Give Me Greed, Now!*

* Use greed with extreme caution

Slides adapted from Ran Libeskind-Hadas and David Kauchak

Administrative

Assignment out today (back to the normal routine)

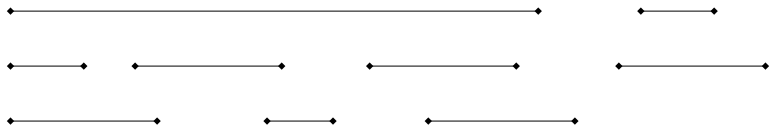
Midterm

Greedy Algorithms

- The idea:
  - When we have a choice to make, make the one that looks the best *right now!*
  - Make a *locally optimal choice* in hopes of a *globally optimal solution*
  - Don't always generate optimal solutions, but can.
    - General characteristics of when greed is good (optimal)

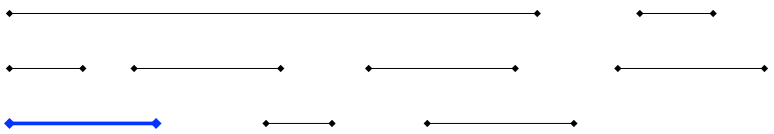
Interval scheduling

Given *n* activities A = [a_1,a_2, .., a_n] where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.
Could also optimize for utilization, rental fees, etc.



Interval scheduling

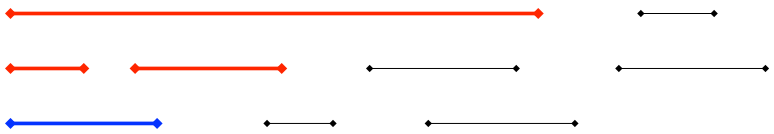
Given *n* activities A = [a_1,a_2, .., a_n] where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.



Which activities conflict?

Interval scheduling

Given *n* activities A = [a_1,a_2, .., a_n] where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.



Which activities conflict?

Simple recursive solution

Enumerate all possible solutions and find which schedules the most activities

```
INTERVALSCHEDULE-RECURSIVE(A)
1  if A = {}
2          return 0
3  else
4          max = −∞
5          for all a ∈ A
6                  A' ← A minus a and all conflicting activites with a
7                  s = INTERVALSCHEDULE-RECURSIVE(A')
8                  if s > max
9                          max = s
10         return 1 + max
```

Simple recursive solution

Is it correct?

- max{all possible solutions}

Running time?

- O(n!)

```
INTERVALSCHEDULE-RECURSIVE(A)
1  if A = {}
2          return 0
3  else
4          max = −∞
5          for all a ∈ A
6                  A' ← A minus a and all conflicting activites with a
7                  s = INTERVALSCHEDULE-RECURSIVE(A')
8                  if s > max
9                          max = s
10         return 1 + max
```

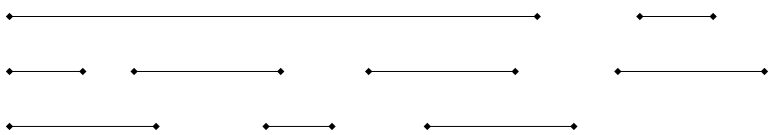
Can we do better?

Dynamic programming (next class)

- $O(n^2)$

Greedy solution – Is there a way to repeatedly make local decisions?

- Key: we'd still like to end up with the *optimal* solution



Overview of a greedy approach

Greedily pick an activity to schedule

Add that activity to the answer

Remove that activity and all conflicting activities. Call this A'.

Repeat on A' until A' is empty

Worksheet

- List possible strategies for selecting an activity to add to our schedule.

Greedy options

Select the activity that starts the earliest, i.e. argmin$\{s_1, s_2, s_3, \ldots, s_n\}$?

