Objectives

- Divide and conquer algorithms
  - Counting inversions
  - Closest pair of points

Review

- Describe the template for divide and conquer solutions
- How can you compute D&C running times?
  - Describe first step
  - 2 ways to solve
- What are you looking for when unrolling the recurrence?

Review: Divide-and-Conquer

Divide et impera.
Veni, vidi, vici.
- *Julius Caesar*

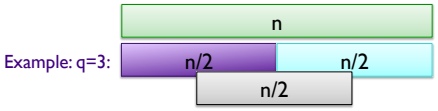
- Divide-and-conquer process
  - Break up problem into several parts
  - Solve each part recursively
  - Combine solutions to sub-problems into overall solution
- Define a **recurrence relation** that describes the running time

Review: Recurrence Relations

- Use recurrences to analyze/determine the run time of divide and conquer algorithms
  - Number of sub problems
  - Size of sub problems
  - Number of times divided (number of levels)
  - Cost of merging problems
- How to solve
  - Unrolling
  - Substitution

Review Example: q>2

- Recurrence relation:
  - For some constant *c*,
    $T(n) \leq q\ T(n/2) + cn$ when $n > 2$
    $T(2) \leq c$



Run time: $O(n^{\log_2 q})$

COUNTING INVERSIONS

Problem Context

- Movie site tries to match your movie preferences with others
  - You rank *n* movies
  - Movie site consults database to find people with similar tastes
    - **Collaborative filtering**
- Meta-search tools
  - Do same query on several search engines
  - Synthesize results by looking for similarities and differences in search engines' results rankings

Comparing Rankings

- To determine similarity of rankings, need a metric
- Similarity metric: number of inversions between two rankings
  - My rank: 1, 2, …, n
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Movies i and j *inverted* if i < j but $a_i > a_j$

Discuss pros and cons of this metric

Movies

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

What are the inversions?

Comparing Rankings

- To determine similarity of rankings, need a metric
- Similarity metric: number of inversions between two rankings
  - My rank: 1, 2, …, n
  - Your rank: $a_1, a_2, \ldots, a_n$
  - Movies i and j *inverted* if i < j but $a_i > a_j$

Naïve/Brute force solution?

Movies

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions:
3-2, 4-2

Brute Force Solution

- Look at every pair (i,j) and determine if they are an inversion
- Requires $\Theta(n^2)$ time
  - Note: Already an efficient algorithm but trying to improve upon runtime

Towards a Better Solution…
• Can't look at each inversion individually

Applications

- Voting theory
- Collaborative filtering
- Measuring the "sortedness" of an array
- Sensitivity analysis of Google's ranking function
- Rank aggregation for meta-searching on the Web
- Nonparametric statistics (e.g., Kendall's Tau distance)

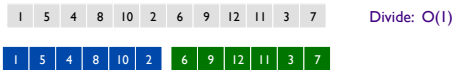
Counting Inversions: Divide-and-Conquer

Assume number represents where item should be in the list

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

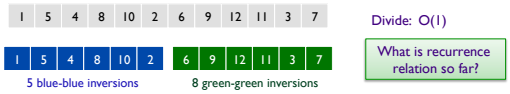
- Divide: separate list into two pieces



What are the inversions in blue and green halves?

Counting Inversions: Divide-and-Conquer

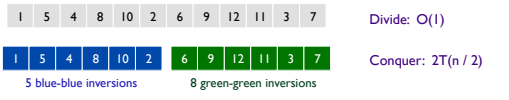
- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half



What is recurrence relation so far?

Counting Inversions: Divide-and-Conquer

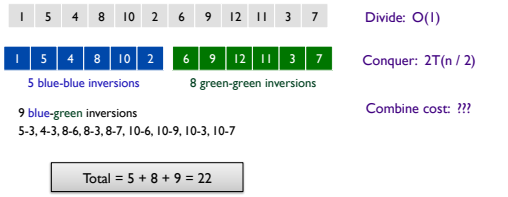
- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half



What do we need to do next?

