Characteristic Vectors
&
BFS&DFS

Laiyan Ding
117010053@link.cuhk.edu.cn

Example: Characteristic Vectors

- Any subsets of the following set:

  digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

  can be indicated by a characteristic vector of 10 bits:

  $\{1, 3, 5, 7, 9\}$

F	T	F	T	F	T	F	T	F	T
0	1	2	3	4	5	6	7	8	9

  $\{2, 3, 5, 7\}$

F	F	T	T	F	T	F	T	F	F
0	1	2	3	4	5	6	7	8	9

- The advantage of using characteristic vectors is that doing so makes it possible to implement the operations `add`, `remove`, and `contains` in any subsets in constant time. For example, to add the element *k* to a set, all you have to do is set the element at index position *k* in the characteristic vector to true. Similarly, testing membership is simply a matter of selecting the appropriate element in the array.

Bit Vectors and Character Sets

- This picture shows a characteristic vector representation for the set containing the uppercase and lowercase letters:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	2
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7

- Storing characteristic vectors as explicit arrays can require a large amount of memory, particularly if the set is large. To reduce the storage requirements, you can pack the elements of the characteristic vector into machine words so that the representation uses every bit in the underlying representation.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Bit Vectors and Character Sets

- Exercise: What is a characteristic vector representation for the set containing the digit letters:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	

Bitwise Operators

- If you know your client is working with sets of characters, you can implement the set operators extremely efficiently by storing the set as an array of bits and then manipulating the bits all at once using C++'s *bitwise operators*.
- The bitwise operators are summarized in the following table and then described in more detail on the next few slides:

x & *y*	Bitwise AND. The result has a 1 bit wherever both *x* and *y* have 1s.
x | *y*	Bitwise OR. The result has a 1 bit wherever either *x* or *y* have 1s.
x ^ *y*	Exclusive OR. The result has a 1 bit wherever *x* and *y* differ.
~*x*	Bitwise NOT. The result has a 1 bit wherever *x* has a 0.
x << *n*	Left shift. Shift the bits in *x* left *n* positions, shifting in 0s.
x >> *n*	Right shift. Shift *x* right *n* bits (logical shift if *x* is unsigned).

The Bitwise AND Operator

- The bitwise AND operator (`&`) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which both *x* and *y* have 1 bits. A table for the `&` operator appears to the right.

	0	1
0	0	0
1	0	1

- The primary application of the `&` operator is to *select* certain bits in an integer, clearing the unwanted bits to 0. This operation is called *masking*.

- In the context of sets, the `&` operator performs an intersection operation, as in the following calculation of odds $\cap$ squares:

	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
&	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The Bitwise OR Operator

- The bitwise OR operator (|) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position which either *x* or *y* has a 1 bit (or if both do, i.e., non-exclusive), as shown in the table on the right.

	0	1
0	0	1
1	1	1

- The primary use of the | operator is to *assemble* a single integer value from other values, each of which contains a subset of the desired bits.

- In the context of sets, the | operator performs a union, as in the following calculation of primes ∪ squares:

	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1
|	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

The Exclusive OR Operator

or but not both

- The exclusive OR or XOR operator (^) takes two integer operands, *x* and *y*, and computes a result that has a 1 bit in every position in which *x* and *y* have different bit values, as shown on the right.

	0	1
0	0	1
1	1	0

- The XOR operator has many applications in programming, most of which are beyond the scope of this text.
- The following example *flips* all the bits in the rightmost three bytes of a word:

	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1
^	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	1	1	0	0	1	1	0	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- Question: What if we apply the same XOR mask twice?

  See “XOR drawing mode” in computer graphics if interested.

The Bitwise NOT Operator

- The bitwise NOT operator (~) takes a single operand *x* and returns a value that has a 1 wherever *x* has a 0, and vice versa.
- You can use the bitwise NOT operator to create a mask in which you mark the bits you want to eliminate as opposed to the ones you want to preserve.
- The ~ operator creates the *complement* of a set, as shown with the following diagram of ~primes:

~	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0	1
	1	1	0	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	1	1	1	1	0	1	0
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- Question: Can you use the ~ operator to compute the set difference operation?

The Shift Operators

- C++ defines two operators that have the effect of shifting the bits in a word by a given number of bit positions.
- The expression x **<<** n shifts the bits in the integer x leftward n positions. Spaces appearing on the right are filled with 0s.
- The expression x **>>** n shifts the bits in the integer x rightward n positions. The question as to what bits are shifted in on the left depend on whether x is a signed or unsigned type:
  - If x is an unsigned type, the **>>** operator performs a ***logical shift*** in which missing digits are always filled with 0s.
  - If x is a signed type, the **>>** operator performs what computer scientists call an ***arithmetic shift*** in which the leading bit in the value of x never changes. Thus, if the first bit is a 1, the **>>** operator fills in 1s; if it is a 0, those spaces are filled with 0s.
- Arithmetic shifts are efficient ways to perform multiplication or division of signed integers by powers of two.

