

Algorithms in the Industry

Kenneth Oksanen

`cessu@iki.fi`

14. Apr 2005

Outline

- Learn basics, and learn to innovate
- Some frequently used but rarely documented tricks
- Traps and pitfalls
- Practical performance
- Reduce “total cost of ownership”

The Basics

- Some basic algorithms shipped with your language (built-in or libraries) — in most new languages (Java, Perl, Python, ...) you'll never implement a hash-table or binary tree of your own!
 - But terminologies vary: Python's "list" is in fact an array with $O(N)$ append cost.
- Many basic algorithms available in various libraries
 1. Search in the Internet
 2. Check the license!
 3. Check the quality of the code, memory management policies, ...
 4. Write "glue" code and/or edit to suit your needs
- Sounds easy? It is harder and needs more knowledge than you think!

Learn to Innovate

- Usually there is no direct answer to your problem. A solution involves
 - a combination of several algorithms and data structures,
 - transforming your problem to another form,
 - understanding the problem's special characteristics,
 - other text-book methods of deriving algorithms (divide-and-conquer, dynamic programming),
 - or an arbitrary combination of these.

- Example 1: IPv4 route lookup
 - Keys are 32-bit IP addresses, find value for the given or largest smaller key
 - Can be done with binary trees, but a little sluggish
 - A 65536-entry table for the uppermost 16 bits, entries contain binary trees for the lowest 16 bits.
 - Even faster in practice if we cache the most frequently used routes.
 - Blindingly fast if the cache is implemented by special purpose hardware!

- Example 2: Split paragraphs into lines beautifully.
 - “Beauty” of a line can be estimated by measuring how condensed or stretched each line is and whether the line break is at a sentence, word, hyphen, or character boundary. The “beauty” of the paragraph is the sum of “beauties” of each line in it.
 - Transform the problem to finding the shortest path from node a to node b in a graph: node a is the start of the paragraph, node b is the end, all other nodes are possible line breaks, nodes are possible line breaks, and the length of an edge is the beauty measure of a line corresponding to the line breaks.
 - Apply well known graph algorithms to this new problem.
 - The resulting paragraph layout algorithm is used e.g. in $\text{T}_{\text{E}}\text{X}$ to produce these slides.
- (Many problems are isomorphic to some graph problem.)

- Example 3: Mines (not very industrial, but illustrative...)

- Which square is mine free?

	a	b	c
1	0	1	
2	1	2	
3			

- Stupid algorithm: 2^N -loop over all N empty squares, test if solutions satisfy known constraints.
- Better: transform into a Diophantes equation

$$\left\{ \begin{array}{l} x_{a,3} + x_{b,3} = 1 \\ x_{a,3} + x_{b,3} + x_{c,3} + x_{c,2} + x_{c,1} = 2 \\ x_{c,2} + x_{c,1} = 1 \end{array} \right.$$

and use algorithms presented in the literature.

- Example 4: Crossword puzzles
 - Enumerate all $N \times N$ crosswords for a given word set

```
static const char *word[] = {  
    "ace",  
    "act",  
    ...  
    "zoo",  
    NULL  
};
```

- Start off by writing $N \times N$ nested loops:

```
for (h0 = 0; word[h0] != NULL; h0++) {
    in_use[h0] = 1;
    for (v0 = 0; word[v0] != NULL; v0++)
        if (!in_use[v0] && word[h0][0] == word[v0][0]) {
            in_use[v0] = 1;
            for (h1 = 0; word[h1] != NULL; h1++)
                if (!in_use[h1] && word[h1][0] == word[v0][1]) {
                    ...
                }
            in_use[v0] = 0;
        }
    in_use[h0] = 0;
}
```

- Depressing running time $\mathcal{O}(c^{2N})$, where c is the number of words.
3 × 3-crosswords are found instantly, 4 × 4 crosswords in a day,
5 × 5-crosswords would take longer than sun shines.
- Avoid transposes, this halves the running time.

...

```
for (v0 = h0 + 1; word[v0] != NULL; v0++)
```

...

- Move checks earlier: test already when placing words whether the letters being placed next to each other occur in some word.
 - * 31.36% of all possible digraphs occur in some word.
 - * 8.05% of all possible trigrams occur in some word.
 - * < 1% of all possible 4-graphs occur in some word.
 - * Not all combinations occur in all places.

* For example, this can't be completed:

	v0		
h0	d	i	<u>e</u>
h1	a	n	<u>i</u>
	y		

And the only 6×6 solution (with my /usr/share/dict/words) is

a	c	c	u	s	e
p	r	o	p	e	l
h	a	n	d	e	d
i	n	v	a	d	e
d	i	e	t	e	r
s	a	y	e	r	s

Tricks of the Trade

Some frequently used tools rarely given in text books:

- Approximate and simplify the original problem. For example, approximate arbitrarily many priority levels with a few hundred, and replace a priority queue with a fixed size array of lists.
- Tries: trees that branch on a few most significant unused bits of the key.
 - Path compression: compress successive single-child nodes into one, store the corresponding key bits in the compressed node.
 - Width compression: get rid of NULL-pointers in the nodes.
 - Level compression: Merge successive levels of sparsely branching nodes into one.