Counting Inversions: Divide-and-Conquer

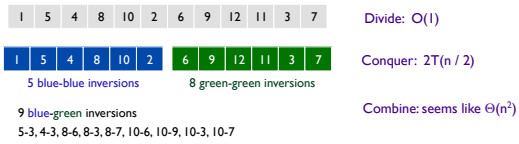
- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities



Total = 5 + 8 + 9 = 22

Counting Inversions: Divide-and-Conquer

- Divide: separate list into two pieces
- Conquer: recursively count inversions in each half
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities



Total = 5 + 8 + 9 = 22

What would make figuring out blue-green inversions easier?

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted
- Count inversions where a_i and a_j are in different halves
- Merge two sorted halves into sorted whole

to maintain sorted invariant

- What does sorting do for us?
- What is our algorithm for counting the inversions and merging?

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted
- Count inversions where a_i and a_j are in different halves
- Merge two sorted halves into sorted whole

to maintain sorted invariant

Count: O(n)

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Merge: O(n)

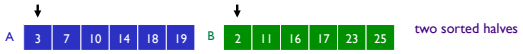
We'll run through an example in a bit…

Merge and Count

```
Merge-and-Count(A,B):
    i=0 (front of list A)
    j=0 (front of list B)
    inversions = 0
    output = □
    while A not empty and B not empty:
       output.append( min(A[i], B[j]) )
       if B[j] < A[i]:
          inversions += A.size – i (remaining elements in A)
       update i or j (whichever had smaller element)
```


Merge and Count Step

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole

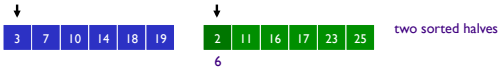


Output array

Total:

Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole

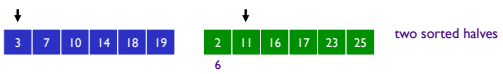


Output array

Total: 6

Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole

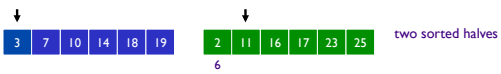


Output array

Total: 6

Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole

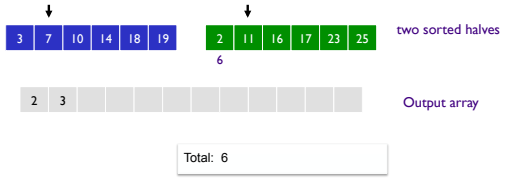


Output array

Total: 6

Merge and Count

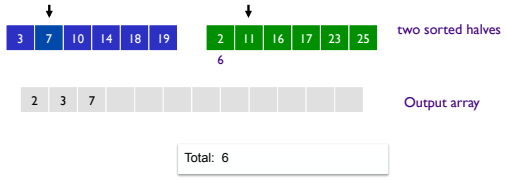
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

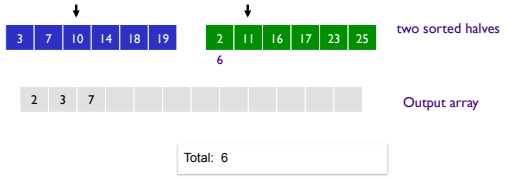
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

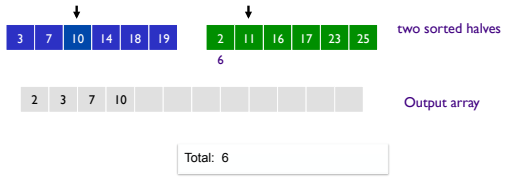
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

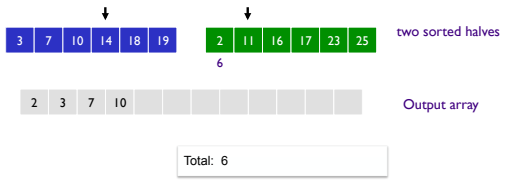
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

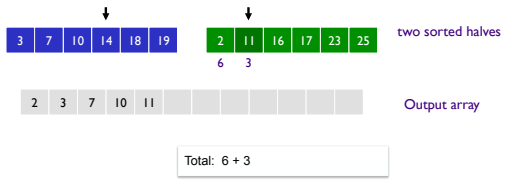
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6

