19: compression / encoding

Binary Files

The basic types of files are binary files and text files.

Text File I/O in C++:

```
// text_io.cpp - download here

#include <fstream>

int main(int argc, char * argv[]){

  std::ofstream fout("output.txt");
  fout << "hello world" << std::endl;
  fout.close();

  //output.txt has:
  //hello world

  //or using a hex editor
  //68 65 6C 6C 6F 20 77 6F 72 6C 64 0A
  return 0;
}
```

Binary File I/O in C++:

```
// binary_io.cpp - download here

#include <fstream>

int main(int argc, char * argv[]){

  std::ofstream fout("output.bin", std::ios::binary);
  int array[4];
  for(int i = 0; i < 4; ++i){
    array[i] = i + 1;
  }
  fout.write((const char *)array, sizeof(int) * 4);
  fout.close();

  //output.bin has:
  //

  //or viewing using a hex editor
  //01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
  return 0;
}
```

Binary Techniques

```
// binary_techniques.cpp - download here


int main(int argc, char * argv[])
{
  int num = 0x12345670;

  num &= 0x0000ffff;
  //num now is equal to 0x00005670

  num |= 1;
  //num now is equal to 0x00005671

  num <<= (4 * 4);
  //num now is equal to 0x56710000

  return 0;
}
```

Run-length Encoding

Run-length encoding encodes runs of the same character as a length and the character:

- AAAABBBB can be compressed as 4A4B

Encoding using a binary format is a way to make sure you never accidentally get confused

- For instance, 11111111111544444 = 1111554 (11 one's, 1 five and 5 fours)

Or you can have a table at the beginning that stores fixed length sizes

- For instance, 11111111111544444 = 011001005-154 (11 one's, 1 five and 5 fours)

Huffman Coding

Huffman Coding was invented by David Huffman while a PhD student at MIT in 1952 [1]. It is a variable length coding. The algorithm is as follows:

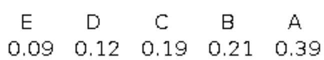
```
algorithm huffman()
  for each symbol create a tree with a single root node and order all
    trees according to the probability of symbol occurance;
  while more than one tree is left
    take two trees, (t_1 and t_2) with the lowest probabilities (p_1 <= p_2)
      and create a new_tree with t_1 and t_2 as the children and then with
      a new_root of probability p_1 + p_2;
  associate 0 with each left branch and 1 with each right branch;
  create a unique codeword for_each symbol by traversing the tree from the
    root to the leaf;

```

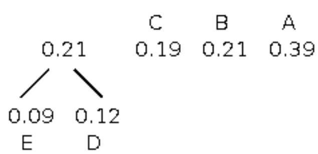
The basic Huffman Coding algorithm needs to know the probability that each input symbol will occur. English language letters have a certain probability, so this can be hard coded. Or you can calculate the probability for a certain file and transmit a table with the file.

Example from [5]:

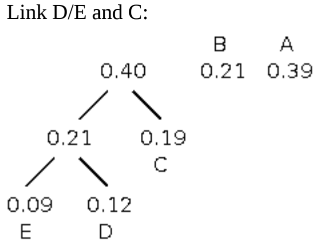
- Start:



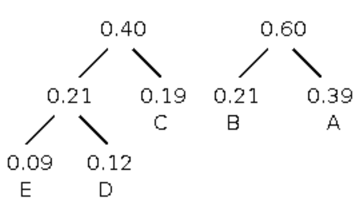
- Link D and E:



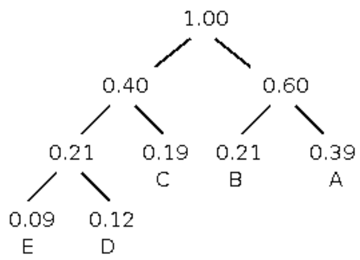
- Link D/E and C:



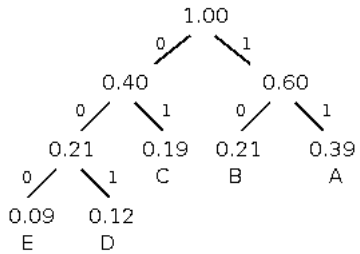
- Link A and B:



- Link C/D/E and A/B:



- Letter left nodes 0 and right nodes 1:



Example of compressing TTTTTSSRQP with huffman:

Letter	Code
T	1
S	01
P	001
Q	0001
R	0000

Bytes: 11111010 10000000 1001---- (3 bytes)

The same thing with run-length:
Characters: 5T2S1R1Q1P

Bytes: 00000101 01010100 00000010 01010011 00000001 01010010 00000001 01010001 00000001 01010000

Basic run-length takes 10 bytes (the original was 10 bytes).

Ziv-Lempel Code

LZ77 is a method that keeps two buffers of characters, one of things that have been sent and one of things that need to be sent.

1. First the buffer is initialized to contain 8 copies of the first letter (buffer size is 8). Then the first character is sent and the number of duplicates.

Input	**Next Buffer**	**Code Transmitted**
aababacbaac	aaaa aaaa	2a

2. Then a triple is sent <0, 0, b> representing the position in the buffer that a match was found, the length of the match, and the first character mismatching.

