Encoding/Decoding, Counting graphs

	Miles Jones	MTThF 8:30-9:50am	CSE 4140

August 23, 2016

11-avoiding binary strings

Let's consider the set of all n-bit binary strings with the property that 11 is not a substring.

example: 100010100001010

How many of these strings are there? of length n

11-avoiding binary strings

Let's consider the set of all n-bit binary strings with the property that 11 is not a substring.

example: 100010100001010

How many of these strings are there?

A. 2^n
B. 2^{n-1}
C. F_{n+2} (Fibonacci number.)
D. $n!$
E. $\binom{n}{k}$

11-avoiding binary strings

Let's consider the set of all n-bit binary strings with the property that 11 is not a substring.

How many of these strings are there?

Let S(n) be the set of all n-bit binary strings of this type.
Then split them into two subsets:

S1(n) is the subset that starts with 1
S0(n) is the subset that starts with 0

Are S1(n) and S0(n) disjoint?

11-avoiding binary strings

Let's consider the set of all n-bit binary strings with the property that 11 is not a substring.

$S(n) = S1(n) \cup S0(n)$ and $S1(n) \cap S0(n) = \emptyset$

$$|S(n)| = |S1(n)| + |S0(n)|$$

All elements of S0(n) are of the form 0[11-avoiding (n-1)-bit binary string]
All elements of S1(n) are of the form 10[11-avoiding (n-2)-bit binary string]

$$|S0(n)| = |S(n-1)|$$
$$|S1(n)| = |S(n-2)|$$

$$|S(n)| = |S(n-1)| + |S(n-2)|$$

11-avoiding binary strings

$$|S(n)| = |S(n-1)| + |S(n-2)|$$

what does this look like?

What is |S(1)|? What is |S(2)|?

11-avoiding binary strings

$$|S(n)| = |S(n-1)| + |S(n-2)|$$

what does this look like?

|S(1)|=2 |S(2)|=3

$|S(n)| = F_{n+2}$ (Fibonacci number.)

$F_1 = 1, F_2 = 1$

11-avoiding binary strings (encoding)

The most straightforward way to encode one of these strings is by using the string itself.

This gives us an upper bound on the number of bits needed to encode these types of strings.

n-bits is enough to encode these strings so this means that

$$|S(n)| = F_{n+2} \leq 2^n$$

Theoretically Optimal Encoding

A **theoretically optimal encoding** for length n 11-avoiding binary strings would use

the ceiling of $\log_2(F_{n+2})$ bits.

How?

- List all length n 11-avoiding binary strings in lex-order
- To encode: Store the position of a string in the list, rather than the string itself.
- To decode: Given a position in list, need to determine string in that position.

Lex Order

E.g. the 13 Length n=5 11-avoiding binary strings:

Original string, s	Encoded string (i.e. position in this list)
00000	0 = 0000
00001	1 = 0001
00010	2 = 0010
00100	3 = 0011
00101	4 = 0100
01000	5 = 0101
01001	6 = 0110
01010	7 = 0111
10000	8 = 1000
10001	9 = 1001
10010	10 = 1010
10100	11 = 1011
10101	12 = 1100

Lex Order: Algorithm?

Need two algorithms, given specific n:

$$s \rightarrow E(s,n)$$

and

$$p \rightarrow D(p,n)$$

Idea: Use recursion (reduce & conquer).

Lex Order: Algorithm?

For E(s,n):

0….
0….
…
1…..
1….
1….

Length n-1 11-avoiding binary strings

Length n-2 11-avoiding binary strings

- Any string that starts with 0 must have position **before** $|S(n-1)| = F_{n+1}$
- Any string that starts with 1 must have position **at or after** $|S(n-1)| = F_{n+1}$

Lex Order: Algorithm?

$\boxed{19}$

Example: Encode 00101001

(Note: this is a length 9 binary string.)

S[1]=0 return E[01010011]

S[1]=0 return E[1010011]

S[1]=1 return 13 + E[010011]

S[1]=0 return 0+E[10011]

S[1]=1 return 5+E[0011]

S[1]=0 return E[011]

S[1]=0 return E[11]

n=1 return 1

Lex Order: Algorithm?

Example: Encode 010101001

(Note: this is a length 9 binary string.)

