Ian Piper
CSCI203

Algorithms and Data Structures

Week 1 – Lecture A

Introduction

- Why study this subject?
- What should I already know?
- How do I study this subject?
- What will I learn?
- How is it assessed?

Why study this subject?

- Because you have to! – It's a core subject. ☺
  - CSCI203 is, at least.
- *But why is it core?*

1. As data sets get large, efficient algorithms become critical.
2. An algorithm is only as good as its implementation.
3. Algorithms provide an insight into how computer scientists think.
4. Knowledge of a wide range of algorithms provides a powerful problem-solving tool kit.
5. An understanding of how algorithms are designed may help you to create your own.

What should I already know?

- How to think.
  - Hopefully, you have this from CSIT113 (or CSCI103)
- How to code.
  - In JAVA , C++, C or Python
  - This subject is not intended to be a coding subject – if you cannot already write and debug programs effectively you are likely to struggle.

How do I study this subject?

- Preview the recorded lectures
- Attend Lecture Q&A.
- Attend Laboratories.
- Read the lecture notes after the lecture.
- Work consistently.
- Start early.
  - Assignments are not designed to be completed in one day!
- Work outside of class.
  - 6CP = 12 hours per week.

What will I learn?

- A range of algorithms that look at common problems in computing.
- A range of data structures that provide useful tools in coding these algorithms.
- How to implement the algorithms:
  - Correctly, Efficiently, Flexibly.
- How to implement the data structures and the functions needed to use them:
  - Correctly, Efficiently, Flexibly.

How is it assessed?

- Laboratory tasks.
  - In your assigned lab each week.
  - This is worth 20% of the total mark.
  - Lab tasks will be assessed in the lab and via moodle.
  - Small and self-contained.
- Assignments.
  - There are three programming assignments.
  - Each is worth 10% of the total mark.
  - Larger and more complex.
- Final Examination.
  - This is worth 50% of the total mark.

Peak Finding

Getting Started:

Peak Finding

- If we have an array of numbers we can define a peak as any number that satisfies the following conditions:

1. If it is the first (or last) number it is a peak if it is greater than, or equal to, its neighbour.
2. For any other position it is a peak if it is greater than or equal to both of its neighbours.
3. (A neighbour is a number immediately next to the current number)

An Example

- Consider the following array:

8	6	4	9	7	7	7	3	10	5

- The following are peaks:

-

$\boxed{8}$	6	4	9	7	7	7	3	10	5	8 > 6

-

8	6	4	$\boxed{9}$	7	7	7	3	10	5	9>4, 9>7

-

8	6	4	9	7	$\boxed{7}$	7	3	10	5	7=7, 7=7

-

8	6	4	9	7	7	$\boxed{7}$	3	10	5	7=7, 7>3

-

8	6	4	9	7	7	7	3	$\boxed{10}$	5	10>3, 10>5

Finding Peaks

- How can we find a peak in such an array?
- We note:
  - There will always be at least one peak.
  - The largest value in the array must be a peak.
- This suggests:
  - Algorithm 1: Find the maximum.

Algorithm 1: Finding the Maximum.

- Assume we have our numbers stored in the array **`values`** and that it contains **`n`** integers **`values[1]`** to **`values[n]`**.
- The following pseudocode will find the maximum:

```
maximum = values[1]
position = 1
for index = 2 to n
    if (values[index] > maximum) then
          maximum = values[index]
          position = index
    fi
rof
print "the maximum value is " ,maximum, " at position ",
    position
```

Is This a Good Algorithm?

- Correct?
  - Yes, it finds the maximum value in the array which is certainly a peak.
  - This is a *good thing* ™.
- Efficient?
  - It looks at every element of the array.
  - This may not be a good thing.
  - What if **n** is very large?

Can we do better?

- Although the maximum is always a peak, it is not the only peak.
- A peak is a *local maximum*.
- We can stop as soon as we find *any* peak.
- This suggests:
  - Algorithm 2: Linear search.