Homework 6a

Greedy options

Select the shortest activity, i.e. argmin{f_1-s_1, f_2-s_2, f_3-s_3, …, f_n-s_n}

Homework 6a

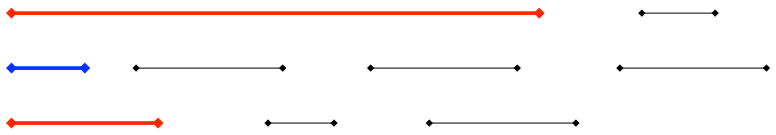
Greedy options

Select the activity with the smallest number of conflicts

Homework 6a

Greedy options

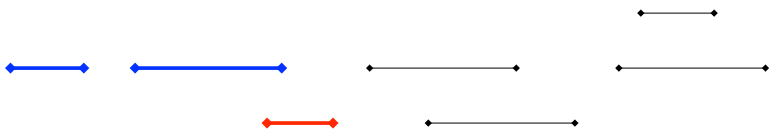
Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?



remove the conflicts

Greedy options

Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?



remove the conflicts

Greedy options

Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?

Greedy options

Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?

Multiple optimal solutions

Greedy options

Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?

Greedy options

Select the activity that ends the earliest, i.e. argmin{f_1, f_2, f_3, …, f_n}?

Efficient greedy algorithm

Once you've identified a reasonable greedy heuristic:

- Prove that it always gives the correct answer
- Develop an efficient solution

An efficient solution

```
INTERVALSCHEDULE-GREEDY(A)
1  sort A based on finish times f_i
2  for i ← 1 to n
3          add a_i to R
4          finish ← f_i
5          while s_i < finish
6                  i ← i + 1
7  return R
```

Is our greedy approach correct?

"Stays ahead" argument:

show that no matter what other solution someone provides you, the solution provided by your algorithm always "stays ahead", in that no other choice could do better

an example of a "safety argument

Is our greedy approach correct?

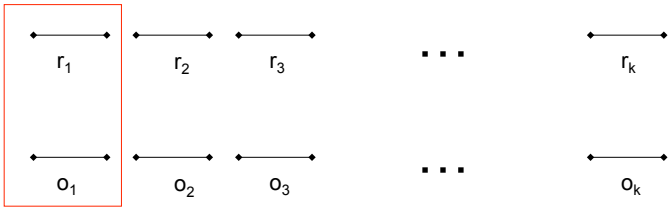
"Stays ahead" argument

Let $r_1, r_2, r_3, \ldots, r_k$ be the solution found by our approach

Let $o_1, o_2, o_3, \ldots, o_k$ of another optimal solution

Show our approach "stays ahead" of any other solution

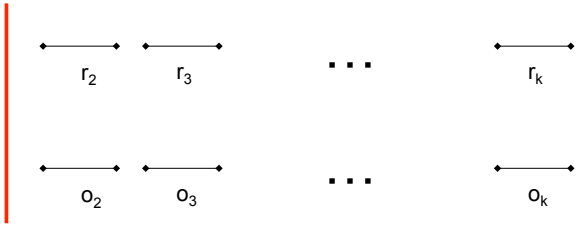
Stays ahead



Compare first activities of each solution

what do we know?

Stays ahead



We have **at least** as much time as any other solution to schedule the remaining 2…k tasks

Greedy Strategy

- How do we find greedy strategies that work?

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always *safe*.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

Proving that the Greedy Algorithm is Correct: Safety and Induction!

Claim: The greedy algorithm finds the largest number of non-overlapping courses

Proof: Strong induction on the number of intervals, n.

Basis: n = 0.

Induction hypothesis: Assume the claim is true for *any* set with *any* number of intervals between 0 and n.

Induction Step: Consider any set of n+1 intervals. We wish to show that the greedy algorithm finds the largest number of non-overlapping intervals. (Notice no induction pitfall here!)

First, we claim that there exists some optimal solution that uses the first course (the one with the earliest ending time) which we'll denote C.

Consider any optimal solution S. If it includes C, our claim is true. If it doesn't include C, let C' be the course in S that ends first.

Note that every other course in S begins after the ending time of C': If it starts before C' ends then it must end before C' even starts since it can't overlap with C'. But this contradicts the assumption that C' ends first in S.

By definition, C ends at the same time or before C'. So, C has no conflicts with any other course in S. Therefore, we can replace C' by C in S and we have an optimal solution that contains C.

Now, we know that C is contained in some optimal solution.

So, that optimal solution must choose from the set of courses that remain after removing C and all courses that overlap with C. Call that set T.

From set T, we clearly want an optimal solution.

Our greedy algorithm now recurses on T.

