CS320 Algorithms: Theory and Practice

Course Introduction

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." - *Francis Sullivan*

Course Objectives

Algorithms:

- Design – strategies for algorithmic problem solving
  - advanced **data structures**
  - and their use in **algorithms**
- Reasoning about algorithm **correctness**
- Analysis of **time** and **space** complexity
- Implementation – create an implementation that respects the runtime analysis

Algorithmic Approaches:

- Divide and Conquer
- Greedy
- Dynamic programming

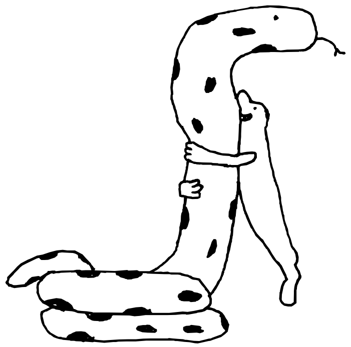
Problem Classes:

- P: Polynomial, NP: Non deterministic Polynomial

Grading

Programming Assignments	20%
Written Assignments	20%
Quizzes	10%
Exams	50%

Implementation



Programs will be written in Python:

- Concise and powerful
- Powerful **data structures**
  - tuples, dictionaries, arraylists
- Simple, easy to learn syntax
- Highly readable, compact code
- Supports object oriented and functional programming
- An extensive standard library
- Strong support for integration with other languages (C, C++, Java)

Python vs. e.g. Java

What makes Python different from Java?

- Java is statically typed, i.e. variables are bound to types at compile time. This avoids run time errors, but makes java programs more rigid.

- Python is dynamically typed, i.e. a variable takes on some type at run time, and its type can change. A variable can be of one type somewhere in the code and of another type somewhere else

```
# line is a String here
line = line.strip().split(" ")
# line is an (Array)List of Strings here
```

- This makes python programs more flexible, but can cause strange run time errors.

Does anyone else use Python?

One of the three “official languages” in Google.
(Guido van Rossum, creator of Python, was a Googler)

Peter Norvig, Director of Research at Google:

"Python has been an important part of Google since the beginning, and remains so as the system grows and evolved. Today dozens of Google engineers use Python, **and we're looking for more people with skills in this language**"

Yahoo groups, Yahoo maps -- 100% python

How will we learn Python?

- The website is our communication medium:
  http://www.cs.colostate.edu/~cs320/
  especially the Progress page, check it regularly

  Check the website for a link to a Python tutorial
  There is a lot more if you look around

- We will work in recitations, amongst others, on Python

Our approach to problem solving

- Formulate it with precision (usually using mathematical concepts, such as sets, relations, and graphs)
- Design an algorithm and its main data structures
- Prove its correctness
- Analyze its complexity (time, space)
  - Improve the initial algorithm (in terms of complexity), preserving correctness
- Implement it, preserving the complexity

Our first problem

Two parties e.g., companies and applicants

- Each applicant has a preference list of companies
- Each company has a preference list of applicants
- A possible scenario:

  cA offers job to aA

  aA accepts, but now gets offer from cX

  aA likes cX more, retracts offer from cA

  now cA offers job to aB, who retracts his acceptance of an offer from cB
- We would like a systematic method for assigning applicants to companies- stable matching
- A system like this is in use for matching residents with hospitals

Stable Matching

Goal. Given a set of preferences among companies and applicants, design a stable matching process.

Unstable pair: student x and company y are an unstable pair **(not in the current matching)** if:

- x prefers y to its assigned company.
- y prefers x to one of its admitted students.

Stable assignment. Assignment without unstable pairs.

- Natural and desirable condition.

Is some control possible?

Given the preference lists of applicants A and companies C, can we assign As to Cs such that

```
for each C
  for each A not scheduled to work for C
    either C prefers all its students to A
        or A prefers current company to C
```

If this holds, then what?

