Lecture 8 - Dynamic Programming I

CMPT 307 D-1

Fall 2024

Instructor: David Mitchell

A Small Example

- The Fibonacci sequence of numbers $F_0, F_1, F_2, \ldots$ is defined by

  $F_0 = 0$; $F_1 = 1$; $F_n = F_{n-1} + F_{n-2}$ // usually

  Eg., 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

- The defining recurrence gives us a (naive) recursive algorithm:

```
fibl(n){
    if n=0 return 0
    if n=1 return 1
    return fibl(n-1) + fibl(n-2)
```

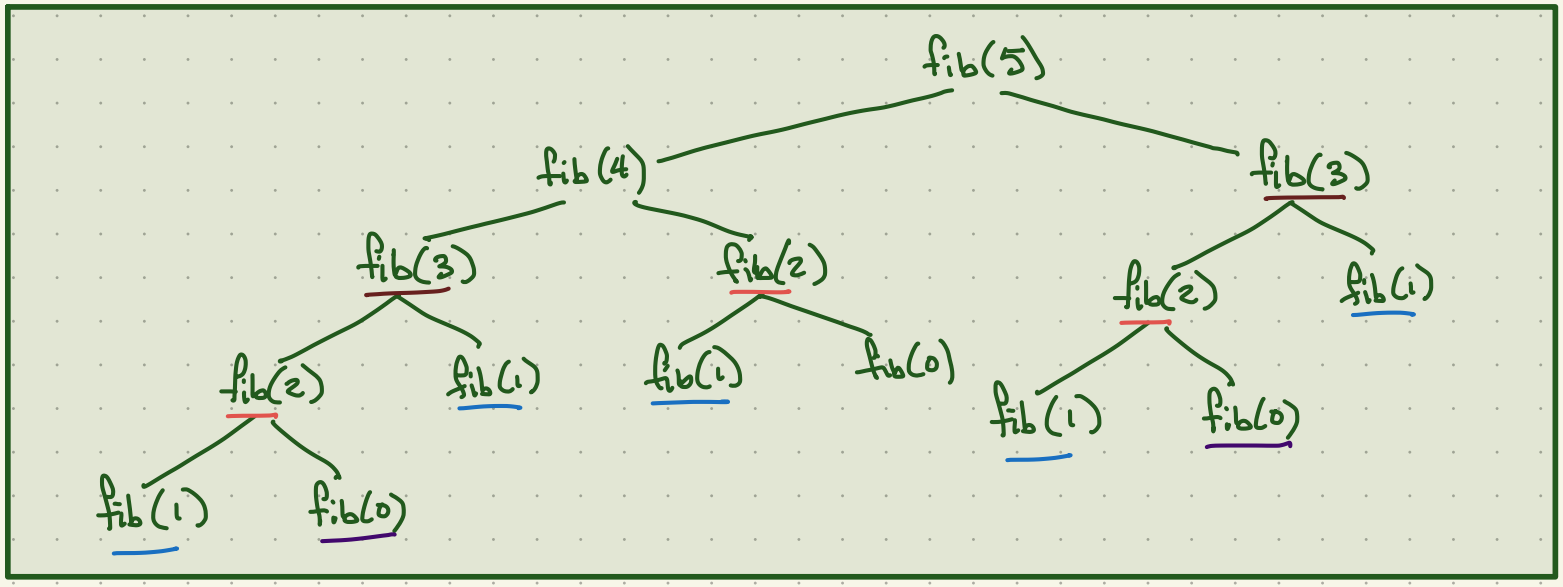
- The running time of fibl() is:

  $T(n) = T(n-1) + T(n-2) + O(1) = \Theta(F_n)$

  $= \Theta(g^n)$, where $g = (1+\sqrt{5})/2 \approx 1.618$

  exponential in n.

- Tree of recursive calls for fib(5):

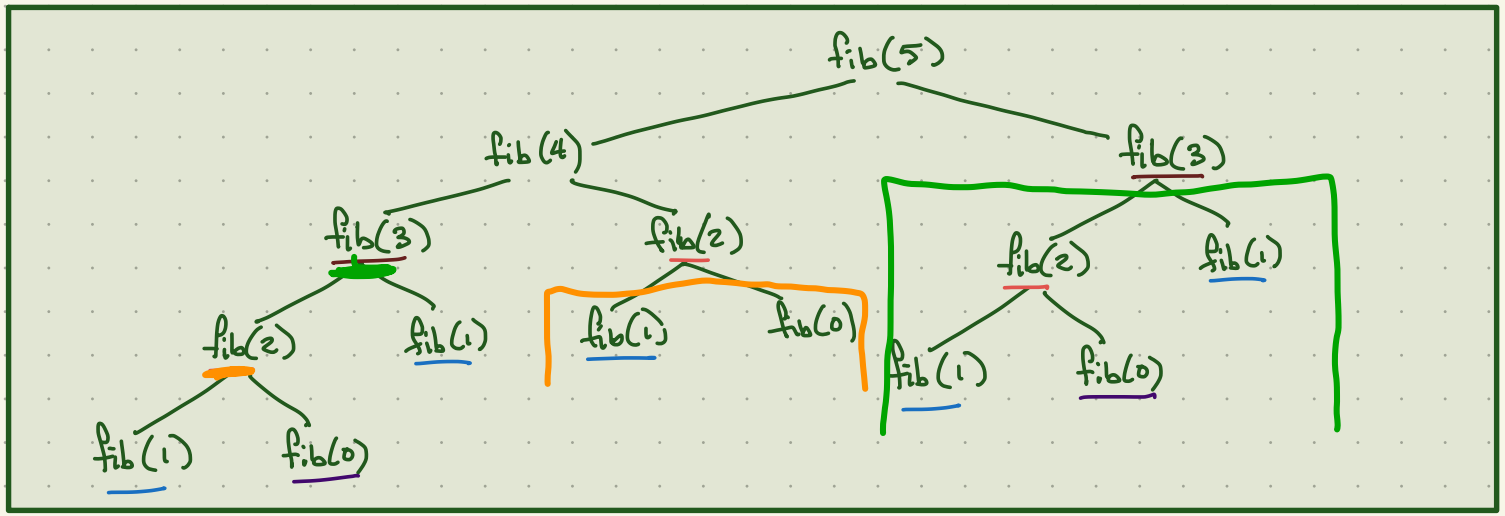


- Suggests a better recursive algorithm using memoization:

```
fib2(n){
   F[0]=0
   F[1]=1
   for i=2..n F[i]=empty
   return fib2r(n)
}
```

```
fib2r(n){
   if F[n]=empty {
      F[n]=fib2r(n-1)+fib2r(n-2)
   }
   return F[n]
}
```

- Tree of recursive calls for fib2r(5)



- The running time of fib2(n) is: $T(n) = \Theta(n)$.
  - A second call to fib2r(i) never generates recursive calls
  - // "recursion is dangerous" is not an appropiate conclusion.