Merge and Count

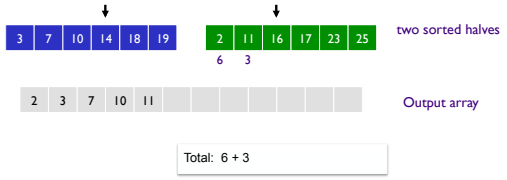
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

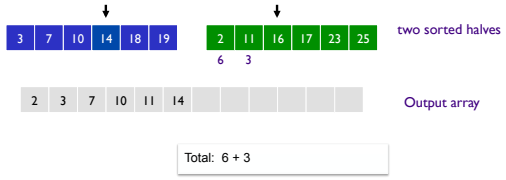
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

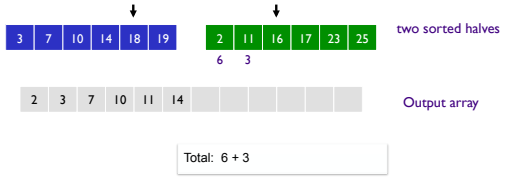
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

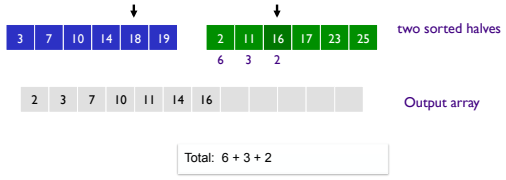
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3

Merge and Count

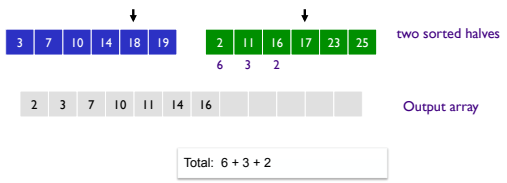
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

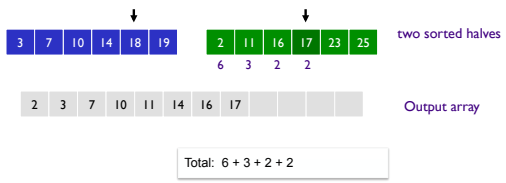
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2

Merge and Count

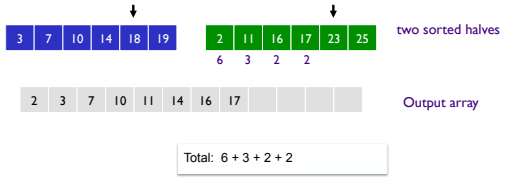
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

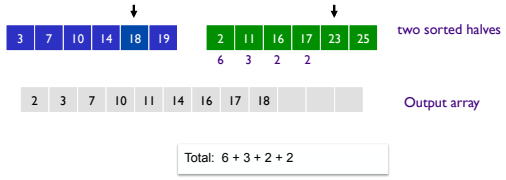
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

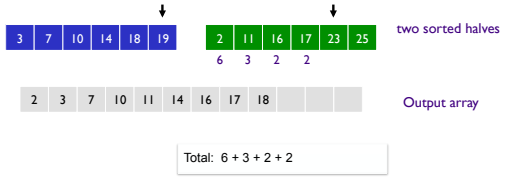
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

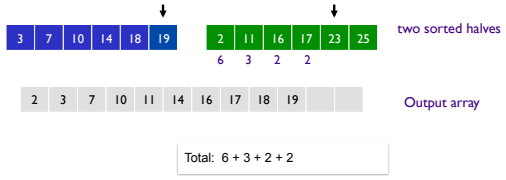
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

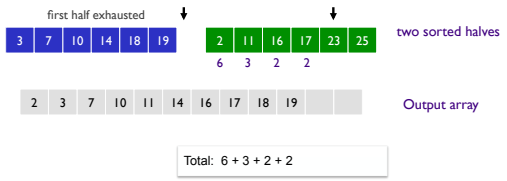
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

