What is Computer Science?

What is it that distinguishes it from the separate subjects with which it is related? What is the linking thread which gathers these disparate branches into a single discipline? My answer to these questions is simple --- *it is the art of programming a computer.* It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex.

C.A.R. (Tony) Hoare

---

Why is programming fun?

What delights may its practitioner expect as a reward?

First is the sheer joy of making things

Second is the pleasure of making things that are useful

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts

Fourth is the joy of always learning

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff.

Fred Brooks

---

Efficient Programming

- Designing and building efficient programs efficiently requires knowledge and practice
  - Hopefully the programming language helps, it’s not intended to get in the way
  - Object-oriented concepts, and more general programming concepts help in developing programs
  - Knowledge of data structures and algorithms helps

- Tools of the engineer/scientist/programmer/designer
  - A library or toolkit is essential, STL or wheel re-invention?
  - Programming: art, science, engineering? None or All?
  - Mathematics is a tool
  - Design Patterns are a tool

---

Course Overview

- Lectures, Recitations, Quizzes, Programs
  - Recitation based on questions given out in previous week
    - Discuss answers, answer new questions, small quiz
    - More opportunities for questions to be answered.
  - Lectures based on readings, questions, programs
    - Online quizzes used to motivate/ensure reading
    - In-class questions used to ensure understanding
  - Programs
    - Theory and practice of data structures and OO programming
    - Fun, practical, tiring, ...
    - Weekly programs and longer programs

- Exams/Tests
  - Semester: closed book
  - Final: open book
Questions
- If you gotta ask, you’ll never know
  - Louis Armstrong: “What’s Jazz?”
- If you gotta ask, you ain’t got it
  - Fats Waller: “What’s rhythm?”
- What questions did you ask today?
  - Arno Penzias

Tradeoffs
- This course is about all kinds of tradeoffs: programming, structural, algorithmic
  - Programming: simple, elegant, quick to run/to program
    - Tension between simplicity and elegance?
  - Structural: how to structure data for efficiency
    - What issues in efficiency? Time, space, programmer-time
  - Algorithmic: similar to structural issues
- How do we decide which choice to make, what tradeoffs are important?

See readwords.cpp
- This reads words, how can we count different/unique words?
  tvector<string> list;
  string filename, word;
  cin >> filename;
  ifstream input(filename.c_str());
  CTimer timer;
  timer.Start();
  while (input >> word) {
    list.push_back(word);
    }
  timer.Stop();
  cout << "read " << list.size() << " words in ";
  cout << timer.ElapsedTime() << " seconds" << endl;

Tracking different/unique words
- We want to know how many times ‘the’ occurs
  - Do search engines do this? Does the number of occurrences of “basketball” on a page raise the priority of a webpage in some search engines?
    - Downside of this approach for search engines?
- Constraints on solving this problem
  - We must read every word in the file (or web page)
  - We must search to see if the word has been read before
  - We must process the word (bump a count, store the word)
  - Are there fundamental limits on any of these operations?
  Where should we look for data structure and algorithmic improvements?
Search: measuring performance

- How fast is fast enough?

```cpp
bool search(const tvector<string> & a, const string & key) {
    // pre: a contains a.size() entries
    // post: return true if and only if key found in a
    int k; int len = a.size();
    for(k=0; k < len; k++)
        if (a[k] == key) return true;
    return false;
}
```

- C++ details: parameters? Return values? Vectors?
- How do we measure performance of code? Of algorithm?
  - Does processor make a difference? PIII, G4, ???

Tradeoffs in reading and counting

- Read words, then sort, determine # unique words?
  - frog, frog, frog, rat, tiger, tiger, tiger, tiger
- If we look up words as we’re reading them and bump a counter if we find the word, is this slower than previous idea?
  - How do we look up word, how do we add word
- Are there kinds of data that make one approach preferable?
  - What is best case, worst case, average case?
- What’s one function spec & implementation to count # unique words in a list/vector of words
  - See readwords3.cpp

Who is Alan Perlis?

