Algorithm II

6. Dynamic Programming I

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Algorithmic paradigms

Greedy. *Myopically* ordering, making *irrevocable* decisions.

- possibly, *no* natural greedy strategy.

Divide-and-conquer. Break up a problem into *independent* sub-problems; solve each subproblem; combine solutions to sub-problems (form solution to original problem).

- Not strong enough: reduce polynomial to faster running time.

Algorithmic paradigms

Greedy. *Myopically* ordering, making *irrevocable* decisions.

- possibly, *no* natural greedy strategy.

Divide-and-conquer. Break up a problem into *independent* sub-problems; solve each subproblem; combine solutions to sub-problems (form solution to original problem).

- Not strong enough: reduce polynomial to faster running time.

Dynamic programming. Break up a problem into a series of *overlapping* sub-problems; combine solutions to smaller sub-problems (form solution to larger subproblem).

- opposite of greedy: work through all possible global optimal.
  - explores exponentially large space, but not examining explicitly.

Weighted Interval Scheduling

Weighted Interval Scheduling Problem

Consider n subjects are sharing a *single* resources.

- job j: start at s_j and finish at f_j
  - has *weight/value* $w_j > 0$
- two jobs are **compatible** if they do not overlap

Weighted Interval Scheduling Problem

Consider n subjects are sharing a *single* resources.

- job j: start at s_j and finish at f_j
  - has *weight/value* $w_j > 0$
- two jobs are **compatible** if they do not overlap

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

	0	1	2	3	4	5	6	7	8	9
9	+	+	+	+	+	+	+			
1	+	+	+	+						
1							+	+	+	+

Earliest-finish-time-first algorithm

Earliest-finish-time-first.

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

Recall. Greedy algorithm is correct if all weights are 1.

Earliest-finish-time-first algorithm

Earliest-finish-time-first.

- Consider jobs in ascending order of finish time f_j.
- 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 for weighted version.

- goal and progress measure are *unrelated*.
- previous greedy decisions are *irrevocable*.

	0	1	2	3	4	5	6	7	8	9
9	+	+	+	+	+	+	+			
1	+	+	+	+						
1							+	+	+	+

EFTF extension

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

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

- i is *rightmost* interval that ends before j begins

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	p
1	+	+	+	+							0
9	+	+	+	+	+	+	+				0
1							+	+	+	+	1

Dynamic programming: binary choice

Def. $OPT(j)$ = max weight of *any subset* of mutually compatible jobs (for subproblem consisting only of jobs $1, 2, ..., j$).

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

Dynamic programming: binary choice

Def. $OPT(j)$ = max weight of *any subset* of mutually compatible jobs (for subproblem consisting only of jobs $1, 2, ..., j$).

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

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

- Optimal solution to *smaller problem*: remaining jobs $1, 2, ..., j-1$.

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

- Collect profit w_j.
- Can't use incompatible jobs $\{p(j) + 1, p(j) + 2, ..., j-1\}$.
- Optimal solution to *smaller problem*: remaining compatible jobs $1, 2, ..., p(j)$.

DP: binary choice (cont.)

Def. $OPT(j)$ = max weight of *any subset* of mutually compatible jobs (for subproblem consisting only of jobs $1, 2, ..., j$).

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

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

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

Bellman equation.

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

Brute-force scheduling

BRUTE-FORCE$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. Compute $p[1], p[2], \ldots, p[n]$ via binary search;
3. RETURN COMPUTE-OPT(n);

COMPUTE-OPT(j)

1. IF $(j = 0)$: RETURN 0;
2. ELSE:
   1. RETURN $\max$ { COMPUTE-OPT$(j–1)$, w_j + COMPUTE-OPT$(p[j])$ };

Quiz: brute-force scheduling

What is running time of COMPUTE-OPT(n) in the worst case?