Initialize p:=0, n:=9
The first bit is 0 so p:=p+0=0, n:=8
The second bit is 1 so $p := p + F_{n+1} = p + F_9 = \mathrm{p} + 34 = 34$, n:=7
The third bit is 0 so p:=p+0=34, n:=6
The fourth bit is 1 so $p := p + F_{n+1} = p + F_7 = \mathrm{p} + 13 = 47$, n:=5
The fifth bit is 0 so p:=p+0=47, n:=4
The sixth bit is 1 so $p := p + F_{n+1} = p + F_5 = \mathrm{p} + 5 = 52$, n:=3
The seventh bit is 0 so p:=p+0=52, n:=2
The eighth bit is 0 so p:=p+0=52, n:=1
The ninth bit is 1 so $p := p + F_{n+1} = p + F_2 = \mathrm{p} + 1 = 53$, n:=0

Lex Order: Algorithm?

Example: Encode 010101001

(Note: this is a length 9 binary string.)

So this string is number 53 in the list and so it will be encoded by the binary expansion of 53 which is 110101. The maximum number of bits needed to store any of these strings is $\lceil \log(F_{11}) \rceil = 7$. So we will pad the left with 0's

E(010101001,9)=0110101

Lex Order: Algorithm?

Example: Decode ~~1001011 = 75~~ into a length 9 binary string.

19

5	4	3	2	1	0
1	0	1	0	0	1

$19 > 13$

$19 - 13 = 6$

$6 < 8$

$6 > 5$

$6 - 5 = 1$

$1 < 3$

$1 < 2$

$1 = 1$

Lex Order: Algorithm?

Example: Decode 1001011 = 75 into a length 9 binary string.

$75>55=F_{10}$ so the first bit is 1	$75-55=20$
$20<34=F_9$ so the next bit is 0	
$20<21=F_8$ so the next bit is 0	
$20>13=F_7$ so the next bit is 1	$20-13=7$
$7<8=F_6$ so the next bit is 0	
$7>5=F_5$ so the next bit is 1	$7-5=2$
$2<3=F_4$ so the next bit is 0	
$2=2=F_3$ so the next bit is 1	$2-2=0$
$0<1=F_2$ so the next bit is 0	

D(1001011,9)=100101010

Theoretically Optimal Encoding

A **theoretically optimal encoding** for length n 11-avoiding binary strings would use

the ceiling of $\log_2(F_{n+2})$ bits.

How big is $\log_2(F_{n+2})$?

$$F_{n+2} < (1.6)^{n+2} \approx \left(2^{0.7}\right)^{n+2} = 2^{0.7n+1.4}$$

So…….

$$\lceil \log_2(F_{n+2}) \rceil < \left\lceil \log_2 2^{0.7n+1.4} \right\rceil = \lceil 0.7n + 1.4 \rceil$$

Another application of counting … lower bounds

Searching algorithm of sorted list:

performance was measured in terms of number of comparisons between list elements

*What's the **fastest possible worst case*** for any searching algorithm of sorted lists?

Another application of counting … lower bounds

Searching algorithm of sorted list:

performance was measured in terms of number of comparisons between list elements

*What's the **fastest possible worst case*** for any searching algorithm of sorted lists?

Tree diagram represents possible comparisons we might have to do, based on relative sizes of elements.

Another application of counting … lower bounds

Searching algorithm of sorted list:

If we construct a tree of all possible comparisons to find all elements, how many leaves will the tree have?

Another application of counting … lower bounds

Searching algorithm of sorted list:

If we construct a tree of all possible comparisons to find all elements, how many leaves will the tree have? n

How tall will this tree be? log(n) is shortest possible

n-1 is longest possible.

Another application of counting … lower bounds

Searching algorithm of sorted list:

If we construct a tree of all possible comparisons to find all elements, how many leaves will the tree have? n

How tall will this tree be? log(n)

So log(n) is the fastest possible runtime and binary search achieves this!!!!

Another application of counting … lower bounds

Sorting algorithm: performance was measured in terms of number of comparisons between list elements

*What's the **fastest possible worst case*** for any sorting algorithm?

Another application of counting … lower bounds

Sorting algorithm: performance was measured in terms of number of comparisons between list elements

What's the ***fastest possible worst case*** for any sorting algorithm?

