Objectives

Dynamic Programming
- Overview
- Fibonacci
- Weighted scheduling

Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, 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

Dynamic Programming History

Richard Bellman pioneered systematic study of dynamic programming in 1950s

Etymology
- Dynamic programming = planning over time
  - Not our typical use of programming
- Secretary of Defense was hostile to mathematical research
- Bellman sought an impressive name to avoid confrontation
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

WARMUP: FIBONACCI SEQUENCE

How Would You Solve Fibonacci Sequence?

Input: the number of fibonacci numbers I want

Output: display the list of fibonacci numbers

Sequence:
- $F_0=F_1=1$
- $F_n=F_{n-1}+F_{n-2}$

Soln 1: Using a List

Typical Solution:

```
fibs = []           # create an empty list
fibs.append(1)      # append the first two Fib numbers
fibs.append(1)
print 1, 1,
for x in xrange(2, N+1):
      newfib = fibs[x-1]+fibs[x-2]
      print newfib,
      fibs.append(newfib)

print fibs          # print out the list
```

Building up solution

Running time? Space cost?

Do we need a whole list?

Soln 2: Using Three Variables

Only need the solutions to the last two problems (F[k-1], F[k-2])

```
lastNum = 1
twoAgo = 1
print twoAgo, lastNum,

for n in xrange (2, N+1):

    nthNum = twoAgo + lastNum
    print nthNum,

    twoAgo = lastNum
    lastNum = nthNum
```

Write as a recurrence

Soln 3: Recursion

```
def fibonacci(n):
    return fibonacci(n-1) + fibonacci(n-2)
```

What is the running time of this algorithm?

Dynamic Programming Memoization Process

Create a table with the possible inputs

If the value is in the table, return it (without recomputing it); Otherwise, call function recursively

- Add value to table for future reference

Memoization Example: Fibonacci

```
memoized_fibonacci(n):
   for j = 1 to n:
      results[i] = -1 # -1 means undefined

   return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
   if results[n] != -1: # value is defined
      return results[n]
   if n == 1:
      val = 1
   elif n == 2:
      val = 1
   else:
      val = memoized_fib_recurs(results, n-2)
      val = val + memoized_fib_recurs(results, n-1)
   results[n] = val
   return val
```

Runtime?

Memoization Example: Fibonacci

```
memoized_fibonacci(n):
   for j = 1 to n:
      results[i] = -1 # -1 means undefined
   results[1] = 1
   results[2] = 1

   return memoized_fib_recurs(results, n)

memoized_fib_recurs(results, n):
   if results[n] != -1: # value is defined
      return results[n]

   val = memoized_fib_recurs(results, n-2)
   val = val + memoized_fib_recurs(results, n-1)
   results[n] = val
   return val
```

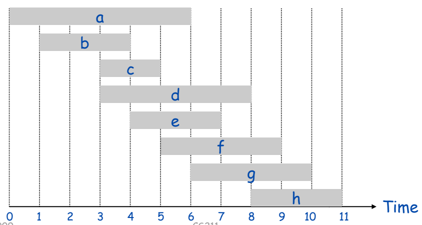

WEIGHTED INTERVAL SCHEDULING

Weighted Interval Scheduling

Job j starts at s_j, finishes at f_j, and has weight or value v_j

Two jobs are compatible if they don't overlap

Goal: find maximum weight subset of mutually compatible jobs



Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

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

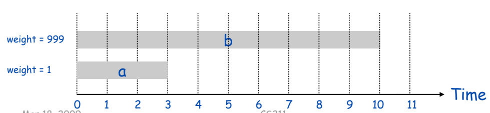
What happens if we add weights to the problem?

Limitation of Greedy Algorithm

Recall. Greedy algorithm works if all weights are 1.

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

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed

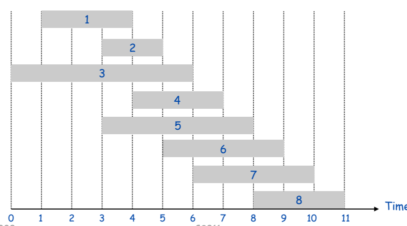


Weighted Interval Scheduling

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



Dynamic Programming

Assume we have an optimal solution

Notation. OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*

- What is something *obvious* we can we say about the optimal solution with respect to job *j*?

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*

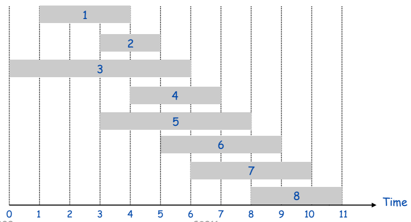
- Case 1: OPT selects job *j*
- Case 2: OPT does not select job *j*
  - Explore both of these cases…
    - What jobs are in OPT? Which are not?
  - Keep in mind our definition of p

Weighted Interval Scheduling

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

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



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*

- Case 1: OPT selects job *j*
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
- Case 2: OPT does not select job *j*
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

optimal substructure

Formulate OPT(j) as a recurrence relation

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the *problem* consisting of job requests 1, 2, ..., *j*

- Case 1: OPT selects job *j*
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
- Case 2: OPT does not select job *j*
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

Two options: Opt(j) = v_j + OPT(p(j))
Opt(j) = Opt(j-1)

Formulate OPT(j) in terms of smaller subproblems
Which should we choose?

Dynamic Programming: Binary Choice

Notation. OPT = value of optimal solution to the problem consisting of job requests 1, 2, ..., *j*

- Case 1: OPT selects job *j*
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(*j*)
- Case 2: OPT does not select job *j*
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

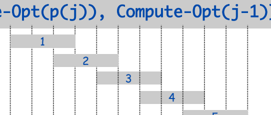
$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Choose the better of the two solutions

Weighted Interval Scheduling: Recursive Algorithm

```
Input: n jobs (associated start time s_j, finish time f_j,
and value v_j)

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n

Compute p(1), p(2), …, p(n)

Compute-Opt(j)
   if j = 0
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

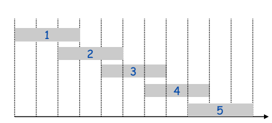
What is the run time?
(Trace for *n* = 5)



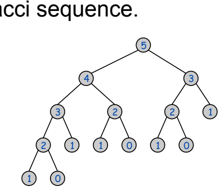
Weighted Interval Scheduling: Brute Force

Observation. Redundant sub-problems ⇒ exponential algorithms

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

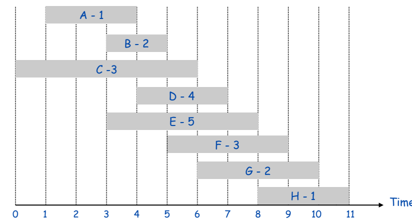
Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n jobs (associated start time sj, finish time fj, and
value vj)

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty      ← global array
M[0] = 0     ← Because we have jobs whose p(j) = 0

M-Compute-Opt(j):
   if M[j] is empty:
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
```

Need to analyze runtime…

Example

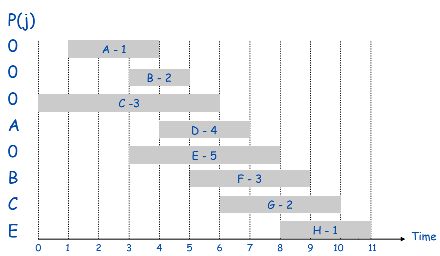
Jobs labeled with name - weight/value



M

0	A	B	C	D	E	F	G	H

Example



M

0	A	B	C	D	E	F	G	H