- This suggests a dynamic programming algorithm:

```
fib3(n){
    F[0]=0
    F[1]=1
    for i=2..n{
        F[i]=F[i-1]+F[i-2]
    }
}
```

- So fib1() is exponential time; fib2() + fib3() linear time, right?

Wrong!

> "Algorithm A is Linear time"
> = "The time taken by algorithm A is $O(n)$, where n is the size of the input"

- A call fib3(x) takes time $\Theta(x)$. // x is the input value, not input size.
  The input size is about $\log_2 x$ bits, so for an input size of n, the value of the input is about 2^n
  On an input of size n, fib3() takes time $\Theta(2^n)$ exponential time!

- A call fib1(x) takes time $\Theta(g^x)$.
  On an input of size n, fib1() takes time $\Theta(g^{2^n})$ doubly exponential time!
  ↑ ($g \sim 1.68$)

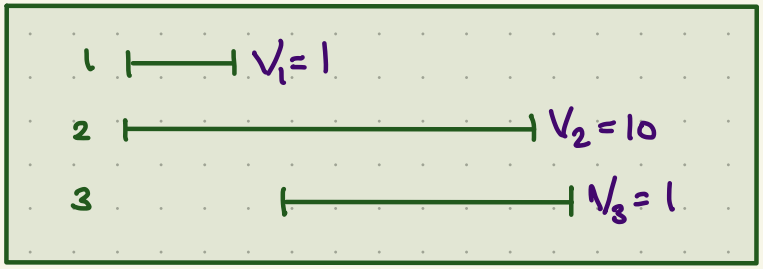
Weighted Interval Scheduling.

Input: A set $I = \{[s_1, f_1], [s_2, f_2], \ldots [s_n, f_n]\}$ of intervals with "weight" values $v_1, v_2, \ldots, v_n$.

- Value of set S of intervals is $\sum_{i \in S} v_i$

Problem: Find a compatible subset of I with maximum value.

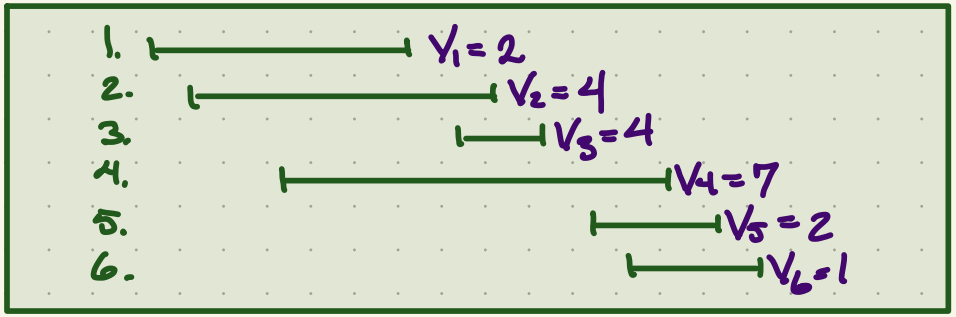
- All weights the same $\Rightarrow$ Interval Scheduling
  - The Earliest Finish Time First greedy alg. is optimal.
- Earliest Finish Time fails if weights are not all the same:



- No greedy alg. known!

Weighted Interval Scheduling – Naive Recursive Algorithm.

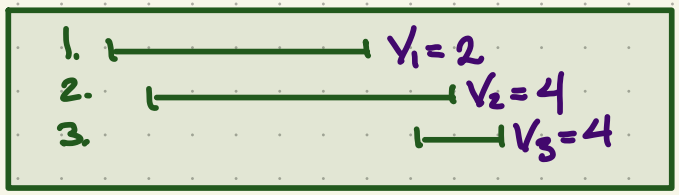
- Suppose we order intervals by non-decreasing finish time:



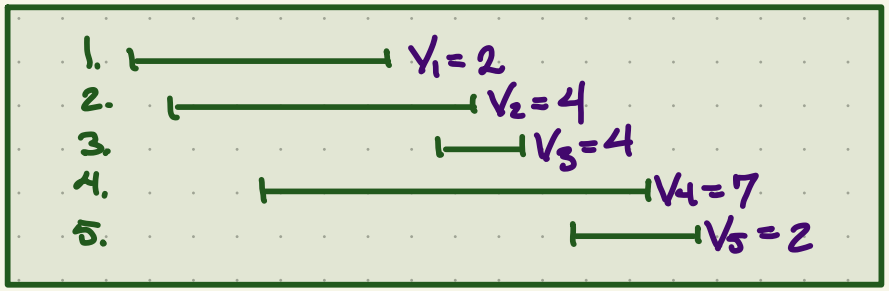
- Let O be an optimal solution, and consider 2 cases – v_6 in O or not:

1) $v_6 \in O \Rightarrow v_5, v_4 \notin O$

$\Rightarrow O$ consists of v_6 plus an optimal solution for:



2) $v_6 \notin O \Rightarrow O$ consists of an optimal solution for:



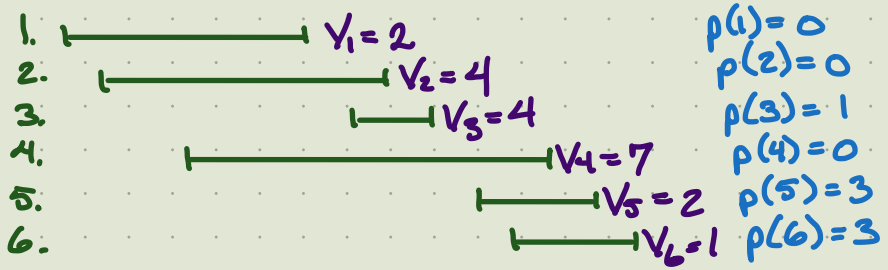
Weighted Interval Scheduling – Naive Recursive Algorithm.

