Edinburgh Napier UNIVERSITY

ALGORITHMS & DATA STRUCTURES SET09117

LECTURE 05:

SEARCHING & SORTING ALGORITHMS

Dr Simon Wells
s.wells@napier.ac.uk
http://www.simonwells.org

Edinburgh Napier UNIVERSITY

TL/DR

Computers have probably spent more time sorting data than performing any other task

At the end of this topic you will be able to:

- Sort data (to make it easier/quicker to find data)
- Recognise and apply a variety of sorting methods
- Have some insight into some of the strategies used in sorting

Edinburgh Napier UNIVERSITY

OVERVIEW

- Look at some common sorting algorithms, e.g.
  - Bubble Sort,
  - Insertion Sort,
  - Selection Sort,
  - Merge Sort,
  - Quicksort

& lot's more distinct algorithms (+ many variations)

SORTING

Intuitively:

A sorting algorithm puts elements of an **unordered collection** into a **particular order**,

Edinburgh Napier UNIVERSITY

SORTING & SEARCHING

- We've built up a pragmatic need for sorting during the last few topics - searching is inefficient on unsorted data as the size of the data grows
- Leads to recognising that:
  - Efficient sorting can be important for optimising the use of other algorithms, e.g. searching a sorted list instead of an unsorted list
    - Simple (but inefficient): Bubble Sort (& variants)
    - Simple (efficient on small data): Insertion & Selection Sorts
    - Efficient: Heap, Merge, QuickSort

- From a research perspective:
  - Sorting is more or less a *solved* problem.
  - That said, new algorithms still being developed
    - Tim Sort (2002) & Library Sort (2006)
  - Niche cases can require special consideration

Edinburgh Napier UNIVERSITY

WHY SORT?

- Generally:
  - Easier to find items
  - Canonicalising data - putting into a standard form for human output
  - Easier to partition dataset
- Specifically:
  - Preprocessing can lead to faster search
  - Grouping for: counting, de-duping,
    - NB. Alternatives, such as hashing can work but sorting can be fast, space-efficient, deterministic, and scalable
  - Simplification: algorithms can be very simple if they are dealing with sorted data

Edinburgh Napier UNIVERSITY

SORTING

- More formally:
  - Input is a sequence s= <e^1, e^2, …, e^n> of n elements
  - Each element has an associated key k^i = key(e^i)
  - Keys come from an *ordered universe* - i.e. there is a linear order over the keys
  - For ease of use: extend the ordering from keys to elements - e <= e' *iff* key(e) <= key(e')
  - The task of ordering is to produce a sequence s' = <e'^1, … e'^n> such that s' is a permutation s and such that $e'^1 < e'^2 < \ldots < e'^n$
  - Where the ordering of equivalent elements is arbitrary

CONDITIONS

- It is expected that a sorting algorithm will also satisfy the following conditions:
  - Output in non-decreasing order
    - *An element is no smaller than the previous element according to the desired ordering - NB. This allows equivalent size items to exist int he same collection*
  - Output is a **permutation**
    - *This is a reordering of the collection that retains all of the original elements*

Edinburgh Napier UNIVERSITY

COMPARISON RELATIONS

- Various comparison relations for data can make sense
  - But numeric and lexicographical order are obvious choices for many sequences, strings, tuples, etc.
  - There can be various lexicographical orderings for strings, e.g.
    - Are upper and lower case characters equivalent?
    - What about accented characters?

Edinburgh Napier
UNIVERSITY

STABLE SORTING

- Sorting algorithms can be distinguished on the basis of whether the sorts that they generate are **stable**
- That is, whether repeated elements in the input appear in the same order in the output
- This doesn't have to be a problem but can be important when considering real-world problems:
  - Two members of the input can sort equivalently but actually refer to different things - sorting doesn't occur in isolation from the rest of the world

Edinburgh Napier
UNIVERSITY

BOGO SORT

Edinburgh Napier UNIVERSITY

OVERVIEW

- "Bogus" sort - also known as slow, stupid, or monkey sort
- Generate all permutations of your data until you get the right (sorted) one
  - *i.e. Consider a deck of cards. Throw them in the air then pick them up on at random. Repeat until you pick up a sorted deck.*
- Pretty bad performance: O((n+1)!) in average & worst cases
- Relies on the idea that there is some probability of getting the right permutation at each try

