## Objectives

- Review: Closest Pair of Points
- Introduction to Dynamic Programming
  - Review: Fibonacci
  - Weighted interval scheduling



## Review

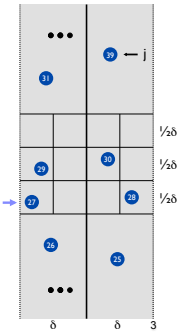
- What is the D&C algorithm for finding the closest pair of points?
- What is the key insight that leads to an O(n logn) algorithm?



## Analyzing Cost of Combining

Prepare minds to be blown…

- Def. Let $s_i$ be the point in the 2δ-strip, with the $i^{th}$ smallest y-coordinate
- Claim. If $|i - j| \geq 12$, then the distance between $s_i$ and $s_j$ is at least δ
  - What is the distance of the box?
  - How many points can be in a box?
  - When do we know that points are > δ apart?






## Review

- What was the new problem-solving technique we started to discuss on Monday?



## 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

How can we apply this template to our Fibonnaci problem?



## 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?

O(n)

## Memoization Example: Fibonacci

Alternative version…

```
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
```

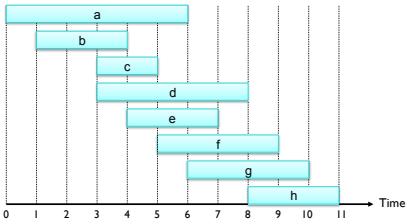


## WEIGHTED INTERVAL SCHEDULING



## 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

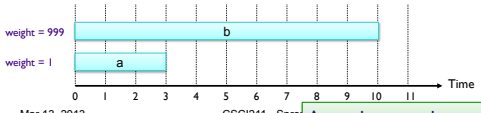
- Recall. Greedy algorithm works if all weights are 1 (or equivalent).
  - Consider jobs in ascending order of finish time
  - Add job to subset if it is compatible with previously chosen jobs

What happens to Greedy algorithm 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




Any other greedy approaches?



## Limitations of Greedy Algorithms

- Need to consider weight
  - No greedy algorithm works
- Need a more complex algorithm to solve problem

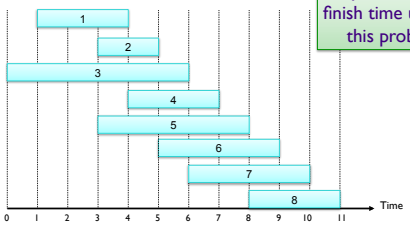
## Weighted Interval Scheduling

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

Why is ordering by finish time useful in this problem?






## Dynamic Programming

- Assume we have an optimal solution
- **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

- 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

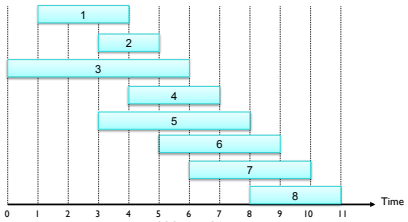


## Weighted Interval Scheduling

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

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: Binary Choice

- 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* — *optimal substructure*
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

Formulate OPT(j) as a recurrence relation



## Dynamic Programming: Binary Choice

- 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* — *optimal substructure*
    - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., *j*-1

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

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

## Dynamic Programming: Binary Choice

- 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

$$\text{Opt}(j) = \begin{cases} 0 & j=0 \\ \max\{ v_j + \text{Opt}(p(j)), \text{Opt}(j-1) \} & \text{Otherwise} \end{cases}$$

*Basecase*

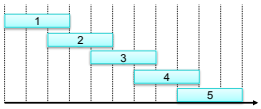
*Choose the "better" of the two solutions*



## 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)        Closest compatible job
Compute-Opt(j):
  if j = 0
    return 0
  else                    Picks j        Doesn't pick j
    return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
```

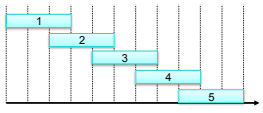
What is the runtime? (Trace for *n* = 5)

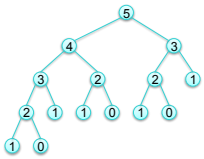





## Weighted Interval Scheduling: Brute Force

- 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

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

```
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)

for j = 1 to n
  M[j] = empty          ← global array
M[0] = 0

M-Compute-Opt(n)        ← Call function with initial input

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



## Looking Ahead

- PS7 due Friday
- SSA EC soon
- Exam 2 handed out Friday