- Order jobs s.t. $i < j \Rightarrow f_i \leq f_j$. // non-decreasing f_i

- Define $p(i) = \begin{cases} \max\{j : j < i \text{ and } f_j < s_i\} & \text{if such } j \text{ exists} \\ 0 & \text{o.w.} \end{cases}$

  ↖ the last job that ends before i starts.

Eg.



- Define: OPT = value of some optimal solution O.
  O_j = an optimal solution for intervals $1..j$ (If O is unique, $O_n = O$.)
  $OPT(j)$ = value of O_j // $OPT(n) = OPT$

- Now: 1) $OPT(j) = \max\{v_j + OPT(p(j)),\ OPT(j-1)\}$ (*)

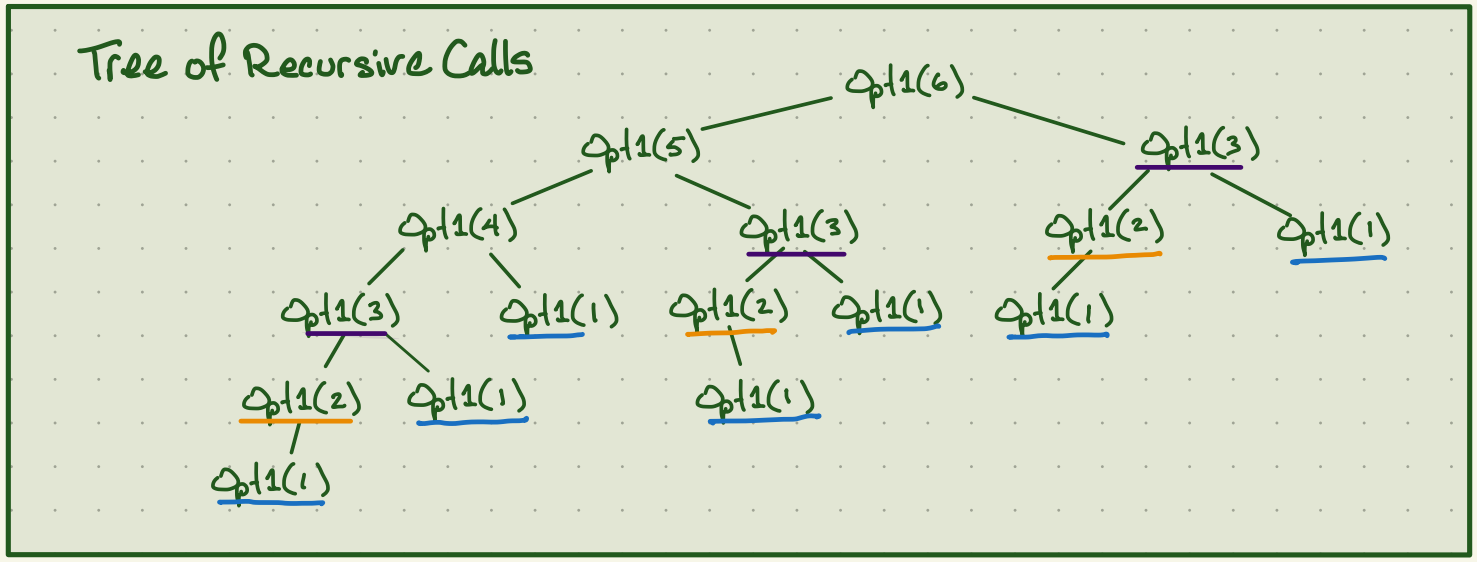
  2) $j \in O_j \Leftrightarrow v_j + OPT(p(j)) \geq OPT(j-1)$

Weighted Interval Scheduling – Naive Recursive Algorithm.

Recurrance (*) gives us:

```
// Assumes jobs ordered by finish time & p(j) computed.
Opt1(j){
    if j=0
        return 0
    else
        return max(vj + Opt1(p(j)), Opt1(j-1))
}
```

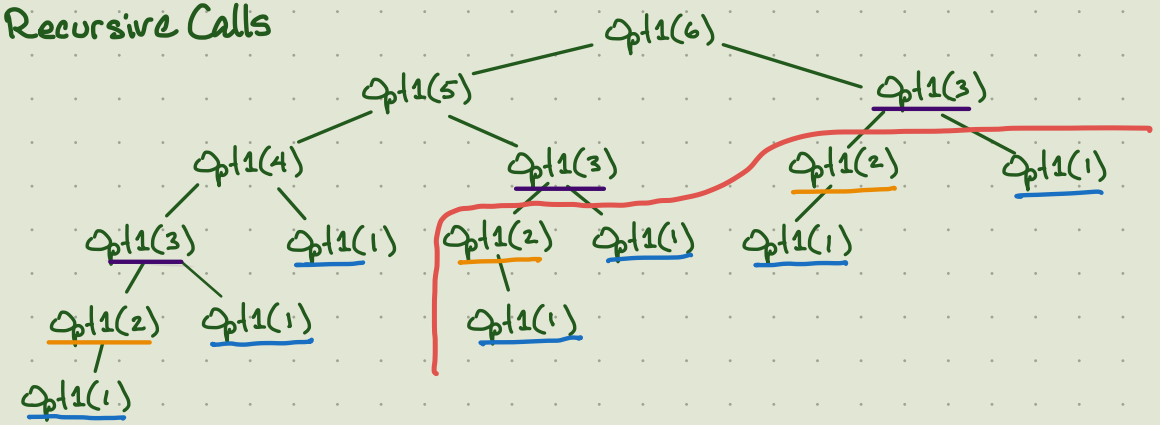
Claim: 1) Opt1(j) returns OPT(j). Pf: See p. 255.
2) Opt1(n) takes exponential time. // p. 255, 256



Weighted Interval Scheduling – Recursion with Memoization

```
// Assume jobs ordered by finish time & p(j) computed.
//
// Uses memoization array M[j] = { OPT(j) if already computed
//                                { empty o.w.
//
Opt2(j){
    if j = 0
        return 0
    else if M[j] not empty
        return M[j]
    else
        M[j] = max(v_j + Opt2(p(j)), Opt2(j-1))
        return M[j]
}
```

Tree of Recursive Calls



Claim:

1) Opt2(j) computes OPT(j)

2) Opt2(n) takes time O(n)

// see p. 257

Each Opt2() call does 1 of: a) a lookup; b) compute a new value for M.