Tree diagram represents possible comparisons we might have to do, based on relative sizes of elements.

Another application of counting … lower bounds

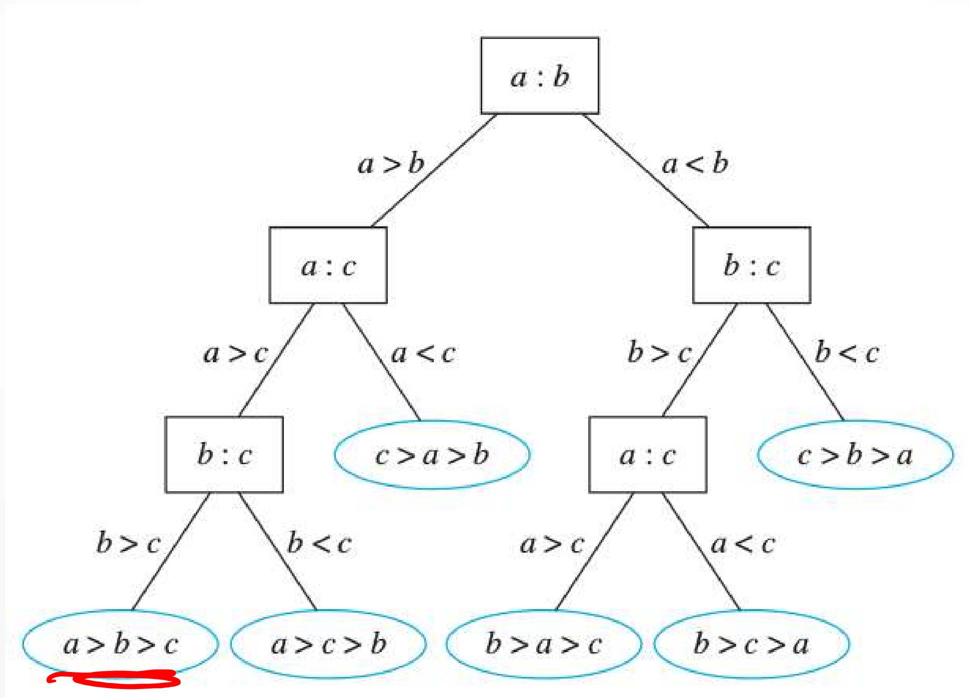
Tree diagram represents possible comparisons we might have to do, based on relative sizes of elements.

321

231

213

123



a, b, c distinct integers

Rosen p. 761

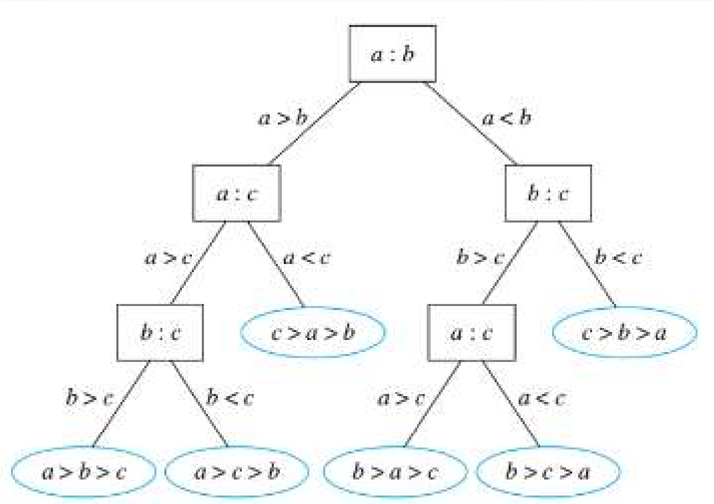
Another application of counting … lower bounds

Sorting algorithm: performance was measured in terms of number of comparisons between list elements

What's the ***fastest possible worst case*** for any sorting algorithm?

Maximum number of comparisons for algorithm is **height** of its tree diagram.

Another application of counting … lower bounds



How many leaves will there be in a decision tree that sorts n elements?

A. 2^n
B. log n
C. n!
D. C(n,2)
E. None of the above.

Another application of counting … lower bounds

Sorting algorithm: performance was measured in terms of number of comparisons between list elements

What's the ***fastest possible worst case*** for any sorting algorithm?