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

COMPUTE-OPT(j)

1. IF ($j = 0$): RETURN 0;
2. ELSE:
   1. RETURN $\max$ { COMPUTE-OPT(j–1), w_j + COMPUTE-OPT($p[j]$) };

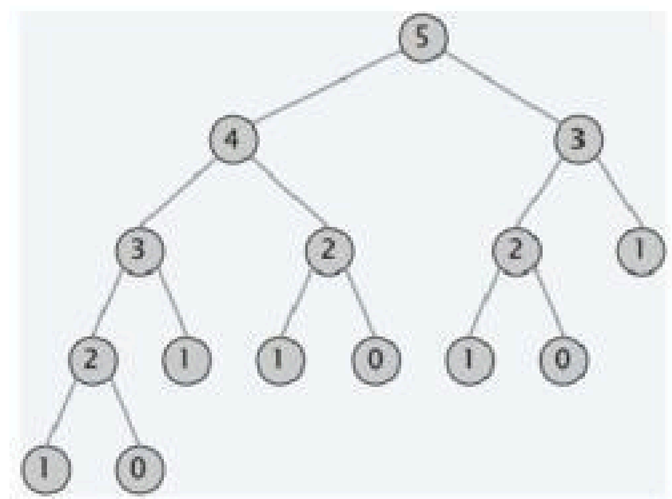
Discussed next.

Brute-force scheduling: analysis

Observation. Recursive algorithm is spectacularly slow because of overlapping sub-problems ⇒ *exponential*-time algorithm.

Ex. # recursive calls for family of “layered” instances grows like Fibonacci sequence.

0	1	2	3	4	5
+	+	+			
	+	+	+		
		+	+	+	
			+	+	+

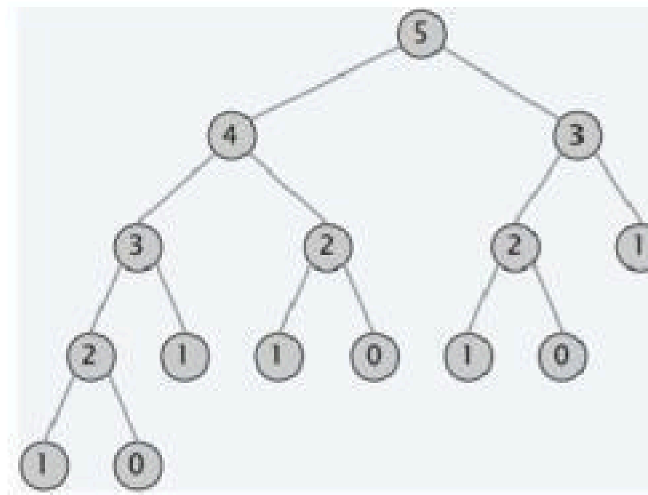


Brute-force scheduling: analysis

Observation. Recursive algorithm is spectacularly slow because of overlapping sub-problems ⇒ *exponential*-time algorithm.

Ex. # recursive calls for family of “layered” instances grows like Fibonacci sequence.

0	1	2	3	4	5
+	+	+			
	+	+	+		
		+	+	+	
			+	+	+



Key insight. Avoid repeated computations using memory.

Memoized scheduling

Top-down dynamic programming (memoization).

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

Memoized scheduling

Top-down dynamic programming (memoization).

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

TOP-DOWN($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. Compute $p[1], p[2], \ldots, p[n]$ via binary search;
3. $M[0] = 0$;
4. RETURN M-COMPUTE-OPT(n);

M-COMPUTE-OPT(j)

1. IF ($M[j]$ is uninitialized):
   1. $M[j] = \max\{$ M-COMPUTE-OPT(j–1), w_j + M-COMPUTE-OPT($p[j]$) $\}$;
2. RETURN $M[j]$;

Memoized scheduling: analysis

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

Pf.

- Sort by finish time: $O(n \log n)$.
- Compute $p[j]$ for each j: $O(n \log n)$ via binary search.
- `M-COMPUTE-OPT`(j): each invocation takes $O(1)$ time and either
  - 1. returns an initialized value $M[j]$
  - 2. initializes $M[j]$ and makes two recursive calls

Memoized scheduling: analysis

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

Pf.