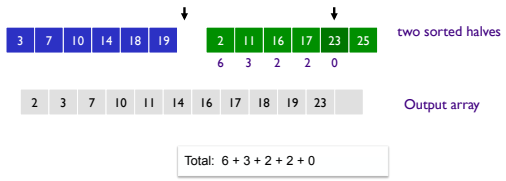
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2

Merge and Count

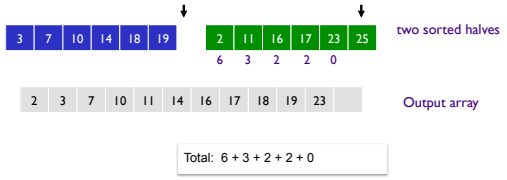
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

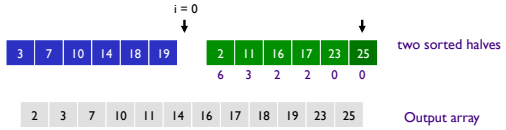
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0

Merge and Count

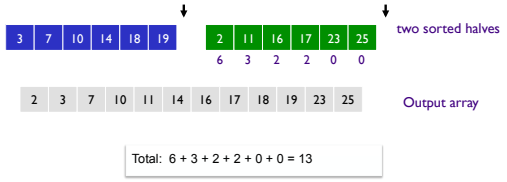
- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0 + 0

Merge and Count

- Given two sorted halves, count number of inversions where a_i and a_j are in different halves
- Combine two sorted halves into sorted whole



Total: 6 + 3 + 2 + 2 + 0 + 0 = 13

Counting Inversions: Implementation

- Merge-and-Count Pre-condition. A and B are sorted.
- Sort-and-Count Post-condition. L is sorted.

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (iA, A) = Sort-and-Count(A)
   (iB, B) = Sort-and-Count(B)
   (i, L) = Merge-and-Count(A, B)

   return i = iA + iB + i and the sorted list L
```

Recurrence relation?

Analysis

Recurrence Relation:

$T(n) \leq T(n/2) + T(n/2) + O(n)$

➔$T(n) \in O(n \log n)$

```
Sort-and-Count(L)
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (iA, A) = Sort-and-Count(A)    T(n/2)
   (iB, B) = Sort-and-Count(B)    T(n/2)
   (i, L) = Merge-and-Count(A, B) O(n)

