Informatics 2 – Introduction to Algorithms and Data Structures

Solutions for Tutorial 5

1. *We are concerned with the best -case running time for QuickSort. We work with n of the form $2^h - 1$ to help ensure equal splits (plus an excluded item at "split") can be achieved recursively.*

Algorithm QuickSort(A, i, j)

(a) **if** $i < j$ **then**
(b) $\quad split \leftarrow$ partition(A, i, j)
(c) $\quad$ quickSort$(A, i, split - 1)$
(d) $\quad$ quickSort$(A, split + 1, j)$

Algorithm 1

(a) *Give* an actual example *of an input array where QuickSort takes only $O(n \lg(n))$ time to sort. I am only really asking what kind of pattern will cause the array to be split into (roughly) half repeatedly.*

The best case for QuickSort occurs when the depth of the recursive structure (the levels) is $O(\lg n)$. We must split the current subarray roughly in half every time (recursively). Assume the keys are $1, 2, \ldots, n$, where $n = 2^k - 1$ for some k.

Intuitively, in order to partition into two equal halves at the top level, we would expect the last element in the original array to be halfway. Careful consideration of partition will show we need $(n+1)/2$ as the pivot. Otherwise, (*for the first call*) we don't mind where the other elements are input, since partition always takes linear time to execute. However, for recursive calls to partition, it will matter greatly how the other elements were placed in the array.

The best way to construct this array is to build from the bottom up.

Suppose $n = 3$. We want 2 in the final position. We will set A to be $1, 3, 2$. Note that on the call to partition, value 1 (less than 2) is first swapped with itself and then j stops at value 3, and then 3 (the value at $i + 1$) is swapped with the value 2 to give $1, 2, 3$ and split index 2.

Suppose $n = 7$. The way to build the $n = 7$ array is to think about joining two copies of the $n = 3$ arrays, then tweaking them. So we take $1, 3, 2$ followed by $5, 7, 6$, and the middle value 4 at the end. However, recall that we will *swap* the pivot 4 with 5 as the last step of Partition, and this would give $7, 6, 5$ for the right subarray, which is not the right pattern for the recursive call: in fact we want to

create the $n = 7$ array as $1, 3, 2$ followed by $6, 5, 7$, followed by 4: the swap of 4 with 6 at the end will then create a right subarray of $5, 7, 6$. Overall we have the pattern

$$1, 3, 2, 6, 5, 7, 4.$$

Testing this with partition, $i = j$ will happen (before j is moved right) for each of the first three items $1, 3, 2$, each of these items "swapping with itself" and then j moving right. Then j will continue right on each of the large items $6, 7, 5$, with i remaining stationary. At the end, i will be moved right so value 6 can be swapped with the pivot, to give a split into $1, 3, 2$ and $5, 7, 6$. These have the right pattern for $n = 3$.

The procedure is the same for $n = 2^k - 1$, $k > 3$:

- Copy the $(n-1)/2$ length array, shifting all keys upwards by $(n+1)/2$.
- Concatenate the $(n-1)/2$ length array, followed by item $(n+1)/2$, followed by the shifted copy;
- Swap the middle item $(n+1)/2$ with the last item, in order to make $(n+1)/2$ the pivot.

For example, for $n = 15$, we would get the following array:

$$1, 3, 2, 6, 5, 7, 4, 12, 9, 11, 10, 14, 13, 15, 8$$

(b) *Show that the best case running time will always be at least* $cn \lg n$*, for some constant* $c > 0$.

answer: The key to QuickSort is partition, which always takes linear time, ie, at least $b(j - i + 1)$ time for some constant $b > 0$.

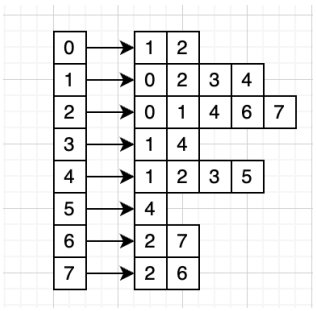
We can give an informal (but accurate) solution by reasoning about the work done at various "levels" of the recursion tree of quicksort. First note that if we consider the first $\lg(n) - 4$ "levels" of the recursion tree, the lowest level can have been broken into, at most, $2^{\lg(n)-4} = n/16$ different subarrays. This is clearly also true for all levels from level 0 (initial recursive call) to level $\lg(n) - 4$ inclusive. Even allowing for the fact that we will crop out $\sum_{h=0}^{\lg(n)-4} 2^h < 2^{\lg(n)-3} = n/8$ "pivots" from the various calls on these levels, at least $7n/8$ items still belong to subarrays of length ≥ 2 throughout. Therefore the total work done by (various calls to) partition across level i (for any $0 \leq i \leq \lg(n) - 4$) is $\Omega(7n/8) = \Omega(n)$. Taking the $lg(n) - 4$ levels together, the total work done by quickSort is at least $\Omega(n \lg(n))$.

Note that sometimes we can do a more formal argument to achieve a proof by induction but these can be fiddly (and involve differentiation). I used to give an alternative proof but it's made more troublesome by this particular variant of QuickSort which crops out the "*pivot*" each time, so I am skipping that alternative.

2. (a) Here are the representations:

0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	1	1
0	1	0	0	1	0	0	0
0	1	1	1	0	1	0	0
0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1
0	0	1	0	0	0	1	0

Table 1: The adjacency matrix representation of G.

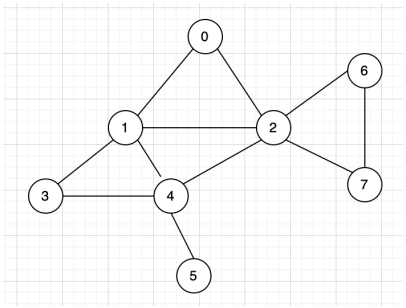


(b) BFS starts at node 0 (the only node in level 0). The exploration of bfsFromVertex will first add the nodes 1 and 2 to the Queue (if that order), these being the "level 1" nodes (Q becomes 1, 2). The next iteration De-Queues 1 and explores its adjacent edges, ignoring the edge to 0 because that one was previously "visited" (Queue becomes 2, 3, 4). The nodes 3 and 4 will belong to level 2 of the search tree. Next, the method De-Queues node 2 and explores all four adjacent edges, pushing non-visited neighbours 6 and 7 onto the Queue(Q becomes 3, 4, 6, 7). 6 and 7 are also level 2.

By now almost all nodes have been visited. The De-Queueing of node 3 actions no changes, but the De-Queueing of 4, causes node 5 to become visited, and the only node on level 3.

(c) DFS starts at node 0 (the only node in level 0), and we run a similar execution, except using the Stack instead of the Queue. We need to be careful with the order of adding items to Stack (according to Adjacency list order) and note that this means that adjacent items to u will then be "popped" in the opposite order. It's a subtle thing that makes a difference.

Here is the graph we are working with:



Our first step is to initialise *visited* to all-false, and to push 0 onto Stack.

Then we do some iterations of the main loop, consisting each time of a "pop", a (possible) updated to the *visited* status of the popped node, and a possible consideration of the nodes adjacent to the popped node.

1st iteration: we pop the 0 ... $finished[0] \leftarrow$ TRUE ... and we push neighbours 1 and 2 on the Stack, in that order.

2nd iteration: we pop the 2 ... $finished[2] \leftarrow$ TRUE, and this implies (0,2) is a edge of the dfs tree.
We must consider all of the adjacent nodes to the popped node: 0, 1, 4, 6, 7
$finished[0]$ is already TRUE, so 0 doesn't get pushed.
The other 4 nodes get pushed on in order 1, 4, 6, 7

3rd iteration: we pop the 7 from the top of the Stack ... we set $finished[7] \leftarrow$ TRUE, (note this implies (2,7) is an edge of the dfs tree.
We must consider all other nodes adjacent to 7 which are 2, 6
We have $finished[2] = True$ already, so 2 doesn't get pushed.
But node 6 gets pushed on top.

4th iteration: we pop the 6 from the top of the Stack ... we set $finished[6] \leftarrow$ TRUE.
Note that the top of the Stack had 2 6s, but we are popping the top one (with parent 7), so (7,6) is an edge of the dfs tree
The Adjacent nodes to 6 are 2 and 7 and these both have $finished[.]$ equals to TRUE. So we push neither of these.

5th iteration: we pop the other 6 from the top of the Stack ... we see that $finished[6]$ is already TRUE.
Hence we don't even look at its adjacent nodes.

6th iteration: we pop the 4 from the top of the Stack ... we set $finished[4] \leftarrow$ TRUE, and note (4 was pushed from 2) the dfs tree has (2,4)
The adjacent nodes to 4 are 1, 2, 3, 5
Node 2 has $finished[2] =$ TRUE, so we skip that, and we push 1, 3, 5 in that order

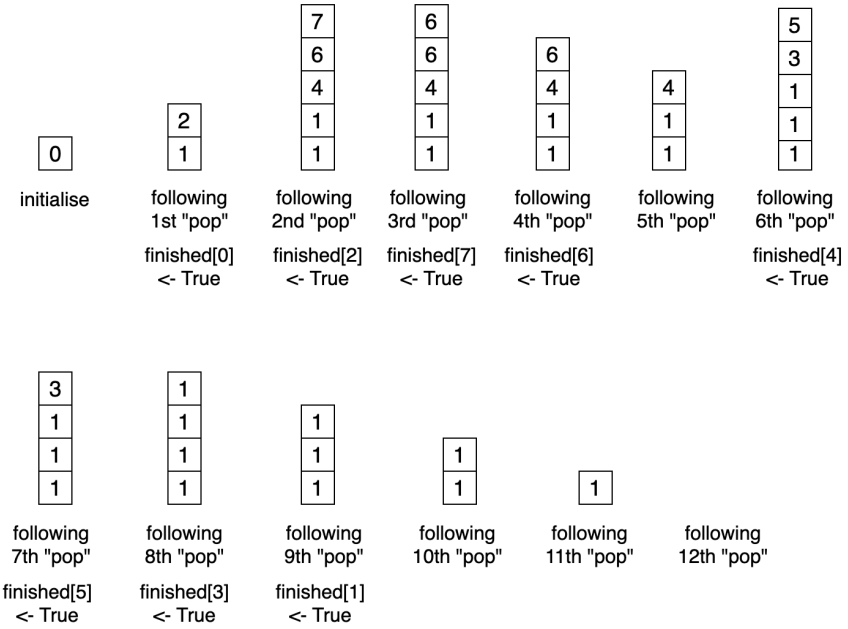
7th iteration: we pop the 5 from the top of the Stack ... we set finished[5] to True, and note (4,5) is an edge of the dfs tree
The only adjacent node to 5 is 4, and finished[4] is already True.
So we don't push anything.

8th iteration: we pop the 3 from the top of the Stack ... we set $finished[4] \leftarrow$ TRUE, and note (4,3) is an edge of the dfs tree
The adjacent nodes to 3 are 1 and 4, and $finished[4]$ is already TRUE.
So we only push node 1 on the top of the Stack.

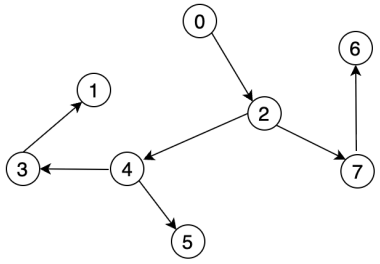
9th iteration: we pop the top 1 from the top of the Stack (there are 4 1s on the Stack at this point) ... we set $finished[1] \leftarrow$ TRUE.
We note (3,1) is an edge of the dfs tree (as 3 was the most recent node to push 1 on top)
The adjacent nodes to 1 all have $finished$ already TRUE.
So we don't push anything onto the top of the Stack

10th-12th iterations: each time we push another 1 off the Stack, but we already have $finished[1] =$ TRUE, so we don't consider adjacent nodes for pushing.

Here is a picture of how the Stack evolves through these iterations:



Here is a picture of the dfs tree created.



(d) We observe that the search trees constructed by BFS and DFS are different.

3. We use proof by contradiction to show this result. To that end, we assume the opposite of what we want - we assume there is *some* edge $(u, v) \in E$ such that $|L(u) - L(v)| > 1$, where $L(\cdot)$ denotes the level of the node. Suppose that u is the node with the lower $L(\cdot)$ value. In that case, we know that u was added to the BFS search tree, and EnQueued onto Q, at a point earlier than v (this is not true for DFS, but it is true for BFS). This means that u was DeQueued before v was, and hence that on u's DeQueueing, that v gets considered in the exploration of adjacent edges of u. There are two possibilities - either that v is not visited yet, and gets marked as visited at this point (in which case $L(v) \leftarrow L(u) + 1$), or that v has become visited in the period after u being visited, in which case $L(v) \in \{L(u), L(u) + 1\}$. In either case this will give $|L(u) - L(v)| \leq 1$, so we have our contradiction.

4. *Suppose we are given an undirected graph $G = (V, E)$ and asked to determine whether the graph is* bipartite - *that is, whether V can be partitioned into two subsets $V = V_1 \uplus V_2$ such that every edge $e = (u, v)$ has one endpoint in V_1 and one endpoint in V_2.*

Show how to answer this question in $O(n+m)$ time.

answer: Let $C \subset V$ be some maximal connected component of G (so the induced subgraph on C is connected, and there is no larger subset $C' \supset C$ which is connected). We note that G is bipartite if and only if each of the maximal connected components is bipartite. Now consider some vertex v in component C, and imagine carrying out breadth-first search from v. We take the convention that we will label this starting vertex "blue". Then we will perform a variant of bfsFromVertex(v) where we colour the nodes on each level as we go, alternating between "red" and "blue" at successive levels (so the neighbours of v will get colour "red", and so on). As we carry out the breadth-first expansion, it may be that case that some neighbours of the current node v have already appeared in the bfs tree (and hence will not be added) - for those, we must check that their previously-allocated colour is the opposite of v's. We carry out this process until there are no new vertices generated.

If this process terminates, and for every v, and every neighbour w of v that was previously visited (and coloured) the already-assigned colour of w is opposite to v, then the connected component containing v is bipartite.

Running this variant of bfs allows us to recognise whether the entire G is bipartite, and the running time is the same as for bfs, $\Theta(n+m)$.

5. *Consider the alternative method for performing topological sorting on a directed acyclic graph $G = (V, E)$: repeatedly find a vertex of in-degree 0, output it, and then remove it and all of its outgoing edges from the graph.*

   (a) *How can you implement this so that the entire algorithm will be $O(|V| + |E|)$?*

answer for (a): To implement this, we will maintain an adjacency list data structure A ($A[v]$ will point to the list of vertices w such that $(v, w) \in E$) to represent the *current digraph* (with some arcs removed), plus an auxiliary array B of length $n = |V|$ such that $B[v]$ is the number of *incoming* edges to v. We will also maintain a list Z of unprocessed "in-degree 0" vertices. Note that we will need to initialise B at the start of our algorithm, and we can do this by first initialising all the $B[v]$ values to 0 ($\Theta(n)$ time), then processing each of the $A[v]$ lists one-by-one, adding 1 to $B[w]$ whenever we see w in an adjacency list. This processing takes $\sum_{v \in V} \Theta(\text{out-degree}(v))$, which is $\Theta(m)$ overall. After this is done, $B[v]$ stores the in-degree of v in the original graph, for every $v \in V$.

Next we do a linear pass ($\Theta(n)$ total) through B searching for vertices which have $B[v] = 0$, adding any such v to Z as we go.

Then we iteratively choose any (the first) item v from L, delete it ($\Theta(1)$ time for a list), and then examine the items in $A[v]$ one-by-one; for every w in the list $A[v]$, we *decrement* the value of $B[w]$ by 1, and if this makes $B[w]$ zero then we also add w to L in $\Theta(1)$ time. Processing the entire list $A[v]$ this way takes $\Theta(\text{out-degree}(v))$ time in total. We only do this work when we remove v from L, which can happen at most once (once $B[v]$ becomes 0 we have exhausted all incoming edges and will never consider $B[v]$ again). Hence overall, we could at most take $\sum_{v \in V} \Theta(\text{out-degree}(v))$ time, which again is $\Theta(m)$ overall.

Therefore we take $\Theta(n) + \Theta(m) + \Theta(n) + \Theta(m)$ time, which is $\Theta(n+m)$.

   (b) *How will you detect that the graph has cycles?*

answer to (b): The graph is acyclic if and only if we can eliminate *all* edges from the graph by following the rule of *delete outgoing edges of a vertex with (current) in-degree 0*. We will notice that the graph has cycles if at some point the list L contains no vertices to work-with, but $A[w]$ is non-empty for some vertices w.

6. *Design an algorithm to sort to sort and return the least k elements of a list that uses the same* partition *subroutine of* quicksort*. How does the worst case execution time of this algorithm compare to that of* quicksort*?*

   answer: The algorithm is similar to quicksort, but where quicksort always recurses on the upper partition, partialQuicksort only recurses when the index of the split is less than $k-1$ (assuming we have 0 as the first index).

Algorithm partialQuicksort(A, i, j, k)

(a) **if** $i < j$ and $k > 0$ **then**
(b) $\quad$ $split \leftarrow$ partition(A, i, j)
(c) $\quad$ partialQuicksort$(A, i, split - 1, k)$
(d) $\quad$ **if** $(split - i) < k$ **then**
(e) $\quad\quad$ partialQuicksort$(A, split + 1, j, (k + i - split))$

Algorithm 2

SHOCKINGLY, the worst-case running-time of this is the same as worst case for quicksort ($\Omega(n^2)$), EVEN WHEN k IS SMALL, SAY $k = 2$. To see this consider the input array

$$1, 2, 3, \ldots, n-1, n.$$

We will show that for the first $n/2$ iterations of partialQuicksort, partition "removes" one extra element each time. This one extra element is always a high element, making the rhs of the partition. Therefore we will not enter lines (d)/(e) of partialQuicksort, and will only have to track one recursive call. The following invariant (IN) will be maintained after each of the first $n/2$ iterations:

(IN) After the hth iteration, the (single) recursive instance of partialQuicksort is:

$$1, 2, 3, \ldots, n-h.$$

Clearly (IN) is satisfied at the beginning, after 0 iterations. Assuming (IN) is satisfied after step h, and $h < n/2$, then on the next $(h+1\text{th})$ iteration of partition, $n-h$ is the pivot. All items are smaller than the *pivot* so partition will do lots of "fake swaps" (item $A[j]$ swapped with itself) before stopping with j and i both pointing at $n-h-1$. Then the final value (the pivot $n-h$) gets swapped with itself and the index/value $n-h$ is returned as the "*split*". Hence we split into the subarray $1, 2, 3, \ldots, n-h-1$ plus the empty array, with $n-h$ returned as the *split*. The pattern of our main (sub)array has not changed, hence (IN) is now satisfied for $h+1$.

Each of the first $n/2$ iterations is guaranteed to take place - after all, we still have not isolated the elements 1 or 2. For each of these first $n/2$ calls, the array contains at least $n/2$ elements, hence partition takes $\Omega(n)$ time. In total we have at least $n/2$ recursive calls, taking $(n/2)\Omega(n)$ in total, which is $\Omega(n^2)$.