COMP20230: Data Structures & Algorithms
Lecture 14: Sorting (2)

Dr Andrew Hines

Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Sorting (2)

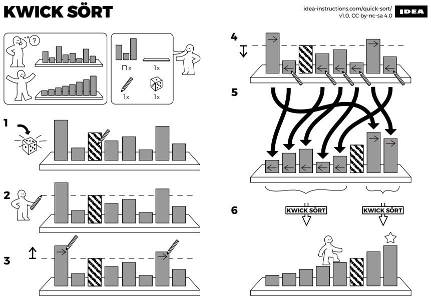
Last Session

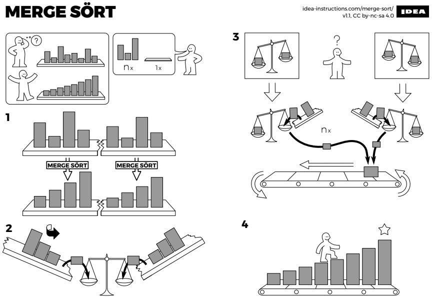
- **bubble sort**: series of *comparison-exchanges* on the n-1 cells of the array (n-1 times)
- **insertion sort**: iteratively “insert” the next unsorted element in the sorted section of the array
- **selection sort:** iteratively select the minimum value in the unsorted section and swap it with the next unsorted element

Outline

Today

Two (recursive) algorithms: Quick sort and Merge sort





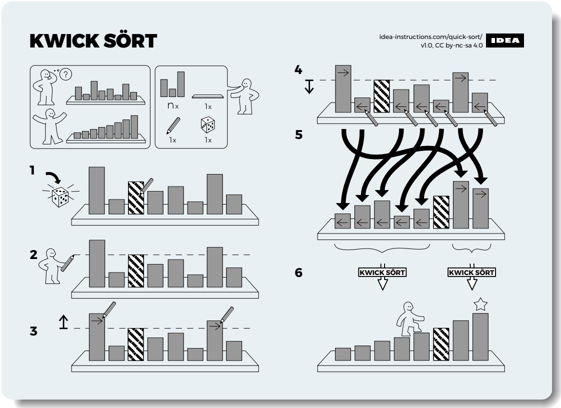
Take home message

Quick sort and merge sort are better: $\mathcal{O}(nlogn)$ in most cases

Image source: `https://idea-instructions.com`

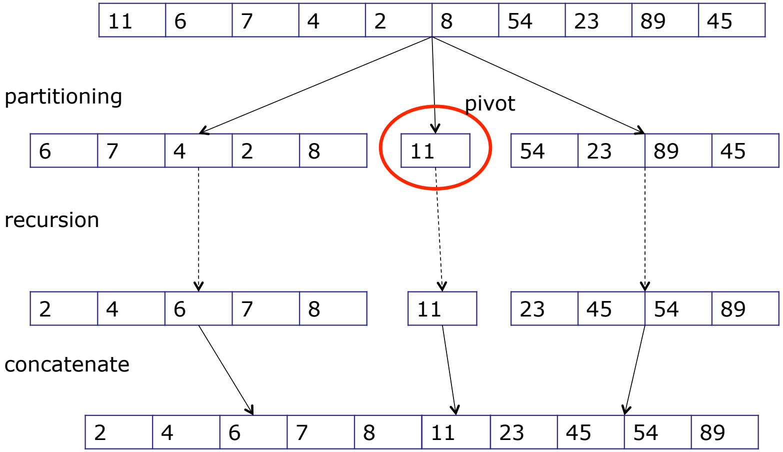
Quick Sort

- Partition list in two "halves", one containing "small" items, the other the "large" ones
- Sort the two "halves" (separately)
- "Glue" two "halves" back together
- The two important notions are: pivot that organises the partitioning – recursion