- Software Caches: small fixed-size lossy hash-tables, to store frequently performed computations or accessed data of some slower data structure. Below we cache the accesses to a binary tree. KEY and value stand for some concrete types, and various comparisons will have to be changed accordingly.

```
typedef struct node_t {  
    KEY key;  
    VALUE value;  
    struct node_t left, right;  
} node_t;
```

```
#define CACHE_SIZE 64

struct {
    node_t *root;
    struct {
        KEY key;
        VALUE value;
    } cache[CACHE_SIZE];
} tree_t;
```

```
VALUE binary_tree_search(KEY k, tree_t *t)
{
    node_t n;
    unsigned long h = hash_fn(k) % CACHE_SIZE;

    /* Probe the cache */
    if (t->cache[h].key == k)
        return t->cache[h].value;
```

```
/* Not found in cache, search from the tree */
n = t->root;
while (n != NULL)
    if (n->key < k)
        n = n->right;
    else if (n->key > k)
        n = n->left;
    else {
        /* Found, put into cache and return. */
        t->cache[h].key = k;
        t->cache[h].value = n->value;
        return n->value;
    }
/* Not found. */
...
}
```

- Improve the hit rate by improving the replacement algorithm:

```
/* Probe the cache */
if (t->cache[h1].key == k) {
    t->cache[h1].hit_count++;
    return t->cache[h1].value;
}
if (t->cache[h2].key == k) {
    t->cache[h2].hit_count++;
    return t->cache[h2].value;
}
```

```
...
/* Found, put into cache and return. */
if (t->cache[h1].hit_count
    < t->cache[h2].hit_count) {
/* Store into t->cache[h1]. */
t->cache[h1].key = k;
t->cache[h1].value = n->value;
t->cache[h1].hit_count = 1; /* Or 2? */
t->cache[h2].hit_count--; /* Or /= 2? */
return n->value;
} else {
/* Store into t->cache[h2]. */
...
}
```

Micro-optimizations

- While algorithms and data structures best be planned ahead, micro-optimizations can often be done last, and only when needed.
- `int`, `char` or a single bit?
- Regrouping information, e.g. the order of variable definitions and fields in a record.
- Using more primitive code, machine idioms, or even hand-written assembly language.
- Loop unrolling, code specialization, rewriting expressions to allow more insn level parallelism, ...

- Benefits from negligible to significant
- Choices often machine-dependent
- Micro-optimizing caused often chaotic and non-intuitive changes in performance

(Simple) Traps and Pitfalls

- Assumptions on the size or distribution of input:
 - Linked lists with over hundred items
 - Quicksort with already ordered data
 - Arrays and hash tables which don't resize when needed
 - Too full hash tables

- Bad hash functions. Here is a relatively good one for C's zero-byte terminated strings:

```
unsigned long string_hash(char *s)
{
    unsigned long h = 0xDEADBEEF;

    while (*s) {
        h += *s++;
        h += h << 10;
        h ^= h >> 6;
    }
    h += h << 3;
    h ^= h >> 11;
    h += h << 15;
    return h;
}
```

- Floating point hazards:
 - Exact equality comparisons
 - Rewrite e.g. $x^2 - y^2$ to $(x + y)(x - y)$

Practical Performance

- Asymptotic complexity analysis often insufficiently accurate.
 - Constant factors are omitted.
 - May measure irrelevant metrics, such as numbers of rotations, when the really expensive stuff is e.g. processor cache misses.

In practice it may be impossible to distinguish an $O(N)$ algorithm from an $O(N \log N)$ algorithm.

- Example: Nobody uses the theoretically most efficient matrix multiplication algorithms. Best ones are based on the trivial $O(N^3)$ algorithm, but use tricks to improve CPU cache hit rates.

- Read scientific papers with a grain of salt: Has the algorithm really been implemented and tested? What has been measured of it?
- Read text books with a grain of salt: Problems and implementations are often overly simplified.
- Read code with a grain of salt: It often includes undocumented assumptions of the input.

Life Is Harder Than Text Books Let You Believe

- CPU Caches: try to place adjacently data which is accessed simultaneously
 - Multi-processor systems
 - Concurrent threads must protect the shared data structures with locks.
 - A plethora of typical programming errors: race conditions, deadlocks, livelocks, lock contention, etc.
- ⇒ Keep it simple!

- Embedded systems: can you give an upper limit on the memory allocated by your program?
 - Real-time systems: can you place reasonable upper limits on the time used by your program?
(Sometimes better give approximate results than to consume too much time and give a perfect result too late.)
 - Fault-tolerant systems: does your program tolerate or recover from hardware faults?
- ⇒ In many projects, libraries can solve only small sub-problems and you really have to think!

Reduce “total cost of ownership”

- When choosing an algorithm or data structure, consider also the cost of implementing and *maintaining* it.
- ⇒ Keep it as simple as possible!
- Example (a bad one): Little benefit of balancing a binary tree, if you are sure the data comes in random order or if keys are randomized.
 - Example: Why use binary trees if the data is static? Use bisection search in a sorted array instead!
 - Use libraries and standard components as much as possible
 - Devise specialized/faster/better algorithms only when needed