Algorithm 2: Linear Search

- The following algorithm uses linear search to find a peak.

```
for index = 1 to n-1
    if values[index] >= values[index+1] then
        print “there is a peak with value ”,
              values[index],” at position ”,index
        stop
    fi
rof
print “there is a peak with value ”,values[n],” at
    position ”,n
```

Is This a Good Algorithm?

- Correct?
  - Yes, it finds the first value which satisfies the following properties:
    - It is greater than or equal to its left neighbor (it there is one)
    - It is greater than or equal to is right neighbor (if there is one)
  - This is the definition of a peak.
- Efficient?
  - This depends on where the peak is:
    - In position 1 – we only do one comparison: Great! (Best case)
    - In position **n** – we do n comparisons: Not Great! (Worst case)
    - In position **`i`** – we do **`i`** comparisons. (General case)
  - On average we will look at roughly half the elements of the array: **`n/(k+1)`** comparisons where there are **`k`** peaks.
  - Again, not good if **n** is large.

Can we do better?

- Consider an arbitrary element of the array, **`values[i]`**.
- There are three possible cases:

1. It is smaller than its left neighbour (if any).
2. It is smaller than its right neighbour (if any).
3. Neither of these is true.

- What does this tell us?

Case 1:

- It is smaller than its left neighbour (if any).
  - This number is certainly not a peak.
  - This is not all it tells us, however.
- There must be a peak to the left of this number!
- WHY?

Case 2:

- It is smaller than its right neighbour (if any).
  - This number is certainly not a peak.
  - Again, this is not all it tells us.
- There must be a peak to the right of this number!
- WHY?
  - (Hint: same reason as before)

Case 3:

- Neither of case 1 or case 2 is true.
  - This number is a peak!
- We are done.
- We can use these observations to create:
  - Algorithm 3: Binary Search.

Algorithm 3: Binary Search

- The following algorithm finds a peak using binary search.

```
start = 1
end = n
repeat
     mid = (start + end)/2
     if values[mid] < values [mid - 1] then
            end = mid – 1
     else if values[mid] < values [mid + 1] then
            start = mid+1
     else
            print “peak found at position ”, mid, “with value”,
                          values[mid]
            stop
     fi
until forever
```

Is This a Good Algorithm?

- Correct?
  - Actually, no!
  - What if we get down to a single value?
  - What if start > end at some point?
    - Can this happen?
  - We need extra code to handle this case.
- Efficient?
  - Each iteration eliminates half of the data (or finds a peak).
  - Best case: we find a peak straight away.
  - Worst case: we work down to a single value (which must be a peak).
    - This will take $\log_2 n$ iterations (which is much less than **n** or **n/k**, especially if n is large).

Algorithm 3a: Binary Search with test for termination.

```
start = 1
end = n
repeat
    mid = (start + end)/2
    if values[mid] < values [mid - 1] then
          end = mid – 1
    else if values[mid] < values [mid + 1] then
          start = mid+1
    else
          print “peak found at position ”, mid, “with value”,
                      values[mid]
          stop
    fi
until start >= end
print “peak found at position ”, mid, “with value”, values[mid]
```

Making the Problem Harder

- What if the data is not a single row of numbers but is, instead, a two-dimensional table.
- We now define a peak as any number which is greater than, or equal to, all of its up to 4 neighbours:
  - Left,
  - Right,
  - Above,
  - Below.
- Again, a number of different approaches are possible.

