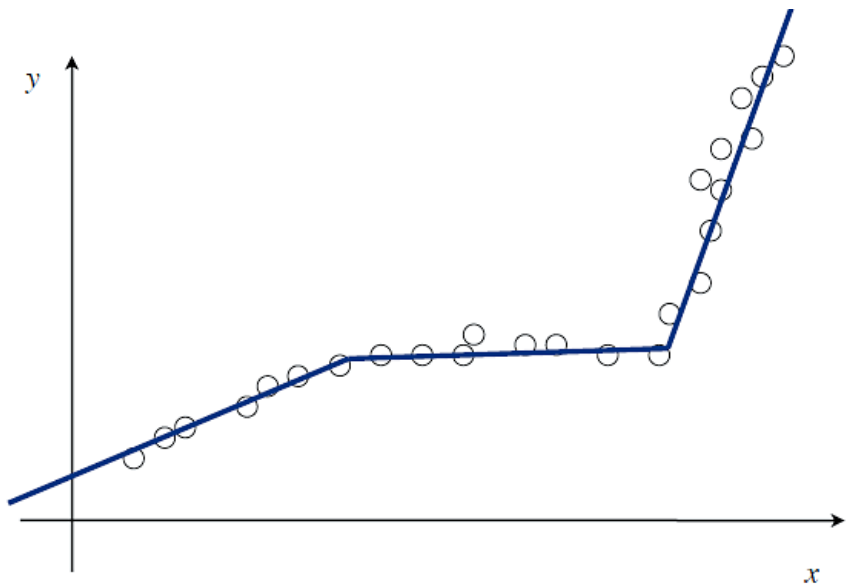
$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2}, \qquad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented least squares

Segmented least squares

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

Q. What is a reasonable choice for $f(x)$ to balance accuracy and parsimony?



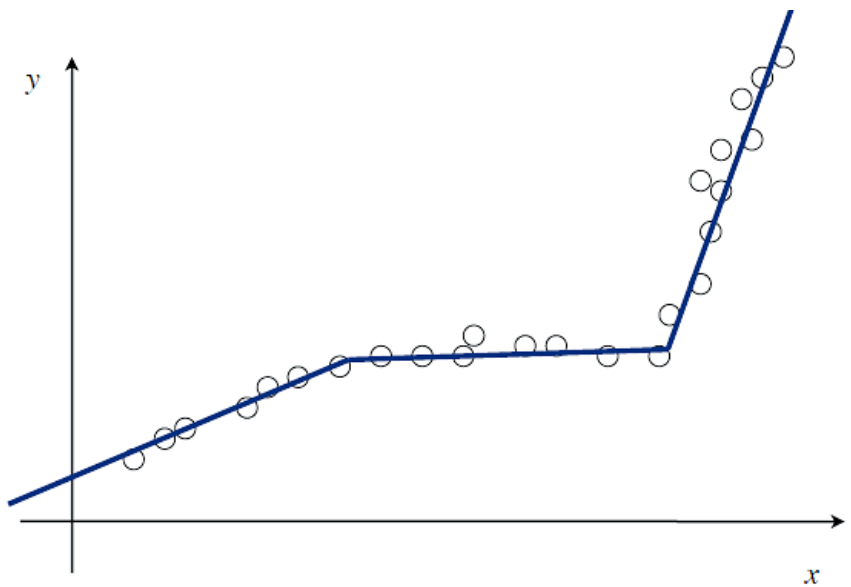
Segmented least squares

Segmented least squares

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

Goal. Minimize $f(x) = E + cL$ for some constant $c > 0$, where

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



Dynamic programming: multiway choice

Notation.

- $OPT(j)$: minimum cost for points $p_1, p_2, \ldots, p_j$.
- e_{ij}: SSE for points $p_i, p_{i+1}, \cdots, p_j$.

To compute $OPT(j)$:

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

Bellman equation.

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \leq i \leq j} \{e_{ij} + c + OPT(i-1)\} & \text{if } j > 0 \end{cases}$$

Segmented least squares algorithm

SegmentedLeastSquares($n, p_1, \ldots, p_n, c$)

for $j = 1$ *to* n **do**
  for $i = 1$ *to* j **do**
    Compute the SSE e_{ij} for the points $p_i, p_{i+1}, \ldots, p_j$;
  end
end
$M[0] \leftarrow 0$;
for $j = 1$ *to* n **do**
  $M[j] \leftarrow \min_{1 \le i \le j} \{e_{ij} + c + M[i-1]\}$;
end
Return $M[n]$;

Segmented least squares analysis

Theorem (Bellman 1961)

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

Proof.