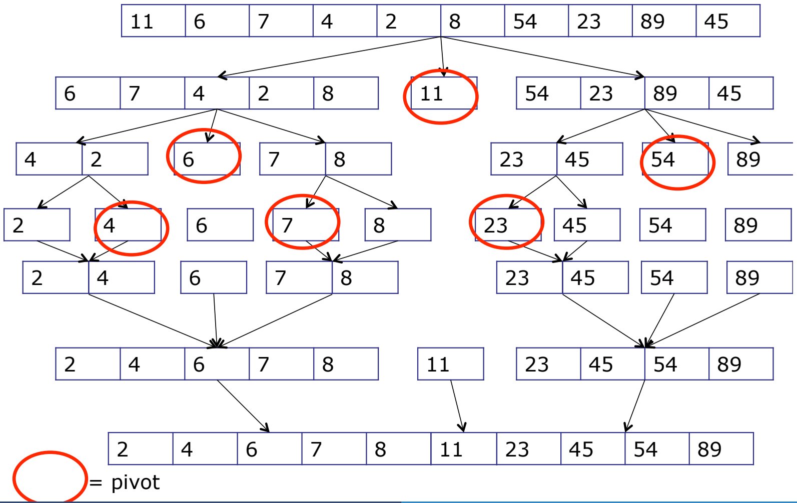
Quick Sort

"Pick a pivot" – partition – recursion (not shown) – concatenate



Quick Sort

"Pick a pivot" – partition – recursion – concatenate



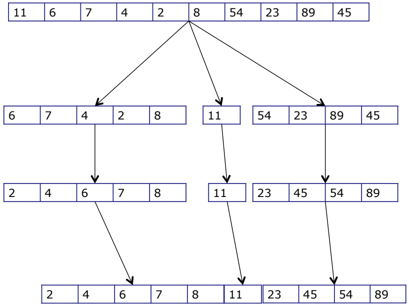
Quick Sort

To sort (sub)Array **A**:

If **A** has fewer than two elements, do nothing

If **A** has at least two elements:

- Select a pivot element **x** from **A**
- Remove elements from **A** and place: those less than x in **S** (smaller); those equal to x in **E** (equal), those greater than x in **G**; those greater than **x** in **G**
- Recursively sort **S** and **G**
- Place elements back in A in the order, first the elements of **S**, then those of **E** and then those of **G**.



Quick Sort: High Level Pseudo-code

```
Algorithm quick_sort:
Input:  A an array
Output:  A is sorted
if $|A| > 1$ then
    pivot $\leftarrow$ some element from A
    Partition elements of A into lists S (smaller
than pivot), E (equal) and G (greater than pivot)
    quick_sort(S)
    quick_sort(G)
    Reconstruct A by copying contents of S, E, G (in
that order) back into A
endif
```

Partitioning elements in S, E and G

```
pivot $\leftarrow$ element (e.g.  first or last) from L
remove pivot element from L
E.add(pivot)
while L is not empty do
    elt $\leftarrow$ get first element of A and remove it
    if elt $<$ pivot then
        S.add(elt)
    else
        if elt = pivot then
            E.add(elt)
        else
            G.add(elt)
        endif
    endif
endwhile
```

Quick Sort: Pseudo-code

Putting it together:

```
Algorithm quick_sort:
Input:  A an array
Output:  A is sorted
if $|A| > 1$ then
    pivot $\leftarrow$ some element from A
    remove pivot element from L
    E.add(pivot)
    while A is not empty do
        elt $\leftarrow$ get first element of A and remove it
        if elt $<$ pivot then
            S.add(elt)
        else
            if elt = pivot then
                E.add(elt)
            else
                G.add(elt)
            endif
        endif
    endwhile
    quick_sort(S)
    quick_sort(G)
    Reconstruct A by copying contents of S, E, G (in that order) back into A
endif
```

