Complexity

Algorithms and Programs

- An *algorithm* is a mechanical procedure written so *human beings* can understand it
- A *program* is a mechanical procedure written so that a *computer* can execute it
- A program needs more information – because computers are dumb!
- Given an algorithm, we are led to ask:
    1. **What is its purpose? (specification)**
    2. **Does it really fulfil its purpose? (verification)**
    3. **How efficiently does it do it? (performance analysis)**

What affects the performance of an algorithm?

1. computer used, the hardware platform
2. representation of abstract data types (ADT's)
3. efficiency of compiler
4. competence of implementer (programming skills)
5. **complexity of underlying algorithm**
6. **size of the input**

- Last two are probably most significant
- Measure of performance is a function of the input

Time complexity vs Space complexity

- For real-world applications need to know how long a computational task will take
- If writing a flight booking system then it is not acceptable for agent and/or customer to wait ½ hour for response to transaction
- This criterion for performance is called ***time complexity***

- If an application stored terabytes of data because a database was loaded into the application → significant memory requirement
- This criterion for performance is called ***space complexity***

- Programmer has to make the trade off between these

Performance analysis - Benchmarking

- Approach
  - Pick some desired inputs
  - Actually run implementation of algorithm
  - Measure time & space needed
- Industry benchmarks
  - SPEC – CPU performance
  - MySQL – Database applications
  - WinStone – Windows PC applications
  - MediaBench – Multimedia applications
  - Linpack – Numerical scientific applications

Performance analysis - Benchmarking

- Advantages
  - Precise information for given configuration based on implementation, hardware, inputs
- Disadvantages
  - Affected by configuration
    - Data sets (usually too small)
    - Hardware
    - Software
  - Affected by special cases (biased inputs)
  - Does not measure *intrinsic* efficiency

Asymptotic Analysis

- Approach
  - Mathematically analyze efficiency
  - Calculate time as function of input size *n*
    - T ≈ O[f(*n*)] read this bit of Maths notation as…

    "T is on the order of f(n)"
    - Make use of Big O" notation (e.g. as covered in Logical Foundations)
- Advantages
  - Does measure intrinsic efficiency
  - Dominates efficiency for large input sizes

An example - searching

- One important problem in computing is that of locating information - *searching* (discussed in earlier lecture)
- The information to be searched has to be represented somehow - *data structures* (discussed in another earlier lecture)
- After a suitable choice of data structure the process is that of searching. This is where algorithms come in!
- TASK: we want to search a collection of integer numbers

  For example 2 5 20 3 88 65 5 7 76

An example - searching

- Arrays are one of the simplest possible ways of representing collections (write its items in order, separated by commas and enclosed by square brackets).

```
list=[2, 5, 20, 3, 88, 65, 5, 7, 76]
```

- This is a 1-D array of size 9
- The positions of the items start from 0,1,...,7,8
- So position 4 contains item 88 and item 47 is not stored anywhere (-1?)

What is its purpose

Specification:

Given an array (called *list*) and integer item (called *x*), find an integer location (called *index*) such that

1. if the item *x* does not exist in the list then the location is set as *index=-1*

2. otherwise if the item *x* does exist then the location is set as *index=i* where the item *x* is stored

Note: The current specification is ambiguous, why? This is not unusual and is not necessarily a problem.

e.g. List of students with highest grades

One solution – linear search

- Lets use a pseudo program to try and describe the algorithm
- We ***assume*** that an array *list* of size *n* and an item *x* are given
- We are supposed to find a location *index* satisfying the specification (previous slide)

```
for i = 0,1,...,n-1
  if list[i] is equal to x then
    we have found a suitable index and hence we stop.
if we reach this point then
  x is not in list and hence we terminate with index set to -1
```

An equivalent Java code would be...

```
for (i=0;i<n;i++)
  if (list[i] == x) {
    index=i;
    return index;
  }
index=-1
return index;
```

Does it really fulfil its purpose?

Verification:

- In this case, easy to see that the algorithm satisfies the specification
- we start counting from zero, the last position of the array is its size minus one (*n-1*).
- Not obvious for complex specifications and/or algorithms
- Need to spend effort verifying that the algorithm is indeed correct -- if it is at all (and very often it is not)
- Testing can be enough for finding out that the algorithm is incorrect
- For large number of inputs maybe more than just testing is needed to be sure an algorithm satisfies its specification. We need a *correctness proof* (relate to Formal Methods course)

How efficiently does it do it?

Performance Analysis

- How good is this search?
  - Before we can answer this we need to understand what we are asking (what do we mean by “good”?)

Best case

- Smallest number of steps required
- Not very useful
- Example ⇒ Find item in first place checked

Worst case

- Largest number of steps required
- Useful for *upper bound* on worst performance
  - Real-time applications (e.g., banking transaction)
  - Quality of service guarantee (e.g. algorithms used in streaming etc)
- Example ⇒ Find item in last place checked

How efficiently does it do it?

Average case

- Number of steps required for *typical* case
- Most useful metric in practice
- Different approaches
  - Average of all cases
  - Expected case

Average of all cases

- Average over all possible inputs
- Assumes some probability distribution, usually uniform

Expected case

- Algorithm uses randomness
- Worse case over all possible input
- average over all possible random values

How efficiently does it do it?

Back to the question...

- How good is this search?
  - In the ***worst case***, searching an array of size *n* takes ***n*** steps (e.g. item missing) – we say "*takes of the order n*"
  - *On* ***average***, searching an array of size *n* takes ***n/2*** steps

- For a big collection of data, e.g. Google search on the world wide web, is this acceptable in practice?

- Can we improve on this algorithm?

- Simple answer is yes, but...
  - we may need to organize the list (data) in such a way that a more efficient algorithm is possible
  - more demand in efficiency → more steps involved → more complicated data structures and/or algorithms

another solution – binary search

- Still using array data structure but, the items are already *sorted* in ascending order. (earlier lecture)
- Instead of working with the list array as

```
list=[2, 5, 20, 3, 88, 65, 5, 7, 76]
```

we now have

```
list=[2, 3, 5, 5, 7, 20, 65, 76, 88]
```

Specification:

Given a sorted array (called *list*) and integer item (called *x*), find an integer location (called *index*) such that

1. if the item *x* does not exist in the list then the location is set as *index=-1*

2. otherwise if the item *x* does exist then the location is set as *index=i* where the item *x* is stored

another solution – binary search

- Use integers (called *l* for left, *m* for middle and *r* for right)
- Lets use a pseudo program to try and describe the algorithm

```
Set l=0, r=n-1
While l<r-1
  set m to the integer part of (l+r)/2
  if x is less than list[m] then
    set r to m
  otherwise if  x is greater than list[m] then
    set l to m
  otherwise if x is equal to list[l]
    set index to l
  otherwise set index to -1
```

another solution – binary search

Performance Analysis

- We divide the search area into two either from *[l,m]* or from *[m,r]*
- The data is sorted so we can effectively discard one half of the search area each time.
- We repeat the division into two halves each time until we find the *index* for the *item*.
- For an array of *n* items this takes $\boldsymbol{\log_2 n}$ searches (earlier lecture)
- For example if n=16, $\log_2 16=4$ so 16 items are reduced to 8 then 4 then 2 then 1 (hooray, found the item). This took 4 steps.
- we say "*takes of the order log n*"

How do we calculate the order of an algorithm?

Guidelines for finding out the time complexity of an algorithm

1. **Single statement**
2. **Loops**
3. **Nested loops**
4. **Consecutive statements**
5. **If-then-else statements**

Guideline 1: Single Statement

The running time of a single statement is a function of the number of operators/operations within the single statement.

```
m = (m + 2)/(4*n);
```
⟵ one addition, one division, one multiplication

Total time = cost of an addition (a) + cost of a division (d) + cost of a multiplication (m) = **O(*constant*)**

e.g. addition=1x10-6 s, division=5x10-6 s, multiplicatipon=2x10-6 s total time= 8x10-6 s

tstart= time start

```
m = (m + 2)/(4*n);
```
⟵ constant time , c (addition and an assignment)

tstop=time stop

Total time = constant c = **O(*constant*)**

e.g. tstart=3.0s, tstop=11x10-6 s total time= 8x10-6 s

Guideline 2: Loops

The running time of a loop is, at most, the running time of the loop process (initialisation, the test and increment) plus the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

executed ***n*** times

```
for (i=1; i<=n; i++)
{
    m = m + 2;
}
```

one initialisation, n tests, n additions)

constant time , c (addition and an assignment)

