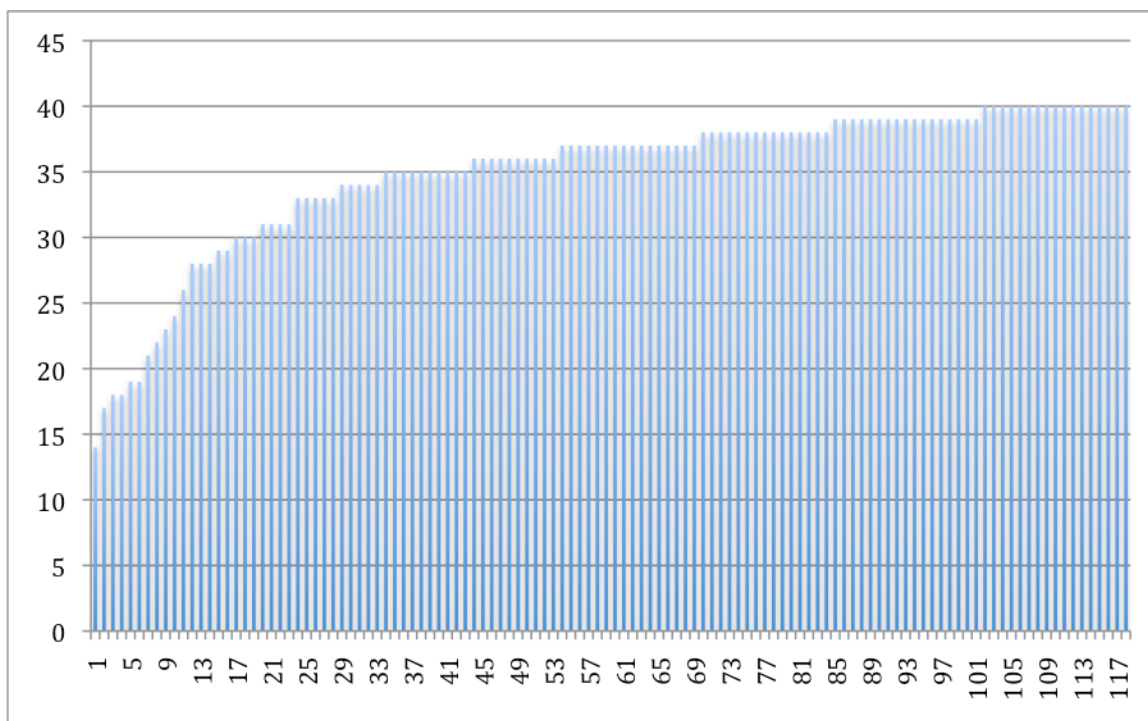


## CS106X Midterm Examination Key

---

Thanks to TA Mike Gummelt and the gaggle of section leaders, your CS106X midterms are graded and available for pickup. I've placed the exams in the lobby about 20 feet from my Gates 192 office, in the wooden handout hangout against the far wall. If you look in the folders on the left side of the wooden structure, you'll see that the exams are distributed across 26 file folders (can you guess why it's 26 and not some other number? ☺)

Feel free to pick up your exam the next time you make it over to Gates.



The height of each bar in the above graph is the score that some student got on the exam, so the chart should give you a sense as to how you did relative to everyone else. The average was a 33.6 out of 35.95, and the median grade was 37. The exam is only a small portion of your grade, but if you did poorly and you want to protect against a similar experience on the second exam, then feel free to come by my office hours to discuss a strategy for doing better.

If you notice a grading error, then you need to submit your exam back to me (Jerry) some time in the next week (before November 7<sup>th</sup>) so I can take a look at it. All regrade requests must go through Jerry.

## Solution 1: Gerrymandering

```

struct coord {
    double longitude;
    double latitude;
};

struct district {
    Set<coord> vertices; // all of size 3, pure districts are always triangles
};

double getDistrictArea(district& d) {
    // assume working implementation that computes area is already
    // provided for you, and it just works. Note, however, that this
    // only works for (triangular) districts, and not the super district
}

double computeSuperDistrictArea(coord coordinate, Vector<district>& districts) {
    double superDistrictArea = 0.0;
    for (int i = 0; i < districts.size(); i++) {
        if (districts[i].vertices.contains(coordinate)) {
            superDistrictArea += getDistrictArea(districts[i]);
        }
    }
    return superDistrictArea;
}

coord findBestCoordinate(Vector<district>& districts) {
    coord bestCoordinate;
    double largestSuperDistrictArea = 0.0;
    for (int i = 0; i < districts.size(); i++) {
        foreach (coord coordinate in districts[i].vertices) {
            double areaOfSurroundingSuperDistrict =
                computeSuperDistrictArea(coordinate, districts);
            if (areaOfSurroundingSuperDistrict > largestSuperDistrictArea) {
                largestSuperDistrictArea = areaOfSurroundingSuperDistrict;
                bestCoordinate = coordinate;
            }
        }
    }
    return bestCoordinate;
}

