Introduction to Algorithms and Data Structures

Tutorial 5

tutor name

University of Edinburgh

Q1: **QuickSort** best-case

Worst-case running time for **QuickSort** is $\Omega(n^2)$.

Average-case running time for **QuickSort** is $\Omega(n \lg n)$.

We are interested in the **best-case** running time.

Q1: **QuickSort** best-case

Worst-case running time for **QuickSort** is $\Omega(n^2)$.

Average-case running time for **QuickSort** is $\Omega(n \lg n)$.

We are interested in the **best-case** running time.

Algorithm QuickSort(A, i, j)

(a) **if** $i < j$ **then**

(b) split $\leftarrow$ **partition** (A, i, j)

(c) **quickSort**(A, i, split - 1)

(d) **quickSort**(A, split+1, j)

Q1(a): **QuickSort** best-case

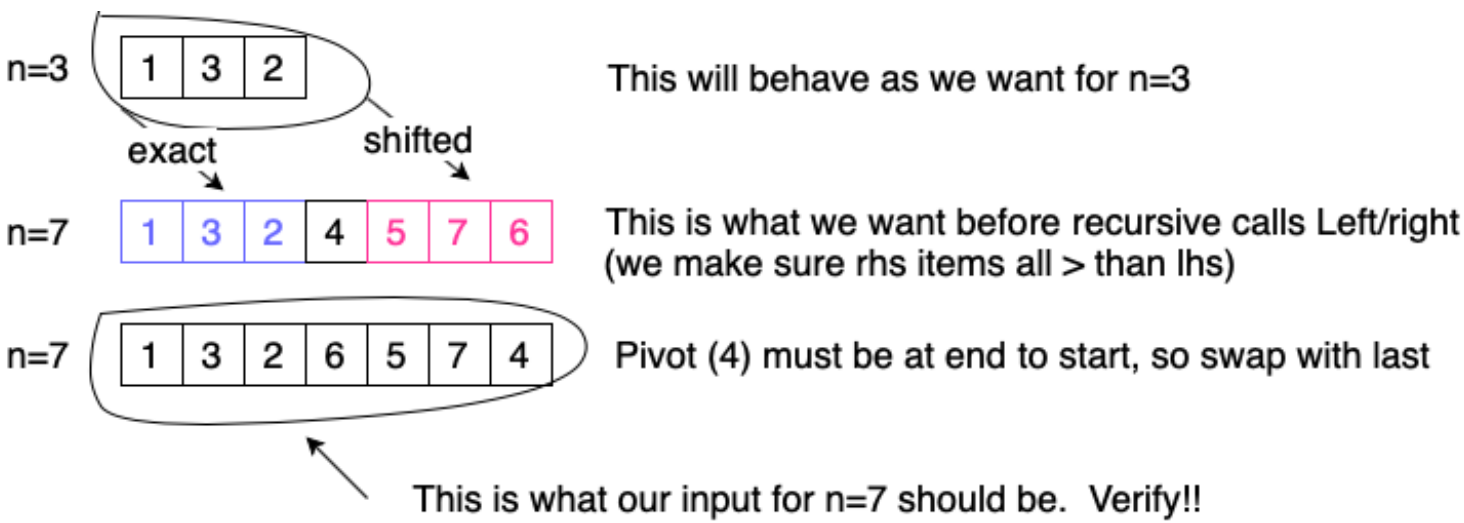
(b) Give an actual example of an input array where **QuickSort** *takes only* $O(n \lg n)$ *time to sort (assume keys are* $1, 2, ..., n$*, where* $n = 2^k - 1$ *for some k).*

We want $(n+1)/2$ as pivot *(why?)

