Part VIII

Complexity and Data structures

Data structures

- Within scientific computing, it is important to understand the availability of different data structures.
- Choosing the wrong one can lead to poor code performance, either due to algorithmic complexity or slow memory access patterns.
- The appropriate data-structure depends on both the algorithm and the architecture being programmed for.
- The correct data-structure for an Intel CPU may be different from that for an NVIDIA GPU.

Computational complexity

- Computational complexity is a measure of how many operations are required to carry out a particular algorithm.
- Here, operation includes any integer or floating point operation, including those required to carry out memory accesses.
- For example, the following (C++) statement takes 5 operations:

```
a[i*10 + j] = sqrt(i);
```

because this is really:

```
*(&a[0] + sizeof(double)*(i*10 + j)) = sqrt(i);
```

- However, because computational complexity is typically only concerned with asymptotics as the size of the data-set tends to infinity, we do not usually worry about taking all these operations into account.

Big-O notation

- Usually, an algorithm will be able to be applied to a particular data-set size, or a particular input value.
- The complexity analysis is then based on how changing this value varies the number of operations required to complete the calculation.
- Some algorithms may depend on more than one input parameter, and therefore so does the complexity.
- The complexity is usually only given as the fastest growing term, e.g. $O(n^3)$, $O(n^2 \log n)$, $O(n!)$

Dot-product

- The dot-product of two vectors `a` and `b` of length N is:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + \ldots a_N b_N$$

- This clearly requires $2N - 1$ operations.
- We usually drop constants and only quote the most significant term, so the complexity is $O(N)$.

Matrix product

- The matrix product of two matrices A $(M \times N)$ and B$(N \times P)$ can be computed as:

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \ldots + a_{iN}b_{Nj}$$

- Each element takes $2N - 1$ operations to compute, and therefore the whole product requires $MP(2N - 1)$ operations.
- The complexity is therefore $O(MNP)$.
- There are many ways of implementing matrix multiplication, however, and some may be more efficient than others, if the hardware characteristics of the machine on which they are implemented are taken into account.
- Complexity usually corresponds to an algorithm, rather than to an implementation of an algorithm.

Matrix product ctd

- The preceding algorithm for matrix multiplication is the most obvious, is the one you were taught at school, and, perhaps surprisingly, not the most efficient.
- More efficient algorithms exist with complexities (for $n \times n$ matrices):
  - Strassen algorithm $O(n^{2.807})$
  - Coppersmith–Winograd algorithm $O(n^{2.376})$
  - Optimised CW-like algorithms $O(n^{2.373})$
- If we make assumptions about the types of matrices involved, then specialised algorithms can be written.
- For example, the product of two diagonal matrices can obviously be computed in time $O(n)$.

Data structures

I shall use C++ nomenclature here, as the C++ STL does use standardised names for data structures.
Other languages may have equivalent data-structures, or you can write code to emulate them.
I shall cover the following:

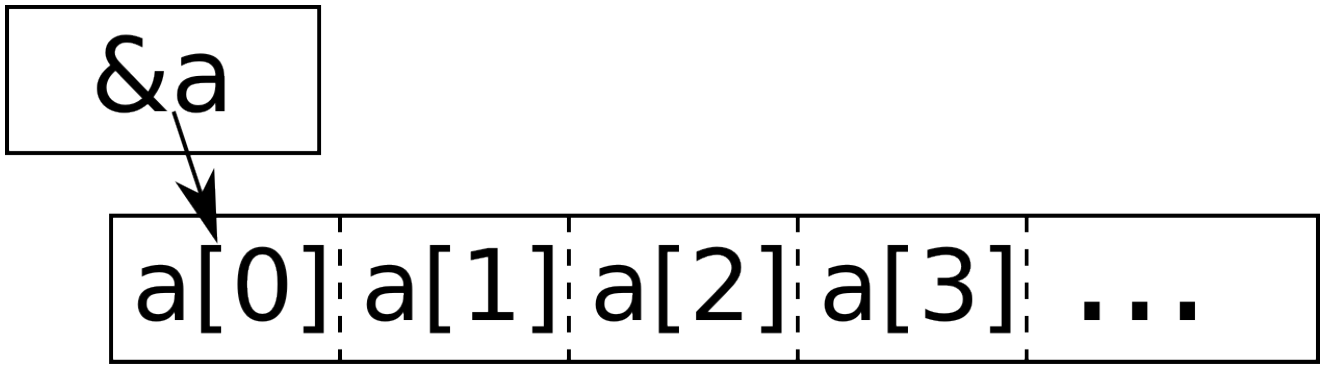
- `vector`
- `list`
- `map`
- `unordered_map`