Maximum number of comparisons for algorithm is **height** of its tree diagram.

For any algorithm, what would be ***smallest possible height***?

What do we know about the tree?

* Internal nodes correspond to comparisons.
* Leaves correspond to possible input arrangements.

Another application of counting … lower bounds

Sorting algorithm: performance was measured in terms of number of comparisons between list elements

*What's the **fastest possible worst case*** for any sorting algorithm?

Maximum number of comparisons for algorithm is **height** of its tree diagram.

For any algorithm, what would be ***smallest possible height***?

What do we know about the tree?

* Internal nodes correspond to comparisons. *Depends on algorithm.*
* Leaves correspond to possible input arrangements. n!

Another application of counting … lower bounds

What's the ***fastest possible worst case*** for any sorting algorithm?

Maximum number of comparisons for algorithm is **height** of its tree diagram.

For any algorithm, what would be ***smallest possible height***?

What do we know about the tree?
    * Internal nodes correspond to comparisons. *Depends on algorithm.*
    * **Leaves correspond to possible input arrangements**. n!

Each tree diagram must have at least **n! leaves**, so its height must be at least $\log_2(n!)$.

Another application of counting … lower bounds

*What's the **fastest possible worst case*** for any sorting algorithm?

Maximum number of comparisons for algorithm is **height** of its tree diagram.

For any algorithm, what would be ***smallest possible height***?

What do we know about the tree?

 * Internal nodes correspond to comparisons. *Depends on algorithm.*

 * **Leaves correspond to possible input arrangements**. n!

Each tree diagram must have at least **n! leaves**, so its height must be at least $\log_2(n!)$.

i.e. fastest possible worst case performance of sorting is $\mathbf{\log_2(n!)}$

Another application of counting … lower bounds

What's the ***fastest possible worst case*** for any sorting algorithm? **$\log_2(n!)$**

How big is that?

Lemma: For n>1,

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} < n! < n^n$$

Proof:

$$\begin{aligned} n! &= (n)(n-1)(n-2)\ldots\left(\frac{n}{2}\right)\ldots(3)(2)(1) \\ &> \left(\frac{n}{2}\right)\left(\frac{n}{2}\right)\left(\frac{n}{2}\right)\cdots\left(\frac{n}{2}\right) \\ &= \left(\frac{n}{2}\right)^{\frac{n}{2}} \end{aligned}$$

$$\begin{aligned} n! &= (n)(n-1)(n-2)\ldots(3)(2)(1) \\ &< (n)(n)(n)\ldots(n)(n)(n) \\ &= n^n \end{aligned}$$

Another application of counting … lower bounds

What's the ***fastest possible worst case*** for any sorting algorithm? **$\log_2(n!)$**

How big is that?

Lemma: for n>1, $\left(\frac{n}{2}\right)^{\frac{n}{2}} < n! < n^n$

Theorem: $\log_2(n!)$ is in $\Theta(n \log n)$

Proof: For n>1, taking logarithm of both sides in lemma gives

$$\frac{n}{2} \log \left(\frac{n}{2}\right) < \log_2(n!) < n \log n$$

i.e.

$$\frac{1}{2}(n \log n - n \log 2) < \log_2(n!) < n \log n$$

Another application of counting … lower bounds

What's the ***fastest possible worst case*** for any sorting algorithm? **$\log_2(n!)$**

How big is that?

Lemma: for n>1, $\left(\frac{n}{2}\right)^{\frac{n}{2}} < n! < n^n$

Theorem: $\log_2(n!)$ is in $\Theta(n \log n)$

Therefore,
the best sorting algorithms will need $\Theta(n \log n)$ comparisons in the worst case.

i.e. it's impossible to have a comparison-based algorithm that does better than **Merge Sort** (in the worst case).

Representing undirected graphs

Strategy:

1. Count the number of simple undirected graphs.
2. Compute lower bound on the number of bits required to represent these graphs.
3. Devise algorithm to represent graphs using this number of bits.

What's true about **simple undirected** graphs?

A. Self-loops are allowed.
B. Parallel edges are allowed.
C. There must be at least one vertex.
D. There must be at least one edge.
E. None of the above.

