Lecture 9

Tuesday, June 6, 2017 10:37 PM

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, 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.

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

Dynamic programming history

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

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

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

Areas.

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

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- De Boor for evaluating spline curves.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.
- ...

4

Example 0: Fibonacci numbers

$F_0 = 1$
$F_1 = 1$
$F_2 = 2$
$F_3 = 3$

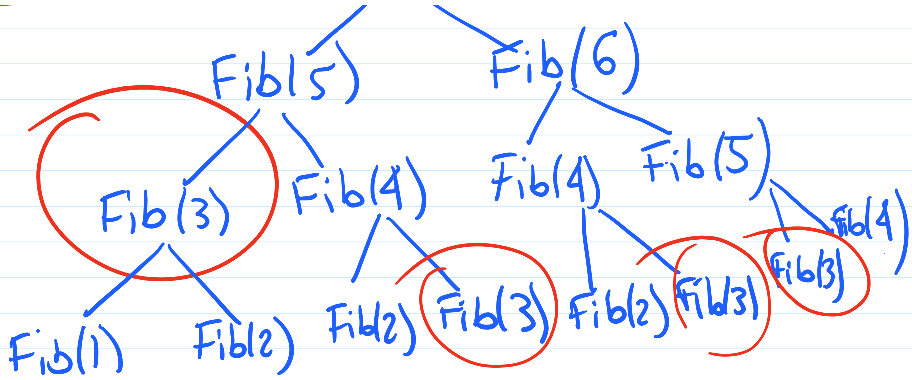
$F_i = F_{i-2} + F_{i-1}$ $(i \geq 2)$

Given n, compute F_n.

Algo 1: Fib(n)

```
if n=0 OR n=1
then return 1
else return Fib(n-1)+Fib(n-2)
endif
```

Example run: Fib(7)



Problem: Many Fib(·) values are recomputed !! $\Rightarrow$ slow runtime

Exponential

Solution: Remember & don't recompute ! Memoization.

algo 2: Fib(n)

make array F[0..n]

F[0] = 1

F[1] = 1

for i = 2 to n

  F[i] = F[i-2] + F[i-1]

endfor

return F[n]

Run time: $O(n)$

Fib(5)

F[2] = 2

$F[0] = 1$

$F[1] = 1$

$F[2] = 2$

$F[3] = 3$

$F[4] = 5$

$F[5] = 8$

Example 1: Wall Climbing

C(i,j) = cost of cell (i,j)

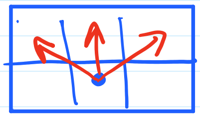
$A(i,j) = C(i,j) + \min\{\cdot, \cdot, \cdot\}$

12

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

m, n

can move
- Directly above, or
- Above Left, or
- Above Right.



greedy: 13

DP: 12

Goal: Get from the first (bottom) row to the last (top) row via a cheapest path.

Cost of a path = sum of costs of the cells on the path.

Greedy algorithm?

Dynamic Programming algorithm

Template:

(1) Describe an array of values (numbers) to compute. Each array entry corresponds to a sub-problem of the original problem.

(2) Give a recurrence to compute the values in the array: a "big" problem can be solved using the solutions to some "small" sub-problems.

(3) Give a program to compute the array values: a "bottom-up" algorithm.

(4) Using the array values, compute an optimal solution to the original problem.

1. $A(i,j)$ = cost of the cheapest path from bottom row to cell (i,j)

$1 \le i \le m$

$1 \le j \le n$

Best cost to get to top row: $\min\{A(m,1), \dots, A(m,n)\}$

2. Recurrence

$$(*)\ A(i,j) = c(i,j) + \min\{A(i-1,j-1),\ A(i-1,j),\ A(i-1,j+1)\}$$

$$(*)\ A(1,j) = c(1,j),\ \forall j$$

3. Use $(*)$ to fill in the array

from bottom row up.

Time: $O(n \cdot m)$

(const time per array element)

4. Recover an actual cheapest path (by "tracing" back through our array $A()$).

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

$C(i,j)$

13	19	16	12	15
11	11	13	7	8
7	9	7	10	5
3	2	5	4	8

$$A(i,j) = \begin{cases} C(i,j) & \text{if } i=1 \\ C(i,j) + \min\{A(i-1,j-1), A(i-1,j), A(i-1,j+1)\} \end{cases}$$

$\min\{3+5, 2+5\}$

algo PrintOpt (i, j) % print optimal path
% ending at cell (i,j)

print (i,j)

if $i=1$ then return

else find $k \in \{j-1, j, j+1\}$ such that

$A[i-1, k] = \min\{A[i-1,j-1], A[i-1,j], A[i-1,j+1]\}$

PrintOpt $(i-1, k)$

endAlgo

The main call is: $\text{PrintOpt}(m, j)$

where j is such that $A[m, j] = \min_{1 \le k \le n} \{A[m, k]\}$

The runtime of $\text{PrintOpt}(m, j)$: $O(m)$.

