Algorithmics I - Tutorial Sheet 1

Sorting and Tries

1. The number of operations required by the (hypothetical) algorithms $B_1, B_2, \ldots, B_5$ to process an input of size n are as follows:

algorithm	number of operations
B_1	$n \log_2 n$
B_2	n^2
B_3	n^3
B_4	2^n
B_5	$n!$

On a computer that executes 10^9 operations per second, calculate the maximum size of input that can be processed by each algorithm in at most 1 minute.

Note: in the cases of B_1 and B_5 use a calculator and trial and error. For the remaining cases you should be able to find a mathematical expression for the solution, which you can then evaluate with a calculator.

Solution: B_1: 2×10^9. B_2: 24498. B_3 : 3914. B_4: 35. B_5: 13.

Explanation: the number of operations that can be completed in 1 minute is $60\times10^9 = 6\times10^{10}$. So, if the number of operations required to process input of size n is $f(n)$, we need to find the largest value of n for which $f(n) \leq 6\times10^{10}$.

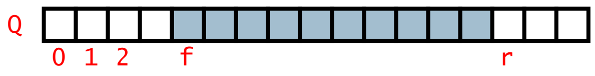
- For B_1, we need to solve the equation $n \log_2 n = 6\times10^{10}$. Formally, this requires techniques from numerical analysis which are beyond the scope of the course. Informally, it can be done on a calculator by "trial and error", giving a result just less than 2×10^9.
- For B_2, we need $n^2 \leq 6\times10^{10}$, i.e., $n \leq \lfloor\sqrt{6\times10^{10}}\rfloor = 244948$.
- For B_3, we need $n^3 \leq 6\times10^{10}$, i.e., $n \leq \lfloor\sqrt[3]{6\times10^{10}}\rfloor = 3914$.
- For B_4, we need $2^n \leq 6\times10^{10}$, i.e. $n \leq \lfloor\log_2(6\times10^{10})\rfloor = 35$.
- For B_5 trial and error on a calculator will (eventually) demonstrate that $13 \leq 6\times10^{10} \leq 14$, so the required answer is 13.

2. Describe in detail how a `Queue` ADT may be implemented using an array in such a way that the operations: `create`, `isEmpty`, `insert` and `delete` all have $\mathcal{O}(1)$ complexity.

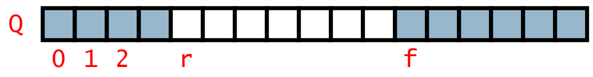
Note. You can assume that the queue has a maximum size $N-1$.

Solution: The queue must 'wrap round' which avoids moving objects once they are placed, since once we start moving objects the complexity will not be $\mathcal{O}(1)$. This can be achieved using two variables f and r which keep track of the front and rear of the queue respectively. More precisely, f is an index to the cell of the array storing the first element of the queue, while r is an index to the next available array cell, i.e. the cell immediately past the last element of the queue.

There are two possible configurations, the normal configuration:



Or the wrapped around configuration:



If an element is removed from the front of the queue, then we can just increment f to index the next cell. If an element is added to the rear of the queue, then we insert it into cell r and increment r to index the next cell. The increments are performed modulo N (where N is the size of the array) to correctly model the wrapped around configuration. In order to tell the difference between a full queue and an empty queue, we require the queue to contain at most $N-1$ elements.

Formally the operations can be defined as follows.

- `create`: sets both the front and rear markers to 0.
- `isEmpty`: tests whether f equals r.
- `insert`: adds 1 (modulo N) to the rear marker, except when the queue has $N-1$ elements (i.e. after inserting r equals f which happens when $(r+1) \bmod N = f$) and instead reports a 'full queue exception'.
- `delete`: deletes the item from the front position and adds 1 (modulo N) to the front marker, except when the queue is empty, in which case it reports an 'empty queue exception'.

Clearly all these operations require constant time, i.e. are $\mathcal{O}(1)$.

