Dynamic Programming

CSE 101: Design and Analysis of Algorithms

Lecture 17

CSE 101: Design and analysis of algorithms

- Dynamic programming
  - Reading: Chapter 6
- Quiz 3 is today, last 40 minutes of class
- Homework 7 will be assigned Nov 29
  - Due Dec 6, 11:59 PM

Dynamic Programming

- Dynamic programming is an algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved

Dynamic Programming

- Dynamic programming is an algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until they are all solved
- Example: fib2

Fibonacci sequence definition

- The sequence of integers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Each number is the sum of the previous two numbers

- $F(1) = 1$
- $F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$

Fibonacci sequence, algorithm 1

function fib1(n)

if $n = 1$ then return 1

if $n = 2$ then return 1

return fib1(n-1) + fib1(n-2)

- Let T(n) be the number of computer steps it takes to calculate fib1(n)

If $n < 3$ then $0 < T(n) < 3$

If $n > 3$ then $T(n) > T(n-1) + T(n-2)$

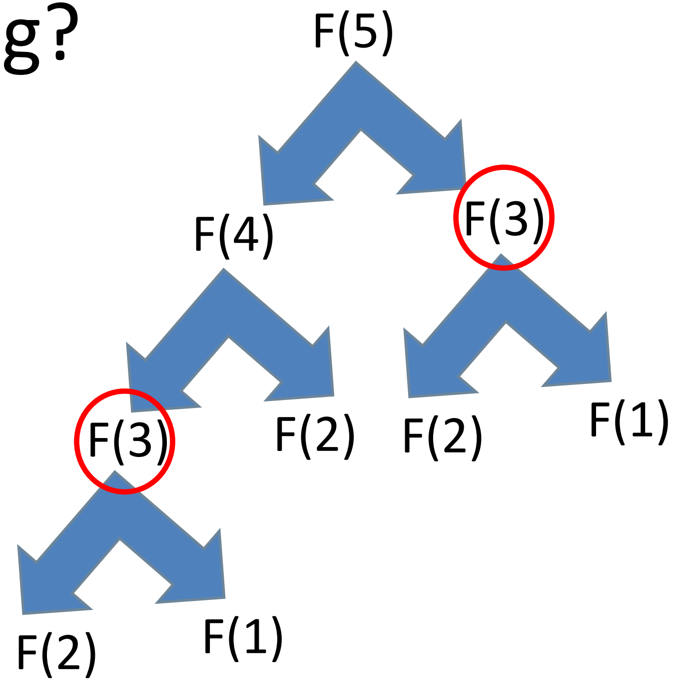
So we have that $T(n) > F(n)$

Fibonacci numbers grow fast!

$F(n) \sim 1.6^n$

Fibonacci sequence, algorithm 1

- Why does it take so long?
  - Recomputing



General principle: store and re-use

- If an algorithm is **recomputing** the same thing many times, we should **store and re-use** instead of recomputing
- Basis for dynamic programming

