Advanced Algorithms

Dr. Malek Mouhoub

Mathematics and Computer Science Department

University of Lethbridge

Spring 2001

1. Introduction

Boss gives the following problem to Mr Dupont, a fresh hired Bsc in computer science (to test him . . . or may be just for fun) :

$$\begin{aligned} T(1) &= 3 \\ T(2) &= 10 \\ T(n) &= 2T(n-1) - T(n-2) \end{aligned}$$

What is $T(100)$?

1. Introduction

Mr Dupont decides to **directly code** a recursive function **tfn**, in Java, to solve the problem :

if (n==1) $\{$ return 3; $\}$
else
  if (n==2) $\{$return 10;$\}$
  else $\{$ return 2 $*$ tfn(n-1) - tfn(n-2); $\}$

1. **First mistake :** no analysis of the problem

   $\Rightarrow$ risk to be fired !

2. **Second mistake :** bad choice of the programming language

   $\Rightarrow$ increasing the risk to be fired !

1. Introduction

$$
\begin{array}{rcl}
n = 1 & \rightarrow & 3 \\
n = 2 & \rightarrow & 10 \\
n = 3 & \rightarrow & 17 \\
n = 35 & \rightarrow & \text{it takes 4.19 seconds} \\
n = 100 & \rightarrow & \text{waits} \ldots \text{ and then kills the program !}
\end{array}
$$

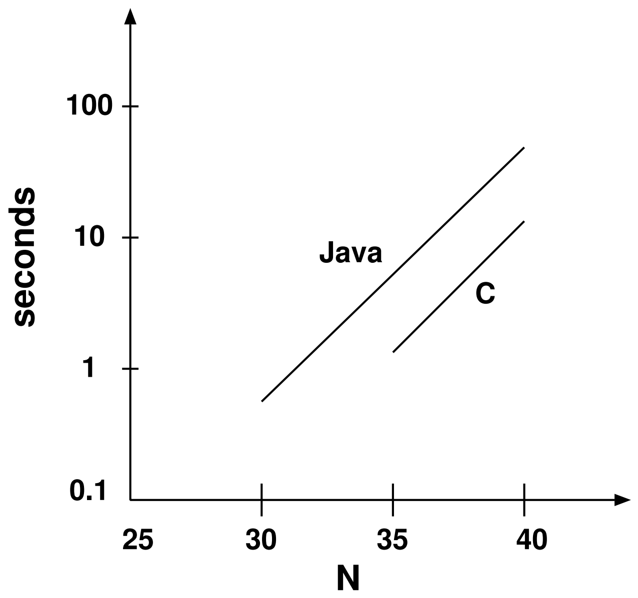
Mr Dupont decides then to use C :

```
if (n==1) return 3;
if (n==2) return 10;
return 2 * tfn(n-1) - tfn(n-2);
```

$$
\begin{array}{rcl}
n = 35 & \rightarrow & \text{it takes only 1.25 seconds} \\
n = 50 & \rightarrow & \text{waits and then kills the program !}
\end{array}
$$

1. Introduction

Finally, Mr Dupont decides to (experimentally) analyze the problem : he times both programs and plots the results.



It seems that each time n increases by 1 the time increases by $\approx$ 1.62. At $n = 40$ the C program took 13.79 seconds, so for $n = 100$ he estimates : $13.79 \times 1.62^{60} \approx 1,627,995$ years!!!

1. Introduction

- Mr Dupont remembers he has seen this kind of problems in one of the courses he has taken (data structures course).
- After consulting his course notes, Mr Dupont decides to use Dynamic Programming :

```
int t[n+1];
t[1]=3;
t[2]=10;
for(i=3;i¡=n;i++)
    t[i] = 2 ∗t[i-1] - t[i-2];
return t[n];
```

1. Introduction

This solution provides a much better complexity in time but despite of the space complexity :

- $n = 100$ takes only a fraction of a second,
- but for $n = 10,000,000$ (a test that may make the boss happy . . . if it succeeds) a core dump occurs. Too much memory required.