Rosen p. 641-644

Representing undirected graphs: Counting

In a simple undirected graph on n (labeled) vertices, how many edges are possible?

A. n^2
B. $n(n-1)$
C. $C(n,2)$
D. $2^{C(n,2)}$
E. None of the above.

$$= \frac{n!}{(n-2)!\,2!} = \frac{n(n-1)}{2}$$

** Recall notation: $C(n,k) = \binom{n}{k}$ **

Representing undirected graphs: Counting

In a simple undirected graph on n (labeled) vertices, how many edges are possible?

A. n^2
B. $n(n-1)$
C. $C(n,2)$ *Possibly one edge for each set of two distinct vertices.*
D. $2^{C(n,2)}$
E. None of the above.

** Recall notation: $C(n,k) = \binom{n}{k}$ **

Representing undirected graphs: Counting

How many **different** simple undirected graphs on n (labeled) vertices are there?

A. n^2
B. $n(n-1)$
C. $C(n,2)$
D. $2^{C(n,2)}$
E. None of the above.

there are $\binom{n}{2}$ possible edges in a graph each edge is possibly there or not there

Representing undirected graphs: Lower bound

How many **different** simple undirected graphs on n (labeled) vertices are there?

A. n^2
B. $n(n-1)$
C. $C(n,2)$
D. $2^{C(n,2)}$
E. None of the above.

For each possible edge, decide if in graph or not.

Conclude:
minimum number of bits to represent simple undirected graphs with n vertices is

$\log_2(2^{C(n,2)}) = C(n,2) = \mathbf{n(n-1)/2}$

Representing undirected graphs: Algorithm

Goal: represent a simple undirected graph with n vertices using n(n-1)/2 bits.

Idea: store adjacency matrix, but since

- diagonal entries all zero — *no self loops*
- matrix is symmetric — *undirected graph*

only store the entries **above** the diagonal.

How many entries of the adjacency matrix are above the diagonal?

A. n^2
B. n(n-1)
C. C(n,2)
D. 2n
E. None of the above.

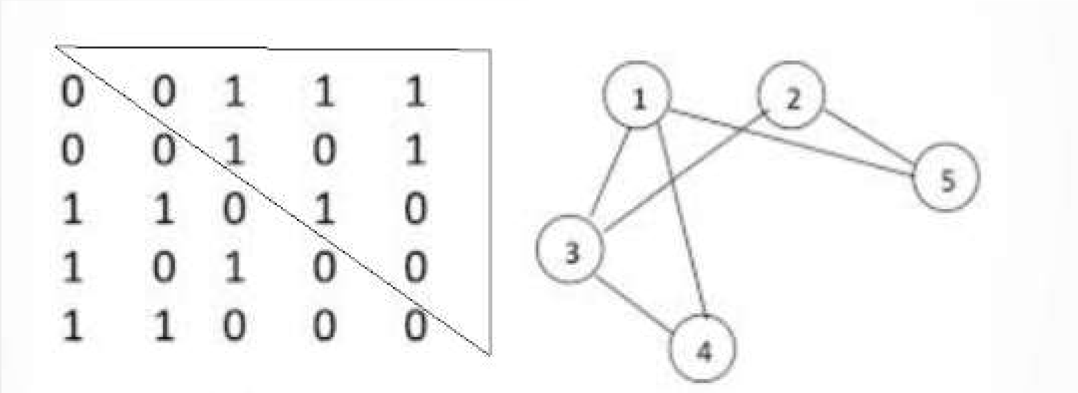
Representing undirected graphs: Algorithm

Goal: represent a simple undirected graph with n vertices using n(n-1)/2 bits

Idea: store adjacency matrix, but since

- diagonal entries all zero — *no self loops*
- matrix is symmetric — *undirected graph*

only store the entries **above** the diagonal.



Can be stored as
0111101100
which uses C(5,2) = 10 bits.

Representing undirected graphs: Algorithm

Decoding: ?

What simple undirected graph is encoded by the binary string

011010 110101 111111 000000 110101 110010 ?

A.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

B.

001101011
000101111
000111000
000000011
000000101
000000110
000000001
000000000

C. Either one of the above.

D. Neither one of the above.

Representing directed graphs: Counting

In a simple **directed** graph on n (labeled) vertices, how many edges are possible?