Fibonacci sequence, algorithm 2

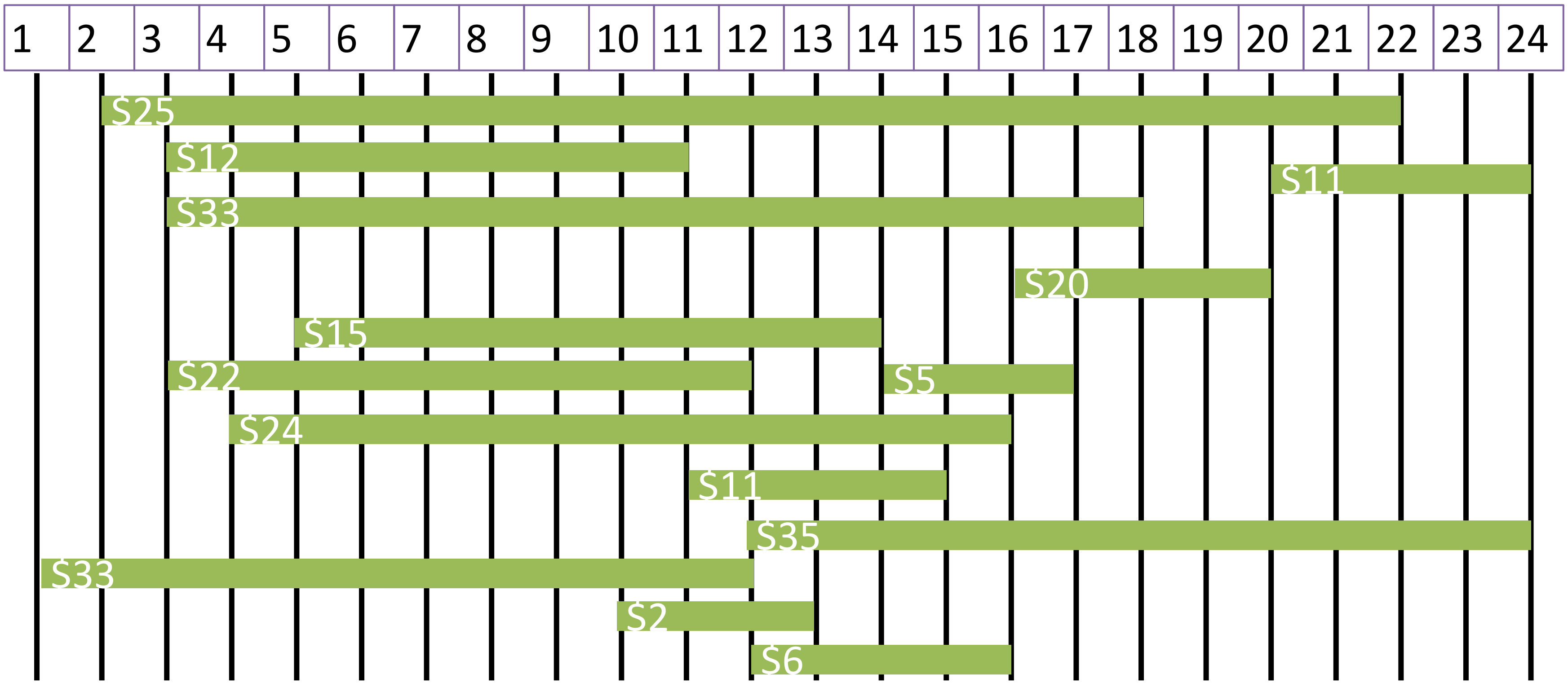
function fib2(*n*)
if *n* = 1 then return 1
if *n* = 2 then return 1
create array f[1…n]
f[1] := 1
f[2] := 1
for i =3 … n:
    f[i] := f[i-1] + f[i-2]
return f[n]

The **for** loop consists of a single computer step, so in order to compute f[*n*], you need *n* – 1 + 2 computer steps!

This is a huge improvement: *linear time (O(n)) vs. exponential time* ($O(1.6^n)$)

WEIGHTED EVENT SCHEDULING

Weighted event scheduling



Weighted event scheduling

- Instance: list of n intervals I= (s,f), with associated values v
- Solution format: subset of intervals
- Constraints: cannot pick intersecting intervals
- Objective function: maximize total value of intervals chosen

Weighted event scheduling

- No known greedy algorithm
  - In fact, Borodin, Nielsen, and Rackoff formally prove no greedy algorithm even approximates
- Brute force? 2^n subsets

Backtracking

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities:
  - Case 1: we exclude I_1, recurse on $[I_2, \ldots, I_n]$
  - Case 2: we include I_1, recurse on the set of all intervals that **do not** conflict with I_1
    - Is there a better way to describe this set? All events that start after I_1 finished, $[I_J, \ldots, I_n]$ for some J

Backtracking, runtime

- Sort events by start time
- Pick first to start. I_1 not necessarily good to include, so we will try both possibilities: exclude I_1 and include I_1