Two’s Complement

- Two's complement is a mathematical operation on binary numbers, and a method of signed number representation.
- The two's complement of an N-bit number is defined as its complement with respect to 2^N, i.e., the sum of a number and its two's complement is 2^N.
- Conveniently, another way of finding the two's complement is inverting the digits and adding one.
- 3-bit and 8-bit signed integers:

0	000
1	001
2	010
3	011
−4	100
−3	101
−2	110
−1	111

0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
−128	1000 0000
−127	1000 0001
−126	1000 0010
−2	1111 1110
−1	1111 1111

Exercise: Bitwise Operators

- What are the outputs:

```
int a = 5;
int b = 10;
cout << (a&&b) << ' ' << (a&b) << endl;
```

```
1 0
```

```
cout << hex << -1 << ' '
<< (-1 << 1) << ' '
<< (-1 >> 1) << ' '
<< unsigned(-1) << ' '
<< (unsigned(-1) << 1) << ' '
<< (unsigned(-1) >> 1) << endl;
```

```
ffffffff fffffffe ffffffff ffffffff fffffffe 7fffffff
```

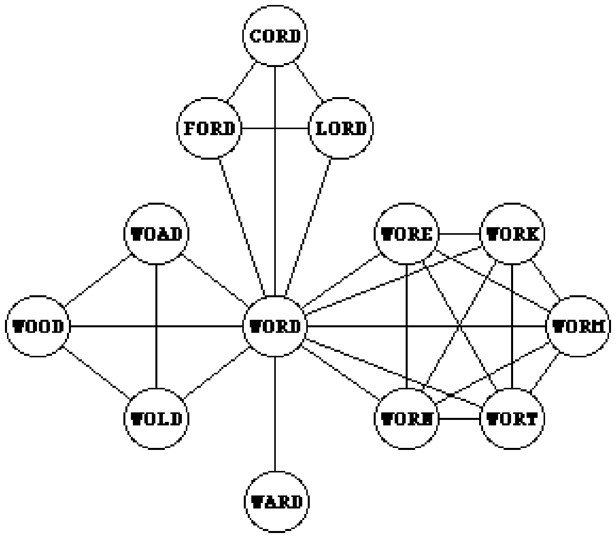
```
-1 -2 -1 4294967295 4294967294 2147483647
```

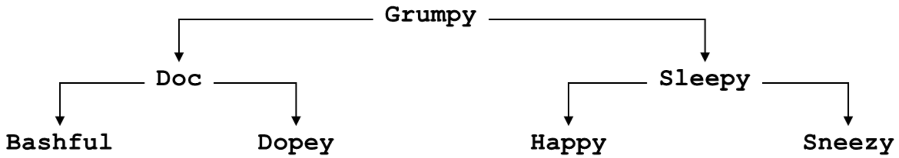
Largest unsigned integer in 32-bit machines, 2^{32}-1.

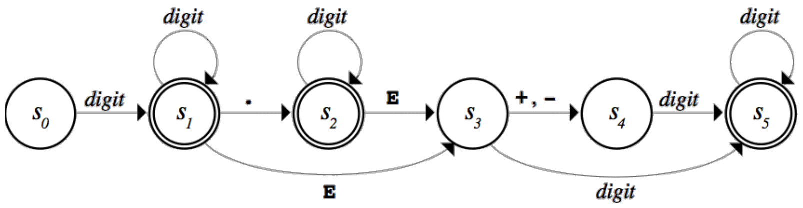
Largest positive integer in 32-bit machines, 2^{31}-1.

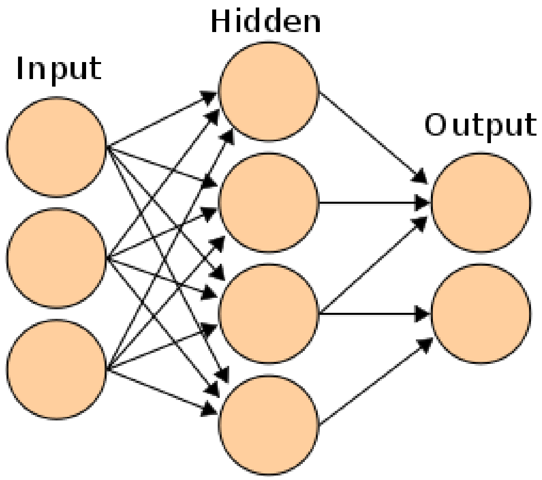
1. `-1`: In binary (32-bit), -1 is represented as all 1s (11111111111111111111111111111111)
2. `(-1 << 1)`: Left shift -1 by 1 bit
    - Result is -2 (shifts all bits left once)
3. `(-1 >> 1)`: Right shift -1 by 1 bit
    - Result is still -1 (sign bit is preserved in arithmetic right shift)
4. `unsigned(-1)`: Convert -1 to unsigned int
    - Result is 2^32 - 1 (FFFFFFFF in hex) = 4294967295
    - This is the largest possible unsigned 32-bit integer
5. `unsigned(-1) << 1`: Left shift the unsigned value by 1
    - Result is 4294967294 (FFFFFFFE in hex)
6. `unsigned(-1) >> 1`: Right shift the unsigned value by 1
    - Result is 2147483647 (7FFFFFFF in hex)
    - This is the largest possible signed 32-bit integer (2^31 - 1)