Edinburgh Napier
UNIVERSITY

ALGORITHM

- While the collection is not in order:
    - Shuffle it
        - Iterate

Edinburgh Napier UNIVERSITY

BAD SORTS

- Not useful
- But an interesting starting place for considering other sort algorithms, i.e. *what is it about Bogo Sort that makes it bad?*
- People are interested in studying perversely poor sorting algorithms (can *give insight into the processes of creating better ones*):
  - Bozosort - If list not sorted: pick two random items & swap them
  - Gorosort - if list not sorted: randomly permute a subset of the collection

Edinburgh Napier
UNIVERSITY

BUBBLE SORT

Edinburgh Napier UNIVERSITY

PROPERTIES

- A type of **comparison sort**
- Simple (but so inefficient that it is not considered practical particularly as data size grows)
- On each pass, sorts largest value to end of collection
  - So sorted list grows back towards beginning on each iteration
- However: can efficiently check if data is sorted:
  - If a single pass has no swaps then data is sorted
  - Means we can exit the algorithm without completing all the iterations

Edinburgh Napier UNIVERSITY

ALGORITHM

- Compare first pair of items
  - If first great than second then swap them
  - Get next pair of items (shift window along so that second item of first pair is now first item of second pair)
  - If first great than second then swap them
  - Repeat until end of data is reached
- Repeat from beginning again until one pass through without any swaps

Edinburgh Napier UNIVERSITY

PERFORMANCE

- Very sensitive to size of dataset
- Positions of elements play a large part in performance
- Best case scenario: data already sorted: O(n)
  - Do a single iteration through the data
- Worst case scenario: data in reverse order: O(n^2)
- Small elements at end take many passes to *bubble* through to the other end

Edinburgh Napier UNIVERSITY

EXAMPLE

First Pass:
(**5 1** 4 2 8) (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) (**1 4** 2 5 8)
(1 **4 2** 5 8) (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) (1 2 **4 5** 8)
(1 2 4 **5 8**) (1 2 4 **5 8**)
Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) (**1 2** 4 5 8)
(1 **2 4** 5 8) (1 **2 4** 5 8)
(1 2 **4 5** 8) (1 2 **4 5** 8)
(1 2 4 **5 8**) (1 2 4 **5 8**)

Edinburgh Napier UNIVERSITY

STANDARD IMPLEMENTATION

```
for(idx=0; idx<SIZE-1; idx++)
{
    for(target=0; target<(SIZE-1)-idx; target++)
    {
        if(arr[target] > arr[target+1])
        {
            tmp = arr[target];
            arr[target] = arr[target+1];
            arr[target+1] = tmp;
        }
    }
}
```

Edinburgh Napier UNIVERSITY

OPTIMISED

```
for(idx=0; idx<SIZE-1; idx++)
{
     swapped = 0;
     for(target=0; target<(SIZE-1)-idx; target++)
     {
          if(arr[target] > arr[target+1])
          {
               tmp = arr[target];
               arr[target] = arr[target+1];
               arr[target+1] = tmp;
               swapped = 1;
          }
     }
     if(!swapped)
          break;
}
```

Edinburgh Napier UNIVERSITY

FURTHER OPTIMISATION

- Assume that we don't have to check the last x items where x = outer loop counter
  - Each iteration sorts largest item to final location at end
  - So inner loop iterates as far as size -x
  - But multiple items could be sorted to the position in each pass
- Could further optimise this:
  - Keep track of position of last swap in an iteration
  - Next iteration only needs go that far
  - Can enable more data to be skipped

Edinburgh Napier UNIVERSITY

IMPROVING BUBBLE SORT

- Interesting point:
  - Element moving towards end of collection moves faster
    - Because comparisons & swaps *overlap* - an element can take part in successive swaps until it reaches it's final destination in a single iteration
- Element moving towards start of collection move more slowly
  - Only moves one place per iteration
  - If smallest element is at the end then this can take *n*-1 passes to get to it's position
- Referred to as rabbits & turtles (*Aesop's* Fables) - Has lead to various *improvements*, & variations e.g. cocktail, sort comb sort