BTWES ($I_1 \ldots I_n$): in order of start times $T(n)$

If n=0 return 0

If n=1 return V_1

Exclude:= BTWES($I_2 .. I_n$) $T(n-1)$

J:=2

Until (J > n or s_J > f_1) do J++

Include:= V_1 +BTWES($I_J .. I_n$) $T(n-J)$

return Max(Include, Exclude)

$$T(n) = T(n-1) + T(n-J) + O(\text{poly})$$
$$T(n) = O(2^n)$$

Backtracking, runtime

- $O(2^n)$ worst case time, same as exhaustive search
- We could try to improve or use *dynamic programming*

Example

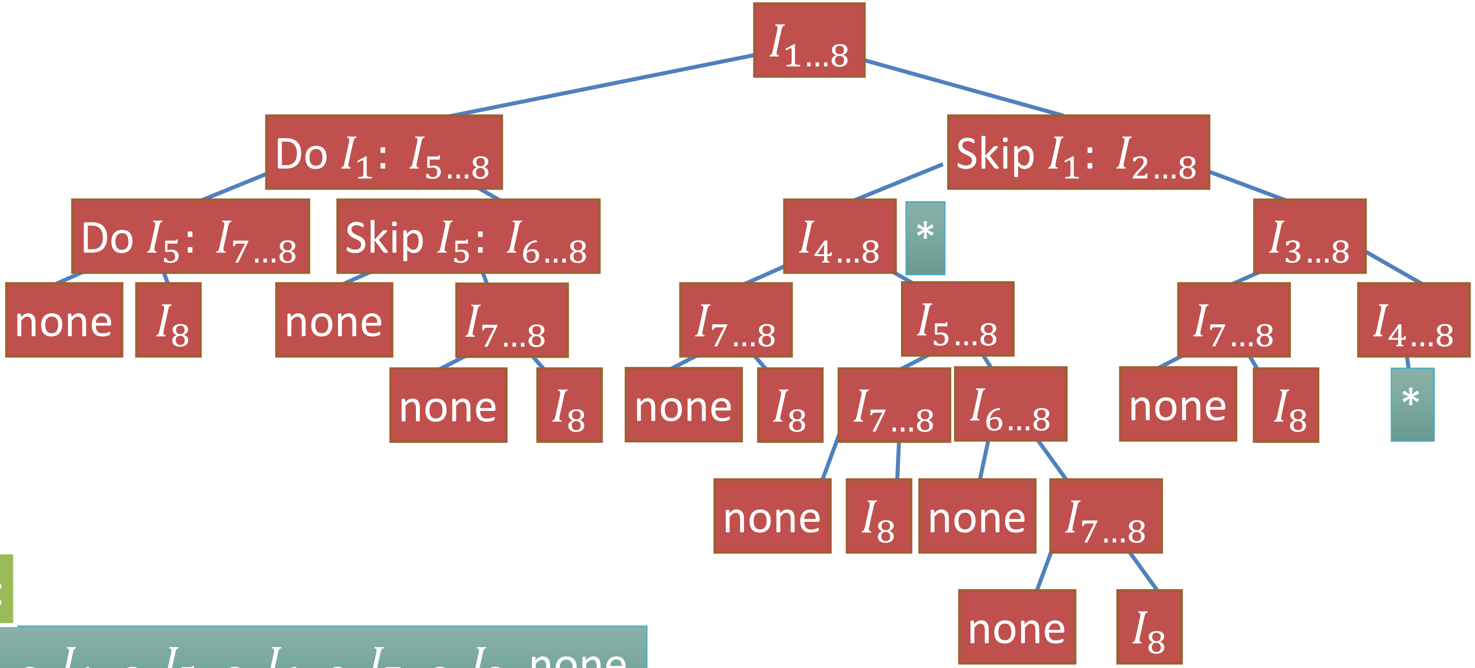
- $I_1 = (1,5), V_1 = 4$
- $I_2 = (2,4), V_2 = 3$
- $I_3 = (3,7), V_3 = 5$
- $I_4 = (4,9), V_4 = 6$
- $I_5 = (5,8), V_5 = 3$
- $I_6 = (6,11), V_6 = 4$
- $I_7 = (9,13), V_7 = 5$
- $I_8 = (10,12), V_8 = 3$