- Bottleneck is computing SSE e_{ij} for each i and j.

$$a_{ij} = \frac{n \sum_k x_k y_k - \left(\sum_k x_k\right)\left(\sum_k y_k\right)}{n \sum_k x_k^2 - \left(\sum_k x_k\right)^2}, \qquad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ to compute e_{ij}.

Remark. Can be improved to $O(n^2)$ time.

- For each i: precompute cumulative sums $\sum_{k=1}^{i} x_k$, $\sum_{k=1}^{i} y_k$, $\sum_{k=1}^{i} x_k^2$, $\sum_{k=1}^{i} x_k y_k$
- Using cumulative sums, can compute e_{ij} in $O(1)$ time.

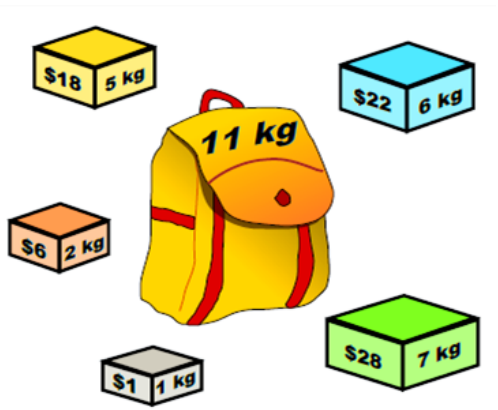
Knapsack Problem

Knapsack Problem

Goal. Pack knapsack so as to maximize total value.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Knapsack has weight capacity of W.

Assumption. All input values are integral.



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

knapsack instance
(weight limit $W = 11$)

Example. $\{1, 2, 5\}$ has value \$35 (and weight 10).
Example. $\{3, 4\}$ has value \$40 (and weight 11).

Dynamic programming: adding a new variable

Def. $OPT(i, w)$: max-profit subset of items $1, \ldots, i$ with weight limit w.
Goal. $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.

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

Case 2. $OPT(i, w)$ selects item i.

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

Dynamic programming: adding a new variable

Bellman equation.

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

Knapsack problem: bottom-up dynamic programming

Knapsack$(n, W, w_1, \cdots, w_n, v_1, \ldots, v_n)$

for $w = 0$ *to* W **do**
  $M[0, w] \leftarrow 0$;
end
for $i = 0$ *to* n **do**
  for $w = 0$ *to* W **do**
    if $(w_i > w)$ **then** $M[i, w] \leftarrow M[i-1, w]$;
    else $M[i, w] \leftarrow \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}$;
  end
end
Return $M[n, W]$;

Knapsack problem: bottom-up dynamic programming demo

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

	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

Knapsack problem: running time

Theorem

The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

Proof.

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

Quiz 4

Does there exist a poly-time algorithm for the knapsack problem?

A. Yes, because the DP algorithm takes $\Theta(nW)$ time.
B. No, because $\Theta(nW)$ is not a polynomial function of the input size.
C. No, because the problem is **NP**-hard.
D. Unknown.

Coin changing

Problem. Given n coin denominations $\{c_1, c_2, \ldots, c_n\}$ and a target value V, find the fewest coins needed to make change for V (or report impossible).

Recall. Greedy cashier's algorithm is optimal for U.S. coin denominations, but not for arbitrary coin denominations.

Example. $\{1, 10, 21, 34, 70, 100, 350, 1295, 1500\}$.
Optimal. $140¢ = 70 + 70$.

