The Appliance of Science: Things in Computer Science that Every Practitioner Should Know

Andrei Alexandrescu

Please Note

- Built 100% using OpenOffice 2.0 on Linux
- OpenOffice makes you love PowerPoint
- Linux makes you love putting up with OpenOffice

Agenda

- Generic algorithm basics
- Big-O notation
- How to append in amortized constant time
- Lambda Functions
- Memory Barriers
- Exception Safety and Transactional Semantics

3

Please keep your seats! Come back from the door!

That was the agenda in 1997

4

2007 Agenda

- Dynamic Programming
- Immutability vs. Aliasing vs. Type Changing
  - GC is not all it's cracked to not be
- Machine Learning
  - The Smoothness Principle
- Transactional Memory

Dynamic Programming

Rationale: Because we might still think `string::find` is the ultimate solution

Dynamic Programming

- Hint: Has nothing to do with
  - Dynamic memory allocation
  - Writing code
- "Programming" = Creating a plan of action
- "Dynamic" = The plan is built from the problem (*dynamically*), not once for all problems (statically)
- "Data-dependent planning"

7

Dynamic Programming

- Solves optimization problems
- Can make an exponential into a polynomial (!)
- Principle of Optimality:
  - An optimal solution to the problem embeds optimial solutions to its subproblems
- Solutions to subproblems must overlap
- Example: longest common subsequence
  - Yes: L(a1, a2, n) includes L(a1, a2, n – 1)
- Example: longest subarray of digits
  - No: no overlapping => no saving work

8

Fibonacci

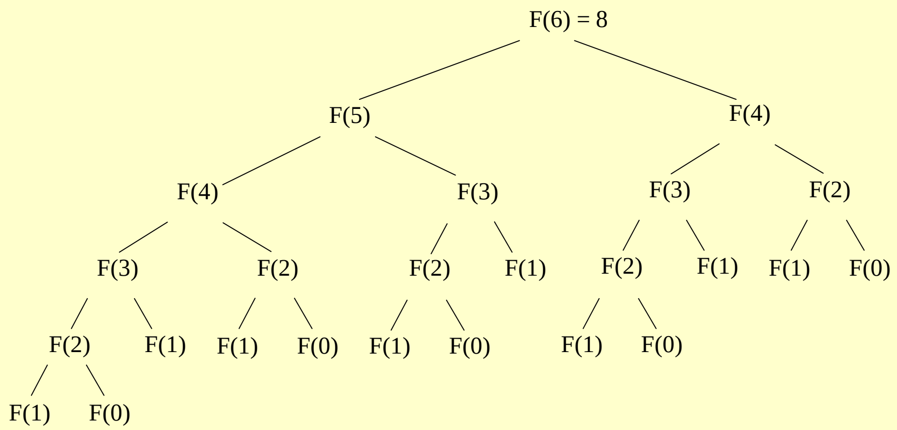
- An absolute classic:

```
unsigned Fib(unsigned n) {
  return n <= 1 ? 1 : Fib(n - 1) + Fib(n - 2);
}
```

- Probably the 2nd best way to turn people away from recursion
- The number of adds is a Fibonacci series itself
  - Exponential!

Fibonacci

- Key pessimization: evaluating Fib(n) evaluates the same overlapping subproblems many times



Fibonacci w/ memoization

- Memoize intermediate results

```
unsigned Fib(unsigned n) {
  vector<unsigned> a(2, 1u);
  for (auto i = 2u; i <= n; ++i)
    a.push_back(a[i – 1] + a[i – 2]);
  return a.back();
}
```

- Turns exponential into O(n)
  - Or even O(1) if we save results between calls
- Memoization is a common DP technique

11

Levenshtein Distance

- Given two strings, figure the minimum number of insertions, deletions, replacements that make one into the other
  - Save typing when fixing typos
  - Spell checkers
  - Protein classification
  - Approximate searching in various databases
- It's an optimization problem
  - Simple solution: delete all original string, replace with all other string
  - But pay £1 per character edited

12

Levenshtein distance

- Idea: LD(s, t) uses LD(s[0..$-1], t) and LD(s, t[0..$-1])
- Fill a 2D cost matrix d[s.size()+1][t.size()+1]
  - d[a][b] is the cost for s.substr(0, a), t.substr(0, b)
- We know what d[0][k] is
- We know what d[k][0] is
- We grow d "around the diagonal"

Implementation

```
int LD(string s, string t) {
  vector<vector<int> > d(s.size() + 1, vector<int>(t.size() + 1);
  for (auto i = 0u; i <= s.size(); ++i) d[i][0] = i;
  for (auto i = 0u; i <= t.size(); ++i) d[0][i] = i;
  for (auto i = 0u; i < s.size(); ++i)
    for (auto j = 0u; j < t.size(); ++j) {
      d[i+1][j+1] := min(min(
                           d[i][j+1] + 1,               // deletion
                           d[i+1][j] + 1),              // insertion
                           d[i][j] + (s[i] != t[j]));   // substitution
      }
    return d[m][n];
}
```

Immutability, Aliasing, Type Changing

Rationale: Because garbage collection is not all it's cracked up to not be

15

What's with all three?

- Principle: In a type system, given:
  - data mutability
  - aliasing
  - retyping memory (= delete)
- You can't have all three and be safe.
- Corollary: Garbage collection makes it safe for you to mutate aliased data!?!