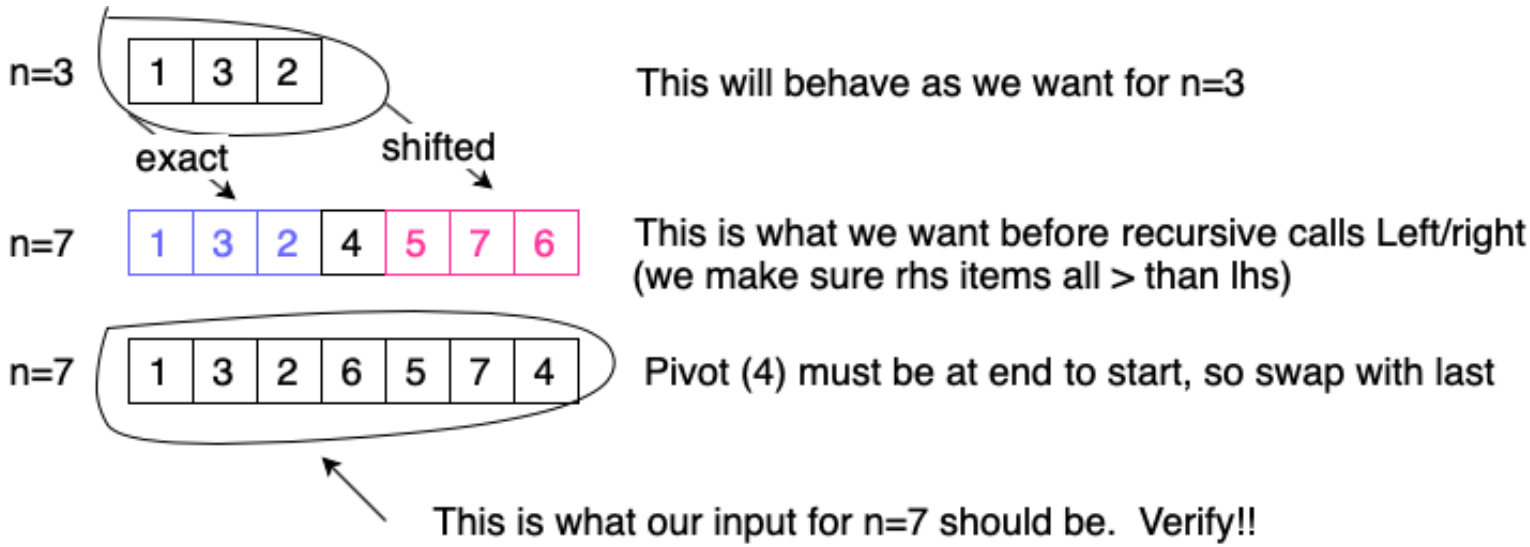
For recursive calls to partition, must carefully place elements in the array.

For $n = 3$, want *last* cell (pivot) holding the *middle item* - eg $1, 3, 2$ order.

Then *build* the answer for $n = 7$ using the pattern for $n = 3$:

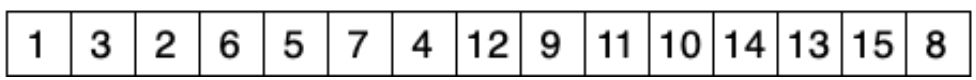


Q1(a): **QuickSort** best-case



n=15: Again we know what we want *after* Partition on lhs and rhs:

Of course pivot started at end ... so n=15 input was:



Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.
(true for all levels from level 0 to level $\ln(n) - 4$ inclusive).

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.
(true for all levels from level 0 to level $\ln(n) - 4$ inclusive).

Must allow for cropping up to $\sum_{d=0}^{\lg(n)-4} 2^d < 2^{lg(n)-3} = n/8$ pivots.

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.
(true for all levels from level 0 to level $\ln(n) - 4$ inclusive).

Must allow for cropping up to $\sum_{d=0}^{\lg(n)-4} 2^d < 2^{lg(n)-3} = n/8$ pivots.

Then $\geq 7n/8$ items still belong to subarrays of length ≥ 2 throughout.

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.
(true for all levels from level 0 to level $\ln(n) - 4$ inclusive).

Must allow for cropping up to $\sum_{d=0}^{\lg(n)-4} 2^d < 2^{lg(n)-3} = n/8$ pivots.

Then $\geq 7n/8$ items still belong to subarrays of length ≥ 2 throughout.

So work by **partition** across level $d = \Omega(7n/8) = \Omega(n)$ (for any d).

Q1 (b): **QuickSort** best-case

(a) Show that **best-case** *running time is always* $> cn \lg n$, $c > 0$.

- key element : **partition**
- partition always takes linear time
- how does partition occur at level i?

Consider the first $\lg(n) - 4$ "levels" of the recursion tree.

Lowest level can have been broken into **at most** $2^{lg(n)-4} = n/16$ subarrays.
(true for all levels from level 0 to level $\ln(n) - 4$ inclusive).

Must allow for cropping up to $\sum_{d=0}^{\lg(n)-4} 2^d < 2^{lg(n)-3} = n/8$ pivots.

Then $\geq 7n/8$ items still belong to subarrays of length ≥ 2 throughout.

So work by **partition** across level $d = \Omega(7n/8) = \Omega(n)$ (for any d).

Total work across all $\lg(n) - 4$ levels $= \Omega(n \lg(n))$.

Q2: Breadth-first search vs. Depth-first search

We represent our collection of found vertices as a set of subtrees.

Each subtree with a particular *root* at level 0.

Found vertices will be at levels 1, 2, 3,...

Q2 (a)

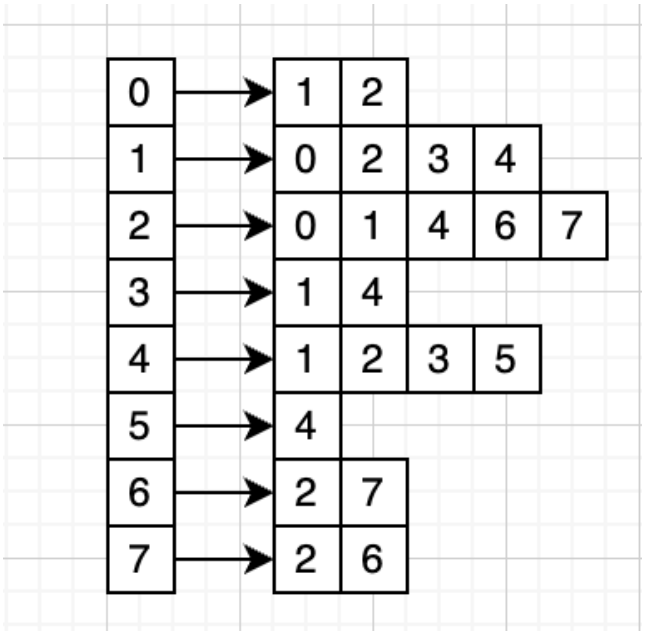
Adjacency matrix

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.

Q2 (a)

Adjcency list



Q2 (b)

BFS starts at node 0 (the only node in level 0).

bfsFromVertex(0) will 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 $(1, 0)$ because 0 was previously "visited".
Queue becomes 2, 3, 4.
The nodes 3 and 4 will belong to level 2 of the search tree.

Next, we De-Queue node 2 and explore 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.

Now almost done. The De-Queueing of node 3 does nothing, but De-Queueing 4, causes node 5 to become visited, and the only node on level 3.

Q2 (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.

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.

Q3

Proof by contradiction:
Assume the opposite of what we want - that 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.

Then u was added to the BFS search tree (and EnQueued onto Q) earlier than v

So u was DeQueued before v was, and hence that on u's De-Queueing, that v gets considered in the exploration of adjacent edges of u. That means *either*

- v is not visited yet, and gets marked as visited at this point (in which case $L(v) \leftarrow L(u) + 1$),
- *OR* 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.

Q4: recognizing bipartite graphs

Given an undirected graph $G = (V, E)$ and asked to determine whether the graph is bipartite *(want to answer in $O(n + m)$ time).*

answer: Ok, $O(n + m)$ is a big hint to use one of bfs or dfs.

With bfs we have better control of "who's on which level (side)"
(level 0 (root) is 1st side, level 1 is 2nd side, level 2 1st-side again, ...)

Assume graph connected (if not, just check each component is bipartite).

Take any vertex v, label it "blue".

Alter bfsFromVertex(v) to *colour* the nodes on each level as we go, alternating between "red" and "blue" at successive levels. Stays $O(n + m)$.