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

Memoized scheduling: find a solution

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

Memoized scheduling: find a solution

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

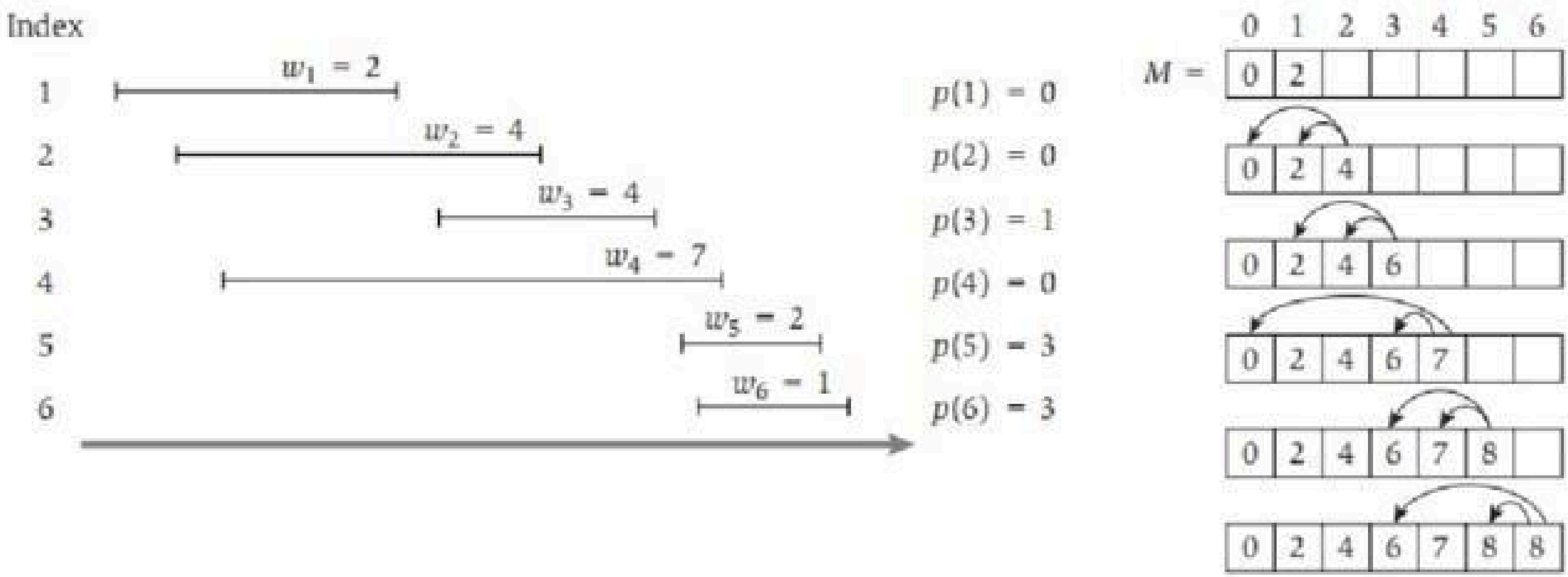
A. Make a second pass, ie., backtrace.

FIND-SOLUTION(j)

1. IF ($j = 0$): RETURN $\emptyset$;
2. ELSE IF ($w_j + M[p[j]] > M[j\text{–}1]$):
   1. RETURN $\{j\} \cup$ FIND-SOLUTION($p[j]$);
3. ELSE:
   1. RETURN FIND-SOLUTION(j–1);

Bottom-up DP

Bottom-up dynamic programming. Unwind recursion.



Bottom-up DP: algorithm

BOTTOM-UP$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. Compute $p[1], p[2], \ldots, p[n]$ via binary search;
3. $M[0] = 0$;
4. FOR $j = 1..n$:
    1. $M[j] = \max\{M[j{-}1], w_j + M[p[j]]\}$;

Bottom-up DP: algorithm

BOTTOM-UP$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

1. SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$;
2. Compute $p[1], p[2], \ldots, p[n]$ via binary search;
3. $M[0] = 0$;
4. FOR $j = 1..n$:
    1. $M[j] = \max\{M[j\text{–}1], w_j + M[p[j]]\}$;

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

Maximum Sub-array Problem

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

Ex.

sum	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, etc.

