# Algorithms: Quick reference

vishvAs vAsuki

November 16, 2012

## Contents

# Part I

# Introduction and problems

## 1 Research effort

For algorithms to numerically solve problems form continuous mathematics, see numerical analysis ref. For proving lower bounds, see complexity theory ref.
Design algorithms with better space and time complexity, parallelism.

## 1.1 Characterization of research effort

### 1.1.1 Identifying problems

Discovering/ inventing new and important problems, and designing algorithms/ data-structures to solve them constitutes most of algorithms research. Often the emphasis is more on the former, than the latter.
Match algorithms/ data structures to applications.

#### 1.1.1.1 Working on established theoretical problems

Read many theory papers, pick problem with lots of previous work (evidence it's interesting), (optional) Add extra complications to the problem to derive other potentially interesting problems, Find an algorithm better than all the previous results, Write it up and publish it in theory conferences and journals.

#### 1.1.1.2 Finding problems from experimental areas

Learn about areas outside of theoretical CS, Choose a problem in one of those application areas where faster or more accurate solutions can make a practical difference, Abstract essential features to get new clean theoretical problem, Find an algorithm better than all the previous results, Write it up and publish it in theory conferences and journals, Implement and communicate your results with the community your problem came from, discover related problems, repeat.

#### 1.1.1.3 Experimental algorithm design

Inclusion of two more steps: Build practical systems to find new problems, Prove conjectures either theoretically or experimentally. Man perfected arches and pyramids before theoretically justifying it.

### 1.1.2 Design, then analyze algorithms

Note that these are separate steps; often algorithm design and analysis is an iterative process.

#### 1.1.2.1 Reasoning about resource bounds

After analyzing an algorithm think about the quantities involved and try to see where they are coming from.
Suppose that algorithm $A$ requires $k = d \log p$ samples or time to solve an algorithm. To see if this is close to optimal, one can reason about $p^d$: perhaps it represents the objects the algorithm has to search through in solving the problem.

## 2 Problems

### 2.1 Secretary problem

Best strategy: Sample around 37% of applicants to get an idea of skill distribution, then select the next applicant with higher skill.

### 2.2 Graph and network problems

See Graph theory ref.

# Part II

# Algorithm and data structure design

Optimization algorithms are considered elsewhere.

## 3 Algorithm design strategies

### 3.1 Bottom up solution

Aka Divide and conquer. A bottom-up way of constructing the solution.
This includes Dynamic programming, which is discussed in the Optimization survey.

#### 3.1.1 Time Analysis

[**Incomplete**]

### 3.2 Greedy algorithms

Eg: shortest path algorithm.

#### 3.2.1 The matroid

This is a pair: (Set, Independence property). Eg: (vertices, Independent sets in a graph). There are efficient algorithms for finding the maximal indpendent subset. So, often sufficient to specify the problem using the matroid formalism.

## 4 Abstract Data types and Data structures

### 4.1 Abstract data types

When data is stored, we often want to perform various operations (ie evaluate mathematical functions) on it.
Abstract data type is a mathematical construct of a data-store, defined by the operations which can be performed over it, and consistency rules (which define the effect of the aforementioned operations on the stored data). For example, a stack can be defined using push, pop operations, with constraints such as pop(push(stack, x)) = x.
This concept may be extended to include rules which limit the cost of various operations.
Abstract data types are thus useful in specifying design goals, which may be met by various data structures.

### 4.2 Data structure for the task

Suppose we want to realize a certain abstract data type on the computer. The same data can be stored in different ways in the computer. For various operations on the data, some ways result in better performance, in terms of memory, time, success rate, than others.
So, one must choose the data structure depending on the types of operations one intends to perform on the data : ie the abstract data-type being implemented.

## 5 Storing sequences

### 5.1 Operations

Already stored data may be mutated with additions and deletions. Searching and sorting are considered elsewhere.

#### 5.1.1 Low level operations

Important special cases defined based on restrictions on insertion and deletion operations include stacks, queues, priority queues. In the case of stacks, elements leave the sequence in a last in first out (LIFO) order. In case of queues, elements leave in a first in first out (FIFO) order.

#### 5.1.2 Higher level operations

These operations are defined in terms of low level operations described earlier: Eg: map, filter etc.. So, an efficient realization of low level operations automatically enables higher level operations.
Coding efficiency in implementing these higher level operations is considered elsewhere.

### 5.2 Iterators

Here, elements are generated one at a time according to some rule. So, with the next() operation one gets a value which may not be produced again. Iterators can be used in iterative statements without wasting memory to store all possible values at once.

### 5.3 Integer indexed Arrays

Integer tuples (indices/ keys) are mapped to values of any general data-type; so that values can be stored at or retrieved from a given indexed location in the array.

### 5.3.1 General implementations

Storage of (key, value) pairs in a way which allows fast search of keys and retrieval of corresponding values is essential. So, (key, value) pairs can be stored using binary search trees, easily calculated map to memory locations, dictionary search trees (using the string representations of the integers) etc..
Values stored with the key can in-fact be pointers to different memory locations containing the corresponding actual-values.

### 5.3.2 Sequential locations implementation

In the most common implementation, Elements stored in sequential memory locations.
But the sequentialness requirement makes deletion of nodes, and insertion of nodes in a certain positions costly.
Their advantage in terms of memory is in locality/ contiguity and in overall - minimality (constant factor better than linked lists, for example). This advantage is usually not significant to bother with.

### 5.3.3 Array buffers

An array buffer stores a list internally using an array. Its length may then be expanded or contracted by replacing the array. So despite the flexibility with the size compared to an array, the access time remains constant, while the amortized update time is also constant.

## 5.4 Associative arrays

These generalize the integer-indexed arrays so that the index is allowed to be of an arbitrary data-type. The aim is to have an O(1) operation for for retrieving elements corresponding to a certain index.

### 5.4.1 Hash-fn map implementation

Here, a hash function maps keys to integers, which together with the values are stored in the usual integer-indexed arrays.
Several keys may be mapped to the same integer, which results in a hash-collision.
In this case, the multiple values associated with different keys but the same integer keys may be stored using any sequential data structure, which may be searched using the query-key. This is aka 'chaining'.

#### 5.4.1.1 Randomization

A hash function can be a fixed rule. But that can be exploited in DoS attacks by forcing collisions. So, ideally the hash function is randomized.
Some implementations randomize iteration over hashmaps the same way.

### 5.4.2 Prefix tree/ treep

Here, the keys are mapped to unique string representations, which are then used as substitute keys. These keys can then be stored as a k-ary tree, where $k$ is the size of alphabet used to form the string. For each string-key stored, characters associated with nodes along the path from the root to a certain leaf is equal to the string key.
Using this, a k-characters long key can be looked up in time O(k).

#### 5.4.2.1 Comparison to hash-fn map

The number of leaf nodes is exactly equal to the number of entries in the associative array. So, this is more efficient when storing a small number of entries, whereas a hash-function implementation would reserve space for a huge table even when the number of entries is small.

## 5.5 Linked list of node objects

Linked lists is a chain of node objects, each of which contains a stored value and the address of the next node. Linked lists occupy more memory than arrays. Accessing the kth element is an O(k) operation, rather than an O(1) operation in the case of arrays.

### 5.5.1 Array list comparison

Linked lists are more dynamic when it comes to 'on-demand' insertion: if one has the memory location of a given node, one can insert a sublist next to it more efficiently than in the case of arrays. For comparison with arrays, see the array-list section.

### 5.5.2 Doubly linked list

Doubly linked lists are extensions of linked lists, where the node also includes the address of the previous node. Their disadvantage relative to a singly linked list is only in terms of memory - usually insignificant in the modern context of cheap memory.

## 5.6 Tree

Tree data-structures of nodes are defined as described in the graph theory survey, with the additional modification that they have a clearly specified root node, and the children of each node are clearly enumerated! Each node can have k child nodes: when $k \leq 2$, we have a binary tree.

### 5.6.1 Traversals

Traversals are enumeration of node values in particular orders. These include in-order (only for binary trees), pre-order and post-order traversals. In each case, one starts at the root node and executes a certain traverse(node) function on that node.
This function, in the case of in-order traversal, executes traverse(left child) (if it exists), then appends the value of the current node, and finally executes traverse(right child) (if it exists).
The function, in the case of pre and post order traversals, executes traverse(node) for each child node in order before and after printing the value of the current node respectively.

### 5.6.2 Search trees

Nodes in search trees have the property that, for any node: the left child node value, the current node value and the right child node value are in non-descending order. Thus, an in-order traversal yields a sorted list. These are very useful because they can be dynamically created - by adding nodes at run-time; and when they are balanced, they yield a sorted list.

### 5.6.3 Heap

[**Incomplete**]

## 6 Set

Basic operations include listing elements, checking set-membership, finding unions and intersections.
Set implementations include the ListSet and the BitSet.

## 6.1 Bit-set

BitSet uses a sequence of bits - one for each element. Theoretically, it can store arbitrary elements which may be mapped to integers, but is often used to store integers. Its size is equal to the largest number N stored in the set, though it may store k elements; so, from this perspective, using it in preference to ListSet makes sense when the set is dense. Unlike listSets, membership check is constant time. But, listing the elements of the set requires O(N) time, often greater than O(k) required by ListSets.

## 7 Sequence operations

## 7.1 Membership query algorithms

The abstract data types relevant here are Sets and multisets (defined in the algebra survey).

### 7.1.1 Over Associative arrays

When the elements are stored as an associative array - eg: hash-fn map or a treep, answering membership queries is O(1).

### 7.1.2 Fingerprinting

Coding theory is useful here: see fingerprinting in information theory ref.

### 7.1.3 Sequential search and sorting

Sequential search: O(n) in the worst case. When the list is unsorted, there is no choice but to do this. But, when the list is sorted, one can do better.

### 7.1.4 Balanced binary trees

Search trees are described in another section. Here, the assumption is that the search tree is balanced.
Starting at the root node, one keeps comparing the query value to the node value and traversing left of right as necessary until one finds a matching node, or reaches a leaf node having searched in vain. Same as doing binary search - except that the sorted list is stored explicitly as a balanced binary tree.
But updates are costly.

### 7.1.5 Binary search

Here an array is viewed as if viewing a balanced binary search tree: this is because one can easily match the id of a node in the binary search tree to the corresponding position in the array list, the latter merely being the result of an inorder traversal of the tree. Time complexity: $O(\log n)$ in the worst case.

### 7.1.6 Splay tree

Splay trees rotate and update themselves so that the most recently accessed item is at the root.

### 7.1.7 Compression, allowing some fast operations

[**Incomplete**]

## 7.2 List Sorting

### 7.2.1 Assumptions

Using only constant extra memory.

### 7.2.2 Lower bounds

$\Omega(n)$ lower bound is easily obtained simply rom the need to look at the values in the $n$ positions.
Any deterministic algorithm involving binary comparisons requires $\Omega(n \log n)$ comparisons in the worst case: At least $\Omega(\log(n!))$ binary decisions are required to distinguish the $n!$ possible permutations of $n$ items.

### 7.2.3 Repeated partitioning sort

Aka quick sort. Move first element to position x such that elements larger than x are to the right, and those smaller than x are to the left; so now you are left with two arrays excluding the partitioning element; repeat this procedure on both these arrays.

#### 7.2.3.1 Time complexity

If the partitioning element is chosen randomly: Average case: $O(nlogn)$.
Worst case: $O(n^2)$ if the partitioning elements always create very unbalanced partitions.

#### 7.2.3.2 Improvements

A median finding algorithm can find the median in O(n) time. This can be used to get roughly $O(nlogn)$ time complexity on most sequences. But, if there are a huge number of duplicates, we still may need $O(n^2)$ time.

### 7.2.4 Merge sort

Divide list into 2 equal parts, apply merge sort on them, merge the sorted lists.
Note that this algorithm can be implemented in a bottom-up manner.

#### 7.2.4.1 Time complexity

Merging two sorted lists of size $k$ takes $O(k)$ time, with atmost $k$ swaps - one uses two pointers, which initially point to the beginning elements of the two lists; one keeps moving the pointers one step at a time while maintaining the property described next using swaps. The property to be maintained is: The

value under the first pointer should always be less than the value under the second pointer, the values between the two pointers should be sorted, the values from the second pointer to the last element should be sorted.
As there are $\log n$ sets of merges, each involving $n$ elements: Worst case, avg case: $O(n \log n)$.

#### 7.2.4.2 Space complexity

In a naive implementation, $O(n)$ memory is required to remember the start and end points of various sublists being sorted and merged. But, these points may be computed dynamically.

### 7.2.5 Other Techniques

Bubble sort: lightest element in the sublist $1:k$ for $k \in n:1$ keeps bubbling to the top. Time complexity: $O(n(n-1)/2) = O(n^2)$

## 7.3 Ordinal selection algorithms

It is assumed that we only use constant extra memory.

### 7.3.1 By Sorting

Median finding after doing merge sort requires $O(n \log n)$ time.

### 7.3.2 By repeated partitioning

One repeatedly does the following: a] pick a pivot. b] derive two partitions of the list such that the left partition only contains elements <= pivot, and right partition only contains elements >= pivot. Then, having determined which partition to seek the kth largest element in, seek the element in that partition by recursively using the same procedure.

#### 7.3.2.1 Time complexity

Avg case complexity: $O(n) = n + n/2 + n/4..$ : for intuition consider the case where the sequence is divided evenly each time. Worst case: $O(n^2)$ due to bad pivots - this can be avoided by randomized choice of pivots.
[**Incomplete**]

# Part III

# Algorithm analysis

## 8 Analysis of algorithms

For analysis of randomized algorithms, see randomized algorithms, probabilistic analysis ref.

### 8.1 Asymptotic behavior

How does the problem scale with increasing input size? See computational complexity ref. Usually care about worst case and average case performance.

### 8.2 Amortized analysis

Got a sequence of operations performed by a deterministic algorithm. Maybe every n operations, a huge cost is incurred. Then, even though, in the worst case, the cost is high, on average, each operation costs much less.

## 9 Resources and their limitations

### 9.1 Online vs offline algorithms

Performance/ Competetive ratio of online algorithms, when compared to an offline algorithm given the entire input sequence.

### 9.2 Randomized algorithms

See randomized algs ref.

### 9.3 Parallel algorithms

Time taken vs total work done.
2 major frameworks: Message passing vs shared memory. In some models, reading a memory location while another is writing is not allowed.
Parallelize around data, rather than tasks: You will have more processors tomorrow than today.

### 9.4 Deal with memory limitations

Use external memory algorithms. [**Incomplete**]