Exercise

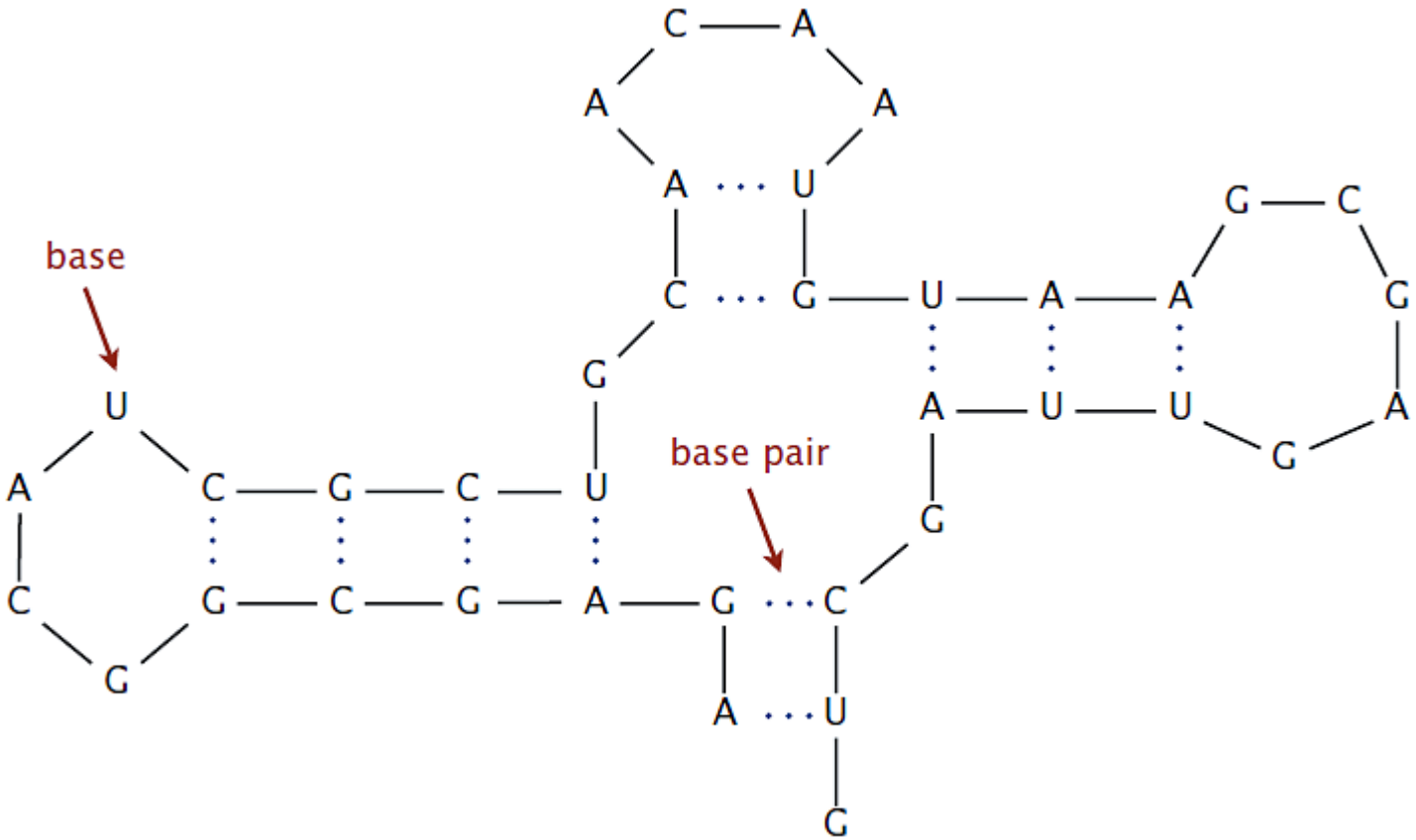
What if infinitely many copies of each item are allowed?

RNA Secondary Structure

RNA secondary structure

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

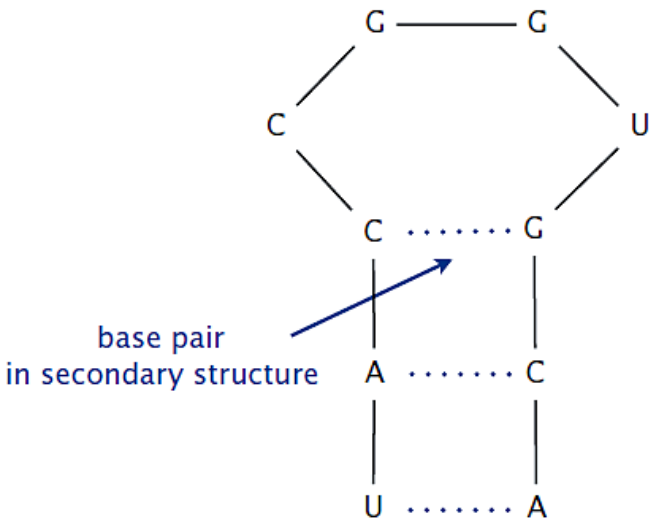
Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.