Maximum Sub-array: brute-force

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

Ex.

sum	12	5	–1	31	–61	59	26	–53	58	97	–93	–23	84	–15	6
187						+	+	+	+	+					

Brute-force algorithm.

- For each i and j : computer $a[i] + a[i+1] + \ldots + a[j]$.
- Takes $\Theta(n^3)$ time.

Maximum Sub-array: brute-force

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

Ex.

sum	12	5	–1	31	–61	59	26	–53	58	97	–93	–23	84	–15	6
187						+	+	+	+	+					

Brute-force algorithm.

- For each i and j : computer $a[i] + a[i+1] + \ldots + a[j]$.
- Takes $\Theta(n^3)$ time.

Apply "cumulative sum" trick.

- Pre-compute cumulative sums: $S[i] = a[0] + a[1] + \ldots + a[i]$.
- Now $a[i] + a[i+1] + \ldots + a[j] = S[j] - S[i-1]$.
- Improves running time $\Theta(n^2)$.

Kadane's algorithm

Def. $OPT(i)$ = max sum of any sub-array of x whose rightmost index is i.

Goal. $\max_i OPT(i)$

Kadane's algorithm

Def. $OPT(i)$ = max sum of any sub-array of x whose rightmost index is i.

Goal. $\max_i OPT(i)$

Bellman equation.

$$OPT(i) = \begin{cases} x_1 & \text{if} \quad i = 1 \\ \max\{x_i, x_i + OPT(i-1)\} & \text{if} \quad i > 1 \end{cases}$$

- take only element i
- take i and best sub-array ending at $i - 1$

Kadane's algorithm

Def. $OPT(i)$ = max sum of any sub-array of x whose rightmost index is i.

Goal. $\max_i OPT(i)$

Bellman equation.

$$OPT(i) = \begin{cases} x_1 & \text{if} \quad i = 1 \\ \max\{x_i, x_i + OPT(i-1)\} & \text{if} \quad i > 1 \end{cases}$$

- take only element i
- take i and best sub-array ending at $i - 1$

Running time. $O(n)$.

Maximum Rectangle Problem

Goal. Given an n-by-n matrix A, find a rectangle whose sum is maximum.

			+	+	+		
	–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

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

Bentley's algorithm

Assumption. Suppose you knew the left and right column indices j and j'.

		j		j'		
–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

x
–7
4
–3
6
5
0
1

Bentley's algorithm (cont.)

An $O(n^3)$ algorithm.

1. Pre-compute cumulative row sums: $\sum_{k=1}^{j} A_{ik}$;
2. For each $j < j'$:
   1. define array x using row-sum differences: $x_i = S_{ij'} - S_{ij}$;
   2. run Kadane's algorithm in array x;

Open problem. $O(n^{3-\epsilon})$ for any constant $\epsilon > 0$.

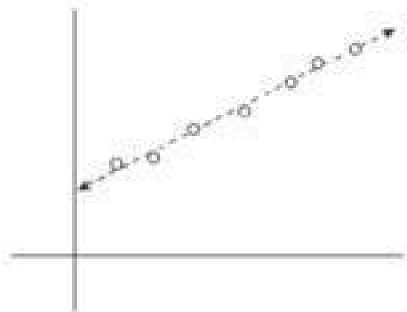
Segmented least squares

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:

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

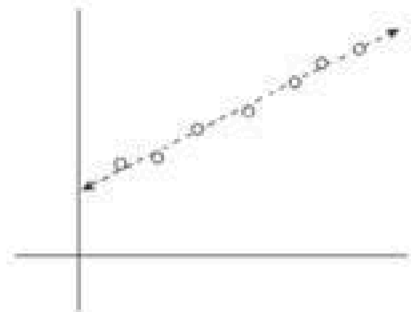


Least squares

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:

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



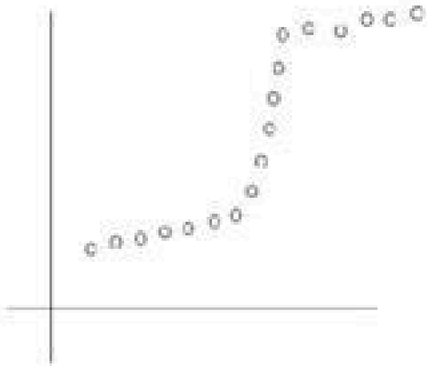
Solution. Calculus $\Rightarrow$ min error is achieved when

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

Segmented Least Squares Problem

Segmented least squares.

