Dynamic Programming Part 1

CS240 Spring 2023

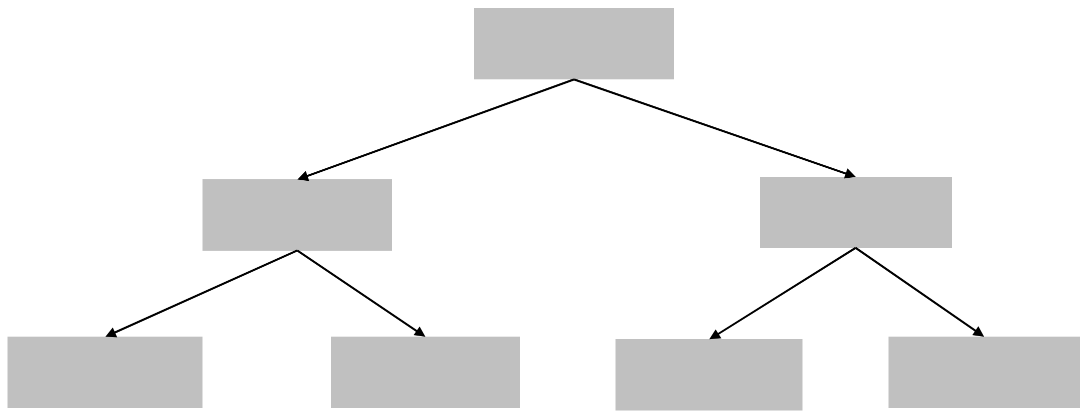
Rui Fan

Algorithmic paradigms

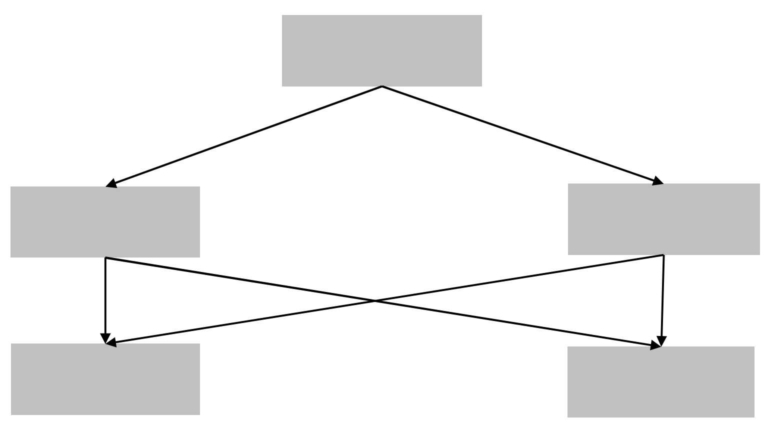
- Greedy Build up a solution incrementally, myopically optimizing some local criterion.
- Divide and conquer Break up a problem into a few sub-problems, solve each sub-problem independently and recursively, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  - Very powerful and widely used technique in CS, OR, info and control theory.
  - Efficiently solves problems that otherwise seem intractable.
  - Name comes from dynamic "schedule" of subproblems the algorithm produces.