Another Example

- All of the marked elements are peaks:

17	$\boxed{19}$	8	$\boxed{19}$	$\boxed{19}$	18	2	1	8	$\boxed{20}$
3	5	2	11	2	11	13	$\boxed{14}$	4	3
10	10	3	5	6	6	$\boxed{14}$	13	12	7
18	$\boxed{19}$	$\boxed{19}$	10	13	6	6	6	6	$\boxed{16}$
10	$\boxed{19}$	13	15	$\boxed{16}$	1	12	13	$\boxed{14}$	5
6	5	$\boxed{15}$	5	12	10	10	12	9	2
8	11	6	10	11	18	$\boxed{19}$	10	4	11
10	$\boxed{14}$	5	4	$\boxed{18}$	4	15	$\boxed{15}$	3	14
$\boxed{17}$	14	8	$\boxed{17}$	11	$\boxed{18}$	12	3	12	$\boxed{17}$
8	11	$\boxed{17}$	11	1	17	6	2	$\boxed{18}$	12

Possible Approaches

- Once again, the global maximum is always a peak.
- If the table has **m** rows and **n** columns we have to look at **m** by **n** entries and compare each with up to 4 neighbours for a total of roughly 4×**m**×**n** comparisons.
- This is a *bad thing*™.
- Perhaps we can use a variation of the best algorithm we found for the one-dimensional case.

Steepest Ascent.

- This algorithm takes an arbitrary element in the array and compares it to its immediate neighbours.
  - If it is smaller than any neighbor select the largest neighbor and repeat the process.
  - Otherwise, we have found a peak.
- Is this better than looking for the maximum?
- Let us try with our example.

Steepest Ascent

- Start at the top left corner

17	19	8	19	19	18	2	1	8	20
3	5	2	11	2	11	13	14	4	3
10	10	3	5	6	6	14	13	12	7
18	19	19	10	13	6	6	6	6	16
10	19	13	15	16	1	12	12	14	5
6	5	15	5	12	10	10	20	9	2
18	11	6	18	11	18	19	10	4	11
10	14	5	4	18	4	15	15	3	14
17	14	8	17	11	18	12	3	12	17
8	11	17	11	1	17	6	2	18	12

- And we have a peak after 1 step!

Lucky?

- Will this always happen so fast?

1	2	3	4	5	6	7	8	9	10
2	3	4	5	6	7	8	9	10	11
3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	9	10	11	12	13
5	6	7	8	9	10	11	12	13	14
6	7	8	9	10	11	12	13	14	15
7	8	9	10	11	12	13	14	15	16
8	9	10	11	12	13	14	15	16	17
9	10	11	12	13	14	15	16	17	18
10	11	12	13	14	15	16	17	18	19

- No!

Lucky?

- Will this always happen so fast?

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	11
35	36	37	38	39	40	41	42	11	12
34	5	6	7	8	9	10	43	12	13
33	6	55	56	57	58	11	44	13	14
32	7	54	9	10	59	12	45	14	15
31	8	53	10	11	12	13	46	15	16
30	9	52	51	50	49	48	47	16	17
29	10	11	12	13	14	15	16	17	18
28	27	26	25	24	23	22	21	20	19

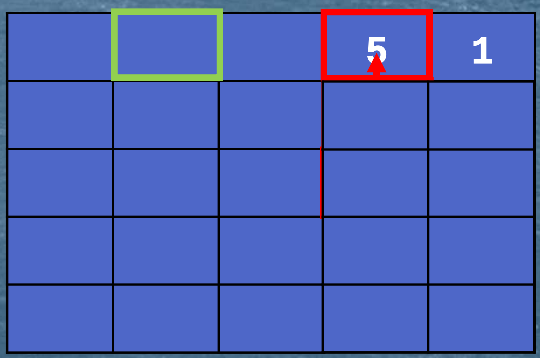
- Not even close!

Can we do better?

- Steepest ascent, while better than finding the global maximum, is not the best we can do.
- Is it possible to use an algorithm that is closer to the best one dimensional one?
- The problem is more complex than it looks.
- If we select an arbitrary row and column and eliminate either the left or right (or top or bottom) half will we always find a peak?
- Consider the following example.

A problem with our approach

- Start in the middle
- 3 > 2 so eliminate the left half
- 4 > 3 so eliminate the bottom half
- 5 > 4 so eliminate the bottom half
- No larger neighbor so we stop
- See the problem?
- 5 is not a peak.
- The algorithm, in fact, removed the only peak in the first step.



