Algorithm Part I Examples Sheet

Dr Robert Harle
Lent 2018

Questions marked (**W**) are warm up questions. They mostly ask you to regurgitate the notes just to check your core understanding. Questions marked (*) are intended to be more time consuming and/or challenging. Conult your supervisor for an appropriate set for you.

Algorithm Specification

1.1. **(W)** Explain the purpose of preconditions and postconditions when specifying an algorithm and assertions when implementing it.

1.2. **(W)** Suggest pre- and post-conditions and possible assertions for the following examples:

i) `double sqrt(double)`: a function that returns the square root of any argument.

ii) `double divide (double a, double b)`: a function that computes a/b.

iii) `void normalise (int[] a)`: a function that takes a 3D vector specified in `a` and normalises it to have magnitude 1.0.

1.3. **(W)** Discuss the following reasoning. *Major programs such as operating systems must run indefinitely. Assertions cause a program's execution to halt abruptly. Therefore assertions are not appropriate when designing such programs.*

Asymptotic Analysis

2.1. You must choose between two sorting algorithms: the documentation indicates both are $O(n^2)$ in comparisons in the worst case. Assuming your goal is to optimise for speed, does it make any difference which you choose? How would you make the decision?

2.2. Show that:

i) $f(n) = 6n^2 + 7n$ is $O(n^2)$

ii) $f(n) = \frac{6n^2+7logn}{n}$ is $O(n)$

iii) $f(n) = 4n^2 + 2$ is $\theta(n^2)$

iv) $f(n) = 1^2 + 2^2 + 3^2 + ... + n^2$ is $\theta(n^3)$

Sorting Algorithms

3.1. **(W)** In the insertion sort pseudocode what is the worst-case number of *assignments* made? Improve the pseudocode to reduce this.

3.2. **(W)** When looking for the minimum of m items, every time one of the $m - 1$ comparisons fails the best-so-far minimum must be updated. Give a permutation of the numbers from 1 to 7 that, if fed to the Selection sort algorithm, maximizes the number of times that the above-mentioned comparison fails.

3.3. **(W)** Of all of the $O(n^2)$ sorting algorithms discussed in the lectures, which would you expect to perform best given a random input? Justify your answer.

3.4. Code up the binary insertion sort algorithm for arrays.

3.5. Show the steps involved in sorting the digits 4632739 using:

i) Mergesort;

ii) Quicksort;

iii) Heapsort.

3.6. Show how to mergesort an array in $\frac{n}{2}$ space.

3.7. Write pseudocode for the bottom-up mergesort.

3.8. For the merge step in mergesort we have two sorted subarrays that we need to merge into one sorted array. A programmer suggests this is just like a half-finished insertion sort and could therefore be achieved using an insertion sort algorithm.

i) Write out the pseudocode for an insertion sort with a precondition that the first half of the data is in sorted order.

ii) If the resultant array is of length n, show that your modified insertion sort is $O(\frac{3n^2}{8})$.

iii) In a merge step, we can additionally make use of the fact that the second half of the array is presorted. What is the Big-O complexity of this new 'insertion merge' step?

3.9. Can picking the quicksort pivot at random really make any difference to the expected performance? How will it affect the average and worst cases?

3.10. What is the smallest number of pairwise comparisons you need to:

i) Find the smallest of n items.

ii) Find the *second* smallest of n items.

3.11. Implement two versions of quicksort in Java:

i) One where the pivot is always chosen as the first element; and

ii) One where the pivot is chosen at random.

How do you know your implementations work?

3.12. What are the minimum and maximum number of elements in a heap of height h?

3.13. Implement counting sort and radix sort in Java. Given that radix sort uses multiple instances of counting sort, why not just use counting sort in the first place?

3.14. For each of the sorting algorithms covered in the notes, establish whether it is stable or not. Justify each answer.

Core ADTs and Data Structures

4.1. **(W)** Write a Java function that tests whether a string is a palindrome (the same backwards as forwards) using only the standard operations of a stack.

4.2. **(W)** Discuss the advantages and disadvantages of the linked-list and array based implementations of a queue.

4.3. Consider implementing a stack using an array. When the array is full and a push() is requested, there are two common strategies to growing the array: increase the array by a constant number of elements or double the size of the array. Analyse both strategies to find the amortized costs associated with a push() operation.

4.4. Write pseudocode or Java code for a deque that uses a circular buffer (i.e. it wraps around to use space at the start of the array and vice-versa)..

4.5. Consider a table implementation based on a linked list. Write pseudocode for the set() function, assuming duplicates are not permitted.

BST, Red-Black and 2-3-4 trees

5.1. **(W)** Describe how to find the predecessor of a node in a BST. Sketch a proof that your alorithm works.

5.2. Using the sequence {10,85,15,70,20,60,30,50,65,80,90,40,5,55} draw:

i) the BST tree

ii) the 2-3-4 tree;

iii) the red-black tree.

5.3. The main issue with BSTs is their tendency to become unbalanced. This can be a particular issue if the input keys have a lot of duplicates (e.g. insert {1,1,1,1,1,1,1} gives a very unbalanced tree. Suggest a way in which duplicate entries in a BST could be addressed such as to produce a more balanced result.

5.4. Draw the BST obtained by the insertion sequence {25,9,5,1,6,20}. How many permutations of this input would have given the same BST?

5.5*. Show how to delete a node from a red-black tree.

5.6. Find a sequence of ten numbers that results in a worst-case red-black tree (i.e. the 10-node red-black tree with the longest path from root to leaf). Justify your answer.

B-Trees

6.1. **(W)** When analysing trees, complexities are often O(height of tree). Given that B-trees can be used to produce trees of lower height than the equivalent BST, explain why:

i) B-trees are not a better choice than Red-Black trees when the tree is stored purely in memory.

ii) B-trees are a better choice than Red-Black trees when the tree is stored on disk.

6.2. What are the minimum and maximum number of keys in a B-tree of minimum degree 700 (i.e. an arbitrary node has between 700 and 1,400 children links) and height h?

Hash Tables

7.1. **(W)** Show how to delete hash table entries when resolving collisions using:

i) chaining;

ii) open addressing.

7.2. Prove the search complexity results for a hash table that uses chaining to resolve collisions. i.e. $O(1+\alpha)$.

7.3. How would you use a hash table to create a basic spell checker (i.e. decide whether or not a word is correctly spelt)? For an incorrectly-spelt word, how would you efficiently provide a set of correctly-spelt alternatives?

Priority Queues

8.1. Consider a priority queue implemented using i) a doubly linked list and ii) a singly linked list. For each case state and justify the complexities of the standard priority queue operations.

8.2. Recall that modern computers give the appearance of running multiple applications simultaneously, whilst actually running each in turn for very short time periods. Priority queues are often used to queue the applications so that those deemed to be more important get to the front of the queue, ahead of any 'background' applications. Unfortunately this can lead to starvation, where the low priority applications never make it to the head of the queue because higher priority applications keep jumping in first. Suggest how you might address this.

8.3*. In some applications the priority values are bounded to be the integers 1 to K for some positive integer K, and the only operations of interest are `first()`, `insert()` and `extractMin` (note we expect multiple values to have the same priority). Devise a custom data structure that gives better performance than a standard heap for this priority queue.

Algorithm Strategies

9.1. **(W)** Briefly explain the basics of the following strategies:

i) Brute-force;

ii) Divide-and-Conquer;

iii) Dynamic Programming;

iv) Back-tracking.

9.2. Write a suduku solver in Java that uses a backtracking strategy. The puzzle should be input as an `int[9][9]`.

9.3.

i) Which strategy was used in the FoCS course to find all ways of making change?

ii) Describe a dynamic programming algorithm to find an optimal solution to the making change problem (i.e. a solution that uses the fewest coins).

iii) What is the computational complexity of your algorithm?

9.4. Propose a divide-and-conquer algorithm to find both the maximum *and* minimum of a set of n integers. What is the complexity of your algorithm?

9.5. For the knapsack problem described in the notes, Provide a small counterexample that proves that the greedy strategy of choosing the item with the highest £/kg ratio is not guaranteed to yield the optimal solution.