3. Suppose we have two integer arrays A and B each with n elements.

Hint: this question concerns using sorting algorithms, so consider sorting one or both of A and B in each case.

(a) Describe an $\mathcal{O}(n \log n)$ algorithm to determine whether all of the elements of A are different.

Solution: The following algorithm meets the requirements:

1. sort A into increasing order (this can be performed in $\mathcal{O}(n \log n)$ time using either `mergesort` or `heapsort`);
2. scan A to determine whether two consecutive elements are equal (this takes $\mathcal{O}(n)$ time).

Considering the complexity of each step, we see that the overall the complexity is $\mathcal{O}(n \log n)$ as required.

(b) Describe an $\mathcal{O}(n \log n)$ algorithm to determine whether A and B have an element in common.

Solution: The following algorithm meets the requirements:

1. sort B into increasing order (this can be performed in $\mathcal{O}(n \log n)$ time using either `mergesort` or `heapsort`);
2. for each element of A in turn, perform a binary search to determine whether it is in B (a binary search takes $\mathcal{O}(\log n)$ time and then there are at most n searches).

Since the complexity of each step is $\mathcal{O}(n \log n)$, the overall complexity is also $\mathcal{O}(n \log n)$ as required.

(c) Given an integer x, describe a $\mathcal{O}(n \log n)$ algorithm to determine whether there is an element a of A and an element b of B such that $a + b = x$.

Solution: Below are are two different methods for answering this question.
Method 1: first sort B into increasing order, then perform a binary search for $x-a$ in B, for each element a of A. Similarly to (b) in follows that the overall complexity is $\mathcal{O}(n \log n)$.
Method 2: first sort both A and B into increasing order. This can be done in $\mathcal{O}(n \log n)$ time. Then execute the following:

```
$i = 1$; // current position in A (start with minimum value of A)
$j = n$; // current position in B (start with maximum value of B)
while ($i \leq n \wedge j \geq 1$)
  if ($A[i] + B[j] < x$) // $A[i]$ is too small
    $i$++;
  else if ($A[i]+B[j]>x$) // $B[j]$ is too big
    $j$−−;
  else // found $i$ and $j$ such that $A[i] + B[j] = x$
    return current values of $i$ and $j$;
return "impossible";
```

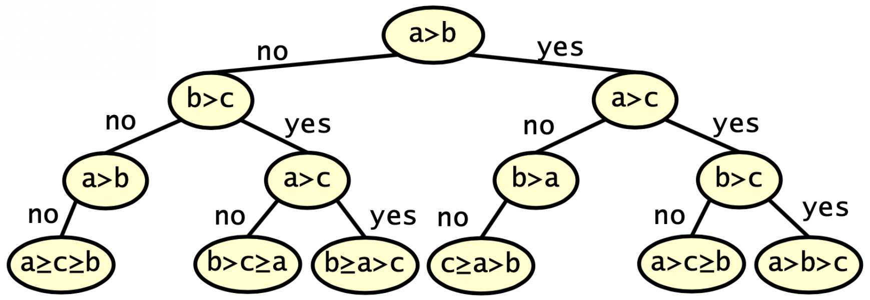
This loop executes in $\mathcal{O}(n)$ time, since the number of iterations is bounded by $2{\cdot}n{-}1$. So the entire algorithm is $\mathcal{O}(n \log n)$.

4. Draw a decision tree to represent the behaviour of bubble sort (given below) when applied to the sequence of elements a, b, c.

```
for (i=0; i<n-1; i++)
 for (j=0; j<n-1-i; j++)
   if (s[j]>s[j+1]) then swap(s,j,j+1)
```

Note: Each internal node of the tree should be labelled by a pair of elements that are compared at that point, and each leaf node by an order between a, b and c (for example $a > c \geq b$).

Solution: A decision tree is given below (explanation for this tree will be given in the tutorial class).



5. Show that no comparison-based algorithm can guarantee to search a sequence of n elements for a particular value in better than $\mathcal{O}(\log n)$ time.

   Hint: recall any comparison-based algorithm can be represented by a decision tree.

Solution: Any comparison algorithm can be represented by a decision tree in which the branch nodes represent the comparisons performed during the search and the leaf nodes the outcome of the search. The number of branch nodes is at least n, since the search must allow for the item searched for being equal to any one of the items in the given sequence. Hence, since the number of branch nodes in a binary tree of size k equals $\lfloor k/2 \rfloor$, we have $n \leq k/2$ which rearranging yields $k \geq n{\cdot}2$, that is the total number of nodes in the tree is at least $2{\cdot}n$. Now, if the height of the tree equals h, we have:

$$\begin{aligned}
2^{h+1} \geq k \Rightarrow\ & 2^{h+1} \geq 2{\cdot}n && \text{since } k \geq 2{\cdot}n \\
\Rightarrow\ & \log_2(2^{h+1}) \geq \log_2(2{\cdot}n) && \text{taking } \log_2 \text{ of both sides} \\
\Rightarrow\ & (h+1){\cdot}\log_2 2 \geq \log_2 2 + \log_2 n && \text{properties of the log function} \\
\Rightarrow\ & h+1 \geq 1 + \log_2 n && \text{since } \log_2 2 = 1 \\
\Rightarrow\ & h \geq \log_2 n && \text{rearranging}
\end{aligned}$$

Now, since the worst case number of comparisons equals to $h{+}1$, it follows that the number of comparisons is no better than $\log_2 n$.

6. Suppose you have in front of you a pile of (several hundred) forms, in random order, each form identified by a unique (7 digit) matriculation number. Describe an algorithm, based on the idea of Radix Sort, that you could use, in practice, to sort the forms into order based on the matriculation number.

Solution: Place the forms in 10 piles, labelled $0, 1, \ldots, 9$, according to the 7th digit. Concatenate the piles in that order. Repeat for the 6th digit, then the 5th and so on. This is a simple variant of radix sort.

7. Draw a `trie` representing the following set of English words that can be formed using the character set $\{a, c, e, r, t\}$:

$$\Big\{\ ace,\ acer,\ acre,\ act,\ are,\ area,\ arete,\ art,\ at,\ ate,\ car,\ care,\ carer,\ caret,\ cart,\ cat,\ cater,\ crate,\ crater,\ create,\ ear,\ eat,\ eater,\ era,\ erect,\ race,\ racer,\ rare,\ rarer,\ rat,\ rate,\ react,\ tact,\ tar,\ tare,\ tart,\ tat,\ tear,\ trace,\ tract,\ treat,\ tree\ \Big\}$$

Solution: It turns out that the number of nodes in the resulting trie is 80. More details are given in the tutorial notes.

How much memory would your `trie` use if

(a) implemented using an array of pointers at each node?

Solution: Using an array of pointers in each node, the trie requires:

- 5 pointers (one for each letter a, c, e, r and t);
- 1 boolean in each node (to say if represents a word);
- the external pointer (for the root node);

therefore we require $5{\cdot}80 + 1 = 401$ pointers and 80 booleans.

(b) implemented using a linked list of pointers at each node?

Solution: Using a linked list of pointers in each node, every node contains:

- two pointers (one for siblings and one for children);
- a character (which the node represents);
- a Boolean (to say if it represents a word);
- the external pointer (for the root node);

therefore we require $2{\cdot}80 + 1 = 161$ pointers, 80 characters and 80 booleans.

Notice the array approach the characters are encoded by the pointers, and hence do not need to be represented explicitly.

8. Describe an algorithm to delete a specified word from a `trie`.

Solution: Care is needed here. The implications of deletion depend on whether the node representing the word is a leaf node. If not, deletion is easy, since no actual `trie` nodes need be deleted; otherwise, the node and its ancestors, as far back as the next node that represents a word, or has another child, must be deleted.