- - - *Quicksort*

Anton Gerdelan <gerdela@scss.tcd.ie>

Assign. Common Probs

- **bad time mgmt** (generally we are wrong by x3)
    - do basic parts and submit on day 1 -> pressure off, resubmit after improving, asking Qs
    - extensions are not *free* time - you have other work later -> compounds problems
- original hash

```
int index = first_hash( name, M );
```

- **searching** for name/key

```
( 0 == strcmp( hash_table[index], name ) )
```

- finding gap for **storing** - check if first index in string is empty, or use boolean flags

```
( '\0' == hash_table[index][0] )
```

- second hash offsets (instead of linear probing by +1, linear probing by +second hash)

```
while ( use one of the above here ) {
    index = index + second_hash( name, M );
}
```

Divide-and-Conquer Algorithm Design

1. divide

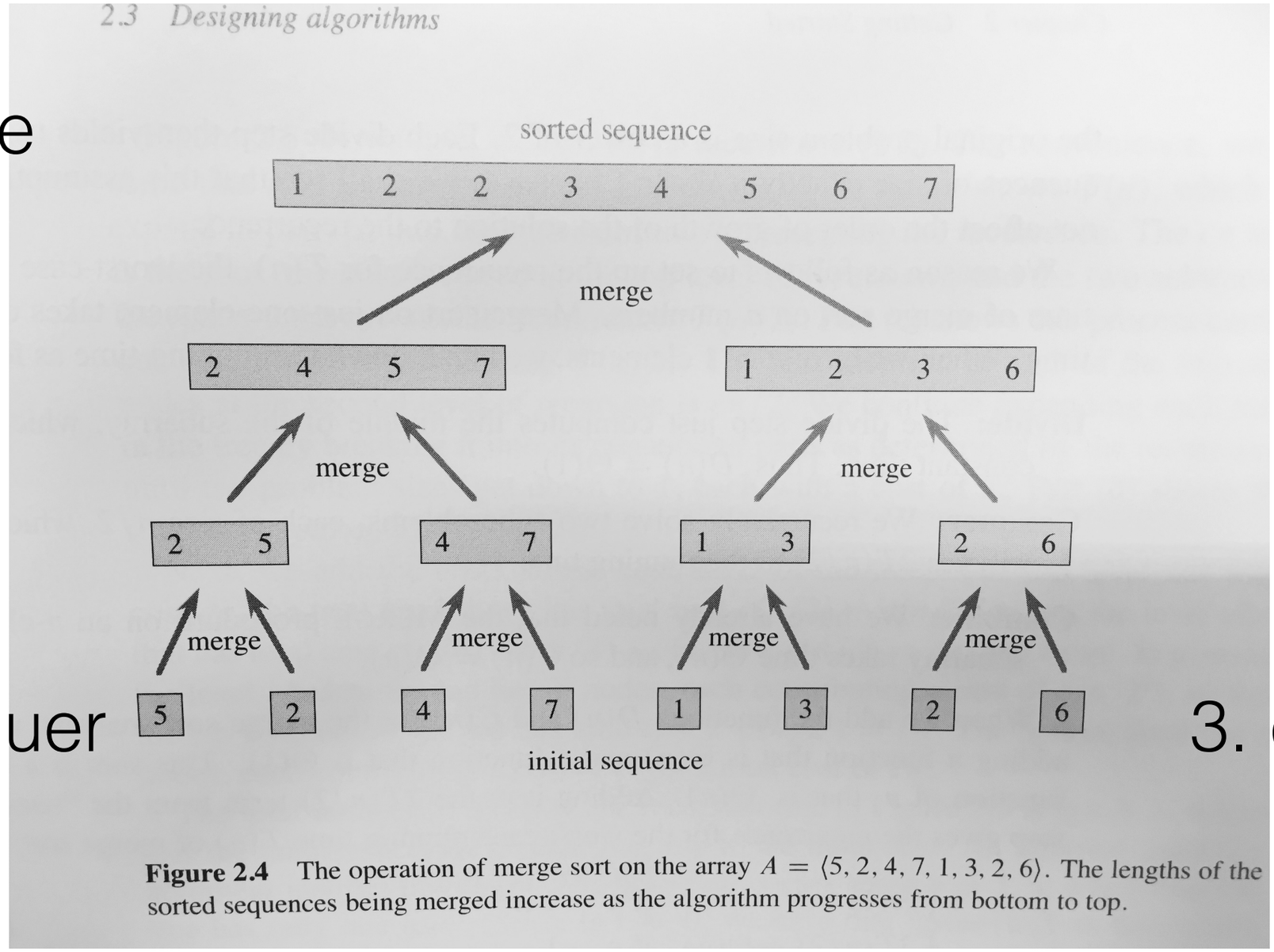

Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

2. conquer

3. combine

Merge sort - from Cormen et al. "Algorithms"

Quicksort

- C.A.R. Hoare, ~1960
- **Divides list** into 2 parts
- Break point is not always middle as in merge sort
- **O(n log(n))** on average with v short inner loop
- O(n^2) worst
- Sorts **in place**.
- Tricky to tune/tweak - if done right is most likely *fastest general sort*

Quicksort Algorithm

- **Divide**: rearrange array into 2 subarrays
  - `A[p..q-1]` and `A[q+1r]`
  - `A[q]` is the **pivot**. It can be any element.
  - All elements left of pivot must be **less than or equal** to `A[p]`
  - All elements right of pivot must be **greater than** `A[p]`

Quicksort Algorithm

- **Conquer**: Sort subarrays by **recursively** calling `quicksort()`

- **Combine**: Entire array is already sorted in place. No merging is required.

Quicksort Pseudo-Code Listing

```
   QUICKSORT( A, p, r )
1      if p < r
2           q = PARTITION( A, p, r )
3           QUICKSORT( A, p, q - 1 )
4           QUICKSORT( A, q + 1, r )
```

- ***A*** is array. ***p*** and ***r*** are first and last (inclusive) indices. ***q*** is pivot index.
- Recursion is fairly clear
- Line 1 halts recursion when array can't be further subdivided
- Writing the `partition()` function is the key

Partitioning Pseudo-Code Listing

```
   PARTITION( A, p, r )
1    x = A[r]
2    i = p - 1
3    for j = p to r - 1
4      if A[j] <= x
5        i = i + 1
6        swap( A[i], A[j] )
7    swap( A[i + 1], A[r] )
8    return i + 1
```

loop once over range
j is *current* index
i is *previous* index

move smaller values leftwards
swap first larger value with end value

Quicksort

- Lots of redundant swapping with self
- Partitioning vaguely resembles elementary sorts
    - likely to sort entire array on first pass
- Efficiency depends on choice of pivot.
- Decide pivot based on data:
    - nearly sorted
    - completed random
    - sorted but in reverse

Reading

- *Cormen et al. Algorithms* has the clearest explanation of quicksort.
- Every algorithms textbook has a chapter on quicksort.
- Lots of extensions/tweaks to quicksort in published papers and code.
- I had a different algorithm/code in the course I took
  - this one has fewer operations
  - you might find better/clearer/optimised code

Blackboard Working

- I'll go over an example that you can find in the *Cormen et al* book.
- This might take a while
- Worth testing your understanding of all algorithms by working through on paper
    - "your pencil is the debugger" + some diagrams

Future Stuff

- Some sorting problems tomorrow (exam-type questions)
- Or we can do some more live coding?
    - requests?
- Sorting assignment
- Trees/Graphs/Heaps
- Searching algorithms