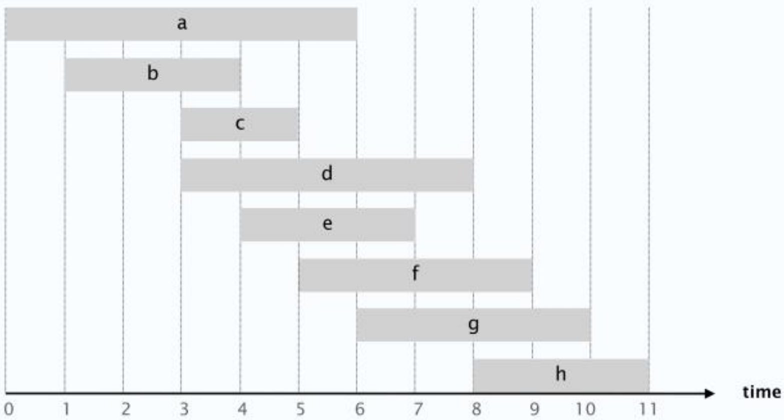
The overall time of the DP algo to find a cheapest path on the wall: $O(m \cdot n) + O(m)$

$\le O(m \cdot n)$

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.



6

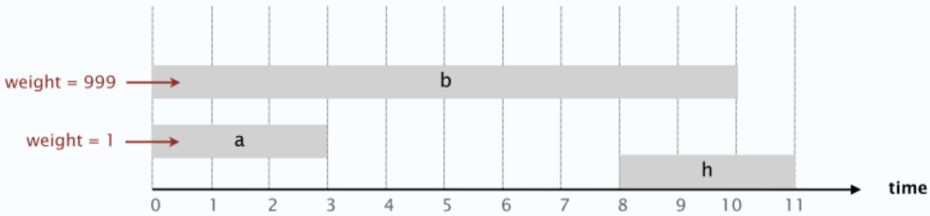
Earliest-finish-time first algorithm

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.



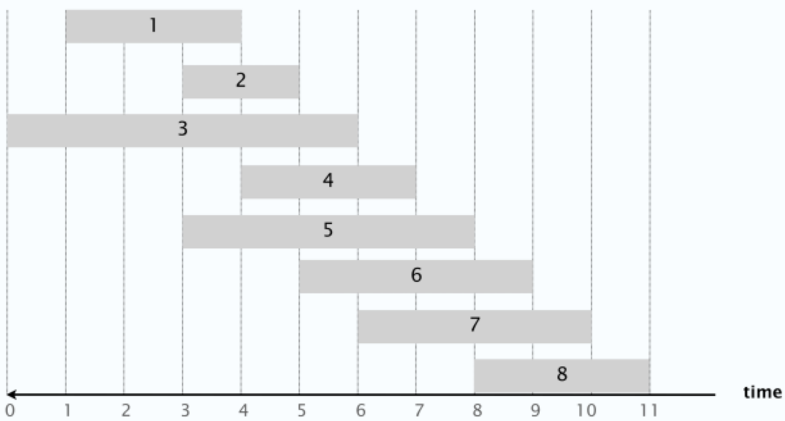
7

Weighted interval scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

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

Ex. $p(8) = 5, p(7) = 3, p(2) = 0.$



8

DP Algorithm

(1) Array: Define M[j] = the value of an optimal solution for the subset of jobs 1,..., j

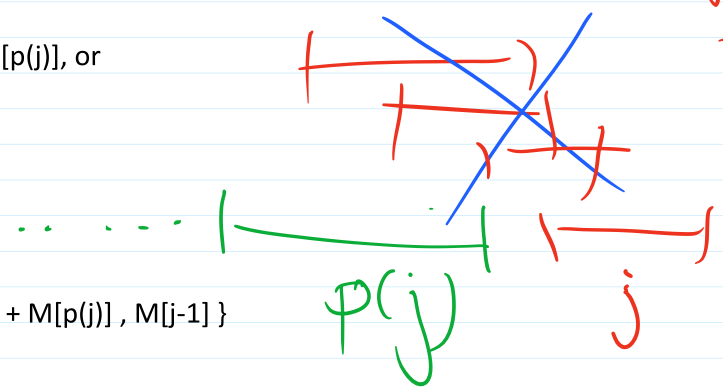
M[n]
||
value
we

(2) Recurrence: Two possibilities: (a) either job j is part of an optimal solution,
or (b) job j is not.