RNA secondary structure

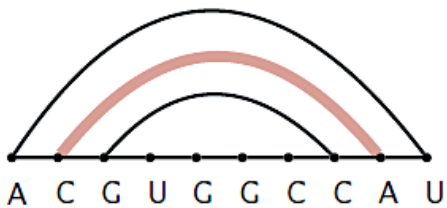
Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick component: $A - U$, $U - A$, $C - G$, or $G - C$.



$B = \text{ACGUGGCCCAU}$

$S = \{ (b_1, b_{10}),\ (b_2, b_9),\ (b_3, b_8) \}$

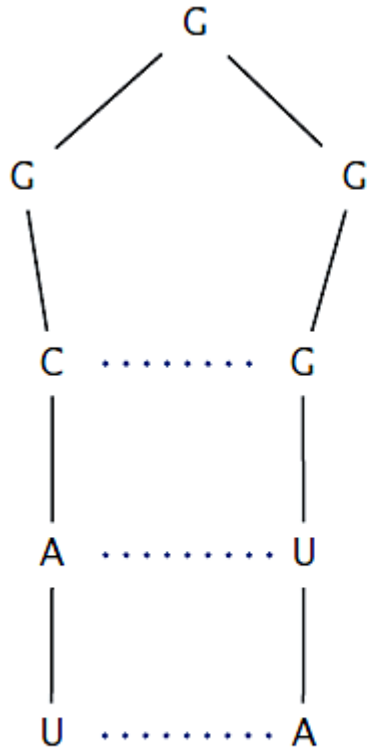


S is not a secondary structure
(C–A is not a valid Watson–Crick pair)

RNA secondary structure

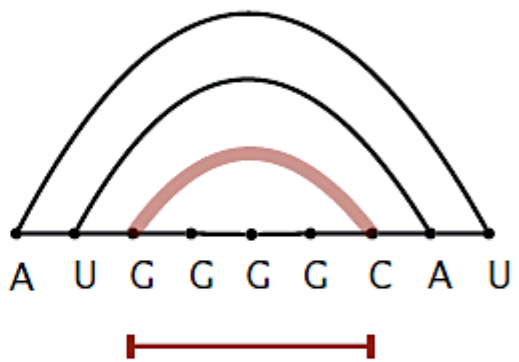
Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick component: $A - U$, $U - A$, $C - G$, 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 $i < j - 4$.



$B = \text{AUGGGGCAU}$

$S = \{ (b_1, b_9),\ (b_2, b_8),\ (b_3, b_7) \}$

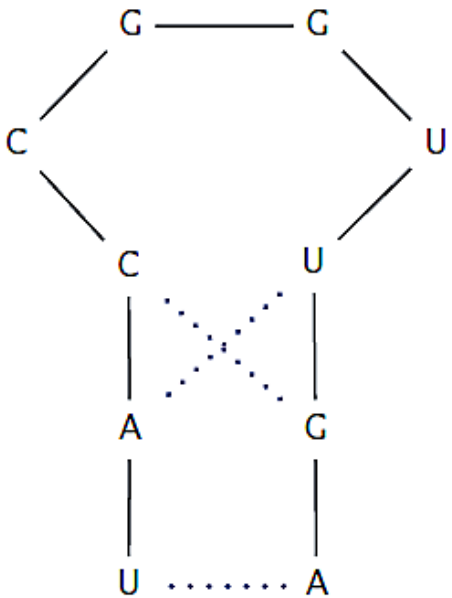


S is not a secondary structure
(≤4 intervening bases between G and C)

RNA secondary structure

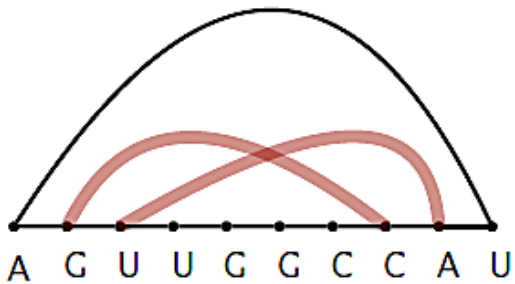
Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick component: $A - U$, $U - A$, $C - G$, 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 $i < k < j < l$.