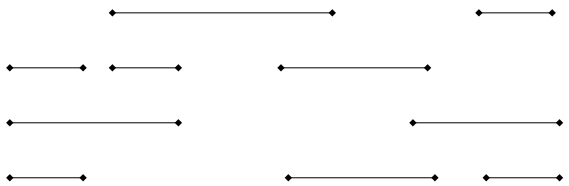
The number of courses in T is between 0 and n. By the strong induction hypothesis the greedy algorithm finds the maximum number of courses in T.

Your turn to be greedy!

- Break into groups of 3-4
- Select one of the following variants to solve:
  - Schedule *all* activities using as few rooms as possible in the Shan
  - If each activity has a value associated with it, select a schedule that maximizes total value of schedule

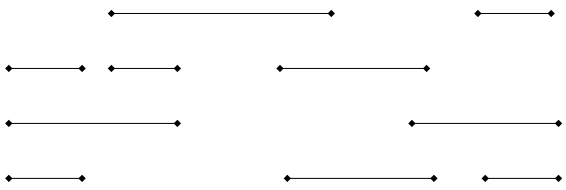
Scheduling *all* intervals

Given *n* activities, we need to schedule **all** activities.
Goal: minimize the number of resources required.

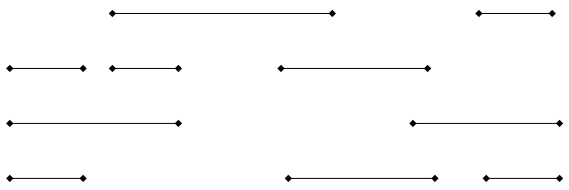


Greedy approach?

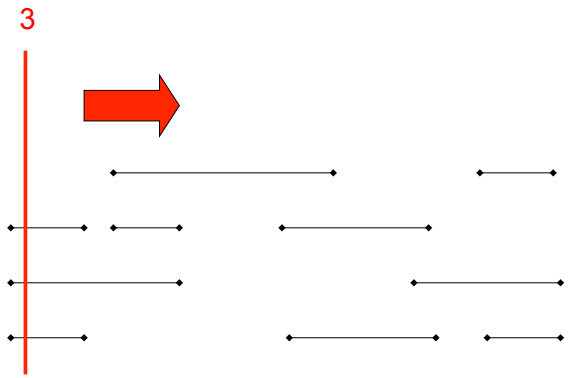
The best we could ever do is the maximum number of conflicts for any time period



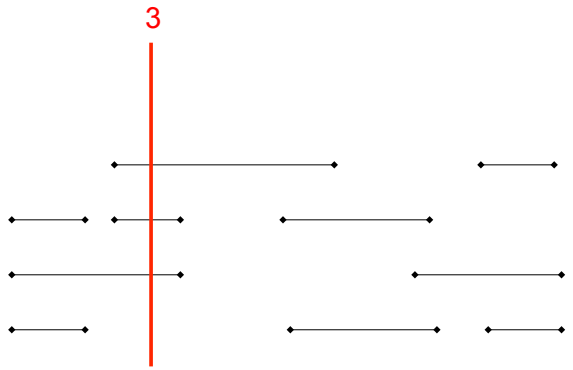
Calculating max conflicts efficiently



Calculating max conflicts efficiently



Calculating max conflicts efficiently



Max-Count-Algorithm

We can do no better then the max number of conflicts. This exactly counts the max number of conflicts.

```
AllIntervalScheduleCount(A)
1   Sort the start and end times, call this X
2   current ← 0
3   max ← 0
4   for i ← 1 to length[X]
5           if x_i is a start node
6                   current++
7           else
8                   current--
9           if current > max
10                  max ← current
11  return max
```

Greedy algorithms

What is a greedy algorithm?

Algorithm that makes a local decision with the goal of creating a globally optimal solution

Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems

What does this mean? Where have we seen this before?

Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:

Break into sum number of sub problems, solve:

then possibly do a little work

Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:

The solution to the general problem is solved with respect to solutions to sub-problems!

Greedy vs. divide and conquer

Greedy

To solve the general problem:

Pick a locally optimal solution and repeat

Greedy vs. divide and conquer

Greedy

To solve the general problem:

The solution to the general problem is solved with respect to solutions to sub-problems!

Slightly different than divide and conquer

Warning: Greed is generally bad!

