SCC.121 ALGORITHMS AND COMPLEXITY
Introduction to Operation Counting

Emma Wilson e.d.wilson1@lancaster.ac.uk

Recap: Last Lecture

Introduced the concept of algorithms and the space and time complexity of algorithms

- An algorithm is an effective method for solving a class of problems
- Algorithms are used everywhere e.g. mobile phones, ATM, etc.
- Space and **Time** complexity
- The running time depends on the size of the input, the organization of the input, optimal operating temperature, number of processor cores

Today's Lecture

Aim: To introduce the concept of operation counting and best vs worst case

Learning objective:

- To be able to calculate the number of operations (in the best and worst case) given an algorithm by performing operation counting

Today’s Lecture

- Running time
- Cost of operations
- Operation counting and T(N)

Today’s Lecture

- **Running time**
- Cost of operations
- Operation counting and T(N)

Running Time

- The running time **depends on the size of the input**
- The running time **depends on the organization of the input**
- Generally, we seek **upper bounds (worst case) on the running time,**
- **Optimal operating temperature?**
- **Number of processor cores?**

Input Size

Lancaster University

- Runtimes in seconds of two fictional algorithms for processing employees' records

# of records	10	20	50	100	1000	5000
Algorithm 1	0.00s	0.01s	0.05s	0.47s	23.92s	47min
Algorithm 2	0.05s	0.05s	0.06s	0.11s	0.78s	14.22s

- Which algorithm is better?

slido

Please download and install the Slido app on all computers you use

Which Algorithm is better?

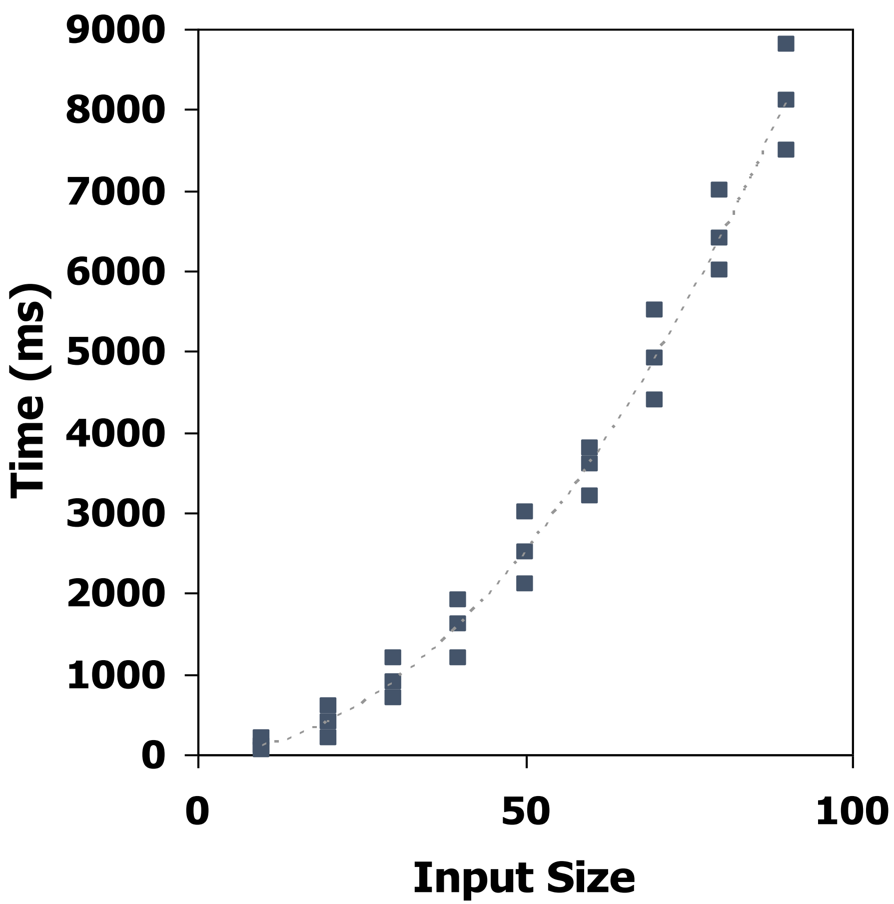
ⓘ Start presenting to display the poll results on this slide.

Input Size

- Which algorithm is better?
  - Algorithm 2
    - Scalable
    - Time efficient algorithm

# of records	10	20	50	100	1000	5000
Algorithm 1	0.00s	0.01s	0.05s	0.47s	23.92s	47min
Algorithm 2	0.05s	0.05s	0.06s	0.11s	0.78s	14.22s

Experimental Approach



- Write a program to implement the algorithm.
- Run this program with inputs of varying size and composition. (e.g. 10, 20, 50, 100, 1000, 5000 employees' records)
- Get an accurate measure of the actual running time (e.g. system call date).
- Plot the results.
- Problems?

Limitations of Experimental Studies

- The algorithm must be implemented, which may take a long time and could be very difficult.
- Results may not be indicative for the running time on other inputs that are not included in the experiments.
- In order to compare two algorithms, the same hardware and software must be used.

- **Solution:**
  - Actual runtime can be estimated
  - We will analyse algorithms, and come up with classes of runtimes based on input size

Today's Lecture

- Running time
- **Cost of operations**
- Operation counting and T(N)

Cost of Operations

- To determine how long an algorithm takes to run we can count-up all the operations it executes
- Let's first consider **what counts as one operation?**
- Want to be able to analyse the algorithm itself, independent of the choice of language used to execute

Example: "array lookup"

```
m = A[0];
```

Assigning a value to a variable

Looking up the value of a particular element in an array

Cost of Operations

Example: “array lookup”

– The "array lookup" statement in one language may compile to different operations in different programming languages

Pascal requires 5 operations for each array access instead of the 2 operation C requires

- Pascal code:

```
m := A[i]
```

- the equivalent of the above Pascal code in C

```
if (i >= 0 && i < n ) {
      m = A[i];
}
```

Cost of Operations

What should we do?

- **Ignore these minor details:** Ignore differences between particular programming languages and compilers and only analyse the idea of the algorithm itself
- Assume each elementary operation takes a fixed time to execute and count as one operation

Operations and Instructions

- Operations in a high-level language may require many machine-level instructions.
- However, we ignore minor details

  - $x = a + b;$ **1 operation**
  - $x = theArray[i];$ **1 operation**
  - $x = x + theArray[i];$ **1 operation**
  - $n < theArray[i]$ **1 operation**
  - $i{+}{+}$ **1 operation**

Today's Lecture

- Running time
- Cost of operations
- **Operation counting and T(N)**

Operation Counting

- Let's start with a simple algorithm – code below
- The first thing we will do is count **how many operations** (ignoring minor details) this code executes.
- Given an array of integers arr of size n:

```
int maxel = arr[0];
for (int i = 0; i < n; i++) {
    if (arr[i] >= maxel ) {
        maxel = arr[i];
    }
}
```

Operation Counting

- The **first line** of this code requires **1 operation**:
  - As we are ignoring minor differences in implementation
- This operations is always required by the algorithm, regardless of the value of ‘n’

```
int maxel = arr[0]; 1 operation

for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

Operation Counting

- The *for* loop initialization code also has to always run.
- This gives us **two more operations**:
  - an assignment (‘i = 0’) and a comparison (‘i < n‘)
  - These will run before the first *for* loop iteration

2 operations
initialization

```
int maxel = arr[0]; 1 operation

for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

Operation Counting

- After each *for* loop iteration, we need **two more operations** to run,
  - an increment of 'i' (i.e. 'i++') and
  - a comparison ('i < n') to check if we'll stay in the loop
  - **Question: How many times do we go around the loop?**

2 operations
initialization

```
int maxel = arr[0];
for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

1 operation

2 operations
Every time we go around the loop

Operation Counting

Lancaster University

- After each *for* loop iteration, we need **two more operations** to run,
  - an increment of ’i’ (i.e. ‘i++’) and
  - a comparison (‘i < n’) to check if we'll stay in the loop
  - **Question: How many times do we go around the loop?**

2 operations
initialization

