Introduction to
Complexity Theory

Complexity

- Analysis of complexity of programs
  - Time complexity
  - Space complexity
  - Big-Oh Notation
- Introduction to complexity theory
  - P, NP, and NP-Complete classes of algorithm

Complexity

- Suppose there is an assignment statement

  x := x +1

  in your program.
- We'd like to determine:
  - The time a single execution would take
  - The number of times it is executed
    Frequency Count
- Product is the total time taken
- Frequency count will vary from data set to data set

Complexity

- Since the execution time will be very machine dependent (and compiler dependent), we neglect it and concentrate on the frequency count
- Consider the following three examples:

Complexity

Program 1

```
x := x + 1
```

Program 2

```
FOR i := 1 to n
DO
   x := x + 1
END
```

Program 3

```
FOR i := 1 to n
DO
   FOR j := 1 to n
   DO
      x := x + 1
   END
END
```


Complexity

- Program 1:
  - statement is not contained in a loop (implicitly or explicitly)
  - Frequency count is 1
- Program 2
  - statement is executed n times
- Program 3
  - statement is executed n^2 times

Complexity

- are said to be different and increasing orders of magnitude (e.g. let n = 10)
- We are chiefly interested in determining the order of magnitude of an algorithm

Complexity

	step	n<0
`procedure fibonacci`	1	1
`   read(n)`	2	1
`      if n<0`	3	1
`         then print(error)`	4	1
`         else if n=0`	5	0
`            then print(0)`	6	0
`            else if n=1`	7	0
`               then print(1)`	8	0
`               else`	9	0
`                 fnm2 := 0;`	10	0
`                  fnm1 := 1;`	11	0
`                  FOR i := 2 to n DO`	12	0
`                     fn := fnm1 + fnm2;`	13	0
`                     fnm2 := fnm1;`	14	0
`                     fnm1 := fn`	15	0
`                  end`	16	0
`                  print(fn);`	17	0

Complexity

	step	n=0
`procedure fibonacci`	1	1
`   read(n)`	2	1
`      if n<0`	3	1
`         then print(error)`	4	0
`         else if n=0`	5	1
`            then print(0)`	6	1
`            else if n=1`	7	0
`               then print(1)`	8	0
`               else`	9	0
`                 fnm2 := 0;`	10	0
`                  fnm1 := 1;`	11	0
`                  FOR i := 2 to n DO`	12	0
`                     fn := fnm1 + fnm2;`	13	0
`                     fnm2 := fnm1;`	14	0
`                     fnm1 := fn`	15	0
`                  end`	16	0
`                  print(fn);`	17	0

Complexity

```
1    procedure fibonacci
2       read(n)
3          if n<0
4             then print(error)
5             else if n=0
6                then print(0)
7                else if n=1
8                   then print(1)
9                   else
10                    fnm2 := 0;
11                     fnm1 := 1;
12                     FOR i := 2 to n DO
13                        fn := fnm1 + fnm2;
14                        fnm2 := fnm1;
15                        fnm1 := fn
16                     end
17                     print(fn);
```