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

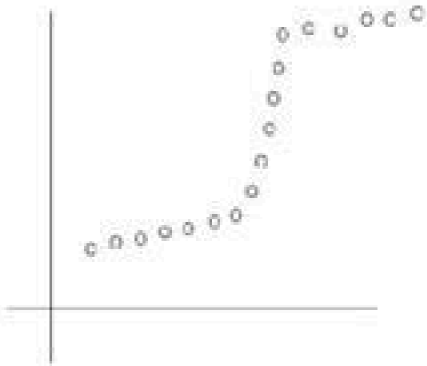


≡

Segmented Least Squares Problem

Segmented least squares.

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



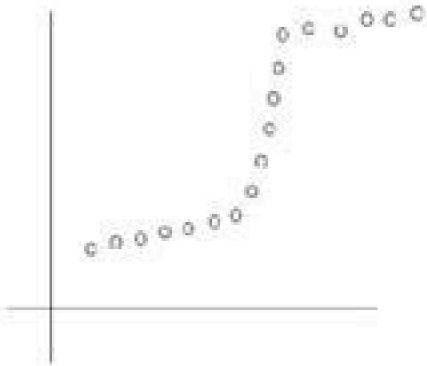
Q. What is a reasonable choice for $f(x)$ to balance *accuracy* and *parsimony*?

- goodness of fit vs. number of lines

Segmented Least Squares Problem

Segmented least squares.

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



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

- goodness of fit vs. number of lines

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

- E = sum of SSEs in each segment.
- L = number of lines.

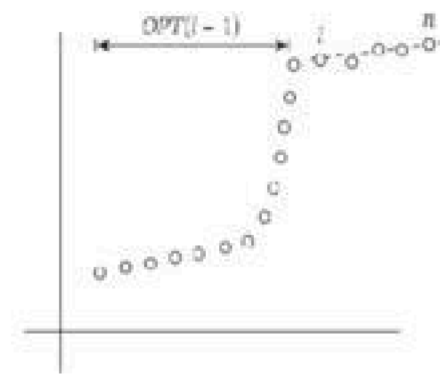
Dynamic programming: multi-way choice

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- e_{ij} = SSE for 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 $i \leq j$.
- Cost = $e_{ij} + c + OPT(i–1)$.

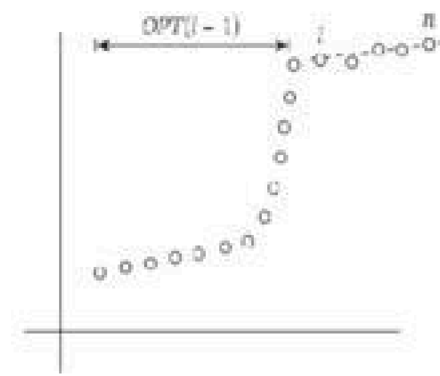


Dynamic programming: multi-way choice

Notation.

- $OPT(j)$ = minimum cost for points $p_1, p_2, \ldots, p_j$.
- e_{ij} = SSE for 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 $i \leq j$.
- Cost = $e_{ij} + c + OPT(i\text{–}1)$.

Bellman equation.

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

Segmented Least Squares: algorithm

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

1. FOR $j = 1..n$:
    1. FOR $i = 1..j$:
        1. Compute the SSE e_{ij} for the points $p_i, p_{i+1}, \ldots, p_j$;
2. $M[0] = 0$;
3. FOR $j = 1..n$:
    1. $M[j] = \min_{1 \le i \le j}\{e_{ij} + c + M[i-1]\}$;
4. 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.