An Improved Approach

- Although eliminating half the array on the basis of an arbitrary element does not work, there is one element in a column which does guarantee that we can use this technique.
- The column maximum.
- The largest element in any column, if it is not a peak itself, will guarantee that a peak exists in the half of the array that contains a larger neighbor.
- This leads to the following algorithm:

2-D Peak Finding

```
repeat
     find the largest element in the middle column of the remaining
array
     if the element to its left is larger then
            throw away the right half of the array
     else if the element to its right is larger then
            throw away the left half of the array
     else
            the maximum is a peak
     fi
until we have found a peak
```

- We need to stop when we get to a single remaining column.
  - Its maximum must be a peak

Is This a Good Algorithm?

- Correct?
  - Yes.
- Efficient?
  - Each complete iteration eliminates half of the data (or finds a peak)
  - Best case: we find a peak straight away. ($\mathbf{m}$ operations to find the maximum).
  - Worst case: we work down to a single column (which must contain a peak)
    - This will take $\log_2 \mathbf{n}$ iterations, each with $\mathbf{m}$ operations to find the maximum.
    - This is a total of $\mathbf{m} \times \log_2 \mathbf{n}$ operations.
- Can we do better?

Even Better?

- There are many approaches to the 2-D peak finding problem.
- You can find more by exploring:
  - The library
  - The Internet
- This proliferation of algorithms for a single task leads to the following question:
- “How do we compare two algorithms that do the same job?”

Comparing Algorithms

Comparing Algorithms

- If we have a number of algorithms that all do the same job how can we compare them?
- How can we find the best algorithm?
- What do we mean by “best”?
  - Fastest?
  - Smallest?
  - Most general?
  - Easiest to understand?
- All of these possible answers are valid.
- We will look at the first of these.
- How do we compare the speed of different algorithms.

Comparing Speed

- If we assume that each operation performed by a computer takes time…
- …we can conclude that, the more operations it performs, the longer it takes.
- We can use this to give us a means of comparison.
- The more operations performed, the worse the algorithm…
- …for the same size of problem.

Problem Size

- The easiest, informal, way to think about problem size is as follows:
- “How many things does the problem contain?”
- As an example, consider a problem you have seen before: sorting.
- Here, the size of the problem is simply the number of items to be sorted.
- Sometimes the size of a problem is not so obvious…

Size and Complexity

- When we compare algorithms we are usually interested in how the number of operations performed grows as the problem gets bigger.
- We usually use a variable, like *n* to refer to the problem size.
- So, if we need to sort a list of 10 numbers, *n* is 10.
- If we need to sort a list of 100 numbers, *n* is 100, and so on.
- We want to know not just that the amount of work performed by an algorithm grows as the problem size grows…

Size and Complexity

- We want to know ***how*** it grows.
- We refer to the *rate of growth* in the number of operations performed by an algorithm as *n* grows as the ***complexity*** of the algorithm.
- We can group algorithms into complexity classes, algorithms with approximately the same complexity.
- We provide an abstract measure of this complexity by expressing it in terms of the problem size, *n*.

Complexity Classes

Complexity Classes

- It is probably easiest to understand the idea of complexity classes by looking at a simple problem: finding a telephone number.
- In this case, *n*, the problem size, is the number of entries, in the directory.
- Is this the only possible size we could use?
- What about the number of pages?
- We will look at a series of problems and see what the complexity of the problem (and it’s solution) is.

Problem 1:

- You are handed the ’phone book and asked the following question.
- “What is the first name in the book?”
- Does the size of the book matter?
- NO!
- This problem is independent of *n*.
- The complexity class of this problem is *constant*.