step	n=1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Complexity

```
1    procedure fibonacci
2       read(n)
3          if n<0
4             then print(error)
5             else if n=0
6                then print(0)
7                else if n=1
8                   then print(1)
9                   else
10                    fnm2 := 0;
11                     fnm1 := 1;
12                     FOR i := 2 to n DO
13                        fn := fnm1 + fnm2;
14                        fnm2 := fnm1;
15                        fnm1 := fn
16                     end
17                     print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

Complexity

step	n<0	n=0	n=1	n>1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	1
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	n
13	0	0	0	n-1
14	0	0	0	n-1
15	0	0	0	n-1
16	0	0	0	n-1
17	0	0	0	1

Complexity

- The cases where $n<0$, $n=0$, $n=1$ are not particularly instructive or interesting
- In the case where $n>1$, we have the total statement frequency of:

  $9 + n + 4(n-1) = 5n + 5$

Complexity

- The notation $f(n) = O(g(n))$ has a precise mathematical definition

Complexity

- If an algorithm has a time complexity of $O(g(n))$ it means that its execution will take no longer than a constant times $g(n)$
- n is typically the size of the data set

Complexity

- $O(1)$ Constant (computing time)
- $O(n)$ Linear (computing time)
- $O(n^2)$ Quadratic (computing time)

Complexity

- Given an algorithm, we analyse the frequency count of each statement and total the sum.
- This may give a polynomial $P(n)$:

  $\mathrm{P}(n) = c_k n^k + c_{k-1} n^{k-1} + ... + c_1 n + c_0$

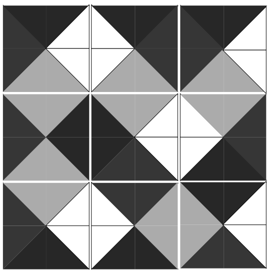
  where the c_i are constants, c_k are non-zero, and n is a parameter

Complexity

- Using big-oh notation, we have:
- On the other hand, if any step is executed 2^n times or more we have:
-

Complexity and Intractability

- The complexity of some algorithms is such that, in effect, they become intractable
- Consider the monkey puzzle problem:
- given nine square cards whose sides are imprinted with the upper and lower halves of coloured figures
- the objective is to arrange the cards in a 5x5 square such that halves match and colours are identical wherever edges meet



Complexity and Intractability

- Assume n, the number of cards, is 25
- The size of the final square is 5x5
- Brute force solution:
  - Go through all possible arrangements of the cards
  - pick a card and place it - there are 25 possibilities for the first placement
  - pick the next card and place it - there are 24 possibilities,
  - Pick the next card, there are 23 possibilities ...

Complexity and Intractability

- there are 25x24x23x22x......x2x1 possible arrangements
- That is, there are factorial 25 possible arrangements (25!)
- 25! contains 26 digits
- If we make 1000000 arrangements per second, the algorithm will take 490 000 000 000 years to complete

Complexity and Intractability

- The order of complexity of this algorithm is O(n!)
- n! grows at a rate which is orders of magnitude larger than the growth rate of the other functions we mentioned before
- It easily dwarfs all functions of the form n^k
- Other functions exist that grow even faster, e.g. n^n
- Even functions like 2^n exhibit unacceptable sizes even for modest values of n

Complexity and Intractability

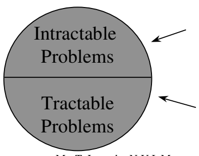
- We classify functions as 'good' and 'bad'
- A polynomial function is one that is bounded above by some function n^k for some fixed value of k
- (Strictly speaking n^k is not exponential but super-exponential)

Complexity and Intractability

- Polynomial-time algorithm
- Super-polynomial, exponential, time algorithm

Complexity and Intractability

- Tractable problem
- Intractable problem

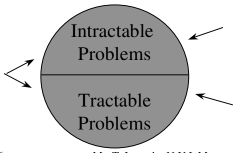


Complexity and Intractability

- There are many (approx 1000) important and diverse problems which exibit the same properties as the monkey puzzle problem
- All admit unreasonable, exponential-time, solutions
- None are known to admit reasonable ones
- But no-one has been able to prove that any of them REQUIRE super-polynomial time
- Best known lower-bounds are O(n)

Complexity and Intractability

- The class of problems are known as
- Lower bounds are linear and upper bounds are exponential



Complexity and Intractability

- Examples of NP-Complete Problems
  - 2-D arrangments
  - Path-finding (e.g. travelling salesman; Hamiltonian)
  - Scheduleing and matching (e.g. timetabling)
  - Determining logical truth in the propositional calculus
  - Colouring maps and graphs

Complexity and Intractability

- All NP-Complete problems seem to require construction of partial solutions (and then backtracking when we find they are wrong) in the development of the final solution
- However, if we could ‘guess’ at each point in the construction which partial solutions were to lead to the ‘right’answer then we could avoid the construction of these partial solutions and construct only the correct solution

Complexity and Intractability

- This approach would allow
  - a polynomial-time solution
  - but it would be non-determinisitic
  - since it requires some guessing
- NP - Nondeterministic Polynomial
- NP-Complete problems admit

Complexity and Intractability

- Imporant property of NP-Compete problems
  - If there exists a polynomial-time algorithm for any single NP-Complete problem, then there would be necessarily a polynomial-time algorithm for all NP-Complete problems
  - If there is an exponential lower bound for any NP-Complete problem, they all are intractable!

Complexity and Intractability

Complexity and Intractability

- We don't know!
- The NP=P? problem has been open since it was posed in 1971 and is one of the most difficult unresolved problems in computer science

12