```

### Criteria for Problem 1: 10 points

- Knows to keep track of both best-coordinate-so-far and best-super-district-area-so-far variables: 3 points (1 point for each, and for an additional point require that 0.0 be attached to one of them to start, but don't require anything for the best-coordinate one to start)
- Ultimately returns the best-coordinate-so-far variable: 1 point
- Properly iterates over all districts: 1 point
- For each district, properly iterates over each vertex: 1 point (for this and the point above, don't penalize for silly errors [**length** instead of **size**, for instance] but do penalize for idioms that are fully made up or incorrect)
- For each vertex, properly computes the area of all districts that it's a part of: 3 points

- Iterates over all districts and properly identifies which of them includes the vertex of interest: 2 points
- Properly invokes the **getDistrictArea** on just districts known to be triangular, and builds a running sum of all relevant triangular districts: 1 point
- Properly updates the running best exactly when it should be updated: 1 point

## Solution 2: Campaign Donations and Kickbacks

There were a few interpretations of the problem, namely around what donor count (either the full number, or just the number who appealed) that should be used to compute fractions. Fortunately, the structure of your solution should have been pretty much the same regardless, so we can cope with these two (and probably other reasonable) interpretations without any trouble.

```
void analyzeRecords(Set<string>& donors, Map<bool>& appeals,
                   double& donorFraction, double& generalFraction) {
    int numDonorReductions = 0;
    int numDonorAppeals = 0;
    int numGeneralReductions = 0;
    foreach (string name in appeals) {
        if (appeals[name]) numGeneralReductions++;
        if (donors.contains(name)) {
            numDonorAppeals++;
            if (appeals[name]) {
                numDonorReductions++;
            }
        }
    }

    donorFraction = ((double) numDonorReductions) / numDonorAppeals;
    generalFraction = ((double) numDonorReductions) / appeals.size();
}
```

### Criteria for Problem 2: 10 points

- Properly declares variables to track the number of approved appeals for both donors and the general public, and initialized them to 0: 2 points
- Properly iterates over the entire name set of general appeals map, using the correct notation to do so: 2 points
- Properly increments the num-general-reductions count when **appeals[name]** is **true** (and they use the proper notation to decide—either **operator[]**, or **get**): 1 point
- Properly identifies when the person filing an appeal is also a donor: 1 point (unless their interpretation doesn't require this, in which case we just give them the point)
- Properly increments the num-donor-reductions count when a donor's is approved: 1 point
- Properly updates **donorFraction**: 2 points (1 for the right math, and 1 point for the double cast)
- Properly updates **generalFraction**: 1 point (just for the right math... don't double-ding if the cast is missing)

### Solution 3: Letter Rectangles and Words

The problem was an adaptation of the domino-oriented **chainExists** example we did in lecture, upgraded to deal with words instead of numbers, and changed so as to not use recursive backtracking (a question asking if it was possible to form a particular word would have benefitted from recursive backtracking, but that, of course, was off limits for this exam.)

```
void gatherWords(
    Vector<string>& rects, Lexicon& english, Lexicon& words, string committed) {
    if (!english.containsPrefix(committed)) return; // (this line was optional)
    if (english.containsWord(committed)) words.add(committed);
    if (rects.size() == 0) return; // (this line was also optional)

    for (int i = 0; i < rects.size(); i++) {
        string pair = rects[i];
        rects.removeAt(i);
        gatherWords(rects, english, words, committed + pair[0] + pair[1]);
        gatherWords(rects, english, words, committed + pair[1] + pair[0]);
        rects.insertAt(i, pair);
    }
}

void gatherWords(Vector<string>& rects, Lexicon& english, Lexicon& words) {
    gatherWords(rects, english, words, "");
}
```