Stable state

Given the preference lists of applicants A and companies C, can we assign As to Cs such that

```
for each C
  for each A not scheduled to work for C
    either C prefers all its students to A
        or A prefers current company to C
```

If this holds, there is no unstable pair, and therefore individual self interest will prevent changes in student / company matches:

Stable state

Simplifying the problem

Matching students/companies problem a bit messy:

- Company may look for multiple applicants, students looking for a single internship
- Maybe there are more jobs than applicants, or fewer jobs than applicants
- Maybe some applicants/jobs are equally liked by companies/applicants (partial orders)

Formulate a "bare-bones" version of the problem

Stable Matching Problem

Perfect matching: Each man matched with exactly one woman, and each woman matched with exactly one man.

Stability: no incentive for some pair to undermine the assignment.

- A pair (m,w) NOT IN THE CURRENT MATCHING is an instability if man m and woman w prefer each other to current partners in the matching.
- Both m and w can improve their situation

Stable matching: perfect matching with no unstable pairs. Stable matching problem: Given the preference lists of n men and n women, find a stable matching if one exists.

The Stable Matching Problem

Problem: Given n men and n women where

- Each man lists women in total order of preference
- Each woman lists men in total order of preference
  - What is a total order? Do you know an example?

    Do you know a counter example?

	1st (favorite)	2nd	3rd (least favorite)
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

Men's Preference Profile

	1st (favorite)	2nd	3rd (least favorite)
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

Women's Preference Profile

find a stable matching of all men and women

Formulation

Men: $M=\{m_1,\ldots,m_n\}$ Women: $W=\{w_1,\ldots,w_n\}$

The Cartesian Product MxW is the set of all possible ordered pairs.

A matching S is a set of pairs (subset of MxW) such that each m and w occurs in at most one pair

A perfect matching S is a set of pairs (subset of MxW) such that each individual occurs in exactly one pair

How many perfect matchings are there?

Instability

Given a perfect match, eg

$S = \{ (m_1,w_1), \ (m_2,w_2) \}$

But m_1 prefers w_2 and w_2 prefers m_1
(m_1,w_2) is an instability for S

(notice that (m_1,w_2) **is not in S**)

S is a stable matching if:
- S is perfect
- and there is no instability wrt S

Example 1

m_1: w_1, w_2 m_2: w_1, w_2

w_1: m_1, m_2 w_2: m_1, m_2

What are the perfect matchings?

Example 1

m_1: w_1, w_2 m_2: w_1, w_2

w_1: m_1, m_2 w_2: m_1, m_2

1. $\{ (m_1,w_1), (m_2,w_2) \}$
2. $\{ (m_1,w_2), (m_2,w_1) \}$

which is stable/unstable?

Example 1

m_1: w_1, w_2 m_2: w_1, w_2

w_1: m_1, m_2 w_2: m_1, m_2

1. $\{ (m_1,w_1), (m_2,w_2) \}$ stable, WHY?
2. $\{ (m_1,w_2), (m_2,w_1) \}$ instable, WHY?

Example 2

m_1: w_1, w_2 m_2: w_2, w_1

w_1: m_2, m_1 w_2: m_1, m_2

1. $\{ (m_1,w_1), (m_2,w_2) \}$
2. $\{ (m_1,w_2), (m_2,w_1) \}$

which is / are unstable/stable?

Conclusion?

Questions…

- Given a preference list, does a stable matching exist?
- Can we efficiently construct a stable matching if there is one?
- a naive algorithm:

```
for S in the set of all perfect matchings :
    if S is stable :  return S
return None
```

Is this algorithm correct?

What is its running time?

Towards an algorithm

initially: no match

An unmatched man m proposes to the woman w highest on his list.

Will this be part of a stable matching?

Towards an algorithm

initially: no match

An unmatched man m proposes to the woman w highest on his list.

Will this be part of a stable matching?

Not necessarily: w may like some m' better, AND?

So w and m will be in a temporary state of engagement.

w is prepared to change her mind when a man higher on her list proposes

While not everyone is matched…

An unmatched man m proposes to the woman w highest on his list that he hasn't proposed to yet.

If w is free, they become engaged

If w is engaged to m':

    If w prefers m' over m, m stays free

    If w prefers m over m', (m,w) become engaged

The Gayle-Shaply algorithm[1]

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
    Choose such a man m
    w = highest-ranked woman on m's list to whom m has not yet proposed
    if (w is free)
        (m,w) become engaged
    else if (w prefers m to her fiancé m')
        (m,w) become engaged, m′ becomes free
    else
        m remains free
```

A few non-obvious questions:

How long does it take?

Does the algorithm return a stable matching?

Does it even return a perfect matching?

[1]D. Gale and L. S. Shapley: "College Admissions and the Stability of Marriage", American Mathematical Monthly 69, 9-14, 1962.

Observations

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
    Choose such a man m
    w = highest-ranked woman on m's list to whom m has not yet proposed
    if (w is free)
        (m,w) become engaged
    else if (w prefers m to her fiancé m')
        (m,w) become engaged, m′ becomes free
    else
        m remains free
```

Each woman w remains engaged from the first proposal
  and the sequence of w-s partners gets better

Each man proposes to less and less preferred women and will not propose to the same woman twice

Observations

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
    Choose such a man m
    w = highest-ranked woman on m's list to whom m has not yet proposed
    if (w is free)
        (m,w) become engaged
    else if (w prefers m to her fiancé m')
        (m,w) become engaged, m′ becomes free
    else
        m remains free
```

Claim. The algorithm terminates after at most n^2 iterations of the while loop.

Observations

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
    Choose such a man m
    w = highest-ranked woman on m's list to whom m has not yet proposed
    if (w is free)
        (m,w) become engaged
    else if (w prefers m to her fiancé m')
        (m,w) become engaged, m′ becomes free
    else
        m remains free
```

Claim. The algorithm terminates after at most n^2 iterations of the while loop.

At each iteration a man proposes (only once) to a woman he has never proposed to, and there are only n^2 possible pairs (m,w)

WHY ONLY n^2?

only n choices for each of the n men

Observations

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
    Choose such a man m
    w = highest-ranked woman on m's list to whom m has not yet proposed
    if (w is free)
        (m,w) become engaged
    else if (w prefers m to her fiancé m')
        (m,w) become engaged, m′ becomes free
    else
        m remains free
```

When the loop terminates, the matching is **perfect**

Proof: **By contradiction.** **Assume there is a free man, m.**

Because the loop terminates, m proposed to all women

But then all women are engaged, hence there is no free man

→**Contradiction**

Proof of Correctness: Stability

Claim. No unstable pairs. **Proof.** **(by contradiction)**

- Suppose (m, w) is an unstable pair: each prefers each other to partner in Gale-Shapley matching S*.

men propose in decreasing order of preference

- Case 1: m never proposed to w.
  $\Rightarrow$ m prefers his GS partner w′ to w
  $\Rightarrow$ (m, w) is not unstable.

S*
m, w′
m′, w
. . .

- Case 2: m proposed to w.
  $\Rightarrow$ w rejected m (right away or later) $\leftarrow$ women only trade up
  $\Rightarrow$ w prefers her GS partner m′ to m.
  $\Rightarrow$ (m, w) is not unstable.

- In either case (m, w) is not unstable, a contradiction. ∎

Summary

Stable matching problem. Given n men and n women and their preferences, find a stable matching if one exists.

Gale-Shapley algorithm. Guaranteed to find a stable matching for any problem instance.

Q. If there are multiple stable matchings, which one does GS find?

Which solution?

m_1: w_1, w_2 m_2: w_2, w_1

w_1: m_2, m_1 w_2: m_1, m_2

Two stable solutions

1: { (m_1,w_1), (m_2,w_2) }

2: { (m_1,w_2), (m_2,w_1) }

GS will always find one of them (which?)

When will the other be found?

Symmetry

The stable matching problem is symmetric wrt to men and women, but the GS algorithm is asymmetric.

There is a certain unfairness in the algorithm:
If all men list different women as their first choice, they will end up with their first choice, regardless of the women's preferences.

Non-determinism

Notice the following line in the GS algorithm:

```
while (some man is free and
         hasn't proposed to every woman
       )
     Choose such a man m
```

The algorithm does not specify WHICH

Still, it can be shown that all executions of the algorithm find the same stable matching.

This ends our discussion of stable matching.

Representative Problems

Remember the problem solving paradigm

1. Formulate it with precision (usually using mathematical concepts, such as sets, relations, and graphs, costs, benefits, optimization criteria)

2. Design an algorithm

3. Prove its correctness, e.g. in terms of pre and post conditions

4. Analyze its complexity

5. Implement respecting the derived complexity

Often, steps 2-5 are repeated, to improve efficiency

Interval Scheduling

You have a resource (hotel room, printer, lecture room, telescope, manufacturing facility...)

There are requests to use the resource in the form of start time **s_i** and finish time **f_i**, such that $s_i<f_i$

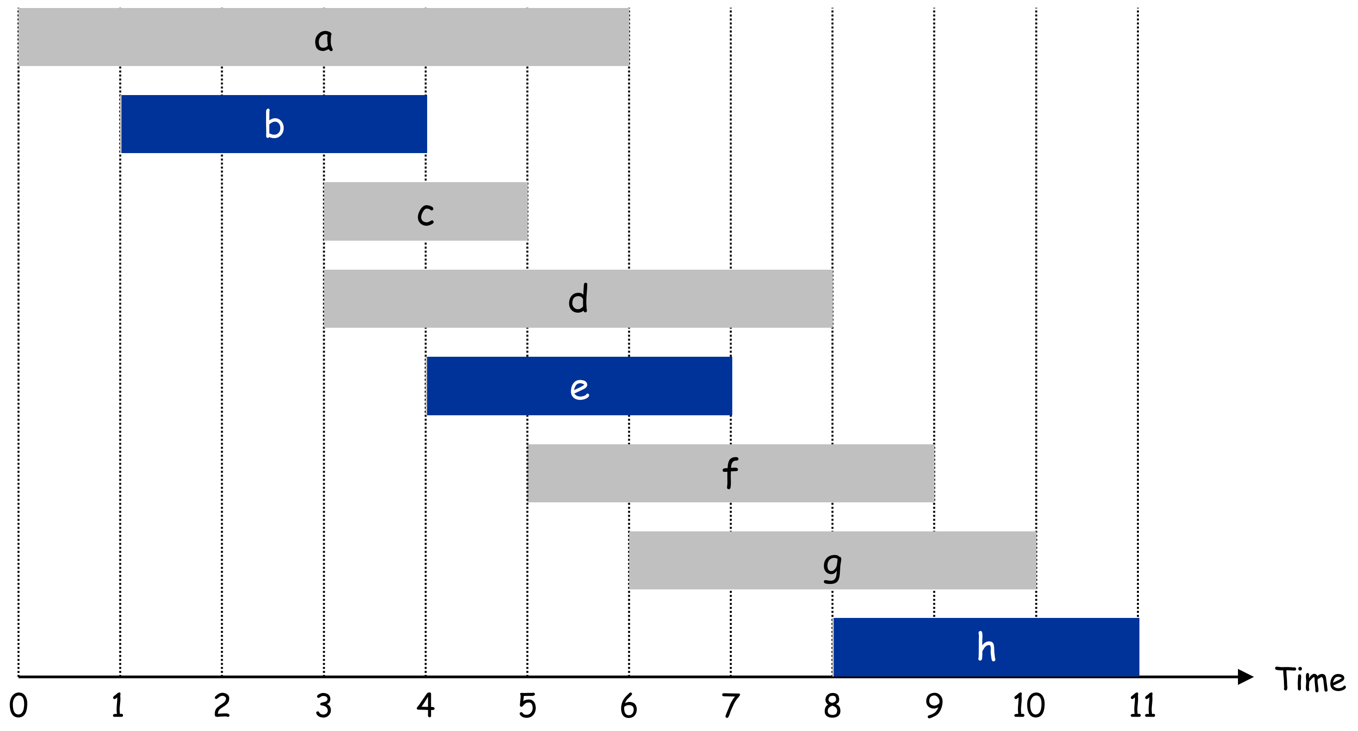
Objective: grant as many requests as possible.

Two requests i and j are **compatible** if they don't overlap, i.e.

$$f_i<=s_j \text{ or } f_j<=s_i$$

Interval Scheduling

Input. Set of jobs with start times and finish times.
Goal. Find maximum cardinality subset of compatible jobs.



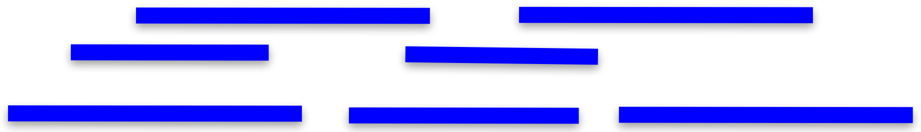
What happens if you pick the first starting (a)?, the smallest (c)? What is the optimum?