Edinburgh Napier
UNIVERSITY

BUBBLE SORT VIDEO

Edinburgh Napier
UNIVERSITY

SELECTION SORT

Edinburgh Napier UNIVERSITY

OVERVIEW

- Simple & can have performance advantages over more complex algorithms
- Inefficient on larger lists & generally performs worse than insertion sort
- Can be implemented *in place* - needs no additional storage beyond input array and loop counters, etc.
- Does no more than *n* swaps so can be useful when swapping is an expensive operation
- Constructs a **final sorted list**
  - After *n* iterations, at least *n* elements are in their correct final place

Edinburgh Napier UNIVERSITY

ALGORITHM

- Consider two lists: sorted (initially empty output) & unsorted (input):
  - (standard) Could be two physically separate arrays
  - (in-place) Could be one array - Imagine a partition wall that slides along from one end separating sorted part from unsorted

- Iterate over the input until you find the smallest value
  - Append the *selected* value to the output
  - Repeat

Edinburgh Napier UNIVERSITY

IMPLEMENTATIONS

- Various ways to implement:
  - We looked at an *in-place* implementation - using the same array to hold the sorted & unsorted data
  - Trivially easy to use a second array if you don't care about memory usage - e.g. consider space constraints
  - Unless you are using an implementation from a standard library (which you should as a general rule) - implementation approach is a pragmatic & problem-dependent decision

Edinburgh Napier UNIVERSITY

ALGORITHM (STANDARD)

- Start with an input (unsorted collection) and an empty output
- Iterate through the collection looking for the smallest element
  - Append the smallest element to the output
  - Continue until input is empty and output contains sorted collection

EXAMPLE: SELECTION SORT (STANDARD IMPLEMENTATION)

```
input = {64 25 12 22 11} | output = {}
input = {25 12 22 64}    | output = {11}
input = {25 22 64}       | output = {11, 12}
input = {25 64}          | output = {11, 12, 22}
input = {64}             | output = {11, 12, 22, 25}
input = {}               | output = {11, 12, 22, 25, 64}
```

Edinburgh Napier UNIVERSITY

ALGORITHM (IN PLACE)

- Start with an input (unsorted collection)
- Iterate through the collection from start to end looking for the smallest element
  - Swap the smallest element with the start position
  - Increment the start position
  - Continue until start position equals end position

EXAMPLE: SELECTION SORT (IN PLACE)

```
input = {64 25 12 22 11}
input = {11 25 12 22 64}
input = {11 12 25 22 64}
input = {11 12 22 25 64}
input = {11 12 22 25 64}
```

CODE: SELECTION SORT (IN PLACE)

```
for(idx=0; idx<SIZE-1; idx++)
{
    for(target=idx+1; target<SIZE; target++)
    {
        if(arr[idx] > arr[target])
        {
            tmp = arr[idx];
            arr[idx] = arr[target];
            arr[target] = tmp;
        }
    }
}
```

Edinburgh Napier
UNIVERSITY

SELECTION SORT VIDEO

Edinburgh Napier
UNIVERSITY

INSERTION SORT

Edinburgh Napier
UNIVERSITY

INSERTION SORT

- A bit like sorting a hand of playing cards
    - Start at one end,
        - if next card is bigger than current then move card to the right place in the hand,
        - otherwise look at the next card
- Relatively efficient on small lists and lists that are mostly sorted
- In place sorting so good use of space but *Involves quite a bit of shifting elements*
- Constructs sorted list one element at a time but *not final sorted list*
- But builds our list as we proceed. Number of sorted items is equal to number of iterations + 1

Edinburgh Napier UNIVERSITY

ALGORITHM

- Simple: Remove element from input, shift everything along until the right position is found. Replace in sorted position

- Take elements from unsorted list one by one,
- Compare to largest in sorted list (next to it)
  - if larger stay otherwise compare to next until smaller
    - Shift all elements up to make space then insert into new gap

EXAMPLE: INSERTION SORT

- Element under consideration is underlined
- Element last considered is **bold**

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

Edinburgh Napier UNIVERSITY

CODE: INSERTION SORT

```
for(cmp=1; cmp<SIZE; cmp++)
{
    for(idx=0; idx<cmp; idx++)
    {
        if(arr[idx] > arr[cmp])
        {
            tmp = arr[idx];
            arr[idx] = arr[cmp];

            for(shift = cmp; shift > idx; shift--)
                arr[shift] = arr[shift-1];
            arr[shift+1] = tmp;
        }
    }
}
```