Weighted Interval Scheduling – D.P. Solution

```
Opt(I){ // DP For Weighted Interval Scheduling.
    order jobs by non-decreasing finish time
    compute values p(j)
    M[0] = 0
    for j = 1..n {
        M[j] = max( v_j + M[p(j)], M[j-1] )
    }
```

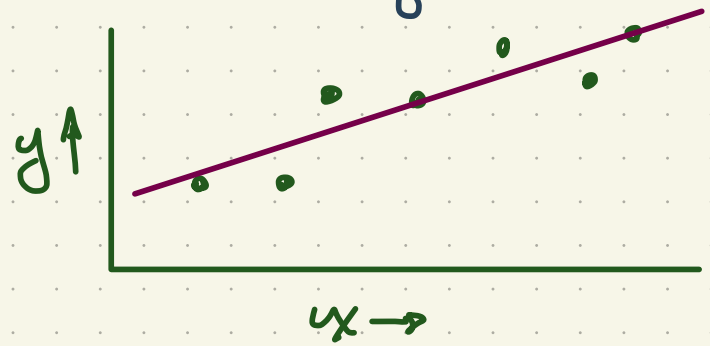
To construct an optimal solution from M:

```
FindSol(j){
    if j=0
        output nothing
    else if v_j + M[p(j)] ≥ M[j-1]
        output j; FindSol(p(j))
    else
        output FindSol(j-1)
}
```

Claim: Computing OPT(n) and Opt(n) by D.P. takes time $O(n \log n)$.

Segmented Least Squares

- Consider data points $(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$, $x_1 < x_2 < \dots < x_n$
- Common way to understand a trend in y with increasing x:
  Line with least squared error:



Line L defined by $y = ax + b$: $\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$

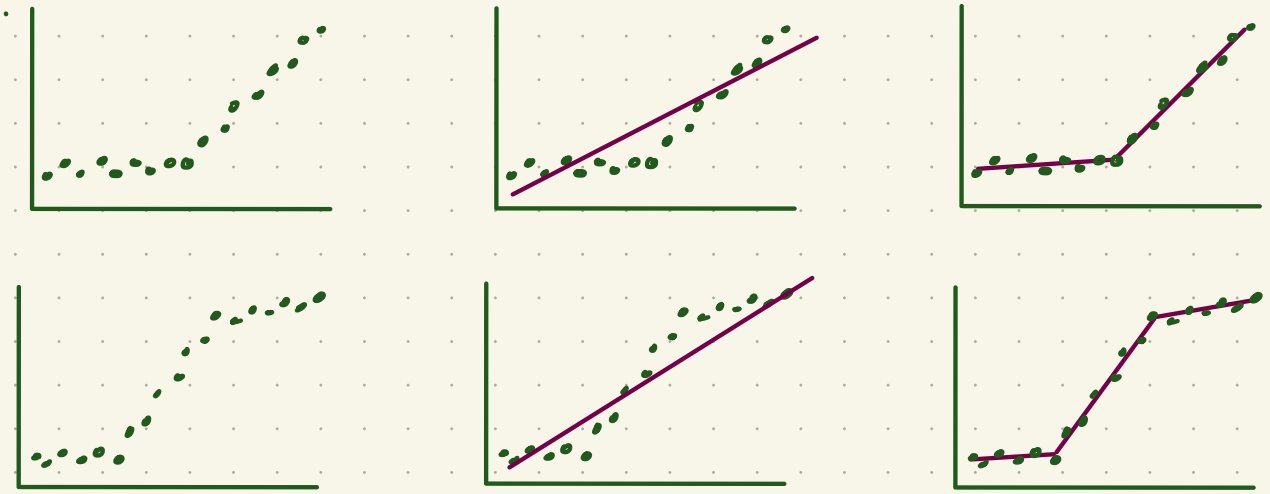
The min. error line has:

$$a = \frac{n\Sigma_i x_i y_i - (\Sigma_i x_i)(\Sigma_i y_i)}{n\Sigma_i x_i^2 - (\Sigma_i x_i)^2}$$

$$b = \frac{\Sigma_i y_i - a\Sigma_i x_i}{n}$$

Segmented Least Squares

What about changes in trends:



Given: $P = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with $x_1 < x_2 < \cdots < x_n$; Penalty C.

Find: A partition S of P into k contiguous segments that minimizes:

$$C \cdot k + \sum_{s \in S} (\text{least squares error for } s)$$

Penalty for fit of line segments

Penalty for # of line segments // C is given; // k is not.

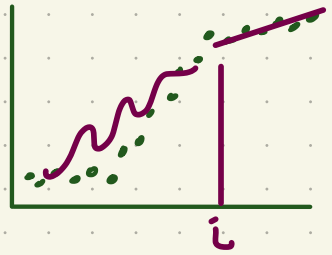
- There are exponentially many possible partitions.

Segmented Least Squares

- Let: $OPT(i)$ = value of optimum solution for $p_1 \dots p_i$ // $OPT(0) = 0$
  $e(i,j)$ = min. error for any line fitting $p_i, \dots p_j$
  O = an optimal partion.