Algorithmic paradigms

Divide and conquer

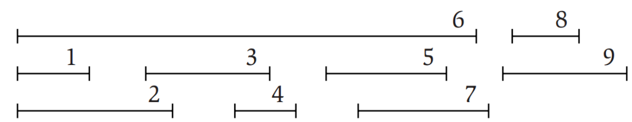


Dynamic programming

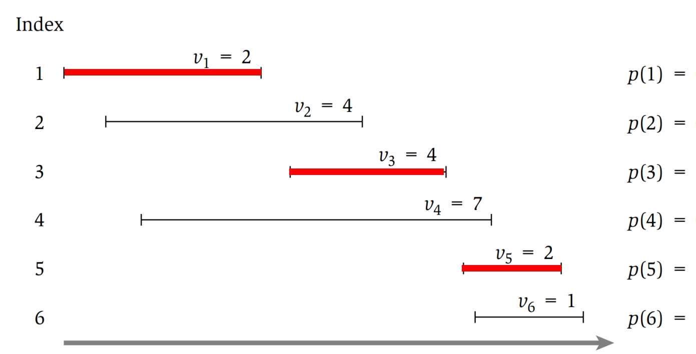


Weighted interval scheduling

- Recall the interval scheduling problem
  - Given a set of intervals, pick the largest set of nonoverlapping intervals.
  - For n intervals, solvable by a greedy algorithm in O(n log n) time.
- Weighted interval scheduling generalizes the problem so the intervals have weights.
  - Pick a set of nonoverlapping intervals with the largest combined weight.
  - No known natural greedy algorithm to solve this.

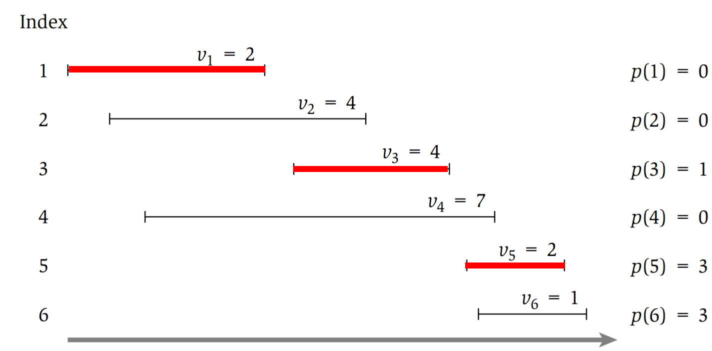


Source: Algorithm Design. Kleinberg, Tardos



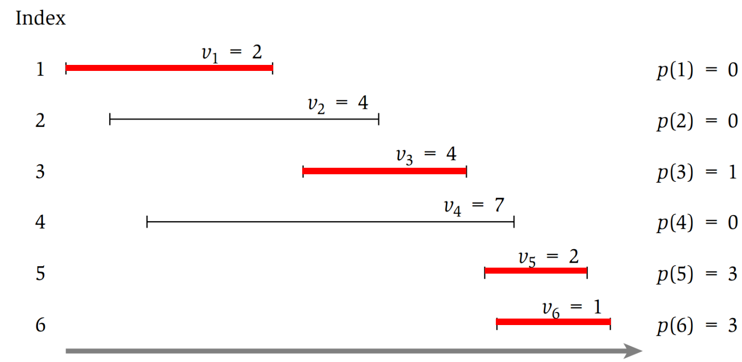
Compatible intervals

- Order the intervals by nondecreasing finishing times, as $I_1, I_2, \ldots, I_n$.
- Given interval I_j, let $p(j)$ be the maximum index k s.t. I_k finishes before I_j starts.
  - If no I_k finishes before I_j starts, let $p(j) = 0$.
- Suppose I_j and I_k are both used in the schedule, and $k < j$, then $k \leq p(j)$.
  - Otherwise I_k overlaps with I_j.



A recursive solution

- Let S^* be an optimal solution. Then either $I_n \in S^*$ or $I_n \notin S^*$.
- If $I_n \in S^*$
  - $I_j \notin S^*$ for all $j \in (p(n), n)$.
  - $S^* - \{I_n\}$ is an optimal solution for the intervals $I_1, I_2, \dots, I_{p(n)}$.
    - I.e. the intervals in S^* besides I_n are a max weight set of non-overlapping intervals from $I_1, I_2, \dots, I_{p(n)}$.
- If $I_n \notin S^*$
  - S^* is an optimal solution for the intervals $I_1, I_2, \dots, I_{n-1}$.