Quick Sort: Complexity

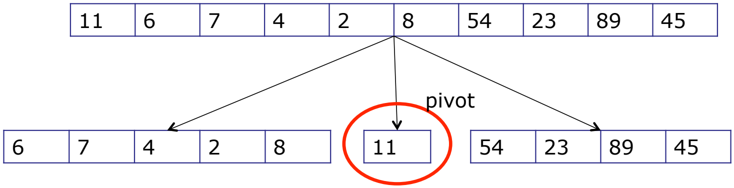
c operations per loop (n elements)= cn

```
Algorithm quick_sort:
Input:  A an array
Output:  A is sorted
if $|A| > 1$ then
    pivot $\leftarrow$ some element from A
    remove pivot element from L
    E.add(pivot)

    while A is not empty do
        elt $\leftarrow$ get first element of A and remove it
        if elt $<$ pivot then
            S.add(elt)
        else
            if elt = pivot then
                E.add(elt)
            else
                G.add(elt)
            endif
        endif
    endwhile

    quick_sort(S)
    quick_sort(G)
    Reconstruct A by copying contents of S, E, G (in that order) back into A
endif
```

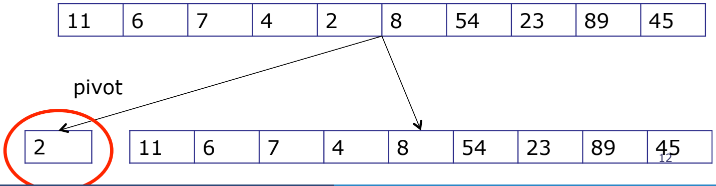
Quick Sort: Complexity

How many calls?

“Normal”/Average case:

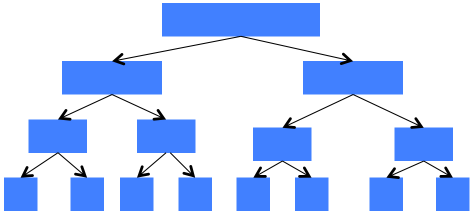


Worse case: (unlucky data input):

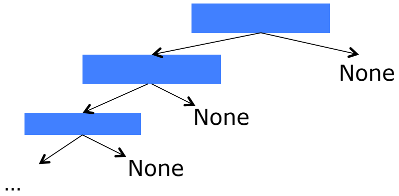


How many calls?

"Normal"/Average case:



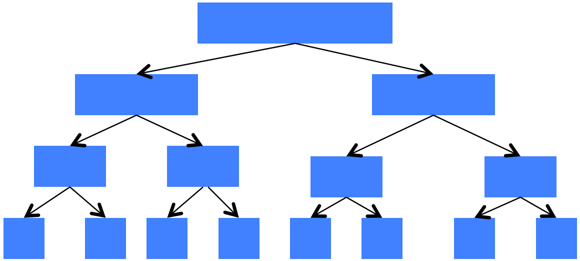
Worse case: (unlucky data input):



Number of Calls (Average)

At each level we have *n* elements in total (in each sub-arrays) hence *cn* operations

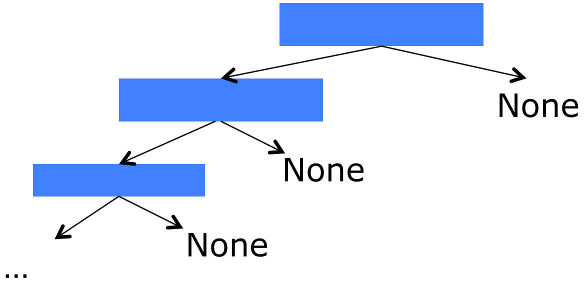
$T(n) = cn * log_2(n)$ which is $\mathcal{O}(nlogn)$



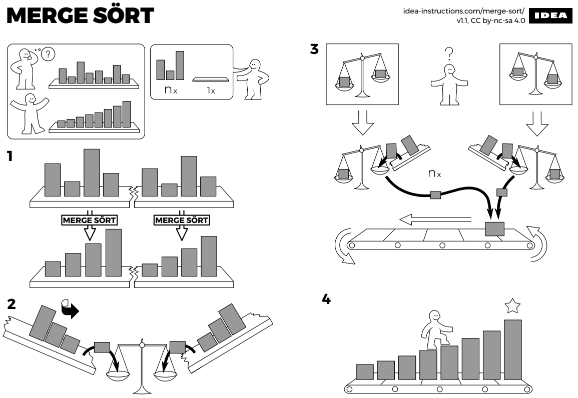
Number of Calls (Worst)

At each level we have 1 element less than the previous one

$T(n) = cn * n$ which is $\mathcal{O}(n^2)$



Merge Sort



Two ideas behind merge sort

Merging **two sorted lists** into one sorted list is easy
Sorting by merging, using carefully chosen sequences of merges

Sounds like a good plan – but how do we do it?

The Merge Problem

Take two (sorted) arrays and merge them into a single sorted array

Input

Two sorted arrays $A1$ and $A2$:

23	45	54	89

12	25	41	96

Output

Single sorted array, A, containing all values from $A1$ and $A2$

12	23	25	41	45	54	89	96

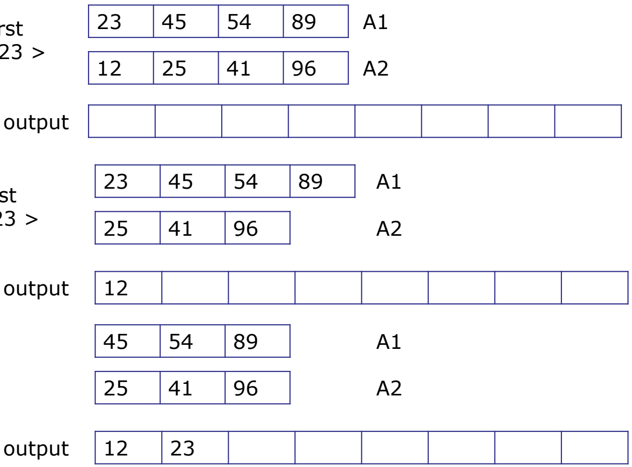
Idea

- Build up A key by key
- at each step remove the smallest remaining key from $A1 \cup A2$
- and append it to the end of A.

Merge Algorithm

compare first elements: 23 > 12?

compare first elements: 23 > 25?



Merge Sort: Pseudo-code

```
algorithm merge_sort
Inputs:  L1, L2 sorted arrays
Output:  L sorted combination of L1 and L2 arrays
while L1 is not empty and L2 is not empty do
    if L1.get(0) $\leq$ L2.get(0) then
        L.add(L1.remove(0))
    else
        L.add(L2.remove(0))
    endif
endwhile
while L1 is not empty do
    L.add(L1.remove(0))
endwhile
while L2 is not empty do
    L.add(L2.remove(0))
endwhile
```

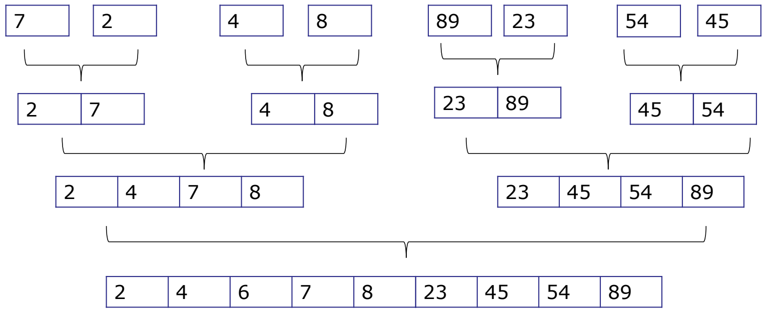
Merge Sort: Complexity

```
algorithm merge_sort
Inputs:  L1, L2 sorted arrays
Output:  L sorted combination of L1 and L2 arrays
while L1 is not empty and L2 is not empty do # 3 op per loop
    if L1.get(0) $\leq$ L2.get(0) then # 3 op per loop
        L.add(L1.remove(0)) # 2 op per loop
    else
        L.add(L2.remove(0)) # 2 op per loop
    endif
endwhile
while L1 is not empty do # 1 op per loop
    L.add(L1.remove(0)) # 2 op per loop
endwhile
while L2 is not empty do # 1 op per loop
    L.add(L2.remove(0)) # 2 op per loop
endwhile
```

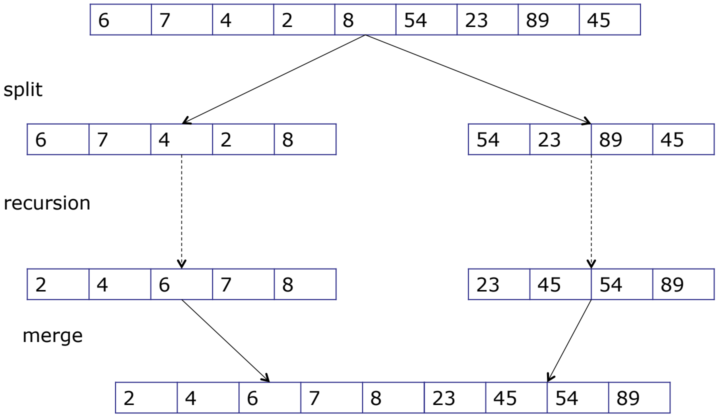
worse case: $8(n+m)$, where n is size of L1, m is size of L2.
$\mathcal{O}(n+m)$ or $\mathcal{O}(n)$

Sorting by merging

Idea: can sort using carefully chosen pattern of merges



Merge sort: sorting a single unsorted array



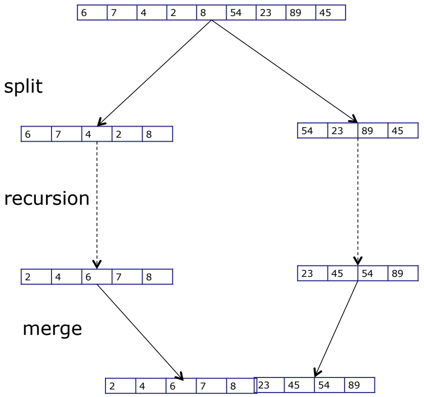
Merge sort

To sort (sub)Array A:

If **A** has fewer than two elements, do nothing

If **A** has at least two elements:

- Split **A** into two arrays **A1** and **A2** of equal size (+/- 1)
- Recursively sort **A1** and **A2**
- Transfer elements back into **A** by merging (sorted) **A1** and (sorted) **A2**.



Merge Sort Pseudo-code

```
algorithm merge_sort
Input:  A an array
Output:  A is sorted
if |A| $>$ 1 then
    for j $\leftarrow$ 0 to |A|/2 do
        add A[j] to A1
    endfor
    for j$\leftarrow$ $|A|/2$ +1 to $|A|$ do
        add A[j] to A2
    endfor
    merge_sort(A1)
    merge_sort(A2)
    A = merge(A1, A2)
endif
return A
```

Merge Sort: Complexity

```
algorithm merge_sort
Input:  A an array
Output:  A is sorted
if |A| $>$ 1 then
    for j $\leftarrow$ 0 to |A|/2 do # 1 op per loop (n/2)
        add A[j] to A1 # 1 op per loop
    endfor
    for j$\leftarrow$ $|A|/2$ +1 to $|A|$ do # 1 op per loop (n/2)
        add A[j] to A2 # 1 op per loop
    endfor
    merge_sort(A1) # 1 op
    merge_sort(A2) # 1 op
    A = merge(A1, A2) # T(n)= 8*(n/2 + n/2)=8n
endif
return A # 1 op
```

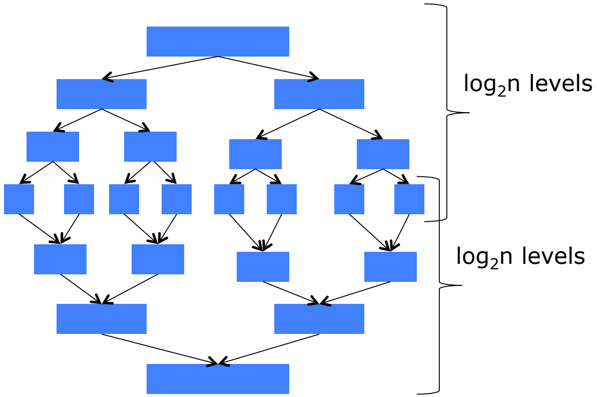
$$T(n) = 1 + 2 * n/2 + 2 * n/2 + 2 + 8n + 1 = 10n + 3$$
$$\mathcal{O}(n)$$

Number of calls

At each split level there are $2n$ operations: $T_{split}(n) = 2nlog_2(n)$ which is $\mathcal{O}(nlogn)$

At each merge level we have $8n$ operations: $T_{split}(n) = 8nlog_2(n)$ which is $\mathcal{O}(nlogn)$

So overall: $\mathcal{O}(nlogn)$



Conclusions

Merge_sort and quick_sort

Improve on the $\mathcal{O}(n^2)$ algorithms seen in last lecture for most realistic scenarios.

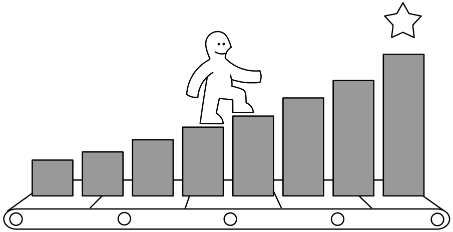
In theory, the data can cause them to be worse but *usually in practice* they are much better: $\mathcal{O}(nlogn)$.

Conclusions

Merge_sort and quick_sort

Improve on the $\mathcal{O}(n^2)$ algorithms seen in last lecture for most realistic scenarios.

In theory, the data can cause them to be worse but *usually in practice* they are much better: $\mathcal{O}(nlogn)$.

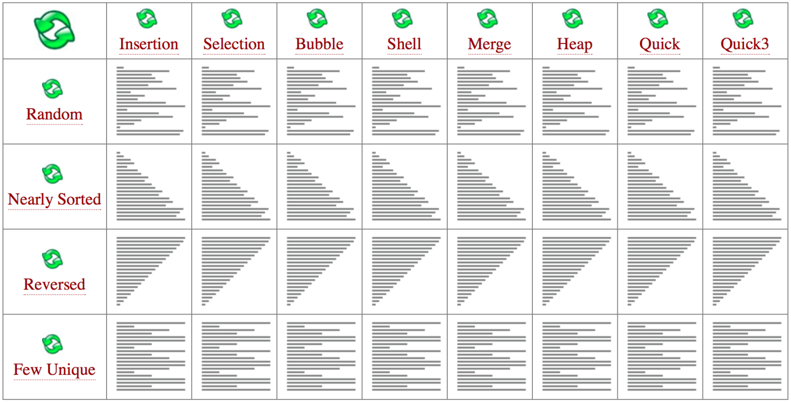


But don't rely on "usually in practice" if you want to be a good software designer!

Visualisation can help

To understand more about sorting and learn tricks that can be applied in your own algorithms read more. Textbooks cover all the main sort algorithms and even Wikipedia has great visualisations.

Quick Sort: `https://www.youtube.com/watch?v=3San3uKKHgg`



From: http://www.sorting-algorithms.com — Try animated version at: https://www.toptal.com/developers/sorting-algorithms