Total number of calls vs number of distinct calls

- We make up to 2^n recursive calls in our algorithm
- But every recursive call has the form $I_J \dots I_n$
- Thus, there are at most $n + 1$ different calls throughout
- Memoization: Store and reuse the answers, do not recompute

Example

$I_1 = (1,5), V_1 = 4$
$I_2 = (2,4), V_2 = 3$
$I_3 = (3,7), V_3 = 5$
$I_4 = (4,9), V_4 = 6$
$I_5 = (5,8), V_5 = 3$
$I_6 = (6,11), V_6 = 4$
$I_7 = (9,13), V_7 = 5$
$I_8 = (10,12), V_8 = 3$



Distinct calls:

$I_{1\ldots8}, I_{2\ldots8}, I_{3\ldots8}, I_{4\ldots8}, I_{5\ldots8}, I_{6\ldots8}, I_{7\ldots8}, I_8$, none

Characterize calls made

- All of the recursive calls BTWES makes are to arrays of the form $I_{K\ldots n}$ or empty with K=1…n
- So, of the 2^n recursive calls we might make, only $n+1$ distinct calls are made
- Just like Fibonacci numbers: many calls made exponentially often
- Solution same: create array to store and re-use answers, rather than repeatedly solving them

Dynamic programming steps

- Step 1: Define subproblems and corresponding array
- Step 2: What are the base cases?
- Step 3: Give recursion for subproblems
- Step 4: Find bottom-up order
- Step 5: What is the final output?
- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

- For analysis:
- Step 7: Correctness proof
- Step 8: Runtime analysis

Dynamic programming steps

- Step 1: Define subproblems and corresponding array

Dynamic programming steps

- Step 1: Define subproblems and corresponding array

Given an input $[I_1, \dots, I_n]$, output the maximum value subset

Dynamic programming steps

- Step 1: Define subproblems and corresponding array

Given an input $[I_1, \dots, I_n]$, output the maximum value subset

Given an input $[I_1, \dots, I_k]$, where $k \leq n$, output the maximum value subset

Dynamic programming steps

- Step 1: Define subproblems and corresponding array

Given an input $[I_1, \dots, I_n]$, output the maximum value subset

Given an input $[I_1, \dots, I_k]$, where $k \leq n$, output the maximum value subset

Let MVS$[k]$ be the maximum value subset of $[I_1, \dots, I_k]$

Dynamic programming steps

- Step 2: What are the base cases?

Dynamic programming steps

- Step 2: What are the base cases?

MVS[0] = 0

MVS[1] = V_1

Dynamic programming steps

- Step 3: Give recursion for subproblems

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

What is MVS[k] in terms of other subproblems?

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

Case 2: I_k is part of the max value subset

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

Case 2: I_k is part of the max value subset

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

MVS[k] = MVS[$k-1$]

Case 2: I_k is part of the max value subset

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

MVS[k] = MVS[$k - 1$]

Case 2: I_k is part of the max value subset

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

MVS[k] = MVS[$k-1$]

Case 2: I_k is part of the max value subset

MVS[k] = $V_k +$ max value subset of intervals not conflicting with k

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

MVS[k] = MVS[$k-1$]

Case 2: I_k is part of the max value subset

MVS[k] = $V_k +$ max value subset of intervals not conflicting with k

MVS[k] = $V_k +$ MVS[j], where I_j is the last interval before I_k starts

Dynamic programming steps

- Step 3: Give recursion for subproblems

MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

What is MVS[k] in terms of other subproblems?

Is I_k part of the max value subset?