1. Introduction

- Mr Dupont analyses the problem again :
  - there is no reason to keep all the values, only the last 2 :

```
if (n==1) return 3;
last = 3;
current = 10;
for (i=3;i<=n;i++) {
   temp = current;
   current = 2*current - last;
   last = temp; }
return current;
```

  - At n = 100,000,000 it takes 3.00 seconds
  - At n = 200,000,000 it takes 5,99 seconds
  - at n = 300,000,000 it takes 8.99 seconds

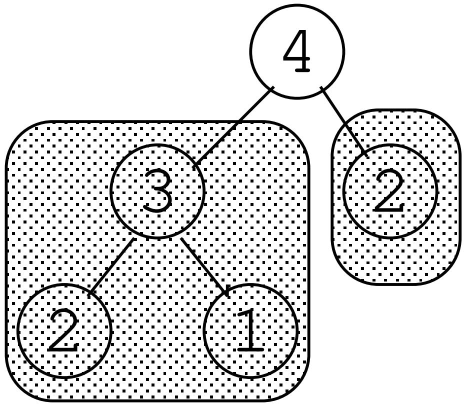
1. Introduction

How to solve such problems ?

1. Analyze the problem **on paper** in order to find an efficient algorithm in terms of time and memory space complexity.
   (a) First look at the problem :
   $$\begin{aligned} T(1) &= 3 \\ T(2) &= 10 \\ T(3) &= 17 \\ T(4) &= 24 \\ T(5) &= 31 \end{aligned}$$
   Each step increases the result by 7.
   (b) Guess : $T(n) = 7n - 4$
   (c) Proof by induction
2. Code : `return 7*n-4`

1. Introduction

Let us look at the recursion tree for the first program at n=4.



Here each circle represents one call to the routine `tfn`. So, for n=4 there are 5 such calls.

In general, a call to tfn(n) requires a recursive call to tfn(n-1)(represented by the shaded region on the left) and a call to tfn(n-2)(shaded region on the right).

1. Introduction

An algorithm analyst might ask :

1. What makes the first program so slow ?
2. How fast are the 3 programs asymptotically ?
3. Is the last version really the *ultimate* solution ?

1. Introduction

If we let $f(n)$ represent the number of calls to compute $T(n)$, then :

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2) + 1 \\
f(1) &= f(2) = 1
\end{aligned}
$$

This is a version of the famous Fibonacci recurrence.

It is known that $f(n) \approx 1.618^n$.

This agrees very well with he times we presented earlier where each increase of n by 1 increases the time by a factor of a little under 1.62.

We say such growth is exponential with asymptotic growth rate $O(1.618^n)$.

This answers question (1).

1. Introduction

In the second and third program there was a loop

```
for (i=3;i<=n;i++)
```

This loop contained two or three assignments, a multiplication and a subtraction.

We say such a loop takes $O(n)$ time.

This means that running time is proportional to n.

Recall that increasing n from 100 million to 300 million increased the time from approximately 3 to approximately 9 seconds.

The last program has one multiplication and one subtraction and takes $O(1)$ or *constant* time. The best measure we may get is ?

This answers question (2).

1. Introduction

The answer to the first of these questions depends partly on how we model computation, and partly the answer lies in the nature of Big-O.

Models also are key in the answer to the second question, but in general the answer is NO.

And this NO answer means that the answer to the last question is also NO. If the boss asked for $T(123456789876543212345566340014733134213)$ we would get integer overflow on most computers.

We need to distinguish the *size* of an integer from its *magnitude*. The size is the number of digits required to write it down.

1. Introduction

The third question, “is the last program the ultimate solution”, begins to cross the line from algorithm analysis to computing science.

A Computer Scientist might ask :

1. How do you justify counting function calls in the first case, counting array assignments in the second case, counting variable assignments in the third, and counting arithmetic operations in the last ?

2. Is it really true that you can multiply two arbitrary large numbers together in constant time ?

3. Is the last program really the ultimate one ?

1. Introduction

More CS questions that occur after thinking about the previous ones :

- What is computation ?
- What is the simplest model of computation ?
- How do various models relate - in terms of what can be computed and how efficiently ?
- What are the theoretical limits of computation ?
- What are the physical limits ?
- What are the practical limits ?
- How should we define a computational problem ?
- Can we prove some problems are hard to compute for all possible programs ?
- How can we classify problems as to how hard they are to compute ?