Problem 2:

- You are handed the 'phone book and asked the following question.
- "What is the 'phone number of Ian Piper?"
- Now the size of *n* does matter.
- But, at least, the names are in alphabetical order.
- A simple and effective algorithm might be:

The Algorithm

```
start with the whole book
repeat
    select the page half way through the remaining pages
    if the name Piper appears on this page then
          select this page.
    else if the name Piper lies before this page
          eliminate the pages after this one
    else
          eliminate the pages before this one
    fi
until we find the right page
look up Ian Piper’s phone number on this page.
```

Analysing the Algorithm

- Our algorithm consists of two tasks:
  - find the right page;
  - find the right entry on the page.
- We will make the following assumptions:
  - Each page contains roughly the same number of names
  - There are far more pages than there are names per page
- Then, as the number of pages gets bigger the two parts of the algorithm are affected differently:
  - *find the right page* gets harder as n grows;
  - *find the right entry on the page* does not change as n grows.
- This means that finding the right page is the critical step in our algorithm.

Finding the Page

- The approach we use to find the right page eliminates roughly half of the remaining pages at each iteration.
- This means that, as *n* grows in size the number of iterations required to find the correct page grows more slowly.
- In fact, it grows as the logarithm of the problem size.
- So, for this algorithm, it's complexity is related to log *n* + *c*, where…
- log *n*, the time to find the correct page, dominates *c*, the constant time to find my name on the page.
- The complexity class of this algorithm is *logarithmic*.

Problem 3:

- You are handed the 'phone book and asked the following question:
- "Who has the 'phone number 42743555?"
- Unfortunately, while the names are in order, the numbers are not!
- The best algorithm that we can use is the following:

The Algorithm:

```
for each entry in the book
     if this entry has the number 42743555 then
          note the name for this entry
          stop
     fi
rof
```

Analysing the Algorithm

- Once again, the number of iterations grows as the number of entries grows.
- This time, however, it grows at the same rate.
- The complexity class of this algorithm is *linear*.

Problem 4:

- You are handed the ’phone book and asked the following question:
- “Will you rearrange the entries in increasing order of ’phone number?”
- The existing order, alphabetical by name, gives us no help.
- We need to use a sort algorithm.
- The best one you should be familiar with, so far, is quicksort.

The Quicksort Algorithm:

```
partition each part of the list as follows:
    entries less than the first entry
    the partition value
    entries greater than the first entry
repeat
    partition each partition
until the list is sorted
```

Analysing Quicksort

- Each partitioning takes linear time in the number of entries.
- It has time proportional to *n*.
- The number of partitionings required is logarithmic in the number of entries.
- The algorithm, as a whole, takes *n* log *n* steps.
- The complexity class of this algorithm is *linearithmic*.
- Usually, we just say the complexity is *n* log *n*.

Problem 5

- You are asked the following question:
- “Every number in the book has an extra zero at the end. We have already printed the books. Can you cover each of the extra zeros with white-out?”
- Only one algorithm seems possible (apart from finding a better job):

The Algorithm:

```
for each book
    for each entry in the current book
        cover the extra zero
    rof
rof
```

Analysing the Algorithm

- There are *n* entries per book.
- Every subscriber gets a copy so…
- there are *n* books.
- The total number of iterations is $n \times n$ or n^2.
- The complexity of this algorithm is *quadratic.*

It gets worse

- This is, by no means, the worst possible complexity.
- Other complexity classes are:
  - *Exponential*, the number of iterations grows as 2^n
  - *Factorial*, the number of iterations grows as $n!$
    - $n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n$

Comparing Classes

- The following table shows the various classes for increasing values of *n*.

n		**2**	**4**	**8**	**16**	**32**	**64**
constant	1	1	1	1	1	1	1
logarithmic	$\log n$	1	2	3	4	5	6
linear	n	2	4	8	16	32	64
linearithmic	$n \log n$	2	8	24	64	160	384
quadratic	n^2	4	16	64	256	1024	4096
exponential	2^n	4	16	256	65536	4.3×10^9	1.8×10^{19}
factorial	$n!$	2	24	40,320	2.1×10^{13}	2.6×10^{35}	1.3×10^{89}

Comparing Classes

- This may be easier to see in a graph: