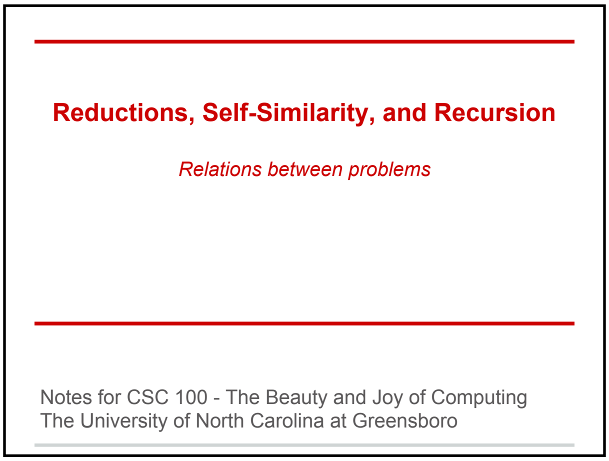


Reductions, Self-Similarity, and Recursion

Relations between problems

Notes for CSC 100 - The Beauty and Joy of Computing
The University of North Carolina at Greensboro

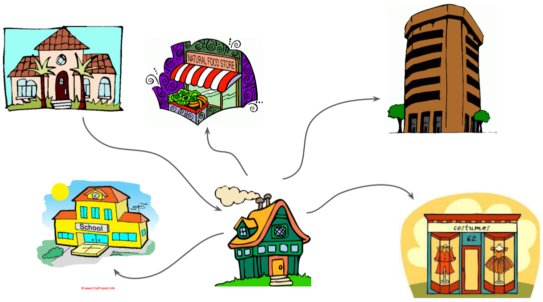
Getting to places from my house...

NATURAL FOOD STORE
School
costumes
62

Now I buy a new house!

NATURAL FOOD STORE
?
School
costumes
62

Get anywhere by first going to old house



Things to notice...

What I want to do...

I can *go anywhere from my new house* by

1. Going to my old house
2. Going to my destination from there

What I know how to do...

Things to notice...

What I want to do...

I can *go anywhere from my new house* by

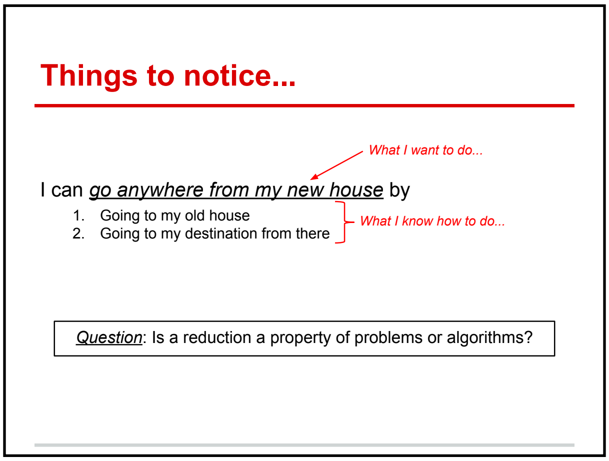
1. Going to my old house
2. Going to my destination from there

What I know how to do...

Terminology: I have *reduced the problem of traveling from my new house to the problem of traveling from my old house*.

Important points:

- Solution is easy to produce (often easier than direct solution)
- Solution is easy and compact to describe
- Solution may *not* be the most efficient to execute



Things to notice...

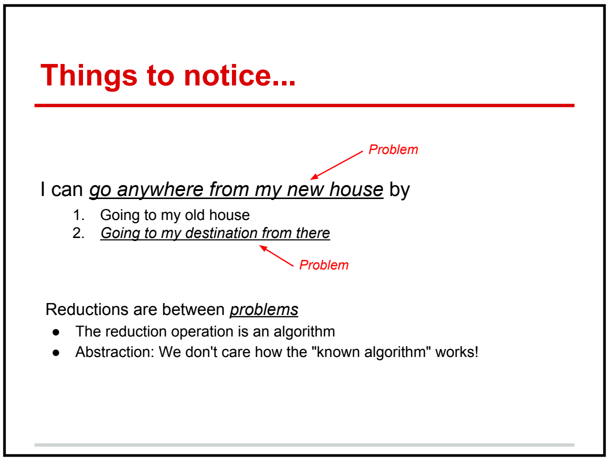
What I want to do...

I can *go anywhere from my new house* by

1. Going to my old house
2. Going to my destination from there

What I know how to do...

Question: Is a reduction a property of problems or algorithms?



Things to notice...