```
int maxel = arr[0]; 
for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

1 operation

‘n’ times

2 operations
Every time we go around the loop

Operation Counting

- So, if we ***ignore the loop body***, the number of operations this algorithm needs is **3 + 2n**.
  - **3 operations** at the beginning of the *for* loop
  - **2 operations** at the end of each iteration of which we have 'n'

2 operations
initialization

```
int maxel = arr[0];
for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

1 operation

'n' times

2 operations
Every time we go around the loop

Operation Counting

- Now, looking at the for body,
- We have an array lookup operation and a comparison that **always happen**
- But the if body **may run or may not run**, depending on what the array values are

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

1 operation

So, n operations in total. Why?

Operation Counting

- If it happens to be so that ’`arr[i] >= maxel`’, then we'll run an additional operation: `maxel = arr[i]`

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

1 operation (n in total)

Operation Counting

- So, we cannot define **T(n)** easily, because our number of operations doesn't depend solely on the size of input n but also **on the organization of the input**
  - For example, for arr = [1, 2, 3, 4] the algorithm will need more operations than for arr = [4, 3, 2, 1]. **Why?**

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
     if (arr[i] >= maxel ) {
          maxel = arr[i];
     }
}
```

Operation Counting

- Different scenarios:
  - **Worst-case scenario** (the algorithm needs the most operations to complete): an example is arr = [1, 2, 3, 4]
  - **Best-case scenario** (the algorithm needs the least operations to complete): an example is arr = [4, 3, 2, 1]
  - **Average-case scenario** Average number of operations

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
    if (arr[i] >= maxel ) {
        maxel = arr[i];
    }
}
```

Operation Counting: Worst Case

- **Worst-case scenario**, example arr = [1, 2, 3, 4]
  - In that case, 'maxel' needs to be replaced every single time and so that yields the most operations
  - So, in the worst case, we have 2 operations to run within the *for* body. **Why?**
  - So we have **T(n) = 3 + 2n + 2n = 4n + 3**

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
     if (arr[i] >= maxel ) {
          maxel = arr[i];
     }
}
```

Operation Counting: Best Case

- **Best-case scenario**, example arr = [4, 3, 2, 1]
  - What is T(n) in the Best-case for the example?

```
int maxel = arr[0];

for (int i = 0; i < n; i++) {
      if (arr[i] >= maxel ) {
            maxel = arr[i];
      }
}
```

Operation Counting Question

Given an input array a of length k. How many operations does the following code execute (ignoring the minor details)?

```
int sum = 0

for (int i = 0; i<k; i++){

	sum +=a[i];

}
```

slido

Please download and install the Slido app on all computers you use

How many operations?

ⓘ Start presenting to display the poll results on this slide.

Operation Counting Answer

Given an input array a of length k. How many operations does the following code execute (ignoring the minor details)?

```
int sum = 0   1 op
                1 op     1 op    1 op
for (int i = 0; i<k; i++){
      sum +=a[i];  1 op
}
```

How many operations before we go around the loop?

- 3 operations (sum = 0, i = 0, i < k)

How many operations each time we go around the loop?

- 3 operations (i < k, i++, sum += a[i])

How many times do we go around the loop?

- k times

Next combine to give overall T(k)

- **T(k) = 3k + 3**

slido

Please download and install the Slido app on all computers you use

Audience Q&A

ⓘ Start presenting to display the audience questions on this slide.

Summary

Today's lecture: introduce the concept of operation counting and best vs worst case

- Limitations of experimental studies for evaluating the time complexity of algorithms
- **we ignore the minor details such as** differences between particular programming languages and compilers and only analysing the idea of the algorithm itself
- We cannot define **T(n)** easily, when the number of operations doesn't just depend on the size of input n but also **on the organization of the input**
  - Best case
  - Worst Case
  - Average case
- **Next Lecture:** More on operation counting, examples for different types of complexity