- It is easier to write an incorrect program than to understand a correct one
- Simplicity does not precede complexity, but follows it
- If you have a procedure with ten parameters you probably missed some
- If a listener nods his head when you’re explaining your program, wake him up
- Programming is an unnatural act
- Won first Turing award

[Link to quote page](http://www.cs.yale.edu/homes/perlis-alan/quotes.html)

Review/Preview: Anagrams/Jumbles

- Brute-force approach to finding anagrams/solving Jumbles
  - Brute-force often thought of as “lack of thought”
  - What if the better way requires too much thought?
  - What if there’s nothing better?
- nelir, nelri, nelir, neri, nelir, nelir, nel, nleri, nlier, nleri, nleir, nleri, nler, nleri, nbr, nleri, nleri, nleri, nleri, ... lenir, lenri, leinr, leirn, lerin, lerin, lerin, lerin, liner
- What’s the problem here?
- Is there a better method?
Brute force? permanc.cpp

```
// find anagram of word in wordSource
// list is a vector [0, 1, 2, ..., n]
Permuter p(list);
int count = 0;
string copy(word);// makes copy the right length

for(p.Init(); p.HasMore(); p.Next()) {
    p.Current(list);
    for(k=0; k < list.size(); k++) {
        copy[k] = word[list[k]];
    }
    if (wordSource.contains(copy)) {
        cout << "anagram of " << copy << endl;
        break;  // find first anagram only
    }
}
```

Quantifying brute force for anagrams

- On one machine make/test a word takes $10^{-5}$ seconds/word
  - 9! is 362,880: how long does this take?
  - What about a ten-letter word?

- We’re willing to do some pre-processing to make the time to find anagrams quicker
  - Often find that some initialization/up-front time or cost saves in the long run
  - We need a better method than trying all possible permutations
  - What properties do words share that are anagrams?

Toward a faster anagram finder

- Words that are anagrams have the same letters; use a letter fingerprint or signature/histogram to help find anagrams
  - Count how many times each letter occurs:
    - “teacher” 1 0 1 0 2 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
    - “cheater” 1 0 1 0 2 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
  - Store words, but use fingerprint for comparison when searching for an anagram
    - How to compare fingerprints using operator ==
    - How to compare fingerprints using operator <

- How do we make client programmers unaware of fingerprints? Should we do this?

Another anagram method

- Instead of fingerprint/histogram idea, use sorted form of word
  - “gable” and “bagel” both yield “abegl”
  - Anagrams share same sorted form

- Similarities/differences to histogram/fingerprint idea?
  - Both use canonical or normal/normalized form
  - Normalized form used for comparison, not for printing
  - When should this normal form be created?

- When is one method preferred over the other?
  - Big words, little words? Different alphabets? DNA vs English?
OO and C++ features we’ll use

- We’ll use an adapter or wrapper class called Anaword instead of a string
  - Clients can treat Anaword objects like strings, but the objects are better suited for finding anagrams than strings
  - The Anaword for “bear” prints as “bear” but compares to other Anaword objects as \texttt{110010000000000001000000000}
- C++ allows us to overload operators to help, not necessary but good cosmetically
  - Relational operators \texttt{==} and \texttt{<}
    - What about other operators: \texttt{>, <=, >=, and !=}\n  - Stream operator \texttt{<<}
- How should we implement overloaded operators?

Overloaded operators

- In C++ we can define what operator \texttt{==} and operator \texttt{<} mean for an object (and many other operators as well)
  - This is syntactically convenient when writing code
  - C++ details can be cumbersome (see Tapestry Howto E)
- In \texttt{anaword.h} there are four overloaded operators
  - What about \texttt{> and >=; what about !=; others?}
  - What about printing, can we overload operator \texttt{<<}\?<
  - How do we access private data for printing? Comparing?
- Overloaded operators are \textbf{not} necessary, \textit{syntactic sugar.}

Overloaded operators (continued)

- Typically operators need access to internal state of an object
  - Relational operators for Date, string, BigInt?
  - Where is “internal state”?
- For technical reasons sometimes operators should not be member functions:
  \begin{verbatim}
  BigInt b = enterBigValue();
  if (b < 2) …
  if (2 > b) …
  \end{verbatim}
  - We’d like to use both if statements, only the first can be implemented using \texttt{BigInt::operator < (why?)}
- Use helper member functions: \texttt{equals, less, toString}
  - Implement overloaded operators using helpers

Anaword objects with options

- Can we use different canonical forms in different contexts?
  - Could have Anaword, FingerPrintAnaword, SortAnaword
  - What possible issues arise? What behavior is different in subclasses?
    - If there’s no difference in behavior, don’t have subclasses
- Alternative, make canonical/normalize method a class
  - Turn a function/idea into a class, then let the class vary to encapsulate different methods
  - Normalization done at construction time or later
  - Where is normalizer object created? When?