Case 1: I_k is not part of the max value subset

MVS[k] = MVS[$k-1$]

Case 2: I_k is part of the max value subset

MVS[k] = $V_k +$ max value subset of intervals not conflicting with k

MVS[k] = $V_k +$ MVS[j], where I_j is the last interval before I_k starts

MVS[k] = max(MVS[$k-1$] , $V_k +$ MVS[j]), where I_j is the last interval before I_k starts

Dynamic programming steps

- Step 4: Find bottom-up order

Dynamic programming steps

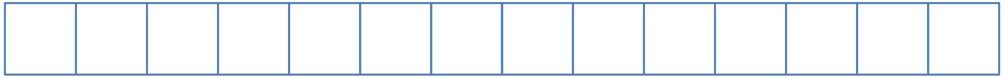
- Step 4: Find bottom-up order

Order from 0 to n

Dynamic programming steps

- Step 4: Find bottom-up order

Order from 0 to n

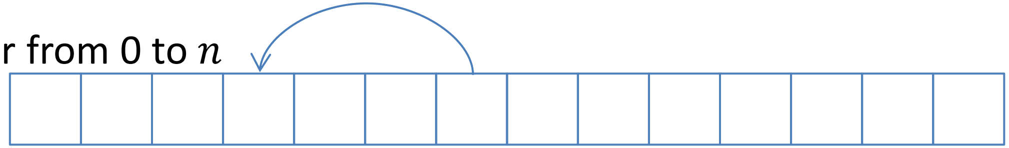
MVS 

Dynamic programming steps

- Step 4: Find bottom-up order

Order from 0 to n

MVS

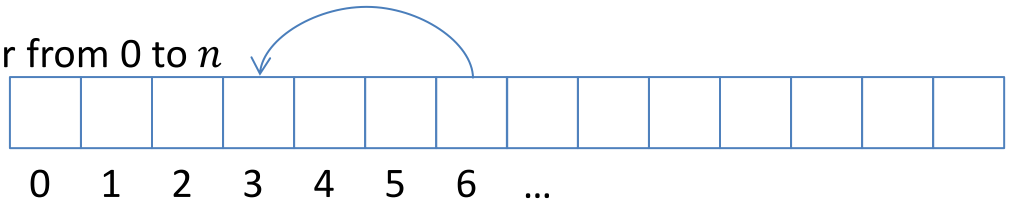


Dynamic programming steps

- Step 4: Find bottom-up order

Order from 0 to n

MVS



for $k = 0$ to n

Dynamic programming steps

- Step 5: What is the final output?

Dynamic programming steps

- Step 5: What is the final output?

MVS[n] is the maximum value subset of $[I_1, \ldots, I_n]$

Dynamic programming steps

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

Dynamic programming steps

- Step 6: Put it all together into an iterative algorithm that fills in the array step by step

maxsubset($I_1 \dots I_n$): in order of start times

    MVS[0] = 0
    MVS[1] = V_1
    Create array MVS[0, …, n]
    for $k = 2 \dots n$
        $j = 1$
        while $f_j \leq s_k$
            j++
        MVS[k] = max(MVS[$k - 1$] , V_k + MVS[$j - 1$])
    return MVS[n]

Dynamic programming steps

- Step 7: Correctness proof

Dynamic programming steps

- Step 7: Correctness proof

Claim: MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$ for $k = 0$ to n

Dynamic programming steps

- Step 7: Correctness proof

Claim: MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$ for $k = 0$ to n

Base cases:

Dynamic programming steps

- Step 7: Correctness proof

Claim: MVS$[k]$ is the maximum value subset of $[I_1, \ldots, I_k]$ for $k = 0$ to n

Base cases:

MVS[0] = 0 is the maximum value of the empty set

MVS[1] = V_1 is the maximum value when there is only one interval

Dynamic programming steps

- Step 7: Correctness proof

Claim: MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$ for $k = 0$ to n

Base cases:

MVS[0] = 0 is the maximum value of the empty set

MVS[1] = V_1 is the maximum value when there is only one interval

Assume for some $k > 1$, MVS[i] is the maximum value subset for intervals $I_1, \dots, I_i$ for all i such that $0 \leq i < k$