Edinburgh Napier
UNIVERSITY

INSERTION SORT VIDEO

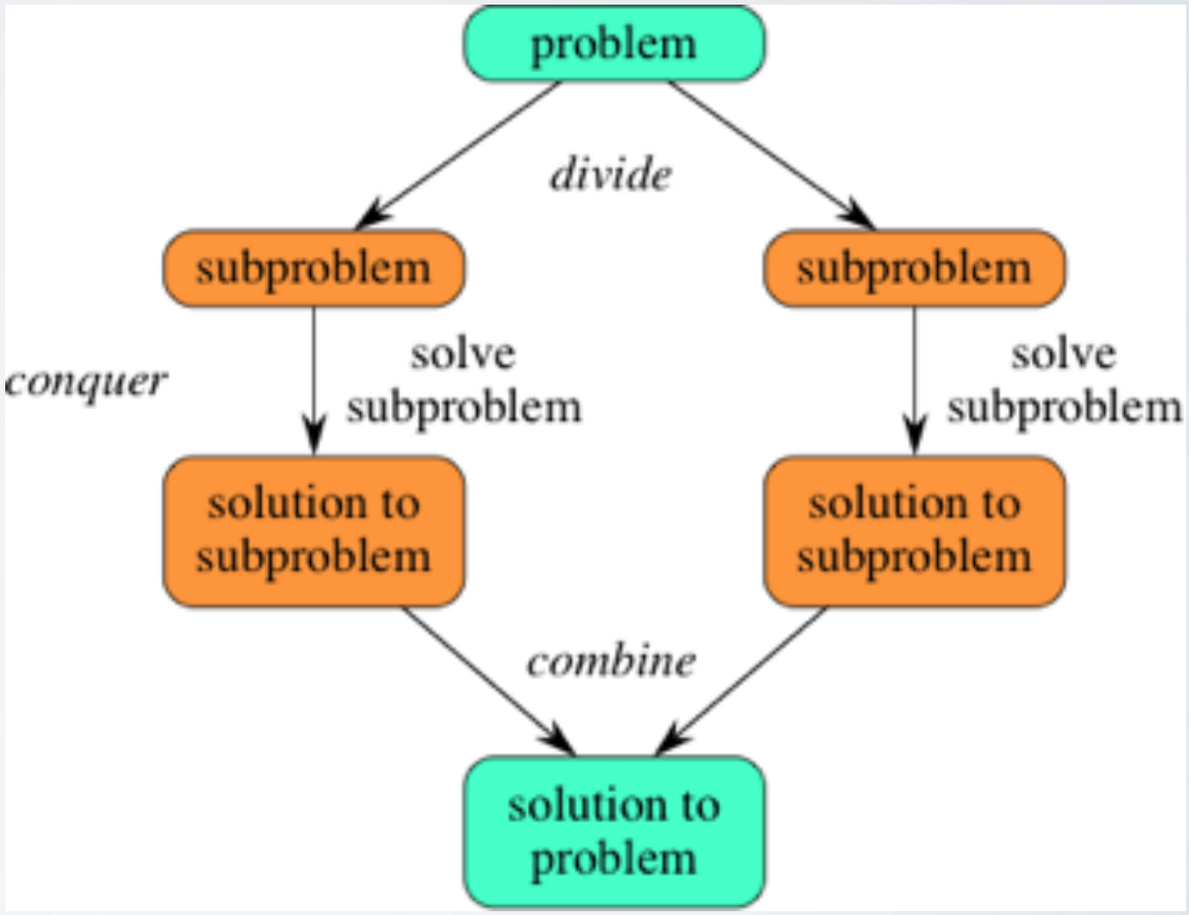
Edinburgh Napier UNIVERSITY

SOME ELEMENTS OF SORTING STRATEGIES

- So far:
  - Some strategies for searching for the target (i.e. lowest) element
    - Compare adjacent pairs
    - Compare between two points
- Some strategies for getting element to a better position:
  - Swap
  - Shift
- Some strategies for building the sort
  - in-place
  - out-of-place