Total time (without loop process) = a constant c * n = cn = **O(*n*)**

Total time (with loop process)= a constant c * n + (1 + n + n) = cn+2n+1 = **O(*n*)**

Guideline 2: Loops (unknown end)

- All loops will have a start point (they are either entered or not) but ...

```
i = 0;
while ( i<=n )
   {
    m = m + 2;
    i = i+1;
   }
```

Total time = **O(*n*)**

```
i = 20.0;
do
   {
    m = m + 2;
    i = i/2.0;
   }
while ( i >= 1.25)
```

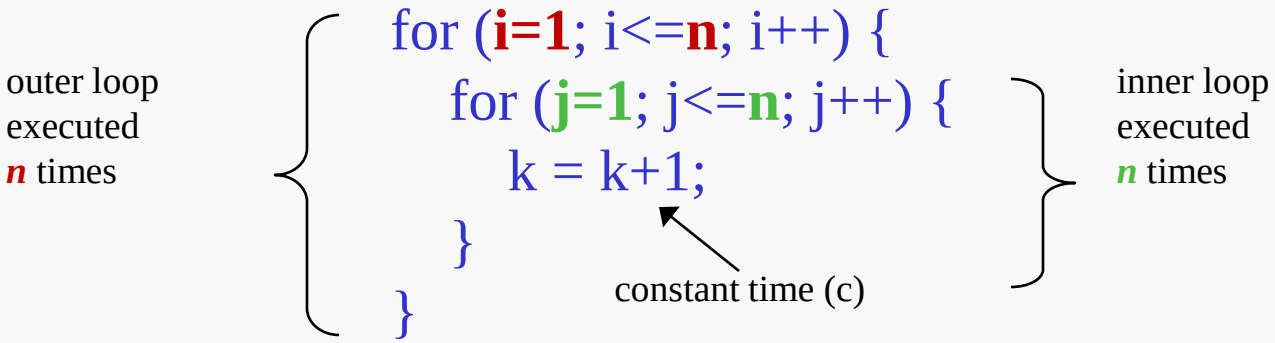
Total time = **O(*5*2*c_1 + c_0)**

```
i = 20.0;
do
   {
    m = m + 2;
    i = i/2.0;
   }
while ( i > 0.0)
```

Total time = **?**

Guideline 3: Nested loops

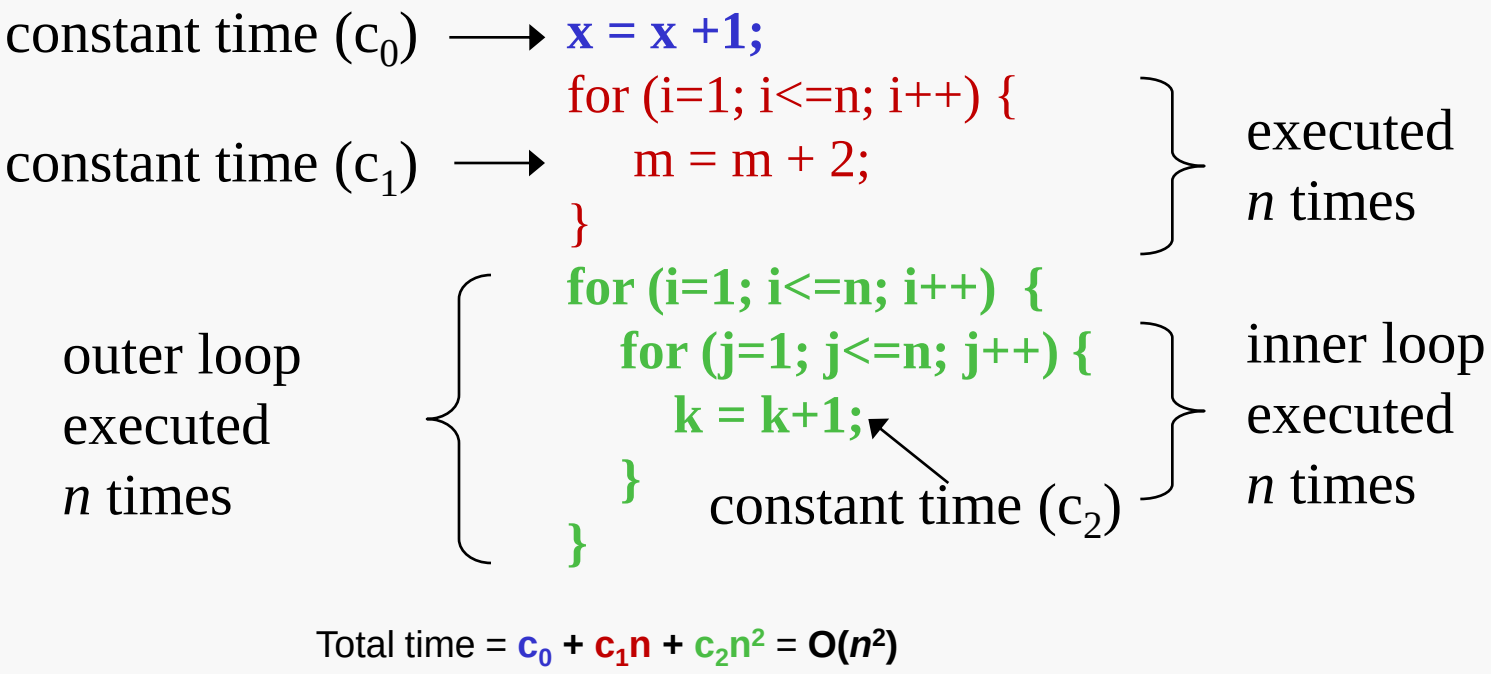
Analyse inside out. Total running time is the product of the sizes of all the loops.



Total time = **c * n * n = cn² = O(*n*²)**

Guideline 4: Consecutive statements

Add the time complexities of each statement.



Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$

Guideline 5: If-then-else statements

Worst-case running time: the test, plus *either* the **then** part *or* the **else** part (whichever is the larger).

test:
constant (c_0)

another if :
constant + constant
(no else part)

```
if (depth( ) != otherStack.depth( ) ) {
   return false;
}
else {
   for (int n = 0; n < depth( ); n++) {
      if (!list[n].equals(otherStack.list[n]))
         return false;
   }
}
```

then part:
constant (c_1)

else part:
(constant c_2 +
constant c_3) * n

Total time = $c_0 + \max\{ c_1 , (c_2 + c_3) * n\} = O(n)$

Big-O and the dominant term

- Programmers have had unexpected (and unwelcomed) surprises when they have moved from small to large data sets
- The big-O notation tells us what happens for very large values of ***n*** – so makes us aware of these potential problems
- The notation focuses on the *dominant* term
- For example, an algorithm that takes $t(n) = 30n^2 + 5n + 2$ can be re-written as $t(n) = n^2 + n/6 + 1/15$ since a different machine and compiler settings will cause a different set of constants

Dominant term

- So we say as n increases then t(n) grows like n^2 or time is of the order n^2
- General guidance – drop lower order terms and constant factors
- Examples,

$t(n) = 7n^5 - 3n^2+6 \rightarrow O(n^5)$

$t(n) = 8n^2\log_2 n + 5n^2+n \rightarrow O(n^2\log_2 n)$

So which is better,

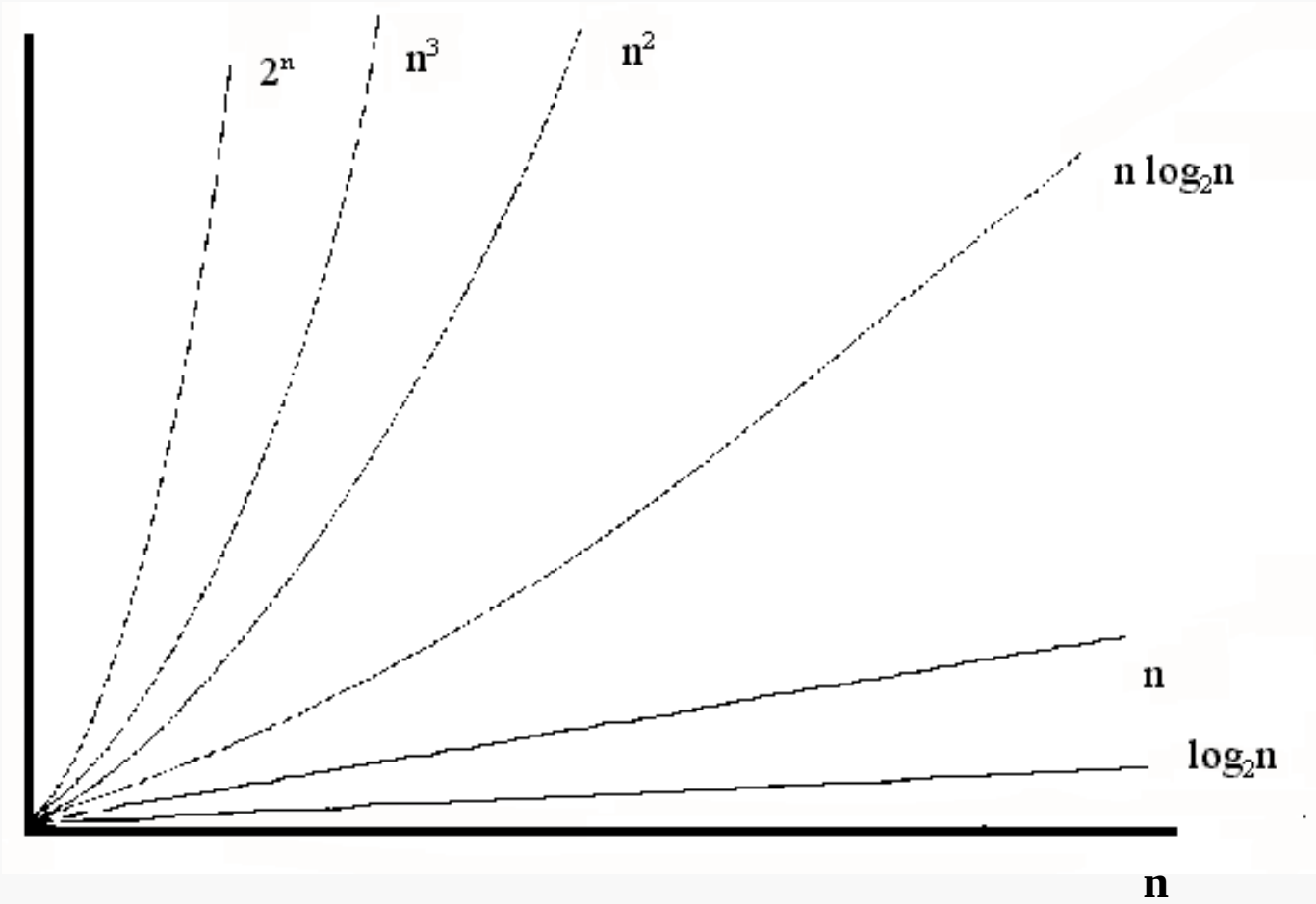
$O(n)$, $O(n^2)$, $O(n^3)$, $O(\log_2 n)$?????

Size does matter !

	logarithmic	linear	n $\log_2$n	quadratic	cubic	3 power n	factorial
n	O($\log_2$n)	O(n)	O(n$\log_2$n)	O(n^2)	O(n^3)	O(3^n)	O(n!)
1	1	1	1	1	1	3	1
2	1	2	2	4	8	9	2
4	2	4	8	16	64	81	24
8	3	8	24	64	512	6561	40320
16	4	16	64	256	4096	43046721	10^{13}
1024	10	1024	10240	1048576	1073741824	10^{488}	10^{2639}
1048576	20	1048576	20971520	10^{12}	BIG	BIGGER	BIGGER STILL

- This gives us a feel for what to expect...

Graphically



- This also gives us a feel for what to expect...

Big-O in action

- Example: *A digital camera produces an image using 4 million pixels. How long does it take to display the image on a screen using the two algorithms below?*
- If these are processed and displayed on a screen using
  - (1) an algorithm $O(n^2)$
  - (2) an algorithm $O(\log_2 n)$
  - at a rate of 1 pixel per microsecond

(1) It takes $4000000^2 \times 10^{-6}$ seconds = 6.5 months!

(2) It takes $\log_2 4000000 \times 10^{-6}$ seconds $\approx$ 22μ seconds!

Is performance everything?

- What are the other considerations for (not) adopting an algorithm
  - How often will the program be used?
  - If only once, or a few times do we care about run time?
  - Will it take longer to code than to run for the few times it is used?
  - What size of data will it be used for? small, medium, large?
  - An efficient algorithm might require careful coding, be difficult to implement, difficult to understand, and difficult to maintain.
  - How much memory will it need?
  - Adaptability to different architectures
  - ...