### Criteria for Problem 3: 10 points

- Wraps the three-argument, user-friendly version around a four-argument version that knows to deal with a **committed** variable as the running prefix built up from 0 or more (possibly rotated) word rectangles: 2 points (1 point for doing it, and 1 point for initially going with "")
- Incidentally adds the running prefix to the second **Lexicon** if it appears in the first: 1 point
- If additional base cases are included as optimizations, they appear in the order given in the solution: 1 point (this point is given if there's no potential for incorrect ordering of base cases)
- Iterates over each of the rectangles: 1 point
- Allows each rectangle, during its iteration, to be the one that extends the running prefix: 3 points (1 point for properly recurring with rectangle in its original position, 1 point for properly recurring with rectangle in rotated position, and 1 point for the correct string syntax for extending the running prefix)
- Ensures that the same rectangle isn't used twice by removing it from **rects** before the recursive call: 1 point
- Ensures the same rectangle can be used deeper in the recursion by putting it back in its proper place after the recursive calls are made: 1 point

### Solution 4: Cubes as Sums of Smaller, Distinct Cubes

It's a well-known theorem among mathematicians that every number—perfect cube or not—can be expressed as a sum of at most 9 (not necessarily distinct) cubes. For more information, you can inspect [http://en.wikipedia.org/wiki/Waring's\\_problem](http://en.wikipedia.org/wiki/Waring's_problem). Your exam question was

concerned with just numbers that themselves were perfect cubes, and I also required all of the contributing cubes be distinct.

```

bool cubicDecompositionExists(int remaining, int side, Set<int>& subCubeSides) {
    if (remaining == 0) return true;
    if (remaining < 0) return false;
    // above not necessary for full credit, though it's absurdly slow without it

    for (int s = side - 1; s >= 0; s--) {
        if (cubicDecompositionExists(remaining - s * s * s, s, subCubeSides)) {
            subCubeSides.add(s);
            return true;
        }
    }

    return false;
}

bool cubicDecompositionExists(int side, Set<int>& subCubeSides) {
    return cubicDecompositionExists(side * side * side, side, subCubeSides);
}

```

#### Criteria for Problem 4: 10 points

- Wraps the function being implemented around one that actually does the recursion, making it explicit that the largest cubic root that could possibly contribute to the cubic decomposition is side itself: 2 points (note that it's possible that some other approach might work as well, but ultimately the recursive backtracking is trying to discover a subset of the set {1, 2, ..., **side**} that identifies the cubic decomposition of interest.
- Returns true at the precise moment a decomposition is discovered: 2 points
- Let's side serve as a hard upper bound on the range of numbers that might recursively contribute to the contribution: 1 point
- Recursively entertains the possibility that a particular value strictly less than side might contribute: 1 point
- Returns **true** if that particular choice recursively leads to a solution: 1 point
- Includes the choice in the **Set<int>** when it's discovered to be a good one: 1 point
- Does not return **false** just because one particular choice didn't work out, but because all possible choices don't work out: 2 points

#### Aside

Although the problem didn't require it, I could have asked for a decomposition that **minimizes** the number cubes in the sum. Had I done that, the problem would have been more difficult, but certainly doable given what we've learned. One particular approach might first limit the number of terms to 3, and unless that produced a solution, to start over with a new limit of 4 terms instead, then 5, then 6, etc. This technique is reminiscent of a technique you'll focus on in artificial intelligence classes called iterative deepening, which is depth-first-search-oriented up to a maximum depth. This, of course, was not required for you to get full credit, and I said exactly that in the problem statement.

```
bool cubicDecompositionExists(int remaining, int side,
                             Set<int>& subCubeSides, int numTermsAllowed) {
    if (remaining == 0) return true;
    if (remaining < 0) return false; // not necessary.. huge perf win, though
    if (numTermsAllowed <= 0) return false;

    for (int s = side - 1; s >= 0; s--) {
        if (cubicDecompositionExists(remaining - s * s * s, s,
                                     subCubeSides, numTermsAllowed - 1)) {
            subCubeSides.add(s);
            return true;
        }
    }

    return false;
}

bool cubicDecompositionExists(int side, Set<int>& subCubeSides) {
    for (int maxTerms = 2; maxTerms <= side; maxTerms++) {
        if (cubicDecompositionExists(side * side * side,
                                     side, subCubeSides, maxTerms))
            return true;
    }

    return false;
}
```