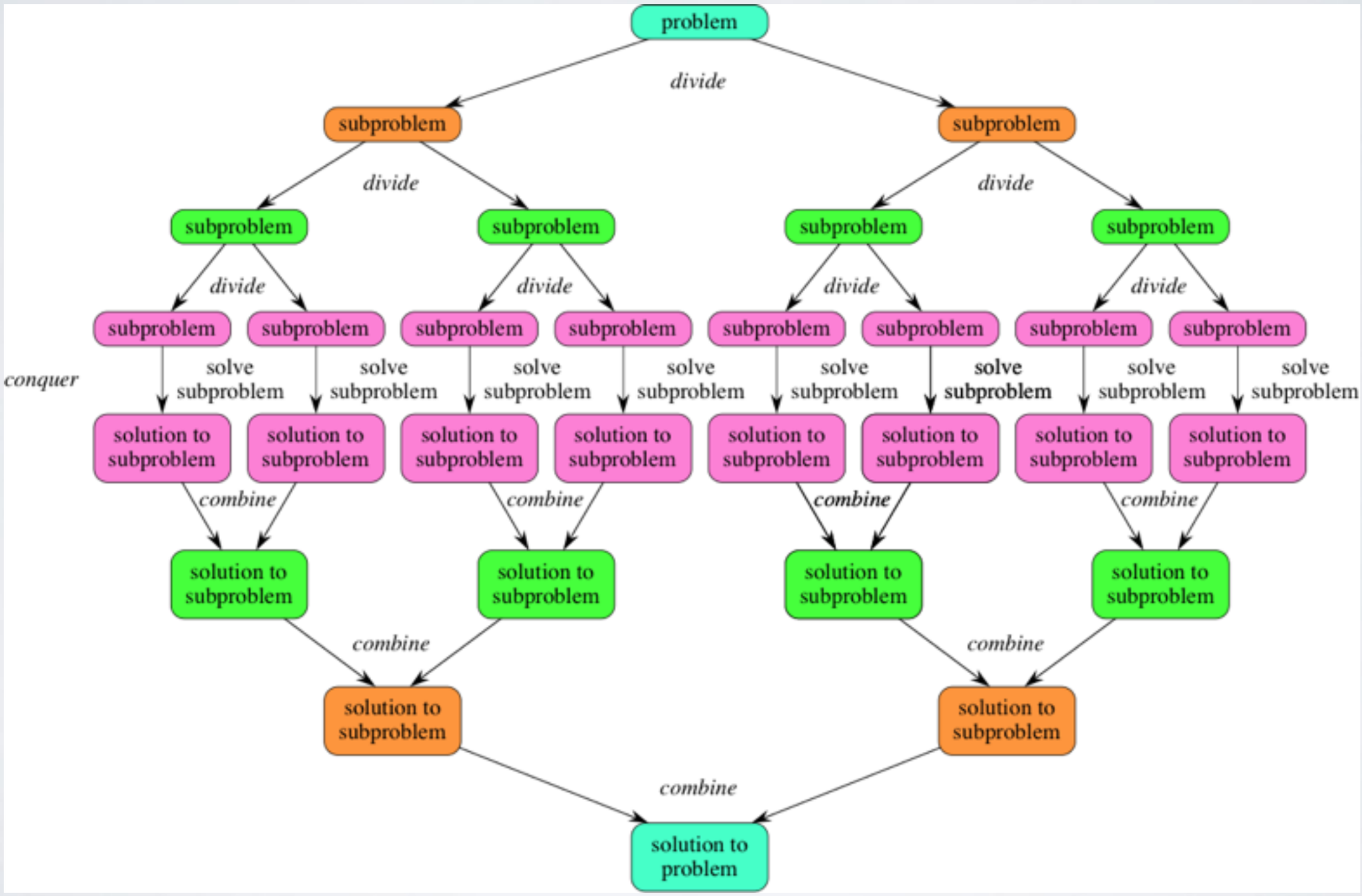
DIVIDE & CONQUER (RECURSIVE) ALGORITHMS

- **Divide** problem into subproblems
- **Conquer** the subproblem by solving recursively
  - If small enough, solve subproblem as base case
- **Combine** solutions to subproblems into solution for original problem



Edinburgh Napier
UNIVERSITY

DIVIDE & CONQUER (3 STEPS)



Edinburgh Napier
UNIVERSITY

MERGE SORT & QUICKSORT

- Algorithms like selection sort & insertion sort can take a long time to run when the input array is large