Optimal substructure

- Optimal substructure property.
  - After making a decision, the rest of the solution should be optimal for the rest of the problem.
  - Ex After deciding whether to include I_n in S^*, the remaining solution $S^* - \{I_n\}$ is optimal for the remaining problem (either $I_1, I_2, \ldots, I_{p(n)}$, or $I_1, I_2, \ldots, I_{n-1}$).
- Optimal substructure is the key feature of dynamic programming.
  - Allows combining current partial solution and optimal subproblem solution to form optimal overall solution.
- Not all problems have optimal substructure.
  - For some problems, the current solution can't be combined with an optimal solution to a subproblem.

A recursive solution

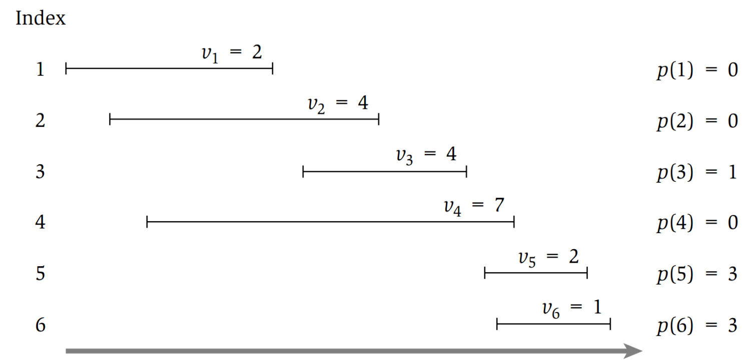
- Let $OPT(j)$ be the weight of a max weight non-overlapping subset of $I_1, \ldots, I_j$.
  - We want to find $OPT(n)$.
- Optimal substructure implies, for any $j \leq n$
  - If $I_j \in S^*$, then $\{I_k \in S^* \mid k < j\}$ is an optimal solution for the intervals $I_1, I_2, \ldots, I_{p(j)}$.
  - If $I_j \notin S^*$, then $\{I_k \in S^* \mid k < j\}$ is an optimal solution for the intervals $I_1, I_2, \ldots, I_{j-1}$.
- Write these as

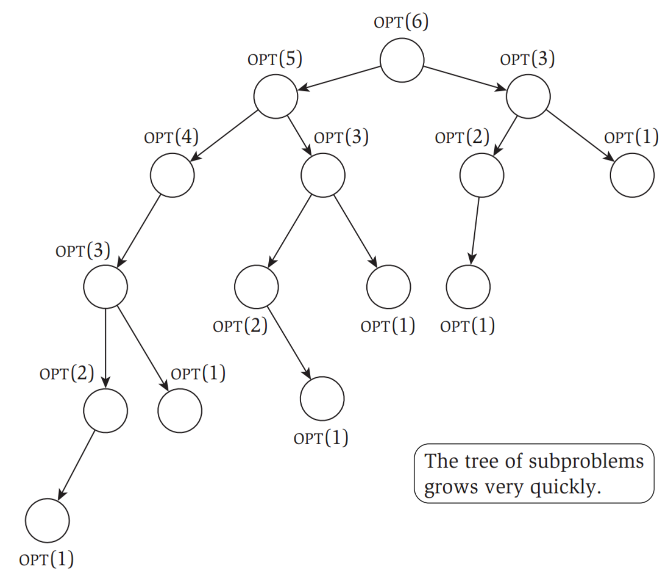
$$OPT(j) = \max\Big(v_j + OPT\big(p(j)\big), OPT(j-1)\Big)$$

  - First part of expression is when $I_j \in S^*$, and second part is when $I_j \notin S^*$.

A recursive solution

- Can use following recursive algorithm.
  - $OPT(j) = \max\left(v_j + OPT\left(p(j)\right), OPT(j-1)\right)$, for $j \geq 1$.
  - $OPT(0) = 0$.
- However, the number of subproblems increases exponentially, so this algorithm takes exponential time.
- But notice many of the calls are the same, e.g. we call $OPT(1), OPT(2), \ldots$ multiple times.
- Instead of computing $OPT(1), OPT(2), \ldots$ multiple times, we can compute them once and store their values.
  - If already computed $OPT(j)$, then when $OPT(k)$ needs to use $OPT(j)$, look up its value instead of running $OPT(j)$.
  - This is called memoization (notice there's no "r"), or the table method.





An iterative solution

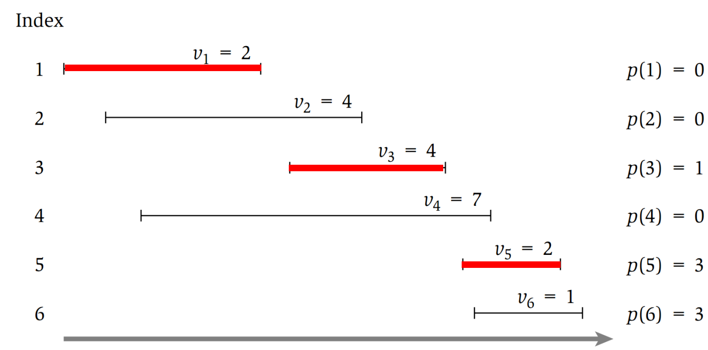
Iterative-Compute-Opt

  $M[0] = 0$

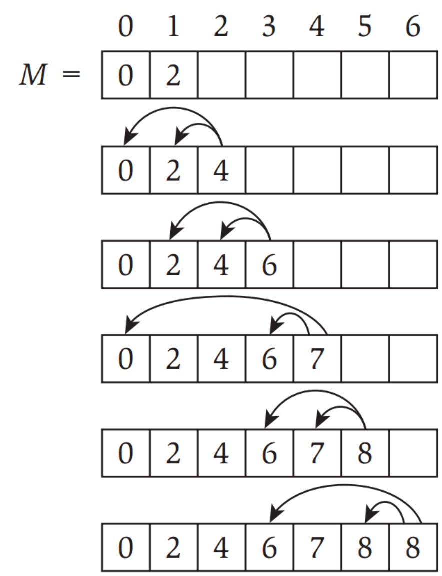
  For $j = 1, 2, \ldots, n$

    $M[j] = \max(v_j + M[p(j)], M[j-1])$

  Endfor



- Use an array M to store the solutions of subproblems we've already solved.
- Solve the subproblems from smallest to largest, i.e. in increasing value of $M[j]$.
- For n intervals, takes $O(n)$ time if we know all the $p(j)$ values.
  - Can compute all the $p(j)$ values by sorting and binary search in $O(n \log n)$ time.

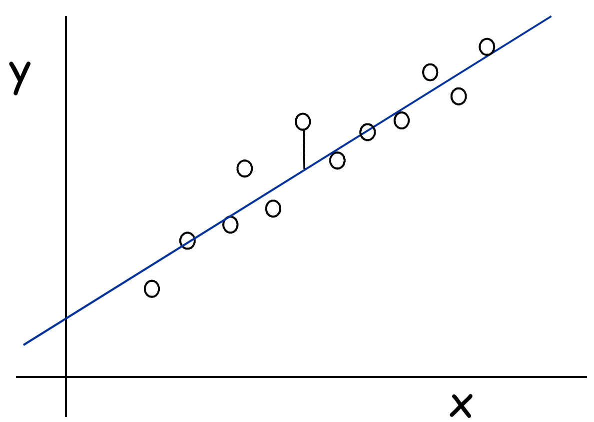


Segmented Least Squares

Least squares.

- Foundational problem in statistic and numerical analysis.
- 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

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

Segmented Least Squares

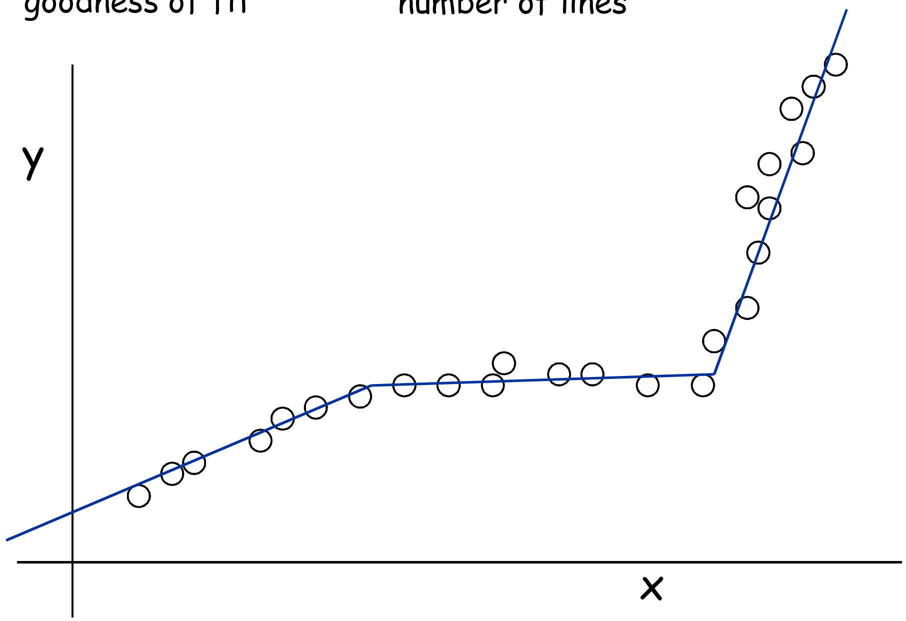
Segmented least squares.

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

Q. What's a reasonable choice for the cost function?

- Shall balance accuracy and parsimony

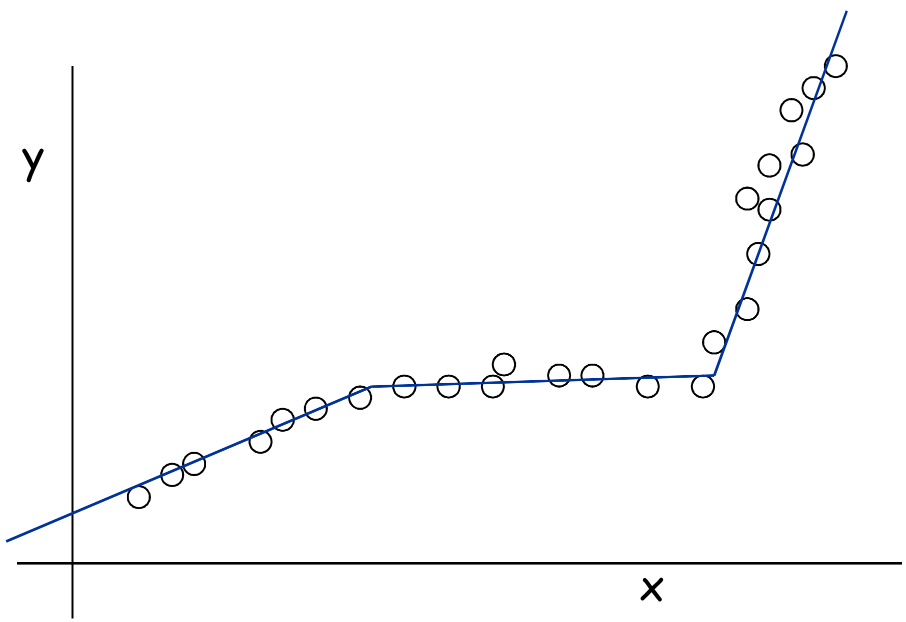
goodness of fit　　　number of lines



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 $E + c\,L$
  - E: the sum of the sums of the squared errors in each segment
  - L: the number of lines
  - c: some positive constant



Dynamic Programming: Multiway Choice

Notation.

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

To compute OPT(j):

- If: last segment uses points p_i, p_{i+1} , . . . , p_j for some i.
- Then: cost = e(i, j) + c + OPT(i-1).

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

Segmented Least Squares: Algorithm

```
INPUT: n, p1,…,pN , c

Segmented-Least-Squares() {
   for j = 1 to n
      for i = 1 to j
         compute the least square error e(i,j) for the
         segment pi,…, pj

   M[0] = 0
   for j = 1 to n
      M[j] = min 1 ≤ i ≤ j (e(i,j) + c + M[i-1])

   return M[n]
}
```

Running time. $O(n^3)$.

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

Subset Sum

- Consider a set of n jobs, where job i takes w_i resource (e.g. time or memory) to process.
- We have W resources total to run the jobs on a computer, and want to use as much resource as possible.
- Find a set of jobs $S \subseteq \{1, \ldots, n\}$ to maximize $\sum_{i \in S} w_i$, subject to $\sum_{i \in S} w_i \leq W$.
- A related problem is Knapsack.
  - Each item also has a value, and want to maximize total value of selected objects subject to weight constraint.
  - Find $S \subseteq \{1, \ldots, n\}$ to maximize $\sum_{i \in S} v_i$, s.t. $\sum_{i \in S} w_i \leq W$.
- Greedy algorithm Sort items from largest to smallest. Insert items sequentially, as many as possible.
  - Always achieves at least 1/2 the max possible sum.
  - Sometimes only achieves $\frac{1}{2} + \epsilon$ fraction of max.
    - Ex $\{\frac{W}{2} + 1, \frac{W}{2}, \frac{W}{2}\}$.

Optimal substructure

- Order the items arbitrarily as $w_1, \ldots, w_n$, and let S be any solution.
- If $w_n \in S$.
  - Sum increases by w_n.
  - The remaining weight is $W - w_n$.
  - $S' = S - \{w_n\}$ should be a max weight subset of $\{w_1, \ldots, w_{n-1}\}$ satisfying $\sum_{w_i \in S'} w_i \leq W - w_n$, by optimal substructure.
- If $w_n \notin S$.
  - Sum doesn't increase.
  - The remaining weight is W.
  - S should be a max weight subset of $\{w_1, \ldots, w_{n-1}\}$ satisfying $\sum_{w_i \in S} w_i \leq W$, by optimal substructure.

Dynamic programming solution

- Let $OPT(i, W')$ be max weight of a subset $S \subseteq \{w_1, \ldots, w_i\}$, subject to $\sum_{w \in S} w \leq W'$. Then

$$\boxed{OPT(i, W') = \max(OPT(i-1, W'), w_i + OPT(i-1, W' - w_i))}$$

- First term in max is where $w_i \notin S$.
  - Then we want a max weight subset of $\{w_1, \ldots, w_{i-1}\}$ with total weight $\leq W'$.
- The second term is where $w_i \in S$.
  - Then our sum increases by w_i, and we want a max weight subset of $\{w_1, \ldots, w_{i-1}\}$ with total weight $\leq W' - w_i$.

Table method for Subset Sum

- Solve dynamic programming equation using an $n \times W$ table M.
  - $M[i, w] = OPT[i, w]$.
  - Base case $M[0, w] = 0$, $\forall w \leq W$.
    - For no items, max sum is 0 regardless of weight limit w.
- $M[i, w]$ depends on $M[i-1, w']$ for some $w' \leq w$.
  - Fill in M in order of increasing i and w.
- The solution to the overall problem is $M[n, W]$.
- Memory complexity is $O(nW)$.
- Time complexity is $O(nW)$.
  - Filling each entry of M requires looking at two other entries in M, which takes O(1) time.

(6.8) *If* $w < w_i$ *then* $\text{OPT}(i, w) = \text{OPT}(i-1, w)$. *Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w - w_i)).$$

```
Subset-Sum(n, W)
  Array M[0 . . . n, 0 . . . W]
  Initialize M[0, w] = 0 for each w = 0, 1, . . . , W
  For i = 1, 2, . . . , n
     For w = 0, . . . , W
        Use the recurrence (6.8) to compute M[i, w]
     Endfor
  Endfor
  Return M[n, W]
```