Algorithmic Approach

The interval scheduling problem is amenable to a very simple solution.

Now that you know this, can you think of it?

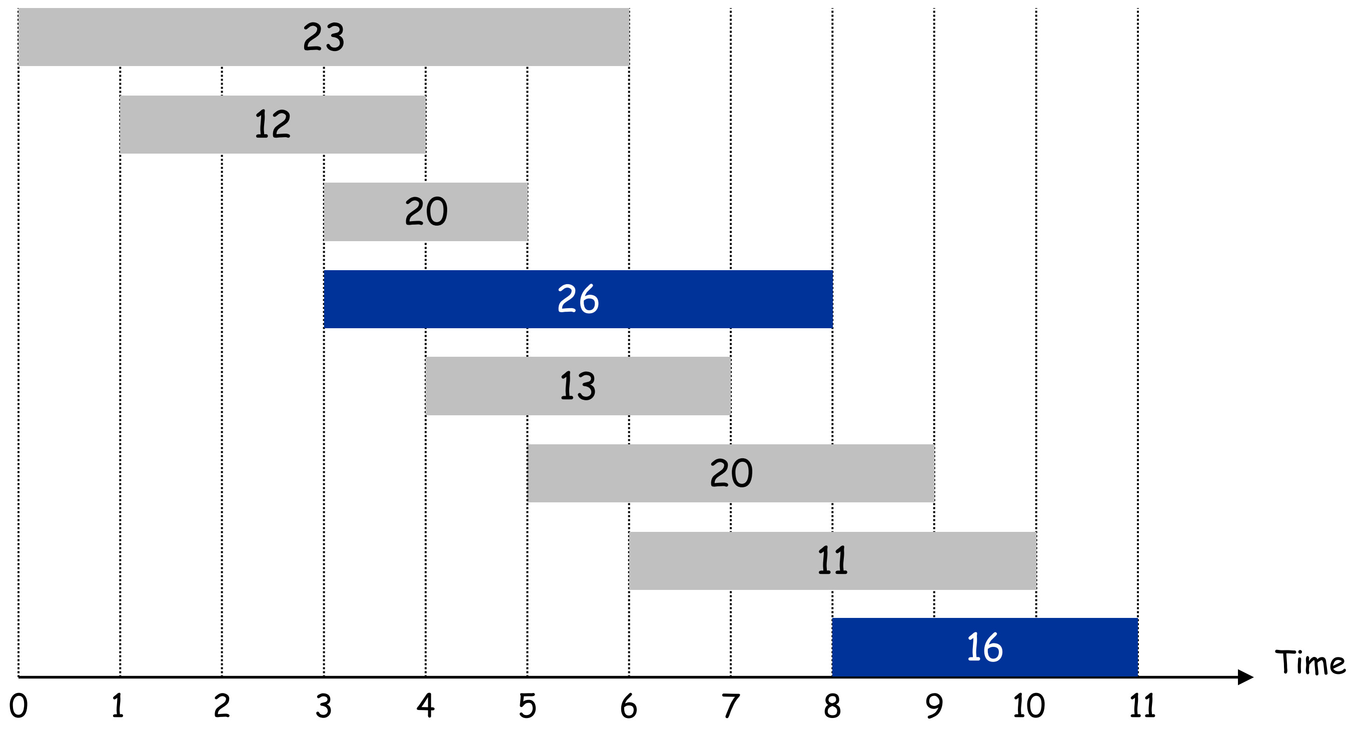
Hint: Think how to pick a first interval while preserving the longest possible free time...



Weighted Interval Scheduling

Input. Set of jobs with start times, finish times, and weights.

Goal. Find maximum weight subset of compatible jobs.



Bipartite Matching

Stable matching was defined as matching elements of two disjoint sets.

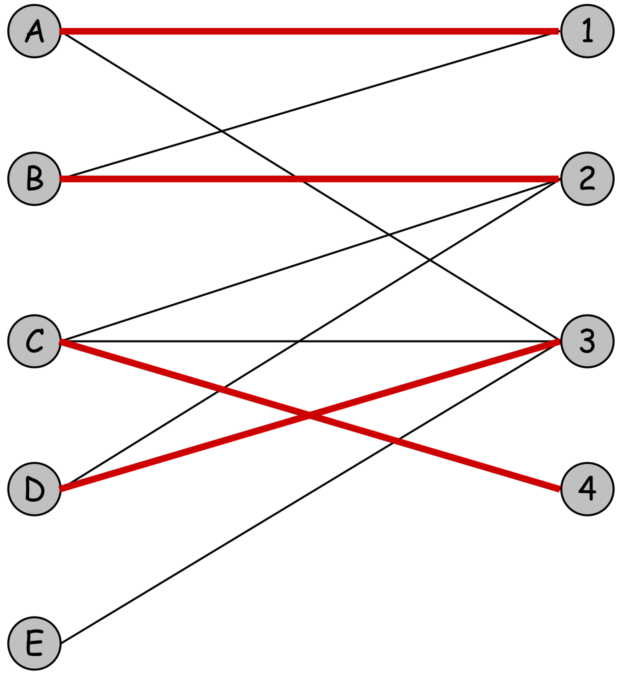
We can express this in terms of graphs.

A graph is **bipartite** if its nodes can be partitioned in two sets X and Y, such that the edges go from an x in X to a y in Y

Bipartite Matching

Input. Bipartite graph.
Goal. Find maximum cardinality matching.

Matching in bipartite graphs can model assignment problems, e.g., assigning jobs to machines, where an edge between a job j and a machine m indicates that m can do job j, or professors and courses.



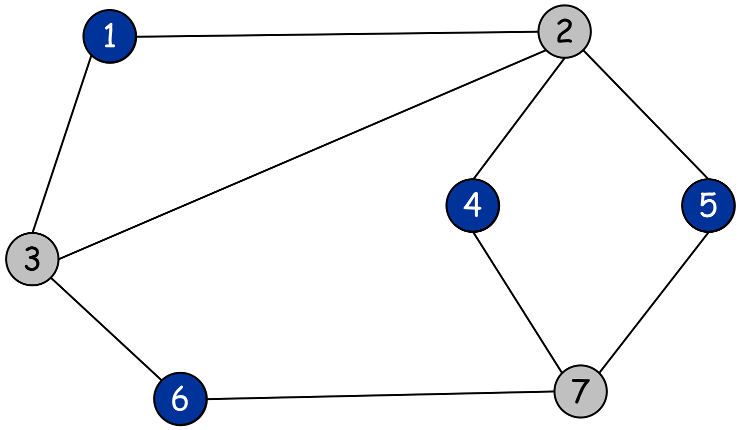
How is this different from the stable matching problem?

Independent Set

Input. Graph.

Goal. Find maximum cardinality independent set:

subset of nodes such that no two are joined by an edge



Can you formulate interval scheduling as an independent set problem? If so, how could you solve the interval scheduling problem?

Independent set problem

- There is no known efficient way to solve the independent set problem.
- But we just said: we can formulate interval scheduling as independent set problem..... ???
- What does "no efficient way" mean?
- The only solution we have so far is trying all sub sets and finding the largest independent one.
- How many sub sets of a set of n nodes are there?

Representative Problems / Complexities

Looking ahead...

- Interval scheduling: n log(n) greedy algorithm.
- Weighted interval scheduling: n log(n) dynamic programming algorithm.
- Bipartite matching: polynomial max-flow based algorithm.
- Independent set: NP (no known polynomial algorithm exists).

Algorithm

Algorithm: effective procedure

- mapping input to output

effective: unambiguous, executable

- Turing defined it as: "like a Turing machine"

- program = effective procedure

Is there an algorithm for every possible problem?

Algorithm

Algorithm: effective procedure

- mapping input to output

effective: unambiguous, executable

- Turing defined it as: "like a Turing machine"
- program = effective procedure

Is there an algorithm for every possible problem?

No, the problem must be effectively specified: "how many angels can dance on the head of a pin?" not effective. **Even if** it is effectively specified, there is not always an algorithm to provide an answer. This occurs often for programs analyzing programs (examples?)

Ulam's problem