Problem

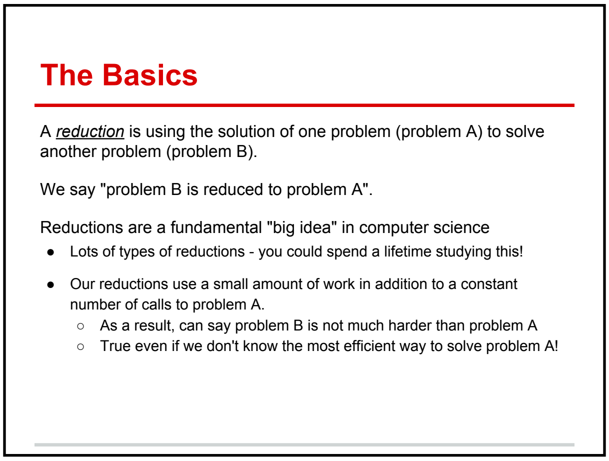
I can *go anywhere from my new house* by

1. Going to my old house
2. *Going to my destination from there*

Problem

Reductions are between *problems*

- The reduction operation is an algorithm
- Abstraction: We don't care how the "known algorithm" works!



The Basics

A *reduction* is using the solution of one problem (problem A) to solve another problem (problem B).

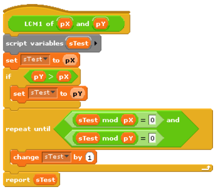
We say "problem B is reduced to problem A".

Reductions are a fundamental "big idea" in computer science

- Lots of types of reductions - you could spend a lifetime studying this!
- Our reductions use a small amount of work in addition to a constant number of calls to problem A.
  - As a result, can say problem B is not much harder than problem A
  - True even if we don't know the most efficient way to solve problem A!

An example from Lab 4

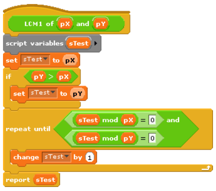
To find least common multiple (LCM):



```
LCM1 of (pX) and (pY)
script variables (sTest)
set [sTest] to (pX)
if <(pY) > (pX)>
    set [sTest] to (pY)
repeat until <<((sTest) mod (pX)) = [0]> and <((sTest) mod (pY)) = [0]>>
    change [sTest] by (1)
report (sTest)
```

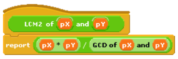
An example from Lab 4

To find least common multiple (LCM):



```
LCM1 of (pX) and (pY)
script variables (sTest)
set [sTest] to (pX)
if <(pY) > (pX)>
    set [sTest] to (pY)
repeat until <<((sTest) mod (pX)) = [0]> and <((sTest) mod (pY)) = [0]>>
    change [sTest] by (1)
report (sTest)
```

But if you already have GCD

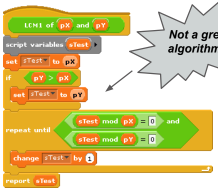


```
LCM2 of (pX) and (pY)
report (((pX) * (pY)) / (GCD of (pX) and (pY)))
```

What have we done? We have *reduced the problem of computing LCM to the problem of computing GCD*.

An example from Lab 4

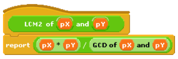
To find least common multiple (LCM):



```
LCM1 of (pX) and (pY)
script variables (sTest)
set [sTest] to (pX)
if <(pY) > (pX)>
    set [sTest] to (pY)
repeat until <<((sTest) mod (pX)) = [0]> and <((sTest) mod (pY)) = [0]>>
    change [sTest] by (1)
report (sTest)
```

Not a great algorithm...

But if you already have GCD



```
LCM2 of (pX) and (pY)
report (((pX) * (pY)) / (GCD of (pX) and (pY)))
```

What have we done? We have *reduced the problem of computing LCM to the problem of computing GCD*.

So: LCM is no harder computationally than GCD. And remember... Euler's algorithm is a very efficient GCD algorithm!

Similarity and Self-Similarity

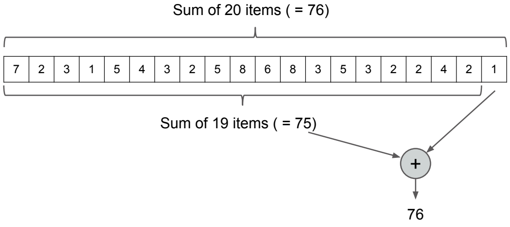
Reducing LCM to GCD identifies similarities between the two problems.

Many problems are structured so that solutions are "self-similar" - large solutions contain solutions to smaller versions of the same problem!

Example: Recall sum of list items as parallel algorithm - each thread solved a smaller version of the same problem!