- Merge sort & Quick sort have better running times

- Rely heavily on recursion & follow the divide & conquer paradigm

  - Problem broken into subproblems similar to the original but smaller & simpler

  - *NB. Must be a base case (the point at which the recursion bottoms out)*

Edinburgh Napier
UNIVERSITY

MERGE SORT

Edinburgh Napier UNIVERSITY

MERGE SORT

- Old - invented ~1945 (von Neumann *again*)
- Because we divide up the elements of the collection & at most compare each element to half the collection -
  - O(*n logn)* performance in best, worst, & average cases

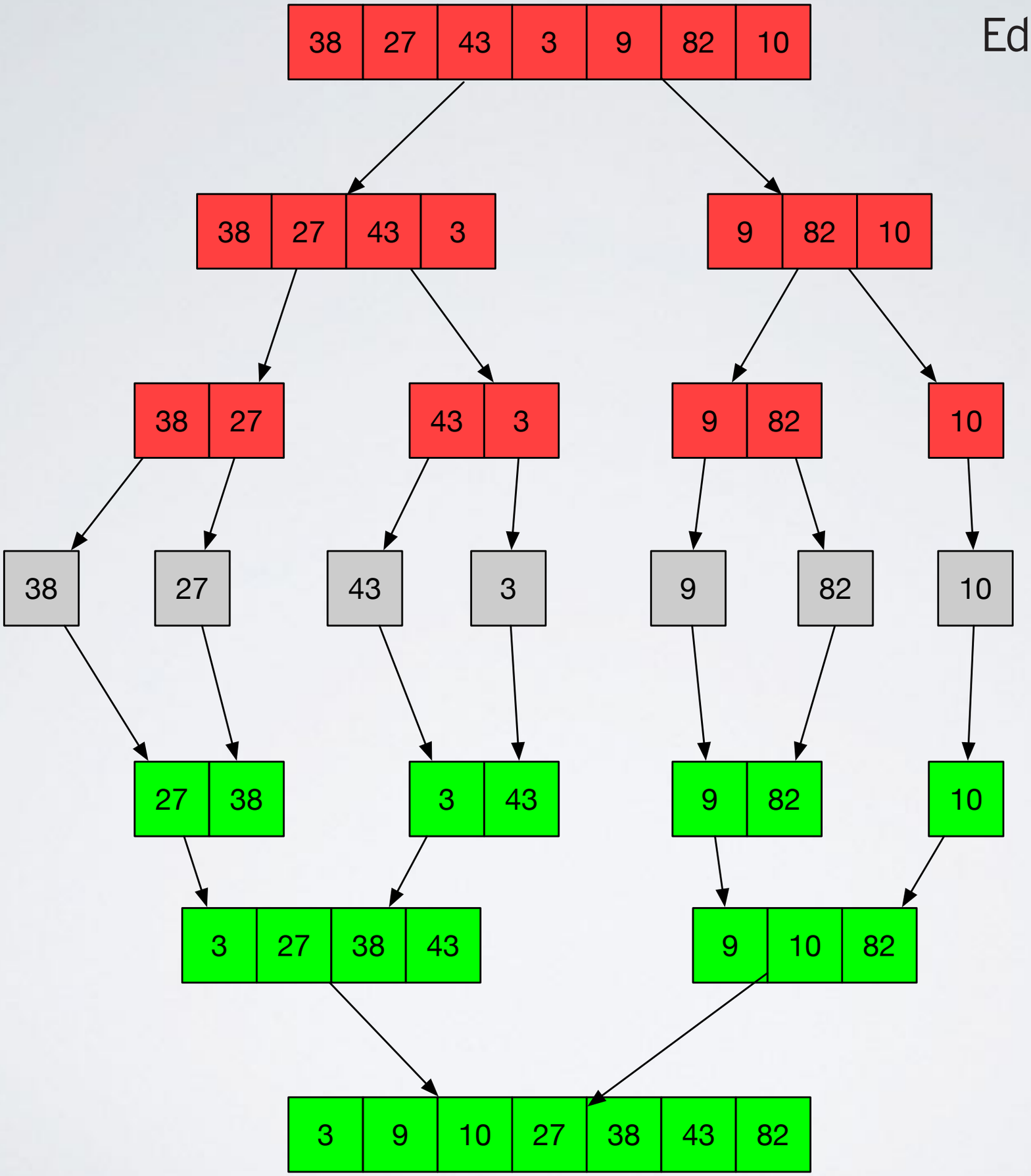
Edinburgh Napier
UNIVERSITY

ALGORITHM

- Repeatedly divide our unsorted data until each portion only contains a single element - *this is trivially sorted.*

- Repeatedly merge the elements, producing larger collections, comparing and placing in sorted order until all elements are back in the collection.

- The collection is now sorted.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

```
void _mergesort(int data[], int left, int right)
{
    if (left < right)
    {
        int middle = left+(right-left)/2;

        _mergesort(data, left, middle);
        _mergesort(data, middle+1, right);

        merge(data, left, middle, right);
    }
}
```

```
void merge(int data[], int left, int middle, int
right)
{
    int left_idx, right_idx, merged_idx;
    int left_size = middle - left + 1;
    int right_size =  right - middle;

    int *tmp_left = (int*) malloc(sizeof(int) *
left_size);
    int *tmp_right = (int*) malloc(sizeof(int) *
right_size);

    for (left_idx = 0; left_idx < left_size;
left_idx++)
        tmp_left[left_idx] = data[left +
left_idx];
    for (right_idx = 0; right_idx < right_size;
right_idx++)
        tmp_right[right_idx] = data[middle + 1+
right_idx];

    left_idx = 0;
    right_idx= 0;
    merged_idx = left;

    while (left_idx < left_size && right_idx <
right_size)
    {
        if (tmp_left[left_idx] <=
tmp_right[right_idx])
        {
            data[merged_idx] = tmp_left[left_idx];
            left_idx++;
        }
        else
        {
            data[merged_idx] =
tmp_right[right_idx];
            right_idx++;
        }
        merged_idx++;
    }

    while (left_idx < left_size)
    {
        data[merged_idx] = tmp_left[left_idx];
        left_idx++;
        merged_idx++;
    }

    while (right_idx < right_size)
    {
        data[merged_idx] = tmp_right[right_idx];
        right_idx++;
        merged_idx++;
    }
    free(tmp_left);
    free(tmp_right);
}
```

Edinburgh Napier
UNIVERSITY

Edinburgh Napier
UNIVERSITY

MERGE SORT VIDEO

Edinburgh Napier
UNIVERSITY

QUICK SORT

Edinburgh Napier UNIVERSITY

QUICKSORT

- A *comparison* sort: can sort items of any type where there is a 'less than' relation defined
- Can operate *in place* on an array so sorts can be very space efficient
- Divide & conquer algorithm that relies on a *partition* operation that is oriented around a *pivot* value
- Elements smaller than pivot are moved before it and elements greater are moved after it. Lesser & greater sublists then sorted recursively