```
def f(n) :
        if (n==1) return 1
        elif (odd(n)) return f(3*n+1)
        else return f(n/2)
```

Ulam's problem

```
def f(n) :
        if (n==1) return 1
        elif (odd(n)) return f(3*n+1)
        else return f(n/2)
```

Steps in running f(n) for a few values of n:

```
1
2, 1,
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
10, 5, 16, 8, 4, 2, 1
```

Does f(n) always stop?

Ulam's problem

```
def f(n) :
        if (n==1) return 1
        elif (odd(n)) return f(3*n+1)
        else return f(n/2)
```

Nobody has found an n for which f does not stop

Nobody has found a proof (so there can be no algorithm deciding this.)

A generalization of this problem has been proven to be undecidable.

A problem P is undecidable, if there is no algorithm that produces P(x) for every possible input x

The Halting Problem is undecidable

Given a program P and input x
will P stop on x?

We can prove (cs420):
the halting problem is undecidable

i.e. there is no algorithm Halt(P,x) that for any program P and input x decides whether P stops on x.

Verification/equivalence undecidable

Given any specification S and any program P there is no algorithm that decides whether P executes according to S

Given any two programs P1 and P2, there is no algorithm that decides $\forall x: P1(x)=P2(x)$

Does this mean we should not build program verifiers?

Intractability

Suppose we have a program,

- does it execute a in a reasonable time?
- E.g., towers of Hanoi (cs200).

Three pegs, one with n smaller and smaller disks, move (1 disk at the time) to another peg without ever placing a larger disk on a smaller

f(n) = # moves for tower of size n

Monk: before a tower of Hanoi of size 100 is moved, the world will have vanished

hanoi

```
# pegs are numbers, via is computed
# empty base case
def hanoi(n, from, to):
  if (n>0) :
    via = 6 - from - to
    hanoi(n-1,from, via)
    print "move disk", n,  " from", from, " to ",  to
    hanoi(n-1,via,to);
```

f(n): #moves in hanoi

$f(n) = 2f(n-1) + 1$, $f(1)=1$

$f(1) = 1$, $f(2) = 3$, $f(3) = 7$, $f(4) = 15$

$f(n) = 2^n-1$

How can you show that?

Can you write an iterative Hanoi algorithm?

Was the monk right?

2^{100} moves, say 1 per second.....

How many years?

Is there a better algorithm?

Is there a better algorithm?

Pile(n-1) must be

off peg1

and

on one other peg

before disk n can be moved to its destination

so (inductively) all moves are necessary

Algorithm complexity

Measures in units of **time** and **space**

Linear Search X in dictionary D

```
i=1
while not at end and X!= D[i]:
  i=i+1
```

We don't know if X is in D, and we don't know where it is, so we can only give **worst** or **average** time bounds

We don't know the time for atomic actions, so we only determine **Orders of Magnitude**

Linear Search: time and space complexity

Space: n locations in D plus some local variables

Time:

In **the worst case** we search all of D, so the loop body is executed n times

In ***average case*** analysis we compute the ***expected*** number of steps: i.e., we sum the products of the probability of each option and the time cost of that option. In the average case the loop body is executed about n/2 times

$$\sum_{i=1}^{n} 1/n * i = 1/n \sum_{i=1}^{n} i = (n(n+1)/2)/n \approx n/2$$