An algorithm can solve a large problem by breaking it down to smaller versions of the same problem - this is called *recursion*.

Example: Adding up a list



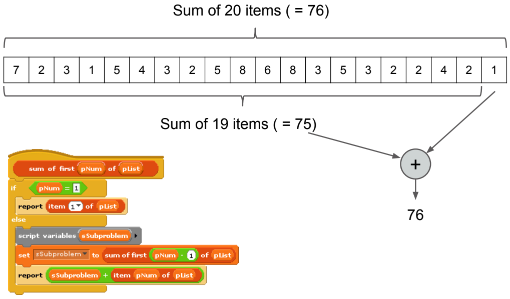
Sum of 20 items (= 76)

7	2	3	1	5	4	3	2	5	8	6	8	3	5	3	2	2	4	2	1

Sum of 19 items (= 75)

76

Example: Adding up a list



Sum of 20 items (= 76)

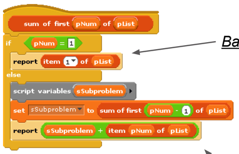
7	2	3	1	5	4	3	2	5	8	6	8	3	5	3	2	2	4	2	1

Sum of 19 items (= 75)

76

```
sum of first pNum of pList
if pNum = 1
  report item 1 of pList
else
  script variables sSubproblem
  set sSubproblem to sum of first pNum - 1 of pList
  report sSubproblem + item pNum of pList
```

Breaking it down



```
sum of first (pNum) of (pList)
if <(pNum) = [1]>
  report (item (1) of (pList))
else
  script variables (sSubproblem)
  set [sSubproblem] to (sum of first ((pNum) - (1)) of (pList))
  report ((sSubproblem) + (item (pNum) of (pList)))
```

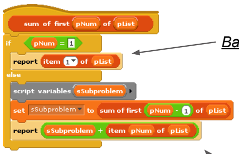
Base case: Handling smallest case directly

Recursive case: Solving a smaller version of the same problem.

Constant amount of work to use answer from subproblem to compute answer to overall problem.

Breaking it down

Workhorse Function



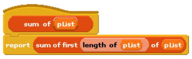
```
sum of first (pNum) of (pList)
if <(pNum) = [1]>
  report (item (1) of (pList))
else
  script variables (sSubproblem)
  set [sSubproblem] to (sum of first ((pNum) - (1)) of (pList))
  report ((sSubproblem) + (item (pNum) of (pList)))
```

Base case: Handling smallest case directly

Recursive case: Solving a smaller version of the same problem.

Constant amount of work to use answer from subproblem to compute answer to overall problem.

Driver Function

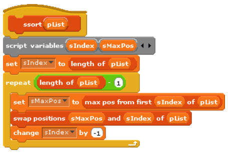


```
sum of (pList)
report (sum of first (length of (pList)) of (pList))
```

Driver function: sets up first call to recursion

Another example: Sorting

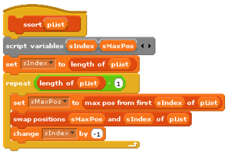
"Selection sort" from algorithms lab:



```
ssort (pList)
script variables (sIndex) (sMaxPos)
set [sIndex] to (length of (pList))
repeat ((length of (pList)) - (1))
  set [sMaxPos] to (max pos from first (sIndex) of (pList))
  swap positions (sMaxPos) and (sIndex) of (pList)
  change [sIndex] by (-1)
```

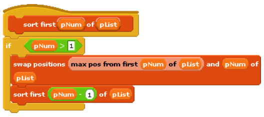
Another example: Sorting

"Selection sort" from algorithms lab:



```
ssort (pList)
script variables (sIndex) (sMaxPos)
set [sIndex] to (length of (pList))
repeat ((length of (pList)) - (1))
    set [sMaxPos] to (max pos from first (sIndex) of (pList))
    swap positions (sMaxPos) and (sIndex) of (pList)
    change [sIndex] by (-1)
```

Recursive version:



```
sort first (pNum) of (pList)
if <(pNum) > [1]>
    swap positions (max pos from first (pNum) of (pList)) and (pNum) of (pList)
    sort first ((pNum) - (1)) of (pList)
```

Base case: One item - nothing to do!

Setting up recursion: Swap max item to last position

Recursion: Sort all the rest

Summary

Finding relations between problems can simplify solutions:

- Sometimes relations between different problems (reductions)
- Sometimes relation to smaller version of the same problem (recursion)

What you should know:

- Recognize reductions and recursion
- Understand the basic principles

We will explore this more in a lab!