Edinburgh Napier
UNIVERSITY

ALGORITHM

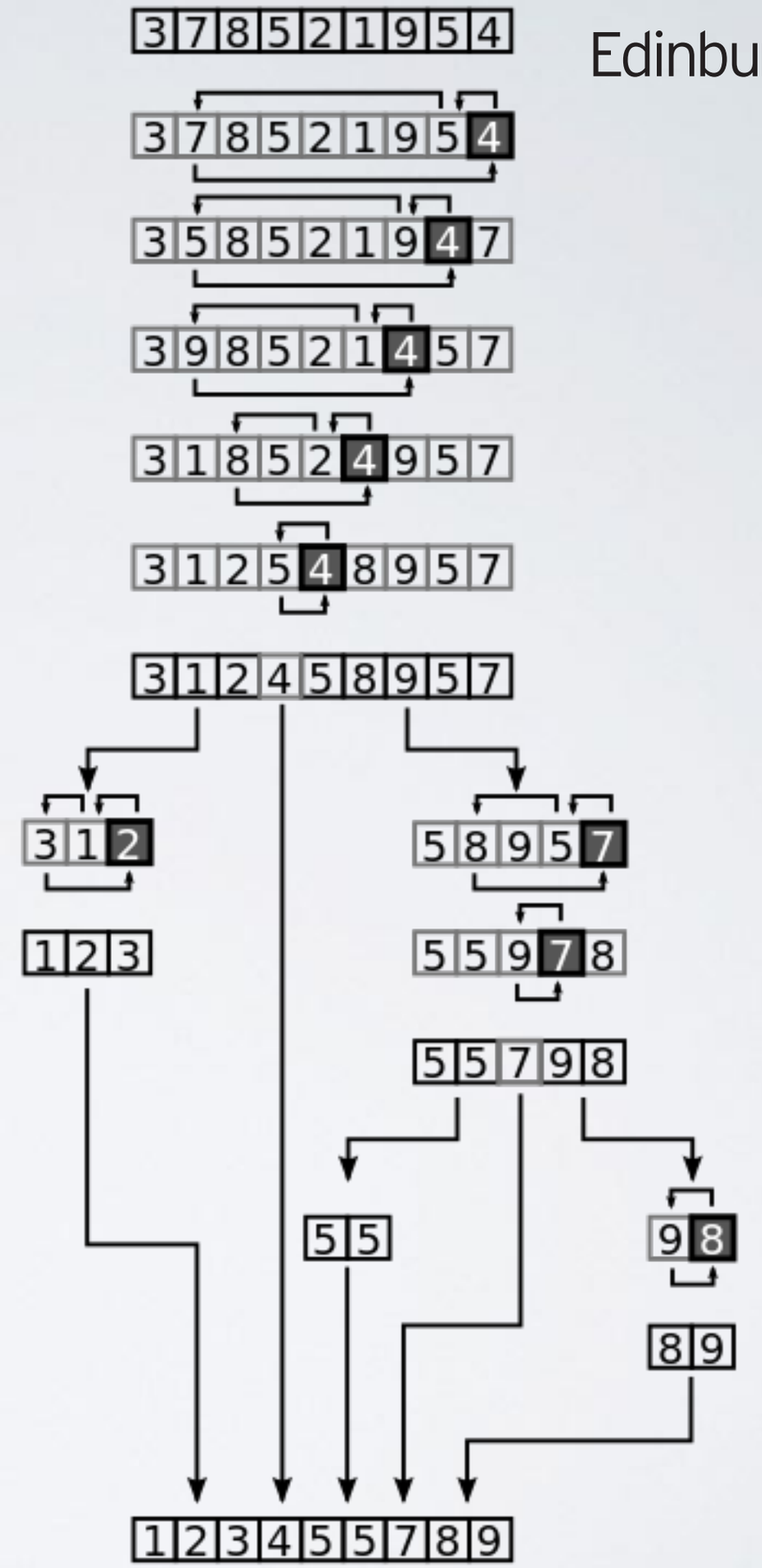
1. Pick an element from the dataset. This element is the pivot.

2. Perform a partitioning operation: Reorder the dataset so that elements greater than the pivot are to one side of it, and items less than the pivot are to the other side of it7. Once complete, the pivot value will be in it's final sorted position.

3. Repeat, applying the pivot selection & partition operation to each separate collection formed respectively from the values smaller than, and greater than the pivot in the previous iteration.

EXAMPLE

- Pick an element (pivot)
- Reorder array so smaller items are before pivot
- Recursively apply to smaller sub-arrays (either side of pivot)



```
void quicksort(int data[], int lower, int upper)
{
    int pivot;
    if(upper > lower)
    {
        pivot = partition(data, lower, upper);
        quicksort(data, lower, pivot-1);
        quicksort(data, pivot+1, upper);
    }
}

int partition(int data[], int lower, int upper)
{
    int pivot = data[upper];
    int idx = lower-1;
    int temp;

    for (int j=lower; j<upper; j++)
    {
        if(data[j] <= pivot)
        {
            idx++;
            temp=data[idx];
            data[idx] = data[j];
            data[j] = temp;
        }
    }

    temp=data[idx+1];
    data[idx+1] = data[upper];
    data[upper] = temp;

    return idx+1;
}
```

Edinburgh Napier
UNIVERSITY

QUICK SORT VIDEO

Edinburgh Napier
UNIVERSITY

15 SORTING ALGORITHMS IN 6 MINUTES

Edinburgh Napier UNIVERSITY

PERFORMANCE OVERVIEW

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

LEGEND

- Excellent
- Good
- Fair
- Bad
- Horrible

Edinburgh Napier UNIVERSITY

SUMMARY

- It can be more efficient to find data within a pre-sorted data-structure (but sorting that data in the first place also has a cost)
- Smaller problems are easier to solve than bigger problems (but knowing how to break a problem up makes all the difference

Edinburgh Napier
UNIVERSITY

QUESTIONS ???

WHAT DID WE LEARN?

- *We can now…*

- Sort data (to make it easier/quicker to find data)
- Recognise and apply a variety of sorting methods
- Have some insight into some of the strategies used in sorting