If ever we "see" (adjacent) node u which already sits in the bfs-tree, check its *existing* colour is the opposite to this recent parent.

If we finish (modified) bfsFromVertex(v), and every "previously seen" u passes the colour-check, then the connected component containing v is bipartite.

Q4: adapt dfsFromVertex(v)

Algorithm isBipartite?(G, v)

```
1.  visited[v] ← TRUE;
2.  colour[v] ← blue
3.  Q.enqueue(v)
4.  while not Q.isEmpty() do
5.          v ← Q.dequeue()
6.          for all w adjacent to v do
7.                  if visited[w] = FALSE then
8.                          visited[w] = TRUE
9.                          colour[w] = opposite(colour[v])
10.                         Q.enqueue(w)
11.                 else if colour[w] = colour[w]
12.                         return FALSE
```

Q4: tiny adaptation to top-level of BFS

Algorithm isBipartite?(G)

1. Initialise Boolean array $visited$, setting all entries to FALSE.
2. Initialise *Queue* Q
3. **for all** $v \in V$ **do**
4. **if** $visited[v] =$ FALSE **then**
5. bfsFromVertex(G, v)
6. **return** TRUE

Q5: "cute" topological sort

Given 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: we need to work with (dynamic) *in-degrees*.

Adjacency list data structure A: $A[v]$ is list of vertices w such that $(v, w) \in E$) (representing *current digraph*).

- We can easily remove the full $A[v]$ when we "eliminate" v.
- But looking at $A[v]$ does not tell us anything about edges *incoming* to v.
- How do we update the details of reduced *in-degree* for remaining vertices?

NOTICE the 'algorithm only wants us to know the NUMBER of (remaining) incoming edges to v, not the specifics

Q5: "cute" topological sort

Given 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.

- Define auxiliary integer array B of length $n = |V|$ such that $B[v]$ is the number of *incoming* edges to v.
- Also maintain a list Z of unprocessed "in-degree 0" vertices.

Need to initialise B at the start of our algorithm - by processing each of the $A[v]$ lists one-by-one, adding 1 to $B[w]$ whenever we see w in an adjacency list.

This pre-processing takes $\sum_{v \in V} \Theta(\text{out-degree}(v))$, which is $\Theta(m)$ overall.

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

Q5: "cute" topological sort

the iteration: Choose the first item v from Z, delete it ($\Theta(1)$ time for a list), then examine the items in $A[v]$ one-by-one;

- for every w in $A[v]$, *decrement* $B[w]$ by 1,
- if the decrement makes $B[w]$ be 0, then also add w to Z ($\Theta(1)$ time).

Processing $A[v]$ this way takes $\Theta(\text{out-degree}(v))$ time in total.

We only do this work when we remove v from Z, which can happen at most once (once $B[v]$ becomes 0 we have exhausted all incoming edges and will never consider $B[v]$ again).

Overall we could 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 $\Theta(n + m)$.

Q5: "cute" topological sort

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

answer: 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.

(draw a picture)

Q6: "partial" QuickSort (optional)

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: See Algorithm 1. This 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)

1. **if** $i < j$ and $k > 0$ **then**
2. $\quad$ $split \leftarrow$ Partition(A, i, j)
3. $\quad$ partialQuicksort$(A, i, split - 1, k)$
4. $\quad$ **if** $(split - i) < k$ **then**
5. $\quad\quad$ partialQuicksort$(A, split + 1, j, (k + i - split))$

Q6: "partial" QuickSort

How does the worst case execution time of this algorithm compare to that of quicksort?

SHOCKINGLY, the worst-case running-time is still $\Omega(n^2)$, EVEN WHEN k IS SMALL, SAY $k = 2$.

To see this consider the input array

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

In the $n/2$ iterations of partialQuicksort, Partition "removes" one extra element each time (always a high element, making the rhs of the partition).

Therefore we will not enter lines (d)/(e) of partialQuicksort, and only have one recursive call.

We have an invariant for these first $n/2$ iterations:

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

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

Q6: "partial" QuickSort (optional)

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

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

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 subarray has $\geq n/2$ 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 $\Omega(n^2)$.