Input	**Next Buffer**	**Code Transmitted**
babacbaac	aaaa aaab	00b

3. Then starting at position 1, there is a two character match followed by an a.

Input	**Next Buffer**	**Code Transmitted**
abacbaac	aaaa aaba	12a

4. Then there is a zero character match of c

Input	Next Buffer	Code Transmitted
cbaac	aaaa abac	00c

5. Then there is a two character match starting at position 2, followed by an a.

Input	Next Buffer	Code Transmitted
baac	aaba cbaa	22a

6. Then there is one character match at 3, followed by a terminate.

Input	Next Buffer	Code Transmitted
c	abac baac	31-

LZW method

Compression:

```
algorithm LZWCompress()
  enter all letters to the table;
  initialize string s to the first letter form input;
  while any input left
    read character c;
    if s+c is in the table
      s = s+c
    else
      output codeword(s);
      enter s+c into the table;
      s = c;
  output codeword(s);
```

Decompression:

```
algorithm LZWdecompress()
  enter all letters to the table;
  read prior_code_word and output one character corresponding to it;
  while codewords are left
    read codeword;
    if codeword is not in the table
      enter in table string(prior_code_word) + firstchar(string(prior_code_word));
      output string(prior_code_word) + firstchar(string(prior_code_word));
    else
      enter in table string(prior_code_word) + firstchar(string(codeword));
      output string(codeword);
    prior_code_word = codeword;
```

Output of compression:

```
compressing: aababacbaac
match: 0 for: a
adding symbol: aa at 26
match: 0 for: a
adding symbol: ab at 27
match: 1 for: b
adding symbol: ba at 28
match: 27 for: ab
adding symbol: aba at 29
match: 0 for: a
adding symbol: ac at 30
match: 2 for: c
adding symbol: cb at 31
match: 28 for: ba
adding symbol: baa at 32
match: 30 for: ac
adding symbol: ac at 33
```

DEFLATE [6]

One of the most popular file compression algorithms is DEFLATE.

Each block is one of three formats:

Header Identifier	Description
00	Raw bytes
01	Huffman Compressed Block with pre-agreed table
10	Huffman Compressed Block with table given

Once the block is made, LZ77 compression is used to remove repeated duplicates in the block.

Base64-encoding

Base64-encoding is not a compression, but an encoding that lets you transfer binary data over a text channel.

- Original Text:

  Items can be inserted in the middle of a singly linked list without moving all the remaining elements over.

- Base64 Encoded:

  SXRlbXMgY2FuIGJlIGluc2VydGVkIGluIHRoZSBtaWRkbGUgb2YgYSBzaW5nbHkgbGlua2VkIGxp
  c3Qgd2l0aG91dCBtb3ZpbmcgYWxsIHRoZSByZW1haW5pbmcgZWxlbWVudHMgb3Zlci4=

A byte stream is encoded by grouping together tokens as 6 bits and then translated to base64 using a lookup table.

Padding characters of '=' are added if the last group does not have 3 input characters.

Text Content	I	t	e	m	s	(space)
ASCII	73	116	101	109	115	32
Bit pattern	0100 1001	0111 0100	0110 0101	0110 1101	0111 0011	0010 0000

6bits per token	0100 10	01 0111	0100 01	10 0101	0110 11	01 0111	0011 00	10 0000
Index	18	23	17	37	27	23	12	32
Base64-encoded	S	X	R	l	b	X	M	g

Base64 index table (from wikipedia):

Value	**Char**	**Value**	**Char**	**Value**	**Char**	**Value**	**Char**
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Base64 is to transmit email attachments in the MIME specification (Multipurpose Internet Mail Extensions).

It has many many other uses (for instance the grading server uses it to convert your binary formatted zip to Base64 to transmit over a text based socket to a server application in a virtual machine).

Checksums

Checksums can be used to make sure a file or message transfered over the internet was no corrupted.

The simplest way to compute a checksum is to let an integer overflow:

```
// checksumOverflow.cpp - download here

#include <iostream>
#include <fstream>

void computeChecksum(std::string filename)
{
  char checksum = 0;
  char prev_checksum = 0;
  int rollovers = 0;
  std::ifstream fin(filename.c_str(), std::ios::binary);
  while(fin.good()){
    char c;
    fin.get(c);
    checksum += c;
    if(checksum < prev_checksum){
      rollovers++;
    }
    prev_checksum = checksum;
  }
  std::cout << "checksum for " << filename << ": " << (int) checksum;
  std::cout << " (" << rollovers << " rollovers)" << std::endl;
}

int main(int argc, char * argv[]){

  computeChecksum("output.bin");
  computeChecksum("output.txt");

  //prints:
  //  checksum for output.bin: 10 (0 rollovers)
  //  checksum for output.txt: 112 (4 rollovers)
  return 0;
}
```

Position dependent hashes

The MD5 and SHA-1 hashes are often used to ensure integrity when downloading a file or program. With these hashes, a small difference in the file produces a very large difference in the hashes (or checksums).

A software publisher will display the hash for a file on a website, and after downloading, a user can verify the hash matches.

Here are some MD5 hashes for ubuntu iso's

MD5 Hash	File
5e427f31e6b10315ada74094e8d5d483	ubuntu-11.10-alternate-amd64.iso
24da873c870d6a3dbfc17390dda52eb8	ubuntu-11.10-alternate-i386.iso
62fb5d750c30a27a26d01c5f3d8df459	ubuntu-11.10-desktop-amd64.iso
c396dd0f97bd122691bdb92d7e68fde5	ubuntu-11.10-desktop-i386.iso
f8a0112b7cb5dcd6d564dbe59f18c35f	ubuntu-11.10-server-amd64.iso
881d188cb1ca5fb18e3d9132275dceda	ubuntu-11.10-server-i386.iso

These can be checked with a program that recomputes the hash from what was downloaded.

References

1. http://en.wikipedia.org/wiki/Huffman_coding
2. http://en.wikipedia.org/wiki/Base64

3. http://www.motobit.com/util/base64-decoder-encoder.asp
4. https://help.ubuntu.com/community/UbuntuHashes
5. Adam Drozdek, "Data Structures and Algorithms in C++". Fourth Edition. Page 594.
6. http://en.wikipedia.org/wiki/DEFLATE
7. http://softsurfer.com/Archive/algorithm_0103/algorithm_0103.htm