Introduction to Algorithms, Data structures and Formal Languages

3 assignments(25%), midterm test(10), final (65)
Textbook: M.J. Dinneen, G.Gimelfarb`, M.C. Wilson
Thanks to A/Prof Georgy Gimel'farb for most of the slides

Gisela Klette
Office hours: Tue, Thu, Fri, 2.30-3.30 p.m.
Room: Tamaki, 371, 318
Phone: 86826
E-mail: g.klette@auckland.ac.nz

What is an Algorithm?

Algorithm
Muhammed ibn Musa al-Khwarizmi (770-840)
-step by step rules for +,-,*,/

Def. 1 (informal):
An algorithm is a list of unambiguous rules that specify successive steps to solve a problem.

In other words:
- a precisely specified procedure for solving a problem
- a computable set of steps to achieve a desired result

Analysis of Algorithms

- Analysis of algorithms is a field in computer science whose overall goal is an understanding of the complexity of algorithms.

- **It is more important to have an efficient algorithm than a fast machine. Why?**

Def. 2 (informal):
An elementary operation is a computer instruction executed in a single time unit (computing step).

Programs are large sets of ordinary arithmetic and logical oprations such as negation, addition, substruction, mutiplication, division, boolean oprations, binary comparisons etc. These basic computer instructions are elementary operations.

Def. 3 (informal):
The running (computing) time of an algorithm is the number of elementary operations.

How to compare algorithms?

- by domain of definition – legal inputs
- by correctness – correct result for a legal input
- by efficiency-time and memory requirements

Efficiency of algorithms

- We are interested to estimate the running time of an algorithm and the memory space in order to compare algorithms independendly from the speed of the computer.

Examples

- **Example 1:**
  To calculate the sum: $s = \sum_{i=0}^{n-1} a_i$

We need n times the same elementary operation (adding a number).

$$T(n) = cn$$

7

Example 2: Euclid`s Algorithm

$$k = GCD(n, m)$$

1) "Brute-force" solution: -test all integers from 1 to the min{m,n}
We would need 7515 tests for GCT(9245,7515).

2) Euclids solution: if k divides both m and n, then it divides the difference (n-m if n>m)

$$GCD(n,m) = GCD(n-m,m)$$

k divides every difference when the substraction is repeated up to i times until n-i*m<m

$$GCD(n,m) = GCD(n \bmod m, m)$$

8

Example for Euclid`s algorithm

9245 mod 7515=1730	540 mod 55=45
7515 mod 1730=595	55 mod 45=10
1730 mod 595=540	45 mod 10=5
595 mod 540=55	10 mod 5=0

8 steps in comparison to 7515 steps!

9

Example 3: Sum of subarrays

Given is an array of size n, to calculate are sums of all subarrays of size m.

For example:

$$s_j = \sum_{k=0}^{m-1} a[j+k]; \quad j = 0,..., n-m$$

$$a = [3,2,5,3,7,1,8], \ m = 3$$

$$s_0 = 10, \ s_1 = 10, \ s_2 = 15, \ s_3 = 11, \ s_4 = 16$$

10

The total number of these subarrays is n-m+1, n is the number of elements in array a and m is the number of elements in the subarrays.

We can calculate the sum for each subarray, that means n-m+1 sums.
In each subarray we have m elements, so the running time is proportional to m*(n-m+1).

If m=n/2 then T(n)=c*n/2*(n/2+1)=0.25cn^2+0.5cn.

11

n	T(n)	0.25n^2	0.5n value	0.5n %
10	30	25	5	16.7
50	650	625	25	3.8
100	2,550	2,500	50	2.0
500	62,750	62,500	250	0.4
1000	250,500	250,000	500	0.2
5000	6,252,500	6,250,000	2,500	0.04

$$T(n) \approx cn^2 \ then \ T(10) \approx 100\,T(1)$$

12

Brute-force program (two nested loops)

```
...
int [ ]s = new int [m + 1];
    for int (k = 0; k ≤ m ; k + + ) {
   s[k ]= 0;
       for (int i = 0; i < m ; i + + )
       s[k ]+ = a[k + i];
 }
```

13

Brute-force Quadratic Approach

Quadratic time (two nested loops):

$$T(n) = c\frac{n}{2}(\frac{n}{2}+1) \approx c'n^2$$

Quadratic relative changes:

$$T(n) = n^2 T(1)$$

14

Brute-force Quadratic Approach

Quadratic time complexity is due to operations that we repeat several times In the innermost loop.

$$s_k = a_k + a_{k+1} + ... + a_{k+m-1}$$

$$s_{k+1} = a_{k+1} + ... + a_{k+m-1} + a_{k+m}$$

15

Better program

```
...
int [ ]s = new int [m + 1];
s[0 ]= 0;
 for (int i = 0; i < m ; i + + );
    s[0 ]+ = a[i ];
 for (int k = 1; k ≤ m ; k + + );
    s[k ]= s[k − 1]+ a[k + m − 1]− a[k − 1];
...
```

16

Linear Time for same problem

$$s_{k+1} = s_k + a_{k+m} - a_k$$

$$T(n) = c(m + 2m) = 1.5cn$$

17

If T(1)=1μs

Array size	n	2,000	2,000,000
Subarray size	m	1,000	1,000,000
Number of subsequences	m+1	1,001	1,000,001
Quadratic algorithm	T(n)	4s	>46 days
Linear algorithm	T(n)	3 ms	3 s

18

How to Run Faster / Save Memory?

- (**RF**) Save results of computations that could be reused later for the same data
- (**RF**) Tabulate functions of one or two integer arguments with relatively small ranges
- (**SM**) **Be careful** with recursive computations (to be sure that the stack is not growing too fast!)
- (**SM**) Free in due time and reuse the allocated memory

19

Running time estimations

We have seen in lecture 1 (2 examples) that a careful analysis of a problem can replace a straightforward brute-force approach with a more effective one.

Simplifying assumptions:
-All elemetary statements take the same amount of time
(e.g. simple arithmetic assignments)

20

Running time estimations

- Simple statement sequence

  s_1; s_2; …. ; s_k

  – $T(n)=c$ as long as k is constant

- Simple loops

```
for(i=0;i<n;i++) { s; }
```

  where s is executed in T(1)=c

  – *Time estimation: $T(n)=cn$*

21

Running time estimations

- Nested loops

```
for(i=0;i<n;i++)
     for(j=0;j<n;j++) { s; }
```

Time estimation: $T(n)=n\ cn=cn^2$

for k nested loops: $T(n)=cn^k$

22

Running time estimations

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
     s;
     h = 2 * h;
     }
```

  – h takes values 1, 2, 4, … until it exceeds n

  – There are $1 + \log_2 n$ iterations

  – Time estimation: $T(n)=c\log_2 n$

23

Running time estimations

•Loop index doesn't vary linearly

General: after m loops:

$$k^{m-1} < n < k^m$$

$$m-1 < \log_k n < m, T(n) = c\log_k n$$

24

Running time estimations

- Loop index depends on outer loop index

```
for(j=0;j<n;j++)
   for(k=0;k<j;k++){
     s;
     }
```

  - Inner loop executed
    - 1, 2, 3, …., n times

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$T(n) = c\,\frac{n(n+1)}{2} = c'n^2$$

Running time estimations

- Conditional switch statements are more complicated:

$$if\ (condition\ )\ then\ (T_1)\ else\ (T_2)$$
$$T(n) = f_{true}T_1 + (1 - f_{true})T_2 < \max\{T_1, T_2\}$$

(f_{true} – relative frequency of the true condition)

Running time estimations

- Function calls:

$$T = \sum_{i=1}^{k} T_i\ ,\ T_i - running\ \ time\ \ of\ \ statement\ \ i$$

- Function composition:

$$T(f(g(n))) = T(g(n)) + T(f(n))$$

Estimation rules

- Running time is proportional to the most significant term in T(n).
- Once a problem size becomes large, the most significant term is that which has the largest power of n
- The most significant term increases faster than the others.

Example:

$$T(n) = 0.25n^2 + 0.5n \cong 0.25n^2$$

n	T(n)	$0.25n^2$	0.5n value	0.5n %
10	30	25	5	16.7
50	650	625	25	3.8
100	2,550	2,500	50	2.0
500	62,750	62,500	250	0.4
1000	250,500	250,000	500	0.2
5000	6,252,500	6,250,000	2,500	0.04

Estimation rules

- Constants of proportionality depend on the compiler, language, processor,…
- Constants are ignored in algorithm analysis
- Constants of proportionality can change by use of faster hardware, it would not effect the behaviour of an algorithm for a large problem

Example