Dynamic programming steps

- Step 7: Correctness proof (continued)

Want to show MVS[k] is the maximum value subset of [$I_1, \ldots, I_k$]

Dynamic programming steps

- Step 7: Correctness proof (continued)

Want to show MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

Case 1: suppose I_k is not part of the max value subset

Case 2: suppose I_k is part of the max value subset

Dynamic programming steps

- Step 7: Correctness proof (continued)

Want to show MVS[k] is the maximum value subset of $[I_1, \ldots, I_k]$

Case 1: suppose I_k is not part of the max value subset

Then, MVS[k] should be the same as the max value subset of $[I_1, \ldots, I_{k-1}]$ which is stored in MVS[$k - 1$]

Case 2: suppose I_k is part of the max value subset

Then, all intervals before I_k that conflict with I_k cannot be included. If I_j is the last interval that does not conflict with I_k, then MVS[k] should be the value of I_k plus the maximum value subset on intervals $[I_1, \ldots, I_j]$ which is stored in MVS[j].

Dynamic programming steps

- Step 7: Correctness proof (continued)

Want to show MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

Case 1: suppose I_k is not part of the max value subset

Then, MVS[k] should be the same as the max value subset of $[I_1, \dots, I_{k-1}]$ which is stored in MVS[$k-1$]

Case 2: suppose I_k is part of the max value subset

Then, all intervals before I_k that conflict with I_k cannot be included. If I_j is the last interval that does not conflict with I_k, then MVS[k] should be the value of I_k plus the maximum value subset on intervals $[I_1, \dots, I_j]$ which is stored in MVS[j].

Since Case 1 and Case 2 are the only possibilities, the maximum value subset should be the maximum of these two values. MVS[k] = max(MVS[$k-1$] , $V_k +$ MVS[j])

Dynamic programming steps

- Step 7: Correctness proof (continued)

Want to show MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

Case 1: suppose I_k is not part of the max value subset

Then, MVS[k] should be the same as the max value subset of $[I_1, \dots, I_{k-1}]$ which is stored in MVS[$k - 1$]

Case 2: suppose I_k is part of the max value subset

Then, all intervals before I_k that conflict with I_k cannot be included. If I_j is the last interval that does not conflict with I_k, then MVS[k] should be the value of I_k plus the maximum value subset on intervals $[I_1, \dots, I_j]$ which is stored in MVS[j].

Since Case 1 and Case 2 are the only possibilities, the maximum value subset should be the maximum of these two values. MVS[k] = max(MVS[$k - 1$] , $V_k +$ MVS[j])

Conclusion: MVS[k] is the maximum value subset of $[I_1, \dots, I_k]$

Dynamic programming steps

- Step 8: Runtime analysis

Dynamic programming steps

- Step 8: Runtime analysis

maxsubset($I_1 \ldots I_n$): in order of start times

    MVS[0] = 0
    MVS[1] = V_1
    Create array MVS[0, …, n] $O(1)$
    for $k = 2 \ldots n$ n times
        $j = 1$
        while $f_j \leq s_k$ $O(n)$
            j++
        MVS[k] = max(MVS[$k-1$] , V_k + MVS[$j-1$]) $O(1)$
    return MVS[n]

Dynamic programming steps

- Step 8: Runtime analysis

maxsubset($I_1 \ldots I_n$): in order of start times

MVS[0] = 0

MVS[1] = V_1

Create array MVS[0, …, n]

$O(1)$

for $k = 2 \ldots n$ $\quad n$ times

$j = 1$

while $f_j \leq s_k$ $\quad O(n)$

j++

MVS[k] = max(MVS[$k-1$] , V_k + MVS[$j-1$]) $\quad O(1)$

return MVS[n]

Total runtime $O(n^2)$

Exercise: can you improve the runtime?

Next lecture

- Dynamic programming
  – Reading: Chapter 6