A. n^2
B. $n(n-1)$
C. $C(n,2)$
D. $2^{C(n,2)}$
E. None of the above.

Simple graph: no self loops, no parallel edges.

Representing **directed** graphs: Counting

In a simple **directed** graph on n (labeled) vertices, how many edges are possible?

A. n^2
B. $n(n-1)$
C. $C(n,2)$
D. $2^{C(n,2)}$
E. None of the above.

Choose starting vertex, choose ending vertex.

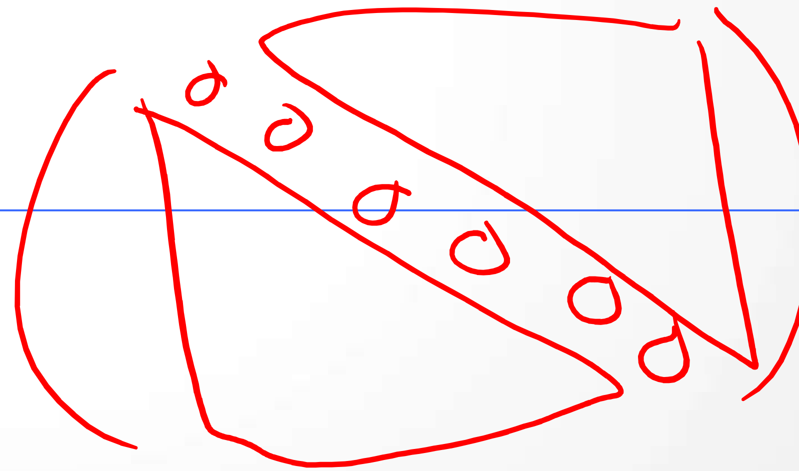
Simple graph: no self loops, no parallel edges.

Representing directed graphs: Counting

How many **different** simple directed graphs on n (labeled) vertices are there?

A. n^2
B. n(n-1)
C. C(n,2)
D. $2^{C(n,2)}$
E. None of the above.

$$2^{n(n-1)}$$



Representing directed graphs: Counting

Another way of counting that there are $2^{n(n-1)}$ simple directed graphs with n vertices:

Represent a graph by

For each of the C(n,2) pairs of distinct vertices {v,w},

specify whether there is

* no edge between them 00
* an edge from v to w but no edge from w to v 01
* an edge from w to v but no edge from v to w 10
* edges both from v to w and from v to w. 11

Representing **directed** graphs: Counting

Another way of counting that there are $2^{n(n-1)}$ directed graphs with n vertices:

Represent a graph by
For each of the C(n,2) pairs of distinct vertices {v,w}, specify whether there is
* no edge between them
* an edge from v to w but no edge from w to v
* an edge from w to v but no edge from v to w
* edges both from v to w and from v to w.

C(n,2) pairs

each has 4 options

Product rule!

$$(4)(4)\ldots.(4) = 4^{C(n,2)}=4^{(n(n-1)/2)}=2^{n(n-1)}$$

Representing directed graphs: Lower bound

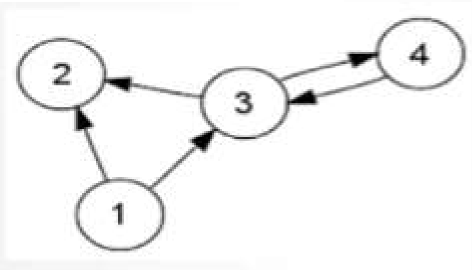
Conclude:
minimum number of bits to represent simple directed graphs with n vertices is

$$\log_2(2^{n(n-1)}) = \mathbf{n(n-1)}$$

Representing **di**rected graphs: Algorithm

Encoding:
For each of the n vertices, indicate which of the other vertices it has an edge to.

How would you encode this graph
using bits (0s and 1s)?



A. 123232443
B. 0110 0000 0101 0010
C. 110 000 011 001
D. None of the above.

Representing directed graphs: Algorithm

Decoding:

Given a string of 0s and 1s of length n(n-1),

- Define vertex set { 1, …, n }.
- First n-1 bits indicate edges from vertex 1 to other vertices.
- Next n-1 bits indicate edges from vertex 2 to other vertices.
- etc.

What graph does this binary string encode? 0110 1001 0001 1011 0100