Today’s Topics

Some last and short topics.

1. Average-case time and expected time.
2. One last asymptotic notation.
3. Revisiting the constant-time assumption.
4. Abstract data types.

Average-Case Time And Expected Time

I prefer to equate them.

But most other people prefer to draw a line:

- average-case time: the input is random, the algorithm is not
- expected time: the input is not random, the algorithm is

They use the same math the same way.
They are both expected values of the same random variable.
They can be easily translated to each other.
They miss one case: the input and the algorithm are both random.
Which uses the same math the same way again.

I cannot defend calling them really different.

One Last Asymptotic Notation

\( f(n) \in \Omega(g(n)) \) means: \( g(n) \in O(f(n)) \).

Examples:

- \( n \in \Omega(n) \)
- \( n^2 \in \Omega(n) \)
- Quicksort takes \( \Omega(n) \) time worst-case

Theorem: \( f(n) \in \Theta(g(n)) \) if and only if \( f(n) \in O(g(n)) \) and \( f(n) \in \Omega(g(n)) \)

The Constant-Time Assumption

We have assumed these to take \( O(1) \)-time:

- arithmetic, e.g., adding/multiplying two numbers
- comparing two strings
- computing a hash code

Arithmetic

Arithmetic operations are crucial in

- algorithms for weighted graphs
- hashing

Adding two 64-bit numbers takes \( O(1) \)-time.
The CPU can do it in 1 clock cycle.

What about 640-bit numbers?
What about 6400-bit numbers?
What about \( m \)-bit numbers?
Cannot possibly be \( O(1) \)-time.

Comparing String

Comparing strings is crucial in binary search trees, heaps, B-trees, etc., if keys are strings.

(Similarly, comparing two numbers, comparing two records consisting of strings and numbers.)

Comparing two 30-character strings takes \( O(1) \)-time.
What about 300-character strings?
What about 3000-character strings?
What about \( m \)-character strings?
Cannot possibly be \( O(1) \)-time.
Computing Hash Codes

Computing hash codes is crucial in hash tables.
Computing the hash code of a 30-character string takes $O(1)$-time.
What about 3000-character strings?
What about $m$-character strings?
(Similarly, hashing other kinds of long keys.)

Examples

“Binary search tree find takes $O(\log(n))$ time.”
True if keys are short or fixed length.
“Binary search tree find makes $O(\log(n))$ comparisons and memory accesses.” This is always true.
“Hash table find takes $O(n/N)$ average time.”
True if keys are short or fixed length.
“Hash table find hashes 1 key and, on average, makes $O(n/N)$ comparisons and memory accesses.” This is always true.

Abstract Data Types

An abstract data type says:

- what operations it supports
- maybe also their time costs and space costs
- but not how they are implemented

Example: “ordered dictionary” is an abstract data type supporting:

- $\text{find}(k)$: ... specification ...
- $\text{insert}(k, v)$: ... specification ...
- $\text{remove}(k)$: ... specification ...
- assuming that keys can be ordered

“Ordered dictionary” does not say that it must be AVL tree, skip list, B-tree, linked list, or array.
But can be implemented by any of those, or other choices.

Abstract Data Types We Have Seen

Ordered dictionary: see previous slide.
Priority queue: insert, find min, remove min; assumes an ordering on priorities.
Can be implemented by heap.
Graph: add vertex, add edge, get vertices, get adjacency, set weight, get weight.
Can be implemented by adjacency matrix or adjacency lists.
Disjoint sets: make set, find set, union.
Can be implemented by union-find.
Hashed dictionary: find, insert, remove; assumes a hash function.
Can be implemented by hash table (many variations).

When to Speak of Abstract Data Types

Recurring solutions to recurring problems.
Division of labour:
One team uses a data structure; doesn’t have time for all implementation details.
Another team implements it; doesn’t have time for all application details.
The abstract data type is their mutual knowledge and agreement.

Tidy organization of a large program:
You can zoom out for a big picture.
You can zoom in for an implementation technique.
You have a limited brain. You can only work on one part at a time.

Exactly the same for every factoring in software.

When Not to Speak of Abstract Data Types

The abstraction is close to the implementation.
The implementation is trivial.
The program is small, is not clearer with more layers of abstraction.
E.g., sometimes we don’t make graph, linked list, array abstract.
Exactly the same for every factoring in software.