Pf.

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

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

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

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

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

Knapsack problem

Knapsack problem

Goal. Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.

Assumption. All values and weights are integral.

Ex. The subset { 1, 2, 5 } has value 35 (and weight 10).

Ex. The subset { 3, 4 } has value 40 (and weight 11).

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

weight limit $W = 11$

Quiz: Knapsack via greedy

Which algorithm solves knapsack problem?

A. Greedy-by-value: repeatedly add item with maximum v_i.
B. Greedy-by-weight: repeatedly add item with minimum w_i.
C. Greedy-by-ratio: repeatedly add item with maximum ratio v_i / v_i.
D. None of the above.

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

weight limit $W = 11$

Quiz: Knapsack via DP

Which sub-problems?

A. $OPT(w)$ = optimal value of knapsack problem with weight limit w.
B. $OPT(i)$ = optimal value of knapsack problem with items $1, \ldots, i$.
C. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$ subject to weight limit W.
D. Any of the above.

Quiz: Knapsack via DP

Which sub-problems?

A. $OPT(w)$ = optimal value of knapsack problem with weight limit w.
B. $OPT(i)$ = optimal value of knapsack problem with items $1, \ldots, i$.
C. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$ subject to weight limit W.
D. Any of the above.

A/B: not eliminating any conflict, thus reduce to brute-force.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

Dynamic programming: two variables

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

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

- $OPT(i, w)$ selects best of $\{1, 2, \ldots, i\text{–}1\}$ subject to 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 $\{1, 2, \ldots, i\text{–}1\}$ subject to new weight limit.

DP: two variables (cont.)

Def. $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w.

Goal. $OPT(n, W)$.

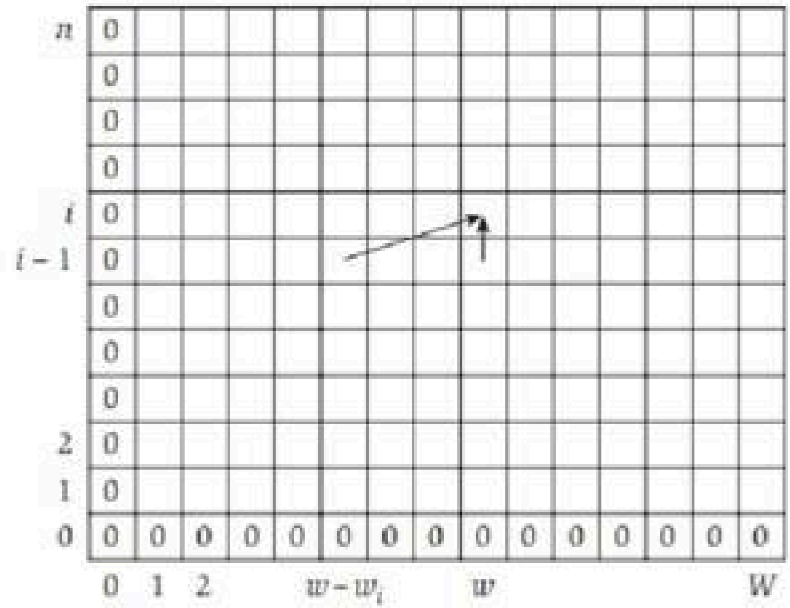
Case 1. $OPT(i, w)$ does not select item i.

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

Bellman equation.

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

DP: two-dimensional table



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

Knapsack problem: bottom-up DP

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

1. FOR $w = 0..W$: $M[0, w] = 0$;
2. FOR $i = 1..n$:
    1. FOR $w = 0..W$:
        1. IF $(w_i > w)$: $M[i, w] = M[i\text{–}1, w]$;
        2. ELSE: $M[i, w] = \max\{M[i\text{–}1, w], v_i + M[i\text{–}1, w\text{–}w_i]\}$;
3. RETURN $M[n, W]$;

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

Knapsack problem: bottom-up DP demo

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

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

Initial values

	0	1	2	3	4	5	6
3							
2							
(1)	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0

Filling in values for $i = 1$

	0	1	2	3	4	5	6
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

Filling in values for $i = 2$

	0	1	2	3	4	5	6
(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

Filling in values for $i = 3$

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

Knapsack problem: analysis

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

Pf.