1. Introduction

CS questions with an engineering orientation :

- What general techniques can we use to solve computational problems ?
- What can these techniques be applied ?
- What data structures are best, and in what situations ?
- Which models should we use to analyze algorithms in practice ?
- When trying to improve the efficiency of a given program, which aspects should we focus on first ?

1. Introduction

Well . . . there is also CS and the BIG questions :

- Is the underlying foundation of the universe computational ?

  Aristote, Newton, Lovelace, Einstein, Turing, Feymann

- Is computation sufficient to explain intelligence and consciousness ?

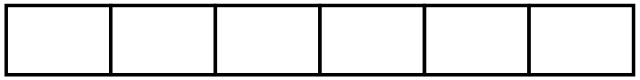
  Babbage, Lovelace, Turing, Searle, Minsky, Penrose

- Is evolution a computational process ?

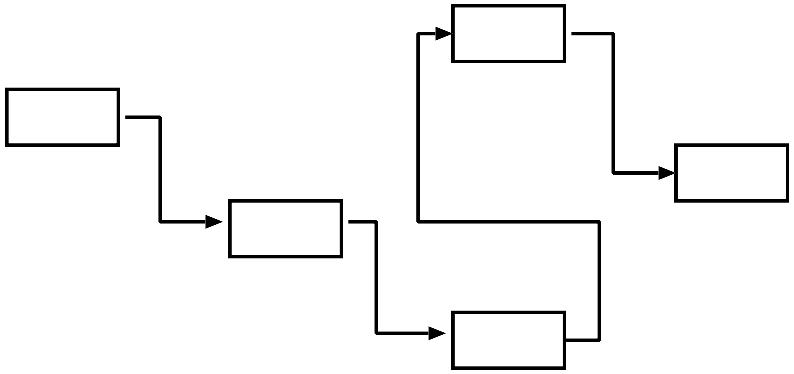
  Holland, Kaufmann

Data Structures

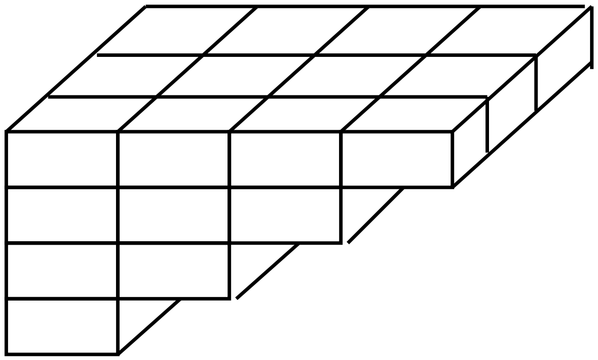
A scalar item



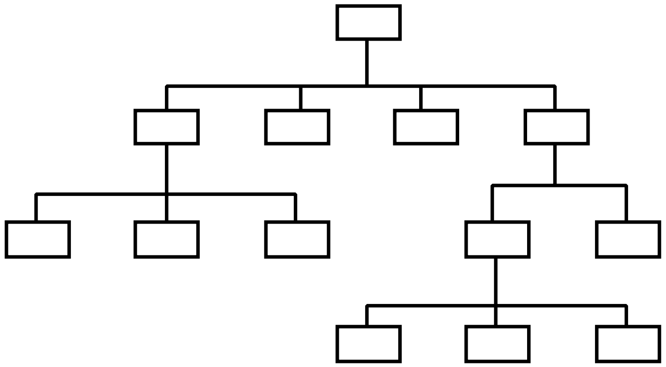
A sequential vector



A linked list



A n-dimentional space



A hierarchical tree

- The most important property to express of any entity in a system is its type.
- In this course we used entities that are structured objects (e.g., an object that is a collection of other objects).
- When determining its type, the kinds of distinguishing properties include :

  Ordering : are elements ordered or unordered ? If ordering matters, is the order partial or total ? Are elements removed FIFO (queues), LIFO (stacks), or by priority (priority queues) ?

  Duplicates : are duplicates allowed ?

  Boundedness : is the object bounded in size or unbounded ? Can the bound change or it is fixed at creation time ?

  Associative access : are elements retrieved by an index or key ? Is the type of the index built-in (e.g. as for sequences and arrays) or user-definable (e.g. as for symbol tables and hash tables) ?

  Shape : is the structure of the object linear, hierarchical, acyclic, n-dimensional, or arbitrarily complex (e.g. graphs, forests) ?