Objectives

- Introduction to Greedy Algorithms
- Interval Scheduling

INTRODUCING GREEDY ALGORITHMS

Greedy Algorithms

At each step, take as much as you can get
→ "local" optimizations

- Need a proof to show that the algorithm finds an optimal solution
- A counter example shows that a greedy algorithm does not provide an optimal solution

Example of Greedy Algorithm

- How do you make change to give out the *fewest* coins?
- Determine for 34¢

Example of Greedy Algorithm

- How do you make change to give out the *fewest* coins?

```
while change > 0:
        if change >= 25:
                print "Quarter"
                change -= 25
        elif change >= 10:
                print "Dime"
                change -= 10
        …
```

Let's generalize …

- Ex: 34¢. 

Coin Changing

- Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.
- Ex: 34¢. 
- Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.
- Ex: $2.89. 

Coin-Changing: Greedy Algorithm

- Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins' denominations by value: c1 < c2 < … < cn.
                  coins selected
S = φ
while x ≠ 0
   let k be largest integer such that ck ≤ x
   if k = 0
      return "no solution found"   ← How could this happen?
   x = x - ck
   S = S ∪ {k}
return S
```

Is cashier's algorithm ***optimal***?

Coin-Changing: Analysis of Greedy Algorithm

- Theorem. Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100
- Pf. (by induction on x)
  - Consider optimal way to change $c_k \le x < c_{k+1}$
    - Greedy takes coin *k*
  - Any optimal solution must also take coin *k*
    - If not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to *x*
    - Table below indicates no optimal solution can do this
  - Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT
1	1	P ≤ 4	-
2	5	N ≤ 1	4
3	10	N + D ≤ 2	4 + 5 = 9
4	25	Q ≤ 3	20 + 4 = 24
5	100	no limit	75 + 24 = 99

Which coin | Coin value | If don't take c_k

Coin-Changing: Analysis of Greedy Algorithm

- Observation. Greedy algorithm is sub-optimal for US postal denominations:
  - 500 100 98 79 78 64 44 28 17 2 1
- Counterexample. 158¢.
  - Greedy: 100, 44, 2, 2, 2, 2, 2, 2, 2.
  - Optimal: 79, 79.

Proving Greedy Algorithms Work

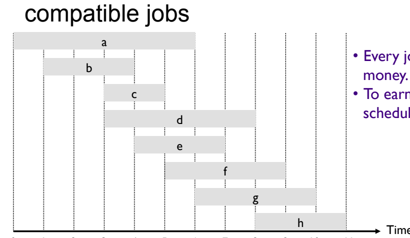
- Specifically, produce an **optimal** solution
- Approaches:
  - Greedy algorithm stays ahead
    - Does better than any other algorithm at each step
  - Exchange argument
    - Transform any solution into a greedy solution
  - Structural argument
    - Figure out some structural bound that all solutions must meet

Greedy algorithm stays ahead

INTERVAL SCHEDULING

Interval Scheduling

- Job j starts at s_j and finishes at f_j
- Two jobs are **compatible** if they don't overlap
- **Goal**: find maximum subset of mutually compatible jobs



- Every job is worth equal money.
- To earn the most money → schedule the most jobs

Greedy Algorithm Template

- Consider jobs (or whatever) in some order
  - Decision: What order is best?
- Take each job provided it's compatible with the ones already taken

What are options for orders? (rhetorical for now)

What is our goal?
What are we trying to minimize/maximize?

What is the worst case?

Greedy Algorithm Pseudo-Code

In some specified order

```
Set Greedy (Set candidate){
    solution = new Set( );
    while candidate.isNotEmpty()
        next = candidate.select() //use selection criteria,
        //remove from candidate and return value
        if solution.isFeasible(next) //constraints satisfied
            solution.union(next)
        if solution.solves()
            return solution

    //No more candidates and no solution
    return null
}
```

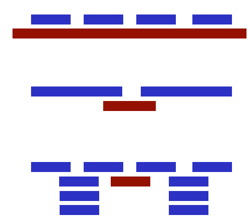

Interval Scheduling

- Earliest start time. Consider jobs in ascending order of start time s_j
  - Utilize CPU as soon as possible
- Earliest finish time. Consider jobs in ascending order of finish time f_j
  - Resource becomes free ASAP
  - Maximize time left for other requests
- Shortest interval. Consider jobs in ascending order of interval length $f_j - s_j$
- Fewest conflicts. For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j

Can we "break" any of these?
i.e., prove they're not optimal?

Counterexamples to Optimality of Various Job Orders

Not optimal when …



breaks earliest start time

breaks shortest length

breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

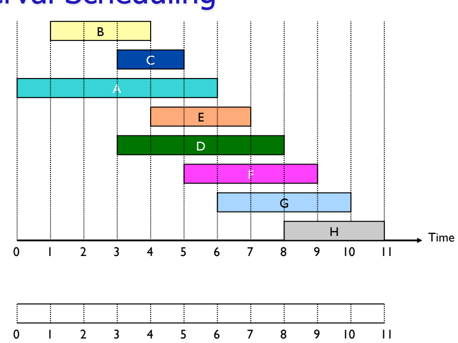
- Consider jobs in increasing order of finish time
- Take each job provided it's compatible with the ones already taken

jobs selected

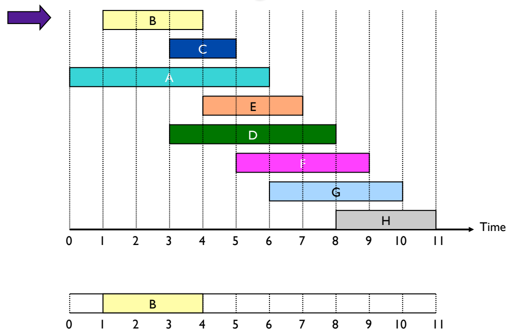
```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn

G = {}
for j = 1 to n
    if job j compatible with G
        G = G ∪ {j}
return G
```


Interval Scheduling

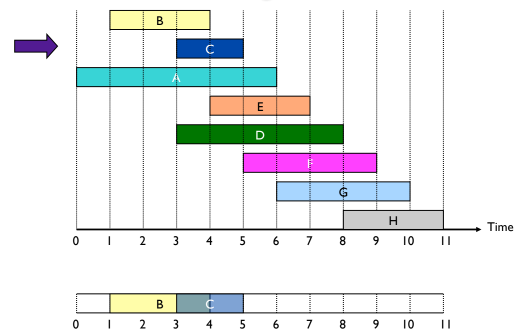


Interval Scheduling



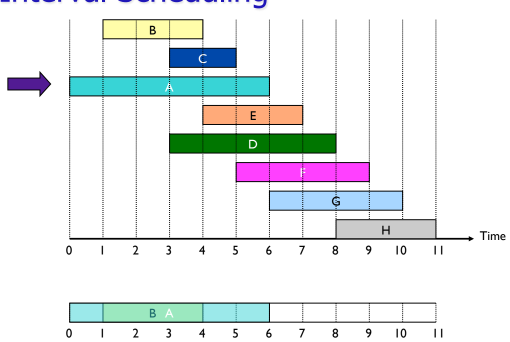
Feb 3, 2012 CSCI211 - Sprenkle 19

Interval Scheduling



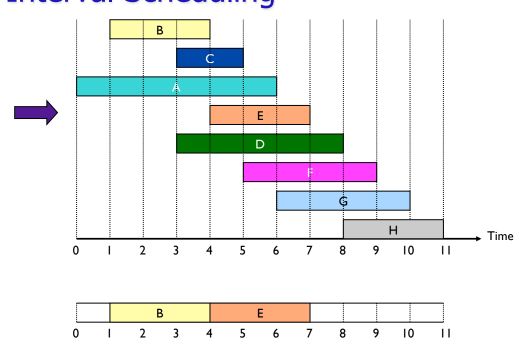
Feb 3, 2012 CSCI211 - Sprenkle 20

Interval Scheduling



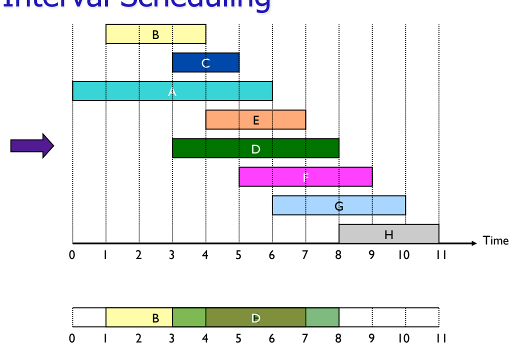
Feb 3, 2012 CSCI211 - Sprenkle 21

Interval Scheduling



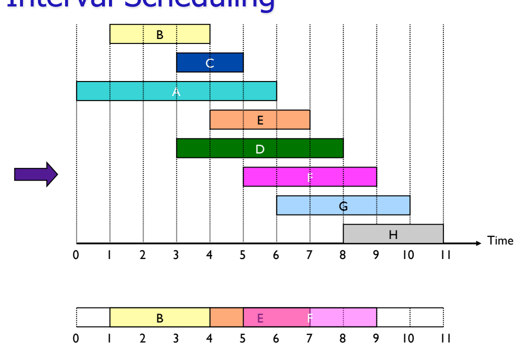
Feb 3, 2012 CSCI211 - Sprenkle 22

Interval Scheduling



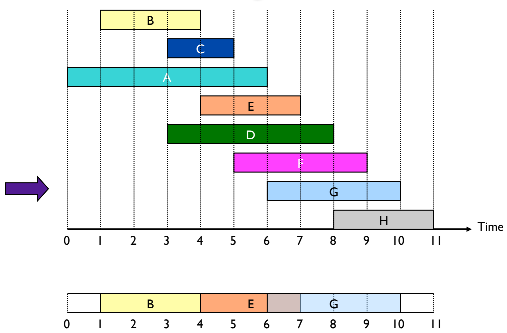
Feb 3, 2012 CSCI211 - Sprenkle 23

Interval Scheduling

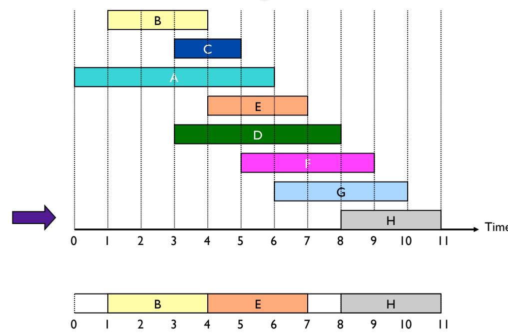


Feb 3, 2012 CSCI211 - Sprenkle 24

Interval Scheduling



Interval Scheduling



Interval Scheduling: Greedy Algorithm

- Consider jobs in increasing order of finish time
- Take each job provided it's compatible with the ones already taken

jobs selected

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn

G = {}
for j = 1 to n
   if job j compatible with G
      G = G ∪ {j}
return G
```

Runtime of algorithm?
- Where/what are the costs?

Interval Scheduling: Greedy Algorithm

- Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

jobs selected

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn      O(n logn)