   return i = iA + iB + i and the sorted list L
```


CLOSEST PAIR OF POINTS

Computational Geometry

- Algorithms and data structures for geometrical objects
  - Points, line segments, polygons, etc.
  - Common motivator: large data sets → efficiency
- Some Applications
  - Graphics
  - Robotics
    - motion planning and visibility problems
  - Geographic information systems (GIS)
    - geometrical location and search, route planning

Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi

fast closest pair inspired fast algorithms for these problems

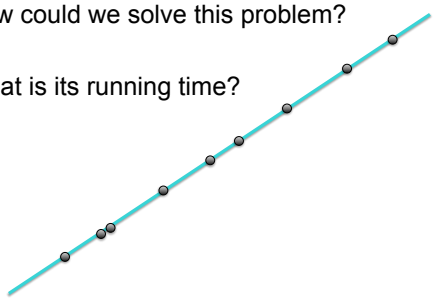
- Brute force?

Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons

Simplify: All Points on a Line

- How could we solve this problem?
- What is its running time?



Simplify: All Points on a Line

- How could we solve this problem?
  - Sort the points
    - Monotonically increasing x/y coordinates
    - No closer points than neighbors in sorted list
  - Step through, looking at the distances between each pair
- What is its running time?
  - O(n logn)

Why won't this work for 2D?

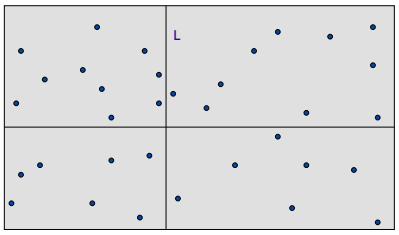
Closest Pair of Points

- Closest pair. Given *n* points in the plane, find a pair with smallest Euclidean distance between them.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons
- 1-D version. O(n log n)
  - Easy if points are on a line
- Assumption. No two points have same x coordinate

to make presentation cleaner

Closest Pair of Points: First Attempt

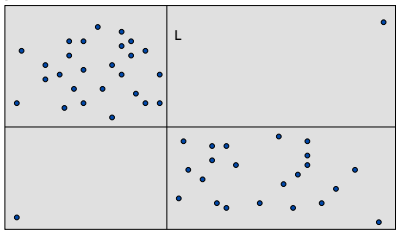
- Divide. Sub-divide region into 4 quadrants



Any problems with implementing this approach?

Closest Pair of Points: First Attempt

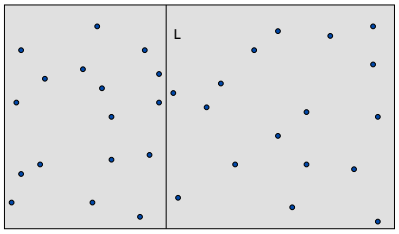
- Divide. Sub-divide region into 4 quadrants
- Obstacle. Impossible to ensure n/4 points in each piece



Closest Pair of Points

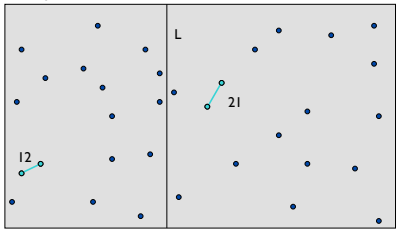
- Divide: draw vertical line L so that roughly ½n points on each side

How do we implement this?



Closest Pair of Points

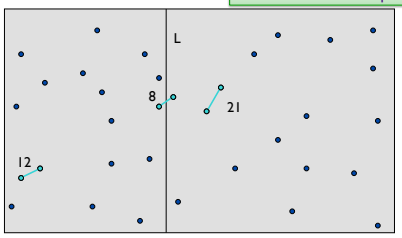
- Divide: draw vertical line L so that roughly ½n points on each side
- Conquer: find closest pair in each side recursively



Closest Pair of Points

- Divide: draw vertical line L so that roughly ½n points on each side
- Conquer: find closest pair in each side recursively
- Combine: find closest pair with one point in each side *seems like $\Theta(n^2)$*
- Return best of 3 solutions

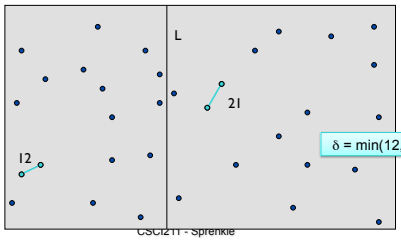
Do we need to check all pairs?



Closest Pair of Points

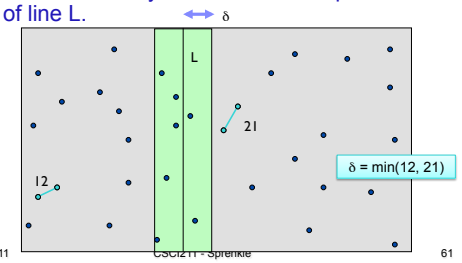
- Find closest pair with one point in each side, assuming that distance < δ

where δ = min(left_min_dist, right_min_dist)



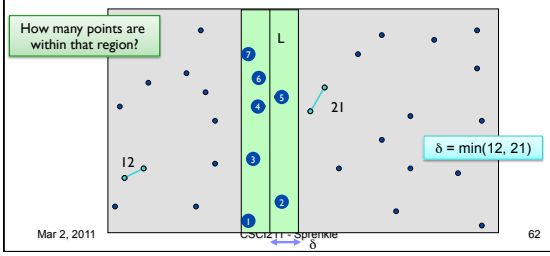
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - Observation: only need to consider points within δ of line L.



Closest Pair of Points

- Find closest pair w/ 1 point in each side, assuming that distance < δ.
  - Observation: only consider points within δ of line L
  - Sort points in 2δ-strip by their y coordinate



Assignments

- Continue reading Chapter 5
- PS5 due Friday
- SSA: opportunities for extra credit
  - If you want to attend "Truth Values", let me know so I can give you a ticket (FREE!)
- PS6 due next Friday