- 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 $M[i, w] > M[i-1, w]$.

Remarks.

- Algorithm depends critically on assumption that weights are *integral*.
  - weights are integers between 1 and W

Coin changing: revisit

Problem. Given n coin denominations $\{d_1, d_2, \ldots, d_n\}$ 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.

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

Coin changing: DP solution

Def. $OPT(v)$ = min number of coins to make change for v.

Goal. $OPT(V)$.

Multiway choice. To compute $OPT(v)$,

- Select a coin of denomination c_i for some i.
- Select fewest coins to make change for $v - c_i$.

Bellman equation.

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min_{1 \le i \le n} \{1 + OPT(v - d_i)\} & \text{if } v > 0 \end{cases}$$

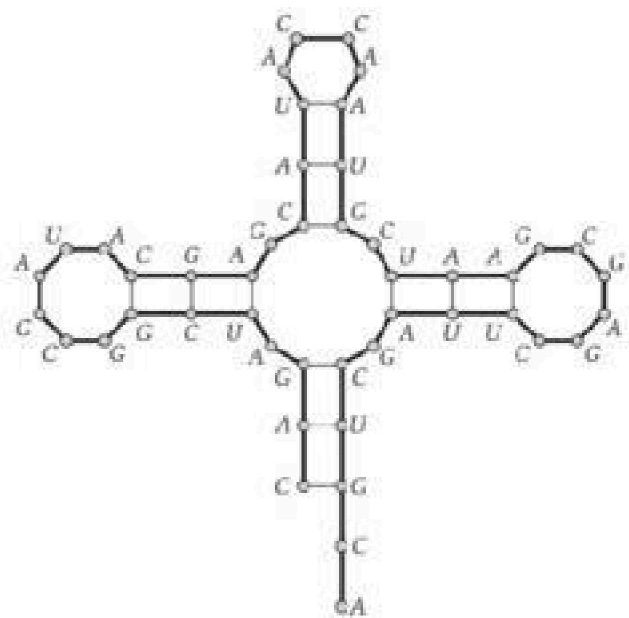
Running time. $O(nV)$.

RNA secondary structure

RNA secondary structure

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

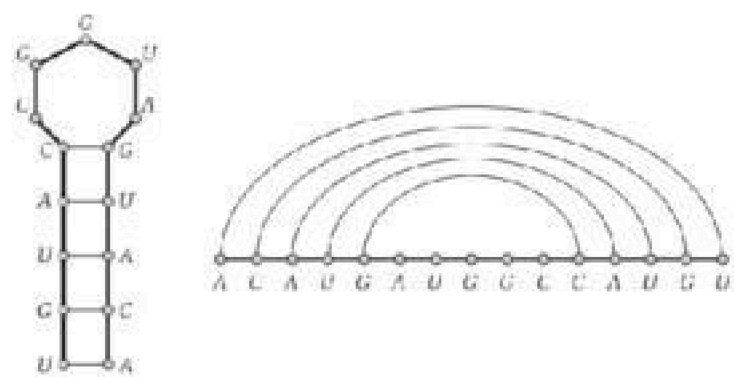
Secondary structure. RNA is single-stranded so it tends to loop back and form *base pairs* with itself. This structure is essential for understanding behavior of molecule.



RNA: matching rule

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, bj) and (b_k, bl) are two pairs in S, then we cannot have $i < k < j < l$.



RNA: hypothesis

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, bj) and (b_k, bl) are two pairs in S, then we cannot have $i < k < j < l$.

Free-energy hypothesis. RNA molecule will form secondary structure with *minimum total free energy*.

