Lecture 12

**DEBUGGING TECHNIQUES**

Some of the common bugs:

* reversed order of parameters
* spelling
* initialization
* object vs value equality – when we use ‘==’
* aliasing – two different ways to get to the same object – deep vs shallow copy(like a list)
* side effects – modifying a parameter when it is not supposed to

Keep record of what you had tried.

Reconsider your assumptions.

When you debug code written by someone else, don’t believe their comments. Read it, but don’t believe it.

If it gets really really tough then

* get help from someone – it is just a fresh set of eyes which can see what we may be missing because it is too obvious.
* walk away and come back and look at it with your own fresh eyes.

Before fixing a bug, after you have found out a bug:

* Haste makes waste – think about the fix
* what are the ramifications of the fix? will it break other things?
* does it allow you to tidy up other things – important
  + code should not always grow – more code you have harder it is to get it right
* **MAKE SURE THAT YOU CAN REVERT back to where you were**
  + **disk space is cheap, don’t hesitate to save the old versions.**

**ALGORITHMS:**

Main aim of 6.00 is: To take a problem, analyze it and get an answer by computing it.

For the next few lectures, we will be looking at optimization problems

It has two parts:

* some function to be maximized or minimized
* some constraints to be honoured

classes of optimization problems:

* shortest path problems
* travelling sales person(TSP) – travelling from one city to other and given the cost what is the least cost you can find for a round trip
* bin packing – how to load up container ships and vans
* sequence alignment problems – biology
* knapsack (bag pack) problem – you have more things than you can fit in the knapsack. decide what to take and what not to take.

Problem reduction : given a problem that you have never seen before. ask yourself if any other people have solved it. we take a new problem and map it into an old problem. then we have a solution.

Most optimization problems don’t have a fast solution.

**CONTINUOUS KNAPSACK PROBLEM:**

Assume that you are a burglar. You have broken into a house and now you are deciding what to take. you can carry 8 pounds of stuff. in a continuous world, you see,

* 4 pounds of gold dust
* 3 pounds of silver
* 10 pounds of raisins

function to be maximized:

cg \* pg + cs \* ps + cr \* pr = function to be maximized.

and the constraints are:

pg+ps+pr<=8

we simply stuff the gold as long as it is there. then silver as long as it is there. and then pour in raisins as long as it fits into.

This is the instance of GREEDY ALGORITHM. at every step you do what maximizes your value at that step. There is no planning ahead. in this case the greedy algorithm gives us the best output. like in the computer player implementation in the word game.

greedy algorithm may not always work. locally optimal decisions do not always lead to global optimum.

in the continous problem, greedy is good.

**0/1 KNAPSACK PROBLEM:**

greedy is not so good.

this is a discrete version of the continuous problem.

there are gold bricks not gold dust. so you either take it or not.

the burglar’s knapsack can hold only 8 pounds.

as given in the handout.

**Greedy thief:**

2 watches and 1 vase.

he could do better though if he took 2 watches and 2 radios. but he is using the greedy algorithm. they may not give the right answer each time but they are very good to use. they are easy to implement and fast to run as a program. but that’s not good enough.

**Slow thief:**

What I will do is I will put stuff in the backpack and compute its value and empty it and again do it on and on till all the possible combinations are done. then I will know which combination was the best and I will take that combination. this is the brute force or exhaustive enumeration algorithm.

but he got caught. inside jail he tries to figure out what is wrong:

the function that he was trying to maximize was:

Σ from 1 to n maximize p(i) \* x(i) where p(i) is the price of item I and x(i) is a list of 0’s and 1’s having n entries. If I took it then I will put x(i) as 1 and if we leave it we take x(i) as 0. all this to subject to the constraint:

Σ from 1 to n w(i) x(i) <= c where c is the maximum weight I can carry.

find the vector x such that the constraint is obeyed and the constraint is obeyed.

we can solve this problem by generating all possible values. 2n values are there.

we have 8 items, combinations are: 28

Brute force algorithm is exponential growth in the number of items that are there.

This shows that exponential algorithms are typically not useful.

n does not have to get very big for exponential algorithms to fail.

**Smart thief:**

Dynamic programming. don’t try to figure out why it is called dynamic programming. it is just a name.

key things are:

* we are looking for a situation where there are over-lapping sub problems
* and optimal sub-structure.

In the recursive implementation of Fibonacci:

we are calculating fib(4) many times again and again.

this is over-lapping sub-problems, fb(4) and fib(3) overlap with each other.