- If the last segment of O is $p_i \dots p_n$, then the value of O is

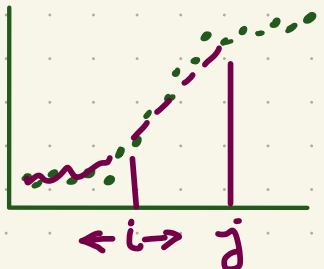
$$OPT(n) = e(i,n) + C + OPT(i-1)$$



- This gives us the recurrence:

$$(*) \quad OPT(j) = \min_{1 \le i \le j} \left(e(i,j) + C + OPT(i-1) \right)$$

$$OPT(0) = 0$$

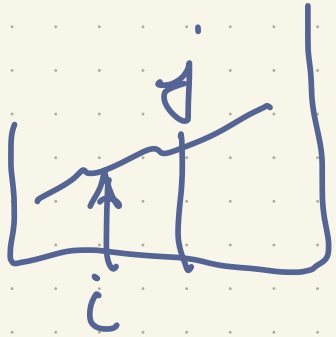


↑ consider all values for i.

$$OPT(1) = e_{1,1} + C + OPT(0) = C$$

Segmented Least Squares - D.P. Solution

```
SegmentedLeastSquares({p_1, ..., p_n}, C){ (*)
    M[0] = 0 // M is an array of size n+1
    for each pair 1 ≤ i ≤ j ≤ n {
        compute error e(i,j) for segment p_i .. p_j
    }
    for j = 1,..,n {
        M[j] = min_{1≤i≤j} ( e(i,j) + C + M[i-1] )
    }
    Return M(n)
}
```



Claim: 1) Alg. (*) returns OPT. // Proof by induction using reccurence (*).

2) Alg (*) runs in polynomial time.

- n^2 $e(i,j)$; time $O(n)$ each // $O(n^3)$; can be done in $O(n^2)$
- n calculations of $M[j]$; time $O(n)$ each // $O(n^2)$

Segmented Least Squares

Constructing the segments from M:

```
Find-Segments(j){
    if j=0 then{
        output nothing
    else
        find i to minimize e(i,j)+C+M[i-1]
        output: (Pi ··· Pj); Find-Segments(i-1)
    }
}
```

Ex: Find-Segments takes time $\Theta(n^2)$.
Modify the D.P. algorithm so that it has a $O(n)$ algorithm to find the segments.

Subset Sum

Instance: · n items with non-negative weights $\{w_1, w_2, \ldots, w_n\}$.
· bound $W > 0$.

Problem: · Find a subset $S \subseteq \{1, \ldots, n\}$ that maximizes $\sum_{i \in S} w_i$
subject to $\sum_{i \in S} w_i \leq W$.

- Can be viewed as a packing problem: load as much as possible up to the max.

or as a scheduling problem: complete as much work as possible before the deadline.

No efficient optimal greedy algorithm known.

Eg: · "order by decreasing weight" fails: $\{1 + w/2, w/2, w/2\}$

· "order by increasing weight" fails: $\{1, w/2, w/2\}$

Subset Sum

- Recall Weighted Interval Scheduling:

$OPT(j)$ = max. value using intervals $1 \ldots j$

$$OPT(j) = \begin{cases} OPT(j-1) & \text{if } j \notin O \\ OPT(p(j)) + w_j & \text{if } j \in O \end{cases}$$

↑ excludes jobs that conflict with j.

- For Subset Sum:
  - no jobs conflict
  - selecting job i reduces capacity by w_i.

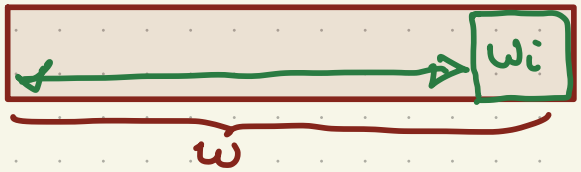
- Let $OPT(i, w)$ = max. possible weight with $w_1 \ldots w_i$ and capacity w.

Then:

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

$$OPT = OPT(n, W)$$

D.P. Algorithm for Subset Sum



Define: $OPT(i,w)$ = max. possible weight with $w_1 \dots w_i$ and capacity w.

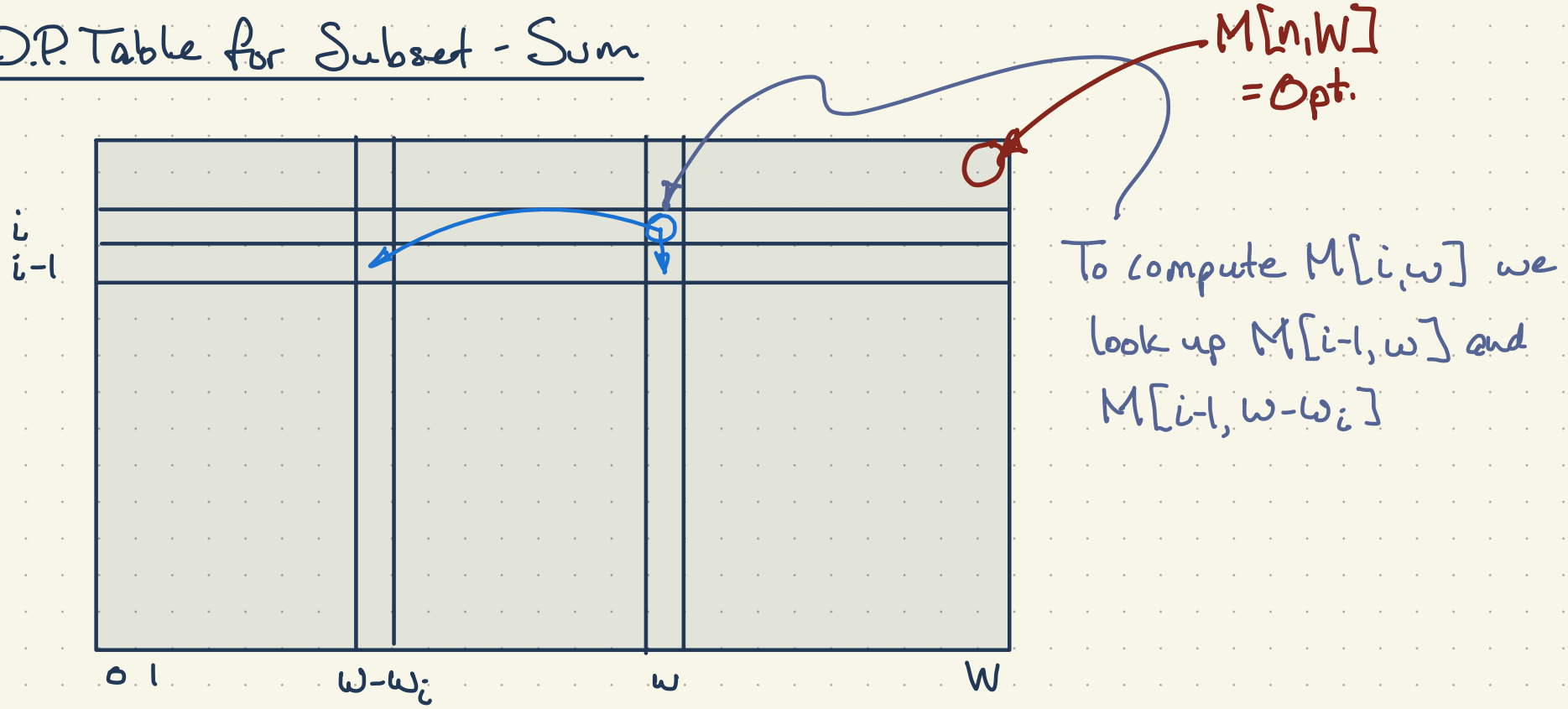
Then: $$OPT(i,w) = \begin{cases} OPT(i-1,w) & \text{if } w_i > w \\ \max(OPT(i-1,w), w_i + OPT(i-1, w-w_i)) & \text{o.w.} \end{cases}$$

$$OPT = OPT(n, W)$$

Build table of all values $OPT(i,w)$ for $i \in \{0 \dots n\}$, $w \in \{0..W\}$.

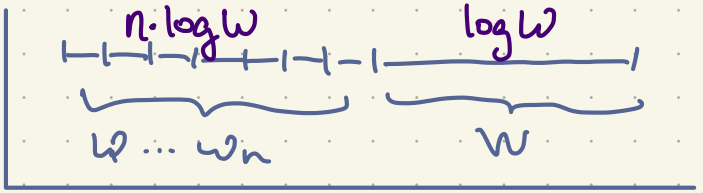
```
Subset-Sum(n,W){
    Array M[0..n, 0..W]
    M[0,w] = 0 for each w ∈ 0..W
    for i = 1..n {
        for w = 0..W{
            M[i,w] = { M[i-1,w]                    if w < w_i
                     { max(M[i-1,w], w_i + M[i-1, w-w_i])  o.w.
        }
    }
    return M[n,W]
```

D.P. Table for Subset-Sum



M[i,w] = max weight possible using boxes 1..i if capacity is w.

Subset-Sum DP - Complexity



- Time: $\Theta(n \cdot W)$

  The table is n by $W+1$, and each table entry needs time $O(1)$.

- This is **not** polytime, because W may be large.

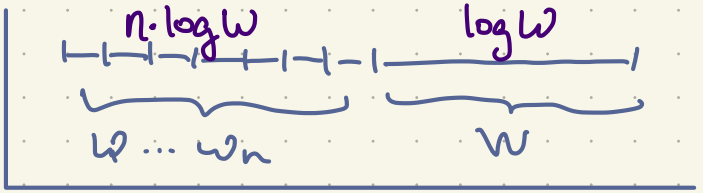
  Suppose: · each of the n weights is $\leq W$,
  · $W = 2^n$

  Then: The size of the input is $N \leq (n+1) \cdot \log_2 W$
  $\leq (n+1)n$
  $\leq 2n^2$

  Table size $= n \cdot W = n \cdot 2^n$, which is not polynomial in N:

$$\frac{\#\text{steps}}{\text{input size}} = \frac{n \cdot 2^n}{2n^2} = \frac{2^n}{2n} = \frac{2^{n-1}}{n}$$

Subset-Sum DP - Complexity



- Time: $O(n \cdot W)$

  The table is n by $W+1$, and each table entry needs time $O(1)$.

- It is "pseudo-polynomial time":

  Polynomial in the size + maximum value in input.

- Consider only inputs where $W \leq n^k$, for some $k \in \mathbb{N}$.

  Then $O(n \cdot W) = O(n \cdot n^k) = O(n^{k+1})$.

  $\Rightarrow$ It is polynomial time if the numbers are small so that the the time is dominated by the number of weights in the input.

- fib_3 is <u>not</u> pseudopolynomial time, because the input is just the number.

The Knapsack Problem

Instance: · n items with weights w_i and values v_i
· Capacity W

Problem: Find subset $S \subseteq \{1, \ldots, n\}$
that maximizes $\sum_{i \in S} v_i$

subject to $\sum_{i \in S} w_i \leq W$

Define: $OPT(i, w)$ = max. value attainable using $w_1 \ldots w_i$ with capacity w.

Then:

$$OPT(i, w) = \begin{cases} OPT(i-1, w) & \text{if } w < w_i \\ \max(OPT(i-1, w), v_i + OPT(i-1, w-w_i)) & \text{o.w.} \end{cases}$$

$$OPT = OPT(n, W)$$

Knapsack DP: $O(n \cdot W)$ time.

Those who cannot remember the past are condemned to repeat it.*

- Dynamic Programming

(Actually: George Santayana, 1905)

Dynamic programming history

Bellman. Pioneered the systematic study of dynamic programming in 1950s.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense had pathological fear of mathematical research.
- Bellman sought a "dynamic" adjective to avoid conflict.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Dynamic programming applications

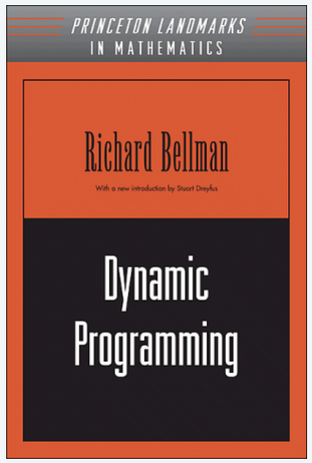
Application areas.

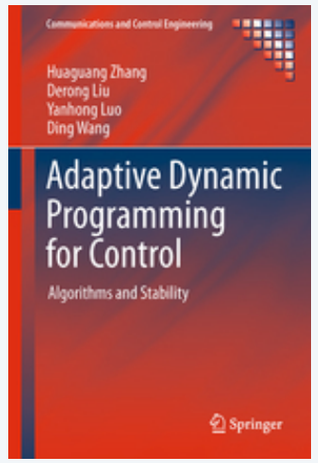
- Computer science: AI, compilers, systems, graphics, theory, ...
- Operations research.
- Information theory.
- Control theory.
- Bioinformatics.

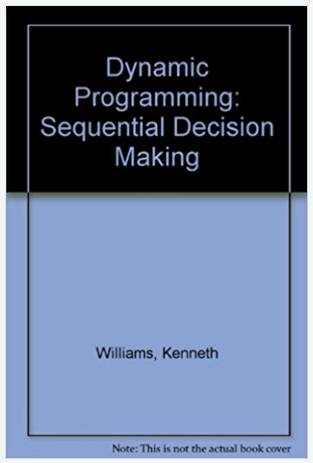
Some famous dynamic programming algorithms.

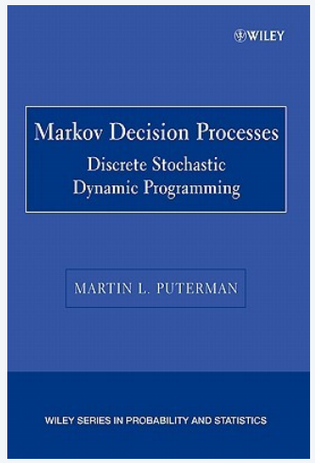
- Avidan–Shamir for seam carving.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Bellman–Ford–Moore for shortest path.
- Knuth–Plass for word wrapping text in TEX.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Needleman–Wunsch/Smith–Waterman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