Safety and Induction Revisited

When is greed good?

- No general way to tell whether a greedy algorithm is optimal
- Two key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem

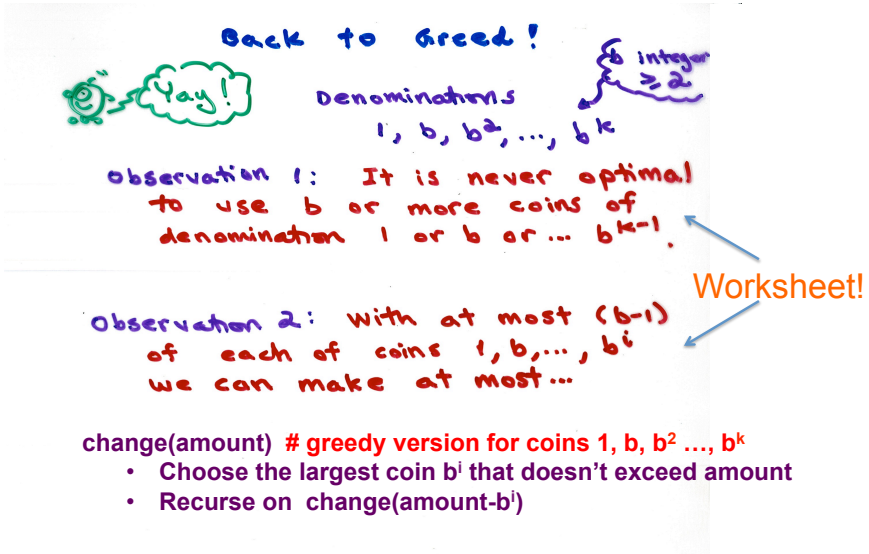
Making….

Worksheet

- How many bills/coins does it take to make the following amounts of change:
  - 31¢
  - $47.32
  - $9.99
- What is the greedy strategy for optimally making change?

Making Change!

```
change(42, [1, 5, 10, 25, 50])
change(42, [1, 5, 21, 28])  # greed?
change(amount, coins[1:i])
  if amount = 0: return 0
  elif i == 0: return Infinity
  elif coinValue[i] > amount:
     return change(amount, coins[1:i-1])
  else:
     useIt = 1 + change(amount-coinValue[i],
                        coins[1:i])
     loseIt = change(amount, coins[1:i-1])
     return min(useIt, loseIt)
```



change(amount) # greedy version for coins 1, b, b^2 …, b^k

- Choose the largest coin b^i that doesn't exceed amount
- Recurse on change(amount-b^i)

Proving Correctness

Proof by strong induction on amount, n
(not on our coin set 1, b, b^2, …, b^k)

Basis: n = 0

Induction hypothesis: Assume that the greedy algorithm uses the optimal number of coins for any amount from 0 to n.

Induction step: Consider an amount n+1. The greedy algorithm uses the largest coin b^i (i between 0 and k) that doesn't exceed n+1. We **first** claim that this choice is safe in the sense that there exists an optimal solution that uses a b^i coin.

Consider an optimal solution S (a multiset of coins) for amount n+1.

If S contains b^i then our claim is true. If not, then S must make up at least b^i from smaller coins 1, b, …, b^{i-1}.

But, by Observation 1, since S is optimal, it uses no more than b-1 of each of these coins.

By Observation 2, we can't make up b^i using at most b-1 of each of these coins. So, S *must* contain coin b^i.

Aha! (Normally left out of proofs) Now, the remaining amount $(n+1)-b^i$ must be made up using the least number of coins. But, our algorithm recurses on this amount and, by the induction hypothesis, it uses the least number of coins for that amount since $(n+1)-b^i$ is between 0 and n. Q.E.D.

Knapsack problems: Greedy or not?

0-1 Knapsack – A thief robbing a store finds n items worth v_1, v_2, .., v_n dollars and weight w_1, w_2, …, w_n pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he wants to maximize value.

Fractional knapsack problem – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of $0.2w_i$ and a value of $0.2v_i$.

Recap

- The idea:
  - When we have a choice to make, make the one that looks the best *right now!*
  - Make a *locally optimal choice* in hopes of a *globally optimal solution*
- Key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem