Algorithms

Example Sheet 1

Petar Veličković

Lent Term 2017

Warm-up

1. Assuming that the `swap(x, y)` function is implemented as {`tmp = x; x = y; y = tmp;`}, propose an improvement to insertion sort as presented in the course notes. What is the time complexity of the improved algorithm?
2. Prove that bubble sort will never need to perform more than n passes of its outer loop.
3. Discuss the benefits to choosing a pivot for quicksort at random, and any potential practical limitations to achieving them.
4. Demonstrate, in a step-by-step fashion, and sketches where appropriate, how heapsort sorts the array {`7, 4, 3, 2, 8, 1, 6, 5`}.

Exercises

1. You have implemented an algorithm that, after paying a constant cost of c time steps, can be recursively applied to an input of size $\sqrt{N}$ to process an input of size N. That is, the recurrence relation for the time complexity $T(N)$ of this algorithm is $T(N) = T(\sqrt{N}) + c$. Derive a non-recurrent expression for $T(N)$ and express it using big-O notation.
2. Propose an algorithm to locate the largest k elements of an array of size n (in any order), and analyse its time complexity.
3. Consider k-ary heaps (heaps where each internal node has k children—for binary heaps, $k = 2$). Provide a modified `heapify` procedure that will convert an array of size n such that it represents a k-ary heap, and analyse its time complexity.
   Hint: You may find the following result useful: for $|x| < 1$,

$$\sum_{m=1}^{+\infty} m x^m = \frac{x}{(1-x)^2}$$

4. Considering the merge sort algorithm:
   (a) Explain how it is possible to merge two sorted linked lists in linear time and constant auxiliary space.
   (b) Encouraged by the above, your colleague proposes merge sorting an array by first converting it into a linked list. Comment on this approach.
5. Describe an algorithm that will determine whether there exist two distinct elements of an array of n positive integers that have the sum s. Analyse its time complexity.
6. Considering the radix sort algorithm:
   (a) Explain the tradeoffs involved with running the algorithm from most to least significant digit, as opposed to the reverse order (as presented in the notes).
   (b) Provide a proof by induction of the algorithm's correctness.

Implementation

Hopefully, the examples covered have demonstrated to you that sometimes, choosing the proper (comparison) sorting algorithm to use can be rather difficult. Therefore, the algorithms that actually make it to standard libraries are quite often hybrid approaches. For this exercise, I would like you to implement the *introsort*[1] algorithm in a language of your choice. This algorithm attempts to combine the excellent average-case performance of quicksort with the safe worst-case performance of heapsort—start with the former, and fall back to the latter if the recursion gets too deep (typically, deeper than $2\lfloor\log(n)\rfloor$).
The pseudocode of one recursive call is as follows:

```
INTRO-SORT(A, d) // Introsort array A with max-depth d of quicksort
1  n ← |A|
2  if n ≤ 1
3      return
4  elseif d = 0
5      HEAP-SORT(A)
6  else
7      p ← PARTITION(A) // Partitions A and returns pivot position
8      INTRO-SORT(A_{0:p}, d − 1)
9      INTRO-SORT(A_{p+1:n}, d − 1)
```

Demonstrate the result of running your algorithm on several randomly generated arrays, as well as arrays that would normally cause "pure" quicksort to exhibit its worst-case ($O(n^2)$) performance.

[1] `https://en.wikipedia.org/wiki/Introsort`
Also, the algorithm used by C++'s `std::sort`.

Job interviews

1. Assume that you have to sort an array of n integers, but you may only fit k integers into working memory at any one time ($k < n$). Describe a strategy for performing this procedure (also known as *external sorting*).

2. Describe an algorithm that will determine whether there exist k distinct elements[2] of an array of n positive integers, $A_1, A_2, \ldots, A_k$, such that $A_1 + \cdots + A_k = s$ for a given integer s.

3. You are given a function `fair_coin()` that will return `"heads"` with probability $\frac{1}{2}$, and `"tails"` with probability $\frac{1}{2}$. Using this function, write a function `biased_coin(p)`, that will return `"heads"` with probability p, and `"tails"` with probability $1 - p$. How many times, on average, will your function call `fair_coin()`? Assume that p can have up to n significant digits.

4. Solve *Exercise 3* from the pre-sheet.

[2] **N.B.** this is a generalisation of Exercise 5 from this sheet.