Hence, either M[j] = v[j] + M[p(j)], or

or (b) job j is not.

value we want



Hence, either M[j] = v[j] + M[p(j)], or
M[j] = M[j-1].

So, the recurrence is

M[0]=0

M[j] = max{ v[j] + M[p(j)] , M[j-1] }

(3) Algorithm to fill in the array:

```
M[ 0 ] = 0
for j=1 to n
    M[ j ] = max { v[j] + M[ p(j) ],  M[j-1] }
end for
```

(4) Recover an actual optimal schedule from M[]:

Weighted interval scheduling: finding a solution

Q. DP algorithm computes optimal value. How to find solution itself?
A. Make a second pass.

```
Find-Solution(j)
if j = 0
   return ∅.
else if (v[j] + M[p[j]] > M[j-1])
   return { j } ∪ Find-Solution(p[j]).
else
   return Find-Solution(j-1).
```

M[j] > M[j-1]

$\underline{\text{Find\_Soln}(n)}$
main call

$T(n) \le T(n-1) + O(1)$
$O(n)$

Analysis. # of recursive calls ≤ n ⇒ $O(n)$.

14

Runtime: $O(n.\log n)$.

- Preprocessing[sorting & computing p(j)'s]: time $O(n \log n)$

• Fill in M[ ]: time O(n)
• Tracing out the opt. solution: time O(n)

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.

24

Dynamic programming: false start

OPT(n) ✓
OPT(1) easy

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

$$OPT(i) = \begin{cases} OPT(i-1) & \text{if } i \notin \text{Opt sol'n} \\ \boxed{v_i} + OPT(i-1, W - w_i) \end{cases}$$

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.

Opt(i, w)

Conclusion. Need more subproblems!

25

Dynamic programming: adding a new variable

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

$OPT(i, w)$

$= OPT(n, W)$

Case 1. OPT does not select item i.

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

$OPT(i-1, w)$

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.

$OPT(i-1, w-w_i) + v_i$

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

$w_i \le W$

$i \notin OPT$ $\qquad$ $i \in OPT$

Knapsack problem: bottom-up

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

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

rows columns

0 1 2 3 . . . .

FOR $i = 1$ TO n
 FOR $w = 0$ 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 $M[n, W]$.

W

W integers

0 1 . . . W

$i=n$

$i=2$

$i=1$

$i=0$ 0 0 0 . . . 0

$M[1, w] = \begin{cases} 0 & w_1 > w \\ v_1 & w_1 \le w \end{cases}$

$n \times W$

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
$i=0$	{ }	0	0	0	0	0	0	0	0	0	0	0	0
$i=1$	{ 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
$i=5$	{ 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.

28

algo PrintOpt(i, w)

% find $S \subseteq \{1, 2, \ldots, i\}$ of max value

% s.t. the weight of $S \leq w$

% The main call will be: PrintOpt(n, W)

if $i = 0$ **then** return $\emptyset$

else if $M[i, w] = M[i-1, w]$

then return PrintOpt$(i-1, w)$

else return $\{i\} \cup$ PrintOpt$(i-1, w - w_i)$

endif

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:

$w = 0\ 1\ 2 \ldots W$

$i=0$

$i=1$

$\vdots$

$i=n$

$n \times W$

$O($

- 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 $M[i, w] > M[i-1, w]$. ▪

$\}\ O(n)$

$n,\ \overset{v_1, \dots, v_n}{w_1, \dots, w_n},\ W$

Input size $n \cdot (\log \max v_i) + \log(\max w_i)) + \log W$

Remarks.

- Not polynomial in input size! ← "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]

usefull: if $\boxed{W \le n^{10}}$

29

n items: $\underbrace{v_1}_{\log_2 v_1}, \underbrace{w_1}, \dots, v_n, w_n, \underbrace{W}_{\log_2 W \text{ bits}}$

$\boxed{\text{Input size}} \ge \log W$

$\le \sum_{i=1}^{n} (\log_2 v_i + \log_2 w_i) + \log_2 W$

Actual Runtime: $\underbrace{O(n \cdot \underline{W})}_{O(n^3)}$

$W \le n^2$

$\#\text{digits of } A = O(\log A)$

$$\begin{array}{r} 1000 = A \\ +\ 3045 = B \\ \hline 4045 = C \end{array}$$

Time: $O(\log A)$

$\underbrace{|||||||||\ \dots\ ||||}_{1000 \text{ times}}$

3048

Time: $O(A)$

"Good" algorithm: $O(\log A + \log B)$

"Good" algo: poly$(\log A, \log B)$