G = {}
for j = 1 to n
   if job j compatible with G     O(1)  } O(n)
      G = G ∪ {j}
return G
```

- Implementation. O(n log n)
  - Remember job j* that was added last to A
  - Job j is compatible with A if $s_j \geq f_j^*$

Analyzing Interval Scheduling

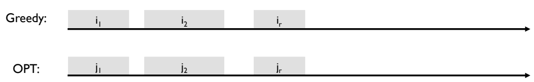
- Know that the intervals are compatible
  - Handled by the if statement
- But is it optimal?
  - What does it mean to be optimal?
  - Recall our goal for maximization

Greedy Stays Ahead Proofs

1. Define your solutions
   - Describe the form of your greedy solution and of some other solution (possibly the optimal solution)
     - Example: Let A be the solution constructed by the greedy algorithm and O be a solution
2. Find a measure
   - Find a measure by which greedy stays ahead of the optimal solution
     - Ex: Let $a_1, \ldots, a_k$ be the first k measures of greedy algorithm and $o_1, \ldots, o_m$ be the first m measures of other solution (sometimes m = k)
3. Prove greedy stays ahead
   - Show that the partial solutions constructed by greedy are always just as good as the optimal solution's initial segments based on the measure
     - Ex: for all indices $r \leq \min(k,m)$, prove by induction that $a_r \geq o_r$ or $a_r \leq o_r$
   - Use the greedy algorithm to help you argue the inductive step
4. Prove optimality
   - Prove that since greedy stays ahead of the other solution with respect to the measure, then the greedy solution is optimal

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume greedy is not optimal
  - Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by *greedy* (*k* jobs)
  - Let $j_1, j_2, \ldots, j_m$ denote set of jobs in *optimal* solution (*m* jobs)
  - Both ordered by finish time for comparison ordering
  - Want to show that $k = m$

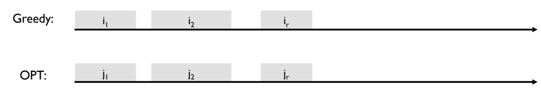


What can we say about i_1 and j_1? $f(i_1) \le f(j_1)$

Interval Scheduling: Analysis

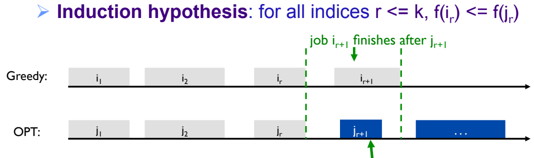
- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Since we picked the first job to have the first finishing time, we know that $f(i_1) <= f(j_1)$
  - Want to show that Greedy "stays ahead"
  - Each interval finishes at least as soon as Optimal's
  - **Induction hypothesis**: for all indices $r <= k$, $f(i_r) <= f(j_r)$

Prove for r+1



Interval Scheduling: Analysis

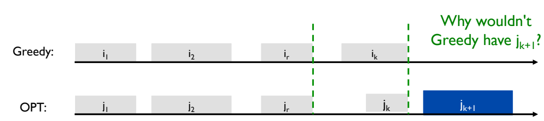
- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Since we picked the first job to have the first finishing time, we know that $f(i_1) <= f(j_1)$
  - Want to show that Greedy "stays ahead"
  - Each interval finishes at least as soon as Optimal's
  - **Induction hypothesis**: for all indices $r <= k$, $f(i_r) <= f(j_r)$



How Greedy stays ahead   why not replace job i_{r+1} with job j_{r+1}?

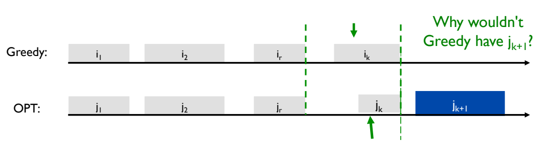
Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume Greedy is not optimal (i.e., $m > k$)
    - Optimal solution has more jobs than Greedy
  - We already showed that for all indices $r \le k$, $f(i_r) \le f(j_r)$
  - Since $m > k$, there is a request j_{k+1} in Optimal



Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - Assume Greedy is not optimal (i.e., $m > k$)
  - We already showed that for all indices $r \le k$, $f(i_r) \le f(j_r)$
  - Since $m > k$, there is a request j_{k+1} in Optimal
    - Starts after j_k ends → after i_k ends
  - So, Greedy could also add j_k
    - Contradiction because now Greedy has another job



Greedy Algorithm Pseudo-Code

In some specified order

```
Set Greedy (Set candidate){
    solution = new Set( );
    while candidate.isNotEmpty()
        next = candidate.select() //use selection criteria,
        //remove from candidate and return value
        if solution.isFeasible(next) //constraints satisfied
            solution.union(next)
        if solution.solves()
            return solution

    //No more candidates and no solution
    return null
}
```

Problem Assumptions

- All requests were known to scheduling algorithm
  - Online algorithms: make decisions without knowledge of future input
- Each job was worth the same amount
  - What if jobs had *different* values?
    - E.g., scaled with size
- Single resource requested
  - Rejected requests that didn't fit

Assignments

- Exam 1
  - Can use book, lecture notes, your notes
  - No "outside" resources
  - Limited access to me
  - Consider typing up answers
  - Due Monday after Mock Con
- No journal for Tuesday