## Problem 1

How much slower does each algorithm get when you (i) double the input size or (ii) increase the input size by 1?

a. $n^2$:
(i) runs $2^2$ times slower.
(ii) takes 2n+1 more steps.
b. $n^3$
(i) runs $2^3$ times slower.
(ii) takes $3n^2 + 3n + 1$ more steps.
c. $100n^2$:
(i) runs $2^2$ times slower.
(ii) takes 200n+100 more steps.
d. $n \log n$
(i) runs in $2n + 2n \log n$ time, which is twice the original runtime, plus a factor of 2n.
(ii) slows by $n \log(\frac{n+1}{n}) + \log(n+1)$.
e. $2^n$
(i) runs in $2^{n^2} = 2^n \times 2^n$ time.
(ii) takes twice as long to run.

## Problem 2

For each of the algorithms, what is the largest input size n that the algorithm could process in an hour, using a computer than can perform $10^{10}$ operations per second?

There are 3600 seconds per hour, so the computer can perform at most $3600 \times 10^{10} = 3.6 \times 10^{13}$ operations:
a. $n^2 : \sqrt{3.6e13} = 6 \times 10^6$.
b. $n^3 : \sqrt[3]{3.6e13} \approx 33019$.
c. $100n^2 : 6 \times 10^6$.
d. $n \log n : n \approx 2.85 \times 10^{12}$.
e. $2^n : \log_2(3.6 \times 10^{13}) \approx 45$.
f. $2^{2^n} : \log_2(45) \approx 5$.

## Problem 3

...Suppose that each line has a length that is bounded by a constant $c$, and suppose that the song runs for n words total. Show how to encode such a song using a script that has length f(n), for a function f(n) that grows as slowly as possible:

1. Give an algorithm for transcribing such a song onto a sheet of paper. The output of A is a document with O(f(N)) words.

Write each unique line down in the order it is to be sung, for example:
0) On the [M]th day of Christmas, my true love gave to me:
1) A partridge in a pair tree.
2) Two turtle doves, and
3) Three French hens,

...
M) [Last line here].

2. Give an algorithm for singing the song from such a document so that the song is sung as above, running for a total of N words.

Singing the song will require nested for loops, with the outer loop counting the line number the song is currently on and the inner loop controlling which line of the song is being sung:

```
For i from 1 to M:
    Sing line 0.
    For j from i to 1:
        Sing line j.
```

3. Explain what your f(N) is. f(N) is defined as the largest document size (in words) for all songs with at most N words. You want f(N) to grow with N as slowly as possible.

There are M lines total, and the length of each line is limited to c words, and we assume that each line has at least one word. Thus the size of the document in terms of M is at least $\sum_{i=0}^{M} i = \frac{M^2+M}{2}$ and at most $c\sum_{i=0}^{M} i = c\ \frac{M^2+M}{2}$. The upper bound on size must be less than N, $c\ \frac{M^2+M}{2} \leq N$, thus $(M^2+M) \leq \frac{2N}{c}$. We can disregard the second M term in the LHS of the inequality to find that $M^2 \leq \frac{2N}{c} \rightarrow M \leq \sqrt{\frac{2N}{c}}$, which can be simplified to $M \leq \sqrt{N}$ in asymptotic analysis. Thus, f(N) = O($\sqrt{N}$).

## Problem 4

### Claim

Let G be a graph on n nodes, where n is an even number. If every node of G has degree at least $\frac{n}{2}$, then G is connected.

### Proof

Let G be a graph on n nodes, where n is an even number, such that every node of G has degree at least $\frac{n}{2}$. Assume, for contradiction, that G is not connected. Then, we can partition nodes in G into at least two subsets, $G_1$ and $G_2$, such that no node in $G_1$ has an edge incident on a node in $G_2$ (and vice versa). Let $G_1$ be a connected subset of minimal size, so that each node in $G_1$ has degree at least $\frac{n}{2}$. $G_1$ must have at least $\frac{n}{2}+1$ nodes, implying that $G_2$ has at most $\frac{n}{2}-1$ nodes. But then any node in $G_2$ can have degree of at most $\frac{n}{2}-2$, contradicting our initial definition of G. Thus, by way of contradiction, G is connected.

## Problem 5

### Description

The problem describes a set of data, sorted temporally in ascending order, consisting of ordered triples $(C_i, C_j, t_k)$ where the first two elements represent computers that have communicated with one another and the third element represents the time at which the contact occurred. Each pair of computers appears in the set at most once. The problem asks for an algorithm that processes the data to determine whether or not computer $C_a$ was communicating with computer $C_b$ at time t.

## Solution

```
// The solution assumes an array of N computers, $C_1 to C_N$. Each element $C_i$
// has an vector of times at which contact was made, and each time, $t_i$ contains a list
// with the indices of computers that were communicating with $C_i$ at $t_i$.
// Assume that there are M triples.
// Input: Infected computer $C_a$, infected time x, suspected infected computer $C_b$, time y.
// Assume the array of N computers is declared but not populated.
For each triple $(C_i, C_j, t_k)$,
    If no data exists at the indices corresponding to $C_i, C_j$,
        Go to the corresponding elements in the array of computers,
            add a vector containing $t_k$, then
                add list beginning with appropriate index of communicating computer.
    Else,
        Go to appropriate element in the array of computers.
        If $t_k$ exists,
            add index of communicating computer.
        Else,
            add vector containing $t_k$, then
                add list beginning with appropriate index of communicating computer.

Go to $C_a$, time x.
For $t_i$ from x to y,
    For each computer $C_i$:
        Search list of $C_i$ for $C_b$ between $t_x$ and $t_y$.
```

## Correctness and Time Complexity

Since the algorithm begins at $C_a$ at time of infectivity $t_x$, every computer that communicated with $C_a$ after that time became infected and then passed on the virus to any other computers it subsequently communicated with. Since time only moves forward in our data set, the algorithm follows a path from the first infected computer to every other computer infected between $t_x and t_y$ by systematically searching the lists of every computer that came into contact with $C_a$ in the given interval.

Constructing the arrays takes the following number of steps: M+k steps for the first loop and at most N(x-y) steps for the second loop, thus M+N steps in the worst case, for large M, N.

# Problem 6

## Description

Suppose that you are given an algorithm as a black box, that is, you cannot see how it is designed. It has the following properties: if you input any sequence of real numbers and an integer k, the algorithm will answer YES or NO indicating whether there is a subset of numbers whose sum is exactly k. Show how to use this black box to find the subset of a given sequence $x_1, ..., x_n$ whose sum is k. You can use the black box O(n) times.

## Solution

```
// Input: A sequence S of real numbers, of length n.
// Output: A subset X of real numbers whose sum is exactly k, or the empty set.
// Let blackbox( input ) denote input of a sequence to the black box.
X = ∅.
If blackbox( S ) returns YES:
    For i from 0 to n:
        If blackbox( $S \setminus \{s_i\}$ ) returns YES,
            S' = $S \setminus \{s_i\}$.

Return X.
```

## Analysis

The algorithm begins by asking the black box whether or not a subset that sums to k exists within the given sequence S. If such a subset exists, the algorithm sequentially removes elements that are not in the desired subset. If no such subset exists, the algorithm returns the empty set. Clearly, this algorithm runs in 0(n) time in the worst case.