Transform and Conquer

Transform and Conquer

- Algorithms based on the idea of transformation
  - Transformation stage
    - Problem instance is modified to be more amenable to solution
  - Conquering stage
    - Transformed problem is solved
- Major variations are for the transform to perform:
  - Instance simplification
  - Different representation
  - Problem reduction

Presorting

- Presorting is an old idea, you sort the data and that allows you to more easily compute some answer
  - Saw this with quickhull, closest point
- Some other simple presorting examples
  - Element Uniqueness
  - Computing the mode of *n* numbers

Element Uniqueness

- Given a list *A* of *n* orderable elements, determine if there are any duplicates of any element

Brute Force:

```
for each x ∈ A
  for each y ∈ {A – x}
      if x = y return not unique
return unique
```

Presorting:

```
Sort A
for i ← 1 to n-1
   if A[i] = A[i+1] return not unique
return unique
```

Runtime?

Computing a mode

- A mode is a value that occurs most often in a list of numbers
  - e.g. the mode of [5, 1, 5, 7, 6, 5, 7] is 5
  - If several different values occur most often any of them can be considered the mode
- "Count Sort" approach: (assumes all values > 0; what if they aren't?)

```
max ← max(A)
freq[1..max] ← 0
for each x ∈ A
   freq[x] +=1
mode ← freq[1]
for i ← 2 to max
   if freq[i] > freq[mode]  mode ← i
return mode
```

Runtime?

Presort Computing Mode

```
Sort A
i ← 0
modefrequency ← 0
while i ≤ n-1
   runlength ← 1;   runvalue ← A[i]
   while i+runlength ≤ n-1  and A[i+runlength] = runvalue
         runlength += 1
   if runlength > modefrequency
         modefrequency ← runlength
         modevalue ← runvalue
   i+= runlength

return modevalue
```

Gaussian Elimination

- This is an example of transform and conquer through representation change
- Consider a system of two linear equations:

  $A_{11}x + A_{12}y = B_1$

  $A_{21}x + A_{22}y = B_2$
- To solve this we can rewrite the first equation to solve for x:

  $x = (B_1 – A_{12}y) / A_{11}$
- And then substitute in the second equation to solve for y. After we solve for y we can then solve for x:

  $A_{21}(B_1 – A_{12}y) / A_{11} + A_{22}y = B_2$

Gaussian Elimination

- In many applications we need to solve a system of n equations with n unknowns, e.g.:

  $A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$

  $A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$

  …

  $A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n$
- If n is a large number it is very cumbersome to solve these equations using the substitution method.
- Fortunately there is a more elegant algorithm to solve such systems of linear equations: Gaussian elimination
  - Named after Carl Gauss

Gaussian Elimination

The idea is to transform the system of linear equations into an equivalent one that eliminates coefficients so we end up with a triangular matrix.

$$\begin{array}{l} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1 \\ A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2 \\ \ldots \\ A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n \end{array} \quad \Rightarrow \quad \begin{array}{l} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1 \\ 0x_1 \quad + A_{22}x_2 + \ldots + A_{2n}x_n = B_2 \\ \ldots \\ 0x_1 \quad + 0x_2 \quad + \ldots + A_{nn}x_n = B_n \end{array}$$

In matrix form we can write this as: $Ax = B \rightarrow A'x = B'$

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & & & \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{bmatrix} \quad B = \begin{bmatrix} B_1 \\ B_2 \\ \ldots \\ B_n \end{bmatrix}$$

Gaussian Elimination

- Why transform?

$$\begin{array}{l} A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1 \\ 0x_1 \quad + A_{22}x_2 + \ldots + A_{2n}x_n = B_2 \\ \ldots \\ 0x_1 \quad + 0x_2 \quad + \ldots + A_{nn}x_n = B_n \end{array}$$

  - The matrix with zeros in the lower triangle (it is called an upper triangular matrix) is easier to solve.

    We can solve the last equation first, substitute into the second to last, etc. working our way back to the first one.

Gaussian Elimination Example

- Solve the following system:

$$2x_1 - x_2 + x_3 = 1$$
$$4x_1 + x_2 - x_3 = 5$$
$$x_1 + x_2 + x_3 = 0$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

subtract 2*row1

subtract ½*row1

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

subtract ½*row2

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Gaussian Elimination

- In our example we replaced an equation with a sum or difference with a multiple of another equation
- We might also need to:
  - Exchange two equations
  - Replace an equation with its nonzero multiple
- Pseudocode:

```
for i ← 1 to n do A[i,n+1] ← B[i]
for i ← 1 to n – 1
  for j ← i+1 to n do
    for k ← i to n+1 do
      A[j,k] ← A[j,k] – A[i,k]*A[j,i] / A[i,i]
```

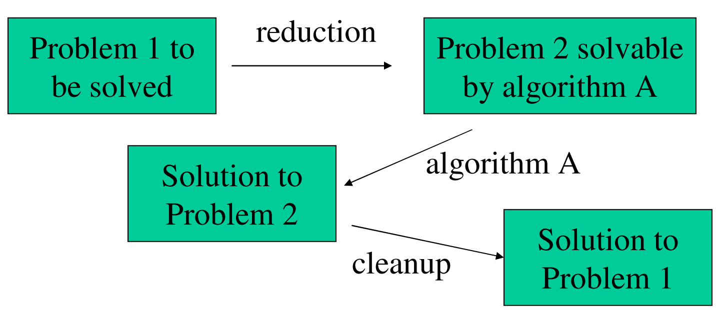
$O(n^3)$ algorithm

Textbook Chapter 6

- Skipping other matrix operations, balanced trees, heaps, binary exponentiation

Problem Reduction

- If you need to solve a problem, reduce it to another problem that you know how to solve; we saw this idea already with NPC problems



