Lecture 21

2's complement numbers

C++ additions

Complexity

Admin

- I'll have office hours from 12:45-4:30.
  - I'll be leaving at 4:30 **sharp**
- Brad's hours moved to 3:30-5:30pm
- Office hours don't restart until the Monday after break.
  - I'll likely be around Friday afternoon during break.

Today

- 2's complement numbers
  - How to compute sign and magnitude
  - How to negate.
- Additional C++ syntax
  - Unsigned, switch statements, bool
- Algorithm complexity

2's complement numbers

- A way to represent signed (+/-) numbers.
  - Leftmost bit is a sign bit.
  - For its value, treat as a binary number, but the last place (MSB) is negated.

1	0	1	0	1	0
-32	16	8	4	2	1

So the above value is -32 + 8 + 2 = -22

2's complement

- To negate a 2's complement number you can just invert all the bits and add 1.
  - 101010 → 010101+1 = 010110 =16+4+2=22
- Practice
  - Find the values of the following 6-bit 2's complement numbers:
    - 101000
    - 001010
    - 111111

Sign extension

- If you want to convert an n-bit 2's complement number to a larger representation you can't just tack on zeros to the end.
  - That would change sign.
- It turns out you can just "extend" the MSB.
  - 000 → 00000
  - 111 → 11111
  - 010 → 00010
  - 100 → 11100
- Why does this work?

Misc. C++ syntax

- First of all, some variables can be declared as unsigned
  - Char and int
  - This means the 8 or 32 bits aren't treated as a 2's complement number
    - Instead just a normal binary number.
  - This can be useful if you are playing with bits or if you have values that can't be negative.
  - It does extend the range of representation a bit, but that usually isn't too helpful.

Examples of unsigned

- unsigned int bob;
- unsigned char mary;

Switch

```
switch(number)
{
    case 1:
    case 2:
    case 3:
            cout << "Low ball" << endl;
            break;
    case 4:
            cout << "Nice number" << endl;
            break;
    case 5:
            cout << "A bit high or ";
    case 6:
            cout << "Maybe way high" << endl;
            break;
    default:
            cout << "I think not" << endl;
}
```

Rules of the switch

- The labels must be **constants**.
- The code continues until a **break**.

Other switch stuff

- I personally dislike switch statements
  - Because the don't handle ranges or variables they are only occasionally useful.
  - If you forget a break things get broken quickly.
  - Nested if/else statements can do the same thing.
- I use them, but only rarely.

Algorithm complexity

- I've been emphasizing that computers are generally "fast enough"
  - To an extent, this is a lie.
    - Think about how often you are waiting for a computer to do ***something***.
    - Maybe logging in, compiling, whatever.
  - Further, bad algorithms can lead to code that is **much** slower than it should be

Input size

- In general it will take longer to perform an algorithm if there is more data as part of the input.
  - As such we generally measure complexity in terms of input size.
- Consider a sorting algorithm.
  - The size of the input is the number of elements to be sorted.

Algorithm complexity

- In general we measure an algorithm's complexity by how the run time is related to the input size.
  - Consider selection sort.
    - For i=I to n
      - Find smallest
      - Copy it to new array
      - Mark old array element as used
  - What is the complexity in terms of n?

Again

- How about bubble sort?

More examples: match from HW2

- This function takes two *sorted* lists, both of which have exactly "size" elements. The lists have the following properties:
  - They are sorted with the smallest value found at index 0.
  - Neither list will have repeated values. (The same value can't occur twice in the same list)
- The function is to return the number of elements shared by the two lists.

A tale of two algorithms

- Compare all pairs
- Walk both lists

So who cares?

- As engineers you will often be working with very large data sets
  - Say 20,000,000 elements.
  - Say for match you can do a comparison in 1 billionth of a second (which is about right on a good day)
    - The smart algorithm will take around $2x10^7/1x10^9$ seconds or 0.02 seconds
    - The n squared algorithm will take $(2x10^7)^2/1x10^9$ or $4x10^5$ seconds which is a bit more than 4 days.

Consider Google

- 8,058,044,651 web pages.
  - Algorithms better be sub-linear…