16

Mutability

- In a system with mutability, you can rebind symbols to values:

```
auto a = new Widget;
if (intractable_condition) a = b;
else a = 0;
```

- In a system with immutability (functional languages), all symbols can't be rebound
- Key observation: in mutable systems the compiler is unable to track the web of references

17

Aliasing

- Aliasing = the ability to have multiple symbols referring to the same object
- Can't free when symbols go out of scope
  - 100% fine in a value-semantics language
- Key observation: can't freely mutate data referred by a symbol
  - other symbols might refer to the same data

18

Memory retyping

- = The ability to pretend that some memory of a type is "from now on" of another type
- A fancy name for `free`
  - `free` is at some later point followed by `new`
  - Ultimately `new` will reuse the memory
- Defining per-type heaps is onerous
  - Also misses the point
- Key observation: you can't retype memory if you have intractable aliases to it

19

Putting Them Together

- You must give up one:
  - Aliases: because they refer the same memory from various parts of the program
  - Mutability: because it makes aliases intractable
  - Memory renaming: because it can't deal with intractable aliases!
- Corollary: Garbage collection deeply connected to fundamental aspects of type safety
  - Far cry from "...just allows incompetent programmers a sense of entitlement"

20

More on GC

- Typical dialog:
  - *Gino:* "Since our company started using GC, speed was up 15%, productivity was up 30%, bugs were resolved 20% faster, code size was 10% smaller, and birth rate was 3% up"
  - *Dino:* "Oh yeah? When our company tried GC, speed was down 16%, productivity was down 31%, bugs were resolved 21% slower, code size was 11% larger, and sterility rate was... oh, never mind."
- We should stop relying on anecdotes (alone)

Hertz & Berger's Tests

- Simulate running the same program on the same inputs:
  - On a GC system; collect the trace
  - On an "oracle" system that manages memory perfectly
    - N.B. This is not always possible
- Simulation on an architecturally-detailed simulator
  - Allows e.g. simulating impact on locality
- No refcounting used
  - Paper mentions 2x slowdown due to smart pointers
- Realistic programs: compress, ray tracing, database, compiler, expert system...

Results

- 5x available memory: < 9% performance *improvements*
- 3x available memory: < 17% performance degradation
- 2x available memory: 70% performance degradation (heavy swapping)
- 1x available memory: 90% performance degradation
- Your RAM and your program's data size?
  - Do the math

Machine Learning

Rationale: Because we must process more data than ever before

Machine Learning

- Traditional programs:
  - Reflect *our* intelligence, knowledge, and skill
  - Good on little data and good prior understanding
- Machine learning programs:
  - Know how to learn; build their own model of reality
  - Good on much data and little prior understanding
- Examples:
  - Simple medical diagnosis
  - Natural language parsing
  - Stock market prediction

25

Machine Learning

- The system learns a function $f : X \rightarrow Y$
  - Inputs X are “features” = “data that might help”
  - Outputs Y are (usually) “labels”
- *Not* heuristics! We don't *know* the function!
- Example: “Is this image a human face?”
  - Features: image pixels
  - Labels: true or false
- Feature preprocessing an important step
  - Infer face contour from pixels first

26

More Examples

- “Will this page layout lead to more sales?”
  - Amazon, Yahoo, Google routinely experiment with data
- “What is the meaning of this word?”
  - Disambiguation is an AI-complete task
- “Who said that?”
  - Speaker recognition
- “Is this C++ code good or bad?”
  - Code quality classification

27

The Smoothness Assumption

- If inputs vary just a little, outputs vary just a little
- Applies to most natural data
- Group together similar features
  - Nearest-neighbors techniques
- Draw a separator to maximize margin
  - Support vector machines
- Compute a smooth function
  - Multi-layer perceptron (= neural net without the hype)

28

"What's in it for me?"

- You've got data
- You want to learn a function
- You don't know the function (complicated, hard to compute, intractable)
- Machine learning techniques learn the function for you

Transactional Memory

Rationale: Because in 15 years we might have a 1023-processor grid on our desks.

Transactional Memory

- We have more transistors than we can power on
  - The "Power Wall"
- We have more computing power than we can feed with data
  - The "Memory Wall" and the "ILP Wall"
- Current technologies cannot evolve > 8 CPUs
  - We need to deal with hundreds/thousands CPUs
- We're in desperate need for a solution
  - Scrumm *that*.

31

Synchronization Models

- Lock-based synchronization
  - "I see deadlocked threads"
- CAS-based programming
- Transactional Memory
- Full/Empty Bits in Memory
- None of the above is definitive
  - Transactional Memory is today the most promising

32

Transactional Memory

- Transaction = finite sequence of memory reads and writes executed by one thread
- Appear atomic to other processors
- Hardware: limited # of reads/writes
- Software: slow
- Virtual: fast in the common case

Example

```
void Widget::Allocate(unsigned n) {
  atomic {
    buffer_.resize(n);
    refresh();
  }
}
```

- Just replaced `synchronized` with `atomic`?!?
  - No! There's *no locking*
  - atomic blocks are composable
  - atomic is system-wide, no data-dependent locking

However

- Undefined changes outside atomic blocks
- Unclear how to implement I/O transactionally
- What to do about O/S calls within a transaction?
  - O/S API should change
- Resumption model unclear
  - Should automatically resume a failed transaction?
- All of the above are of less concern in functional languages

35

Transactional Memory redux

- To conclude:
  - Efficient code becomes important again
  - Concurrency knowledge comes to the forefront
  - Functional-style programming will recrudesce
  - Um, GC makes it easier too

36

Conclusions

- Dynamic Programming
- Immutability vs. Aliasing vs. Type Changing
- Machine Learning
- Transactional Memory

Famous last (coherent) words: **Questions?**

37