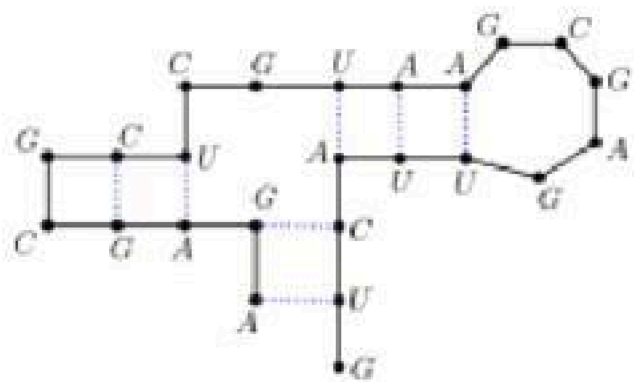
- approximate by # base pairs: more base pairs $\Rightarrow$ lower free energy

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

Quiz: matching rule

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: RNA secondary structure

Which sub-problems?

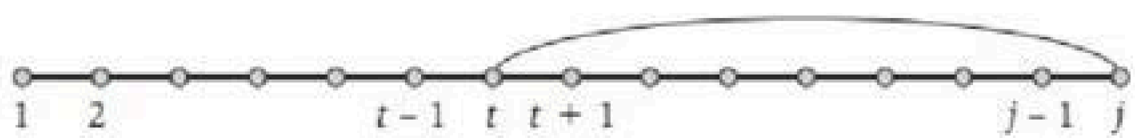
A. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_1 b_2 \dots b_j$.
B. $OPT(j)$ = max number of base pairs in secondary structure of the substring $b_j b_{j+1} \dots b_n$.
C. Either A or B.
D. Neither A nor B.

RNA secondary structure: sub-problems

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

Goal. $OPT(n)$.

Choice. Match bases b_t and b_j.

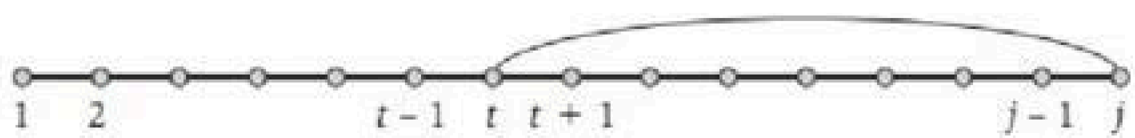


RNA secondary structure: sub-problems

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

Goal. $OPT(n)$.

Choice. Match bases b_t and b_j.



Difficulty. Results in two sub-problems (but one of wrong form).

- Find secondary structure in $b_1 b_2 \ldots b_{t-1}$: $OPT(t-1)$.
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{j-1}$.
  - break sub-structure: first base no longer b_1

DP: intervals

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

Case 1. If $i \geq j–4$.

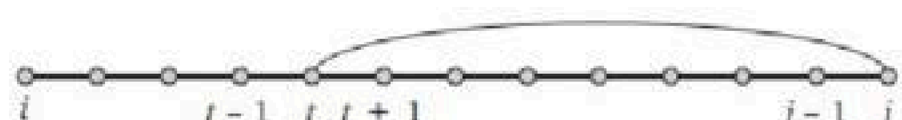
- $OPT(i, j) = 0$ by no-sharp-turns condition.

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 sub-problems.
  - $OPT(i, j) = 1 + \max_t OPT(i, t–1) + OPT(t + 1, j–1)$.



Quiz: DP for RNA

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**.

Quiz: DP for RNA

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**.

B

Bottom-up DP over intervals

Q. In which order to solve the sub-problems?
A. Do shortest intervals first—increasing order of $|j - i|$.

Ex. 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 $k = 8$

DP for RNA: algorithm

RNA-SECONDARY-STRUCTURE$(n, b_1, \ldots, b_n)$

1. FOR $k = 5..n-1$:
   1. FOR $i = 1..n-k$:
      1. $j = i + k$;
      2. Compute $M[i, j]$ using formula;
2. RETURN $M[1, n]$;

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 (polynomial number of) sub-problems.
- Solution to original problem can be computed from sub-problems.
- Natural ordering of sub-problems from "smallest" to "largest" that enables determining a solution to a subproblem from solutions to smaller sub-problems.

Techniques.

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

Top-down vs. bottom-up DP. recursive vs. iterative.