$B = \text{ACUUGGCCAU}$

$S = \{ (b_1, b_{10}),\ (b_2, b_8),\ (b_3, b_9) \}$

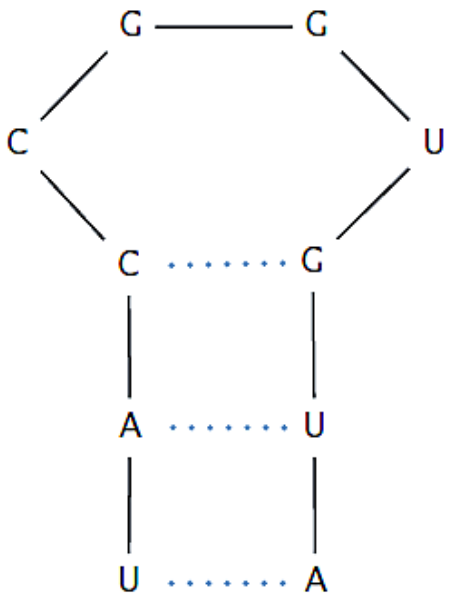


S is not a secondary structure (G–C and U–A cross)

RNA secondary structure

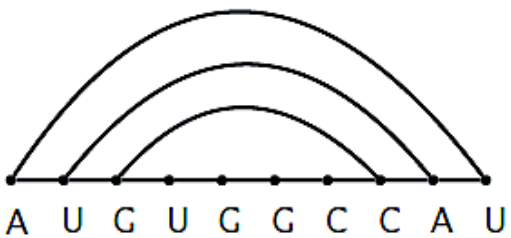
Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick] S is a matching and each pair in S is a Watson-Crick component: $A - U$, $U - A$, $C - G$, 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 $i < k < j < l$.



$B = \text{AUGUGGCCAU}$

$S = \{ (b_1, b_{10}),\ (b_2, b_9),\ (b_3, b_8) \}$



S is a secondary structure (with 3 base pairs)

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 component: $A - U$, $U - A$, $C - G$, 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 hypothesis. RNA molecule will form the secondary structure with the minimum total free energy, approximated by number of base pairs.

- more base pairs $\Rightarrow$ lower free energy.

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.

Quiz 5

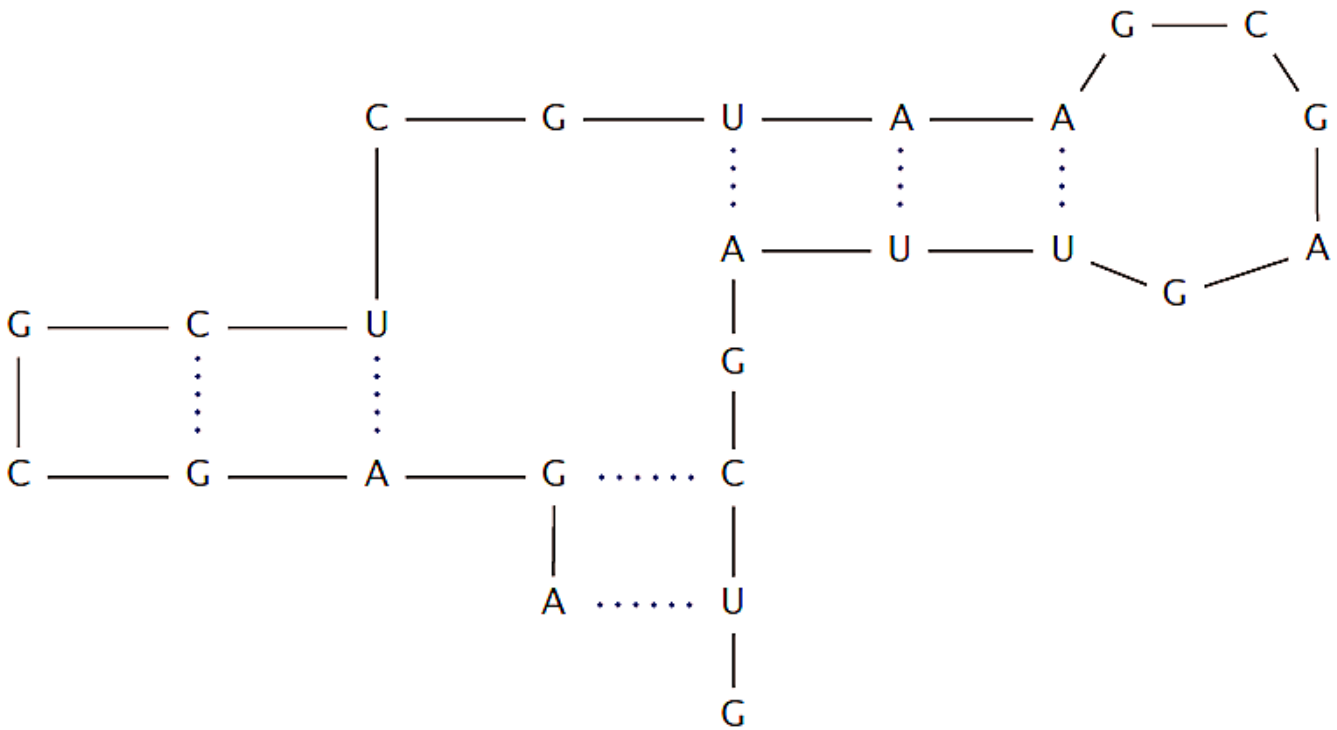
Is the following a secondary structure?

