Programmeren (FEB22012)

Example exam

Block 1 / 2011

Instructions

All the questions are *essay questions*: write concisely everything you know about the topic. Describe what the topic entails, and reflect on the impact of theory on the practice of programming. If you do not know something, it is better to write less than to try and invent an irrelevant answer. All questions should be answerable within 1 page of text with normal size hand writing.

In the actual exam, no extra material (neither dictionaries nor calculators) is allowed. Take with you only a pencil, a sharpener and an eraser.

Questions

1. Programming paradigms
2. Running time analysis of algorithms
3. Binary search trees

Example answer (q1)

Programming paradigm is essentially the idea behind the design of a programming language. There are three main paradigms: imperative/procedural, object oriented and functional programming. In imperative paradigm (with languages such as C and Java) the core idea is that the state of computation is altered by executing statements, and the complexity inherent in large problems is being controlled by breaking larger problem into smaller parts that are solved by separate methods (functions or procedures). Although in object oriented paradigm the actual computation is made in an imperative way as well by describing the state changes, the focus is on constructing classes that are instantiated to objects that communicate with each other (through method calls). Another difference is that the additional abstraction techniques in object oriented languages make it more suitable for implementing larger pieces of software, especially for less experienced programmmers.

By understanding the "way of programming" in a single paradigm, changing languages within a paradigm is easy. That is, the way to implement programs in Java is essentially different from the way to implement programs in Matlab, which in turn is similar to implementing programs in R, Python, or C. Also, although actual paradigm in Matlab and Java is different, they both adhere partly to imperative paradigm as can be seen in similarity of the language structures (for- and while-loops and if-statements).

Example answer (q2)

Running time analysis of algorithms refers to how many primitive operations are required for an algorithm to complete with a certain input. The goal in the analysis is to describe the running time in terms of the input size n. This size may refer to, for example, size of the array to be sorted or to the number of bits

in the integers to be summed. The running time often changes not only according to the input size, but also according to the input contents. For example, the number of operations required by insertion sort depends on ordering of the elements in an array - the algorithm performs a lot faster with an array that is already (almost) sorted than with one that is not.

When analysing the running time, we are usually interested in the growth rate of the running time regarding growth of the input size. That is, when input size increases by 1, how much does the running time increase? The differences in programming languages and computer hardware cause calculation of the running time in terms of exact number of operations to be irrelevant, and the running time is instead often analyzed regarding it's asymptotic groth rate as a function of n. Upper bound to this is given with the O-notation, that describes the highest order term of the growth rate without the associated constants (e.g. f(n) = O(n^2)). Note that larger running time (e.g. $O(n^3) > O(n^2)$) does not necessarily mean that the algorithm is slower: it means that with input size larger than a certain value it is. This value can be so large that the asymptotically larger running time algorithm is faster in practice.

Most of the time we are interested only in the worst case running time of an algorithm, as it definitely happens sometimes and can be considerably higher than the best case one. For example, if an algorithm usually takes O(n) operations to complete, but with some inputs it takes O(n!) operations, it is useless in practice as with these worst case inputs cause the computation to take a few hundred years.

Example answer (q3)

Binary search trees (BST) are a data structure in which the elements are stored in nodes of a tree: each node has exactly one parent node, and in a binary tree each node can have a maximum of 2 child nodes. In binary search tree one additional condition holds: for each node in the tree, the left child <= node < right child. The structure of a BST is shown below.

(Here a fancy drawing of the structure of a BST)

Due to this ordering property we can search a node in the tree by starting from the root node (top) and comparing the key of the node to find with the key of the root: if the key to find is smaller we proceed to do the same in the left subtree, or if it is larger then we search in the right subtree. In this way we will eventually find the node with the given key.

Complexity of looking for a node in a binary search tree depends on whether the tree is balanced or not. In balanced trees the difference in minimum and maximum depths (amount of edges from root to the node) of leaf nodes (nodes without children) is small. If the tree is complete (all leaf nodes are of depth x or x-1), then finding any value in the tree is of complexity O($\log_2 n$), where n is the amount of nodes in the tree. In case of an unbalanced tree the worst case complexity to find a node is the same as in a list, O(n). Due to dynamic allocation of nodes and for the low worst-case complexity of search in the balanced case, binary search trees are often used for storing a changing set of elements that needs to be frequently searched.