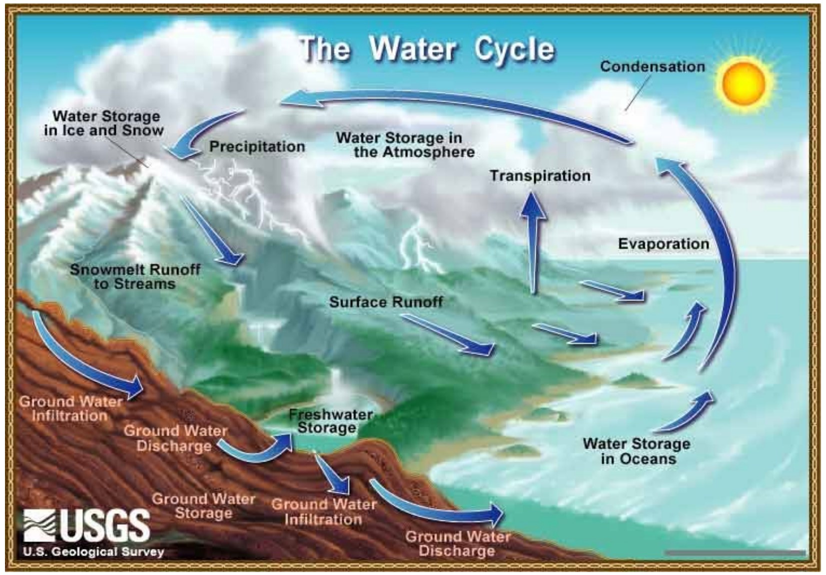
Examples of Graphs





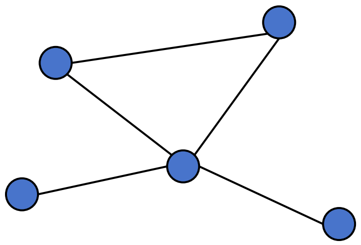




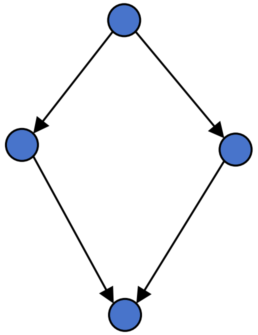


The Definition of a Graph

- A *graph* consists of a set of *nodes* together with a set of *arcs*. The nodes correspond to the dots or circles in a graph diagram (which might be cities, words, neurons, or who knows what) and the arcs correspond to the links that connect two nodes.
- In some graphs, arcs are shown with an arrow that indicates the direction in which two nodes are linked. Such graphs are called *directed graphs*.
- Other graphs, arcs are simple connections indicating that one can move in either direction between the two nodes. These graphs are *undirected graphs*.



Undirected Graph



Directed Graph

BFS

```
Graph:          A
               / \
              B   C
             / \ / \
            D  E F  G
           /  /
          H   I
```

```
Queue (BFS):
Initial:     [A]
After A:     [B,C]         # Level 1
After B:     [C,D,E]       # Level 1,2
After C:     [D,E,F,G]     # Level 2
After D:     [E,F,G,H]     # Level 2,3
After E:     [F,G,H,I]     # Level 2,3
After F:     [G,H,I]       # Level 2,3
After G:     [H,I]         # Level 3
After H:     [I]           # Level 3
After I:     []
Order: A -> B -> C -> D -> E -> F -> G -> H -> I
```

DFS

```
Graph:        A
             / \
            B   C
           / \ / \
          D  E F  G
         /  /
        H  I
```

```
Stack (DFS):
Initial:    [A]
After A:    [C,B]        # A's children
After C:    [G,F,B]      # C's children
After G:    [F,B]        # G has no children
After F:    [B]          # F has no children
After B:    [E,D]        # B's children
After E:    [I,D]        # E's child
After I:    [D]          # I has no children
After D:    [H]          # D's child
After H:    []           # H has no children
Order: A -> C -> G -> F -> B -> E -> I -> D -> H
```

Online Demo

- https://visualgo.net/en/dfsbfs