A. Yes.

B. No, violates Watson–Crick condition.

C. No, violates no-sharp-turns condition.

D. No, violates no-crossing condition.



Quiz 6

Which subproblems?

A. $OPT(j)$: max number of base pairs in secondary structure of the substring $b_1 b_2 \ldots b_j$.

B. $OPT(j)$: max number of base pairs in secondary structure of the substring $b_j b_{j+1} \ldots b_n$.

C. Either A or B.

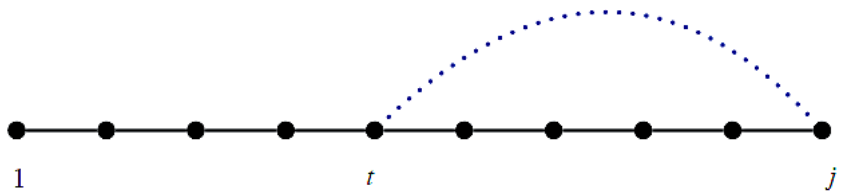
D. Neither A nor B.

RNA secondary structure: subproblems

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

Goal. $OPT(n)$

Choice. Match bases b_i and b_j.



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

- Find secondary structure in $b_1 b_2 \ldots b_{t-1}$.
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{j-1}$.

Dynamic programming over intervals

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

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 $i \leq t < j–4$.

- Non-crossing condition decouples resulting two subproblems.
- $OPT(i, j) = 1 + \max_t \{OPT(i, t - 1) + OPT(t + 1, j - 1)\}$.

Quiz 7

In which order to compute $OPT(i, j)$?

A. Increasing i, then j.

B. Increasing j, then i.

C. Either A or B.

D. Neither A nor B.

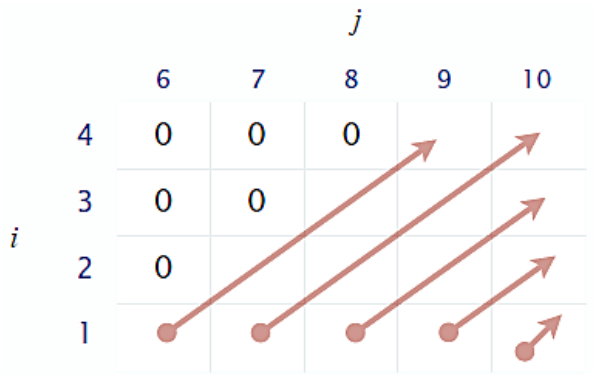
Bottom-up dynamic programming over intervals

Q. In which order to solve the subproblems?

A. Do shortest intervals first—increasing order of $|j - i|$.

RNASecondaryStructure$(n, b_1, \ldots, b_n)$

for $k = 5$ *to* $n - 1$ **do**
  for $i = 1$ *to* $n - k$ **do**
    $j \leftarrow i + k$;
    Compute $M[i, j]$ using formula;
  end
end
Return $M[1, n]$;



Complexity result

Theorem

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

Dynamic programming summary

Outline.

- Define a collection of subproblems.
- Solution to original problem can be computed from subproblems.
- Natural ordering of subproblems from "smallest" to "largest" that enables determining a solution to a subproblem from solutions to smaller subproblems.

Techniques.

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

Top-down vs. bottom-up dynamic programming. Opinions differ.