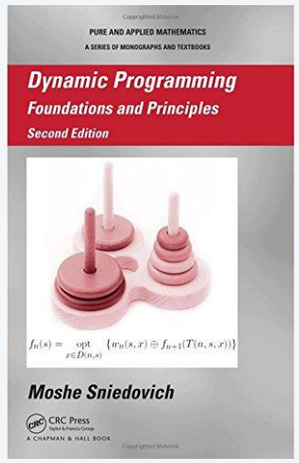
Dynamic programming books

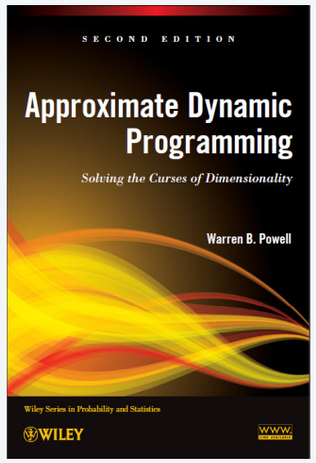


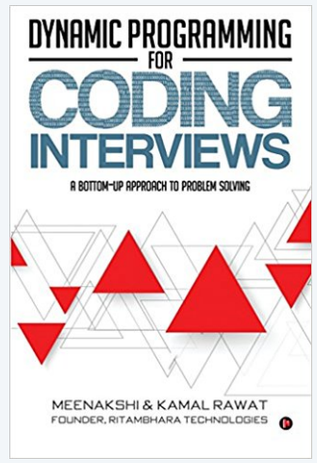


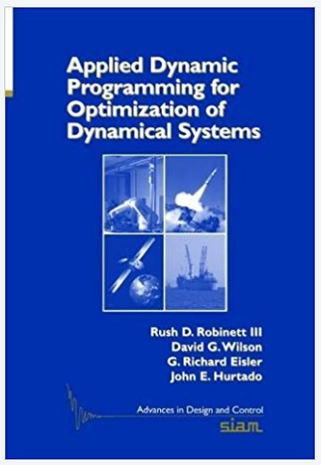


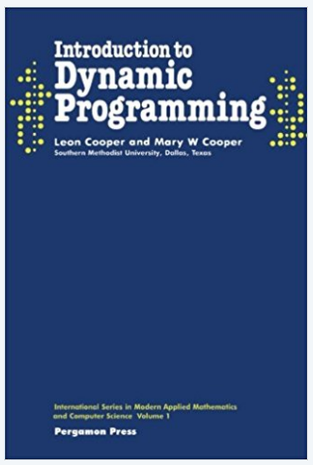


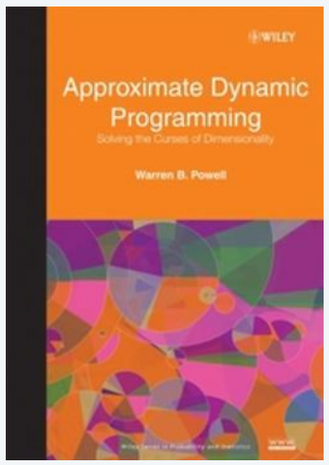


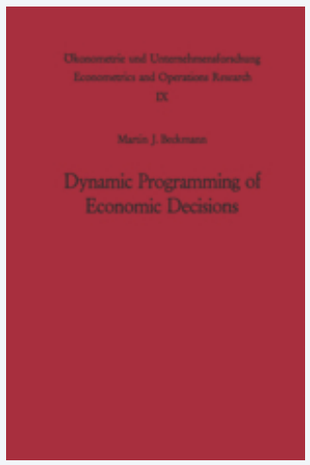


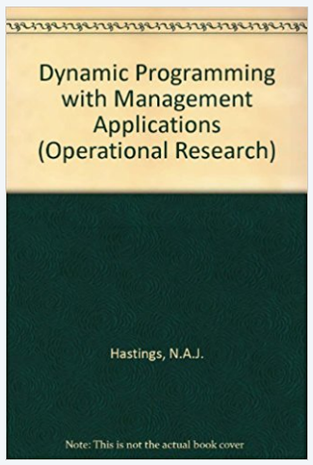


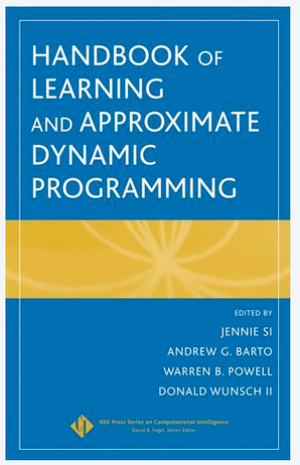


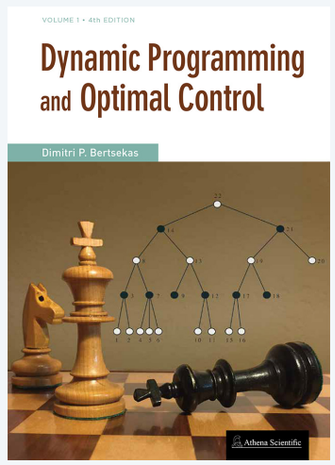


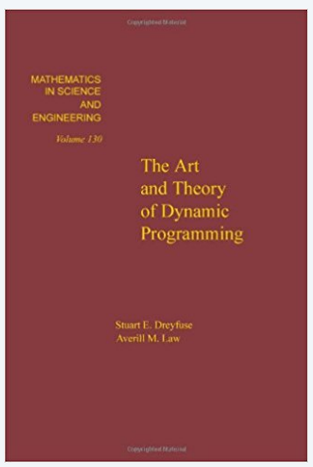


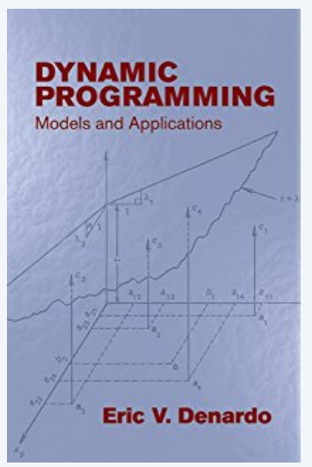


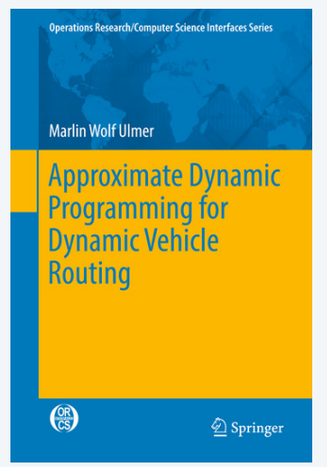


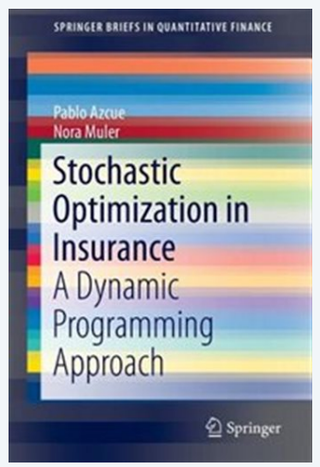


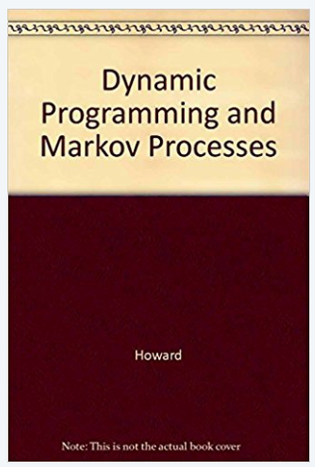


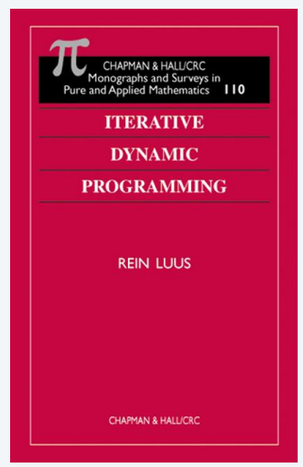


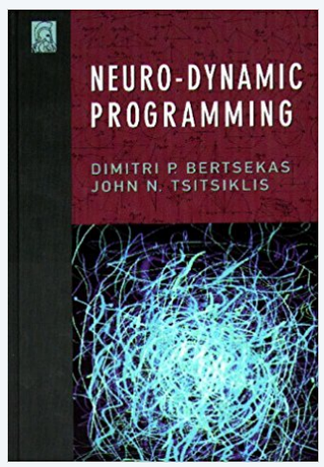










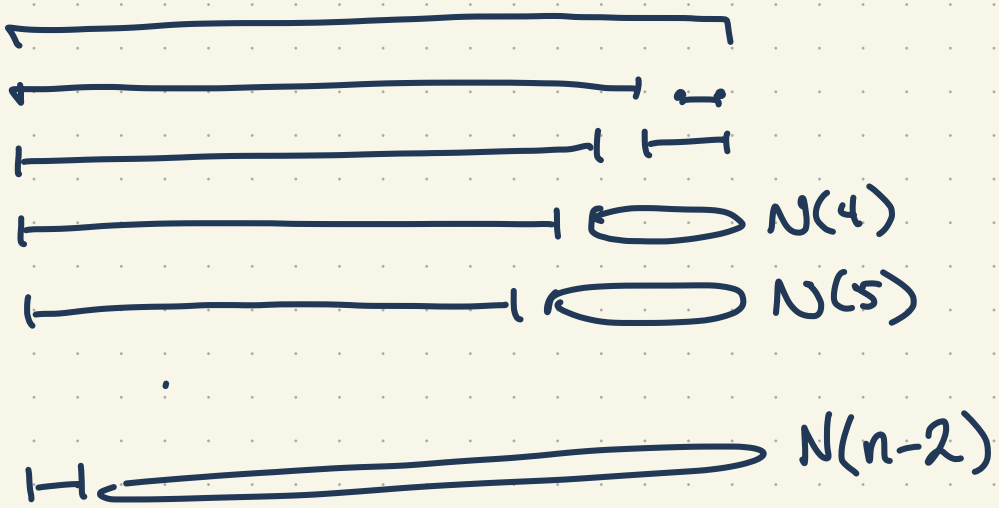


End

$N(1) = 1$

$N(2) = 1$

$N(3) = 1$



$$N(n) = N(n-2) + N(n-3) + \cdots + N(5) + N(4) + 3$$