Linear Programming

- One more example of problem reduction; linear programming
- A Linear Program (LP) is a problem that can be expressed as follows (the so-called Standard Form):
  - minimize (or maximize) cx
  - subject to
    - Ax = b
    - x >= 0
- where x is the vector of variables to be solved for, A is a matrix of known coefficients, and c and b are vectors of known coefficients. The expression "cx" is called the objective function, and the equations "Ax=b" are called the constraints.

Linear Programming Example

- Wyndor Glass produces glass windows and doors
- They have 3 plants:
  - Plant 1: makes aluminum frames and hardware
  - Plant 2: makes wood frames
  - Plant 3: produces glass and makes assembly
- Two products proposed:
  - Product 1: 8’ glass door with aluminum siding (x1)
  - Product 2: 4’ x 6’ wood framed glass window (x2)
- Some production capacity in the three plants is available to produce a combination of the two products
- Problem is to determine the best product mix to maximize profits

Wyndor Glass Co. Data

	Production time per batch (hr)		Production time available per week (hr)
	Product		
Plant	1	2	
1	1	0	4
2	0	2	12
3	3	2	18
Profit per batch	**$3,000**	**$5,000**	

Formulation:

Maximize $z = 3x_1 + 5x_2$ (objective to maximize $$)

Subject to

$x_1 <= 4$ (Plant One)

$2x_2 <= 12$ (Plant Two)

$3x_1 + 2x_2 <= 18$ (Plant Three)

$x_1, x_2 >= 0$ (Non-negativity requirements)

Example 2: Spacing to Center Text

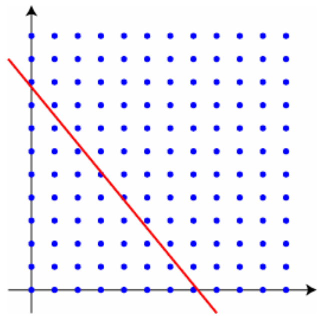
- To center text we need to indent it ourselves by using an appropriate number of space characters. The complication is that we have two types of spaces: the usual space and option-space (also known as non-breaking space). These two spaces are different widths.

- Given three numbers
  - a (the width of a normal space),
  - b (the width of an option-space), and
  - c (the amount we want to indent),
- Find two more numbers
  - x (the number of normal spaces to use), and
  - y (the number of option-spaces to use),

- So that ax+by is as close as possible to c.

Spacing Problem

- Visualize problem in 2 dimensions, say a=11, b=9, and c=79. Each blue dot in the picture represents the combination of x option-spaces and y spaces. The red line represents the ideal width of 79 pixels.
- We want to find a blue dot that's as close as possible to the red line.



Spacing Problem

- If we want to find the closest point below the line then our equations become:

  $x \geq 0$

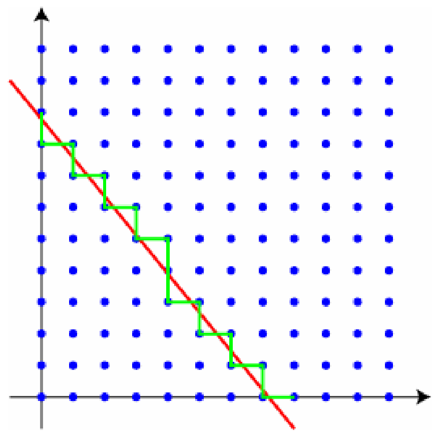
  $y \geq 0$

  $ax + by \leq c$

- The linear programming problem is to maximize $ax + by \leq c$ to find the closest point to the line

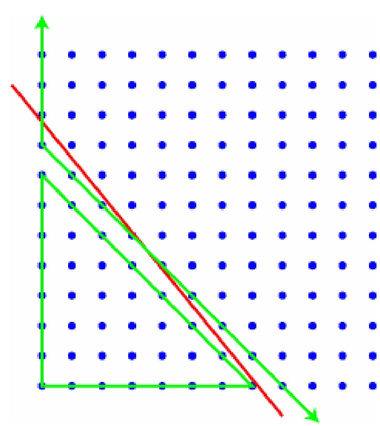
Possible Solutions

- Brute Force



Possible Solutions

- Simplex method
  - Consider only points along boundary of “feasible region”
  - Won’t go into the algorithm here, but it finds solutions in worst case exponential time but generally runs efficiently in polynomial time



Knapsack Problem

- We can reduce the knapsack problem to a solvable linear programming problem
- Discrete or 0-1 knapsack problem:
  - Knapsack of capacity W
  - n items of weights $w_1, w_2 \ldots w_n$ and values $v_1, v_2 \ldots v_n$
  - Can only take entire item or leave it
- Reduces to:

Maximize $\sum_{i=1}^{n} v_i x_i$ where $x_i = 0$ or 1

Constrained by: $\left(\sum_{i=1}^{n} w_i x_i \right) \leq W$