Other language equivalents

C++	**Python**	**Java**
vector	list	ArrayList
list		LinkedList
map		TreeSet
unordered_map	dictionary	HashSet

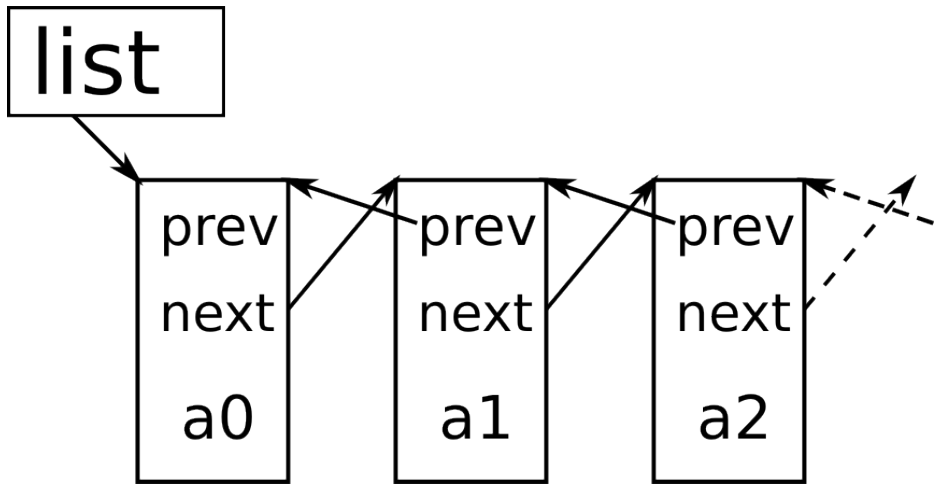
vector

- A `vector` is an indexed set of elements `a`, which are guaranteed to be contiguous in memory.
- The ith element can be found at memory location `&a[0] + i*n` where `n` is the size of an element in bytes.
- Accessing a single numbered element therefore requires constant complexity (a multiplication and addition).



list

- A `list` is a set of ordered elements which have pointers to the preceding and following elements in the list (and no other information).
- Given one list element, it is very easy to find the next but, in order to find the ith element, it is necessary to start at the beginning and go forward $i - 1$ elements.
- Accessing a single numbered element therefore requires $O(n)$ complexity on average, where n is the number of elements in the list.



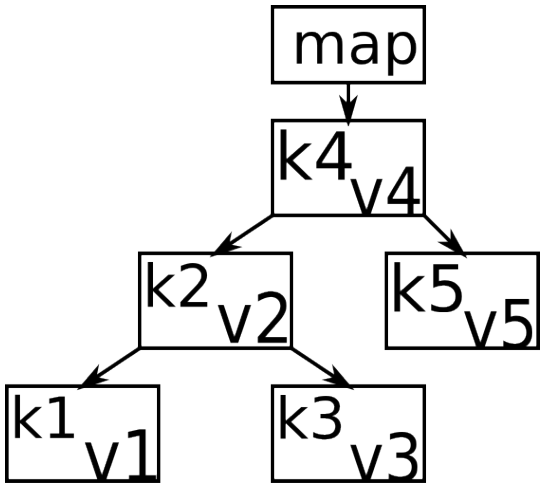
Insertion/Deletion

- Consider what must happen if we want to insert a new element into a `vector` or a `list` at a particular position.
- For a `vector`, we must move all elements $m \ldots N$ to $m+1 \ldots N+1$ and then write the new element into position m.
- This takes at least $N - m$ operations. On average, insertion into a vector has complexity $O(N)$.
- For a `list`, assuming we already know which element is at the insertion location m, we need only
  - Redefine `m->prev_elt->next_elt` to point to the new element.
  - Redefine `m->prev_elt` to point to the new element.
  - Define the new element's `prev_elt` and `next_elt` to point to the appropriate elements.
- This therefore requires only 4 operations, or $O(1)$.
- Deletion for both structures requires a similar set of operations.

map

- A `map` is a one-to-one mapping from, say, a set of strings to a set of integers (a mapping from names to telephone numbers, for example).
- The set being mapped from (the key-type) is required to have a well-defined ordering.
- The way that a `map` is stored is very likely to be in a tree-based structure.
- You should not rely on any assumptions about the way the map is implemented, though.

map ctd



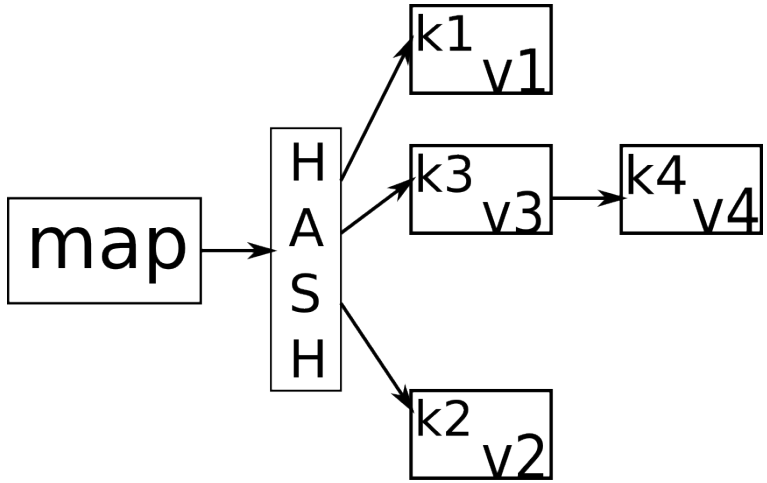
- Each element consists of a key k and a value v.
- The elements are ordered so that those on an element's left have a smaller key, and those on the right have a larger key.
- Assume that $k_1 < k_2 < k_3 < k_4 < k_5$
- "Smaller" and "larger" are based on the well-defined ordering specified by the programmer.
- In the case of strings, the ordering is alphabetical by default (for C++).

Insertion into a map

- Insertion of an element into a map relies on the value of its key.
- Finding the correct location requires working from the top of the tree downwards, comparing the key to each tree node, until the correct location is found.
- At the correct location, updating the pointers of nearby nodes is a constant-cost operation.
- Assuming a well-balanced binary tree, its height is approximately $\log_2 n$, so the cost of inserting a new element is $O(\log_2(n))$.

Unordered map

- An unordered map is based on a hash table, and does not require an ordering on the keys.
- We require a hash function h, typically mapping a key to an integer.
- If the hash function maps multiple keys to the same value, we have a hash collision, and multiple elements must be stored under the same hash value. For example, hash(k3)=hash(k4).
- The performance of an unordered map therefore depends on the hash function and the data in it.
- The hash function should be defined to avoid collisions in general.



Other data-structures

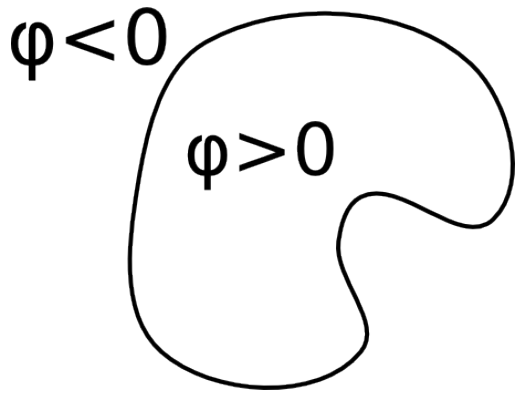
- C++ has several other data-structures with slightly different properties.
- A `deque` is a double-ended queue - somewhere between a vector and a linked-list.
- A `set` is like a `map` but without any value storage - it only stores the set of keys.
- A `multimap` is like a `map` but allows multiple values for a single key.
- These can be implemented in other languages, requiring some or more effort.

Choosing a data-structure

- When choosing a data-structure to use, you need to consider:
  - How do I get values to put into the container?
  - What information do I need out of the container?
  - What operations do I need to invoke on the container?
- Ideally, you then choose the data structure that has the lowest cost for all of the operations.
- If this is not possible, you should make an educated choice based on which will be the most often used operations.
- If using C++ you may be able to write classes so that you can easily try different data structures without modifying too much external code.

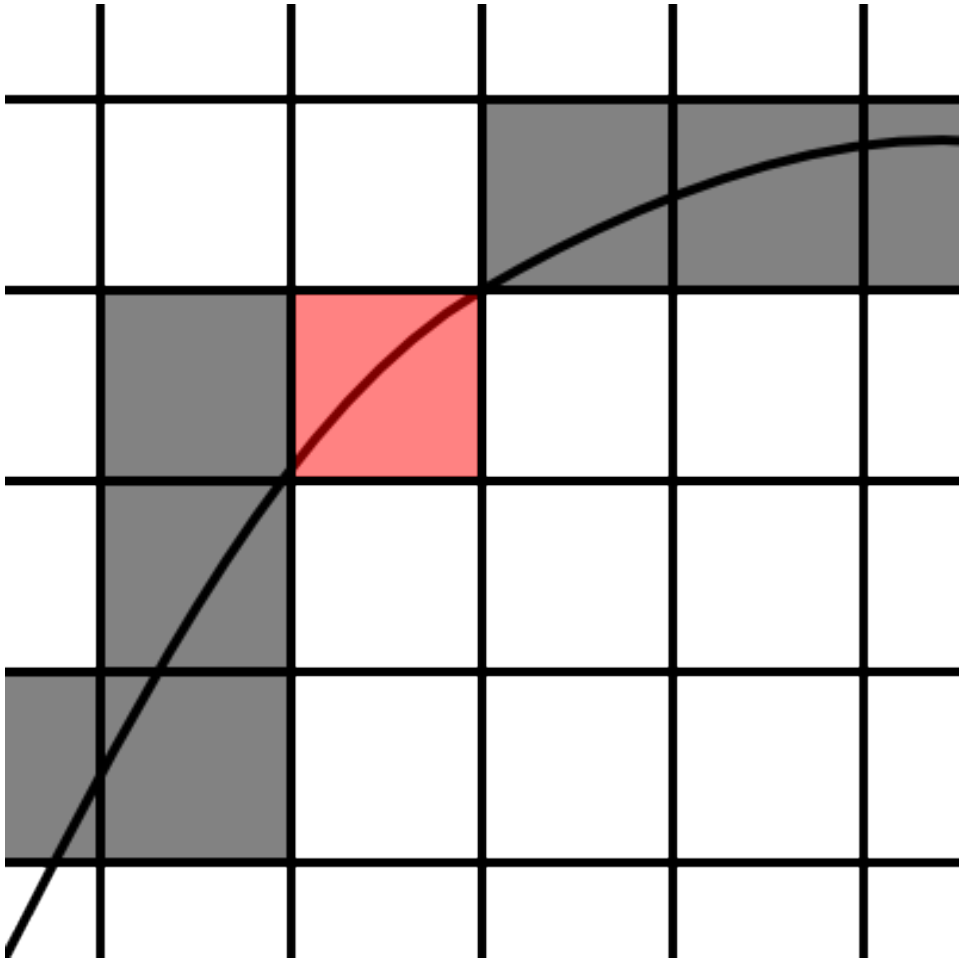
Example - Fast Marching Method

- Given an approximate signed-distance function on a 2D grid, we use the Fast-Marching Method to sweep across a domain to initialize a better approximation everywhere.
- A signed-distance function ϕ allows us to represent an interface between two materials on a 2D grid.
- The interface is represented by the implicit contour $\phi(\mathbf{x}) = 0$.
- We update the signed-distance function using a finite-difference method.
- This may not preserve the signed-distance nature of the function.



Fast Marching Method

- Start with all interface elements.
- (These are defined to be cells where a neighbouring ϕ has a different sign.)
- (*) For the cell (i, j) with the smallest signed distance, loop over its immediate neighbours.
- If an updated ϕ has not been found, compute one from already-computed values.
- Place the newly computed cell and value into the sweeping set.
- Remove cell (i, j) from the set.
- Go to (*) unless the set is empty.



Fast Marching Method

- We need to store a set of cells with their signed distances.
- These need to be ordered by their signed distances.
- We always need to extract the cell with smallest signed distance.
- The cells being inserted may have any signed distance.
- This suggests we need an association from a signed distance to a cell, with an inexpensive insertion into an already-sorted set.
- The answer seems to be a `map` from signed distance to a cell-index.
- A little thought suggests we need a `multimap` as multiple cells may have the same signed distance.
- Insertion then has cost $O(\log(N))$ and extracting the smallest distance cell has cost $O(1)$.

Summary

- When implementing any algorithm, it is worth considering the data structure(s) required.
- A knowledge of the available data structures in your chosen language is useful.
- An overall knowledge of standard data structures is also useful in case an appropriate one is not available in your language.
- It may be easier to implement a sub-optimal algorithm, but to leave your options open to implement a different one if it proves a bottle-neck.