```
m=1
for(j=1;j<n;j++)
  if j=m then m=m(n-1);
   for(k=0;k<n-1;k++){
     s;
     }
     endfor
   endif
  endfor
```


Running time

We defined running time as the number of elementary operations for an algorithm depending on the number of input data.
-T(n) = cf(n)
-c depends on the computer, compiler, language, etc.
-type of function depends on the number of elementary operations

If c is unknown then we still know that the running time increases by a factor:

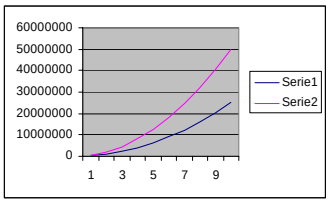
Relative running time:

$$\frac{T(n_2)}{T(n_1)} = \frac{cf(n_2)}{cf(n_1)} = \frac{f(n_2)}{f(n_1)}$$

Running time

We realized that the quadratic part in T(n) increases much faster as the linear part. We say: "the algorithm has an order of n^2 " or "is roughly proportional to n^2".

$$T(n) = 0.25n^2 + 0.5n \le 0.5n^2$$
$$0.25n^2 = 0.25n^2$$
$$0.5n \le 0.25n^2 \ \forall \ n > 1$$



Relative growth: G(n)=f(n)/f(5)

Complexity		Input size n			
Function	f(n)	5	25	125	625
Constant	1	1	1	1	1
Logarithm	log n	1	2	3	4
Linear	n	1	5	25	125
nlogn	n log n	1	10	75	500
Quadratic	n^2	1	25	625	15,625
Cubic	n^3	1	125	15,625	5^9
Exponential	2^n	1	2^{20}	2^{120}	2^{620}

We can approximate running time.

Question: How can we precisely express these approximations ?

We would like to find a way to say that an algorithm has the same time complexity as another one or an algorithm is more complex as another one.

For this reason we introduce the following standard mathematical tools:

Big Oh, Big Theta, Big Omega

"Big-Oh" tool

Def.4:
Let f(n) and g(n) nonnegative valued functions defined on nonnegative integers n.
Then g(n) is O(f(n)) (read: "g(n) is BigOh of f(n)") iff there exists a positive real constant c and a positive n_0 such that g(n) < cf(n) for all $n>n_0$.

“Big-Oh” tool

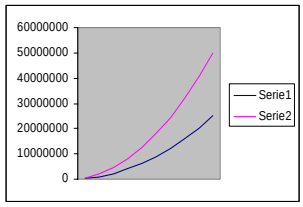
If g(n) is O(f(n) then an algorithm with running time g(n) runs, for large n , at most as fast, up to a constant factor, as an algorithm with running time f(n).

$T(n) = 0.25n^2 + 0.5n \le 0.5n^2$ for n>1

$0.25n^2 = 0.25n^2$

$0.5n \le 0.25n^2$

T(n) – Serie1
f(n) – Serie2



37

“Big-Oh” tool

We also say: g(n) is **asymptotically** f(n) for large values of n.
It means that g(n) for large n may come closer and closer to f(n).
O(f(n)) specifies an **asymptotic upper bound**.

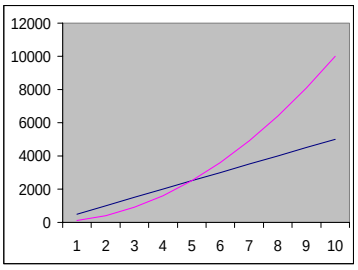
g(n) is O(f(n)) means: The function g(n) is an element of the set of all functions with time complexity O(f(n)).

Looking at the graphs of these functions:
g(n) is O(f(n)) iff there exist a constant c such that the graph of g(n) is always below or equal cf(n) after a certain point n_0.

38

“Big-Oh” tool - Examples

$g(n) = 5n, \quad f(n) = n^2/100$



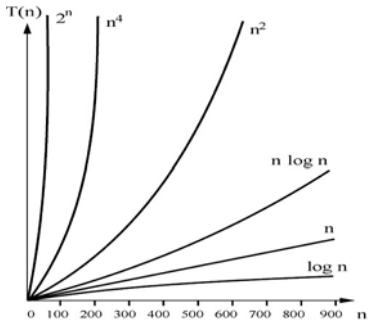
$5n = n^2/100$

$\Rightarrow n_0 = 500$

(times 100 input data)

39

“Big-Oh” tool - Examples



40

Some rules for “Big Oh”

1. Scaling:

cf is $O(f)$ $\forall$ $c > 0$

2. Transitivity:

If h *is* $O(g)$ *and* g *is* $O(f)$
then h *is* $O(f)$

41

Some rules for “Big Oh”

3. Rule of sums:

If g_1 *is* $O(f_1)$ *and* g_2 *is* $O(f_2)$,
then $g_1 + g_2$ *is* $O(\max\{f_1, f_2\})$

4. Rule of products:

If g_1 *is* $O(f_1)$ *and* g_2 *is* $O(f_2)$,
then $g_1 g_2$ *is* $O(f_1 f_2)$

42

“Big-Oh” tool – Examples-1

Linear function:

$g(n) = an + b;\ a > 0,\ is\ O(n)$

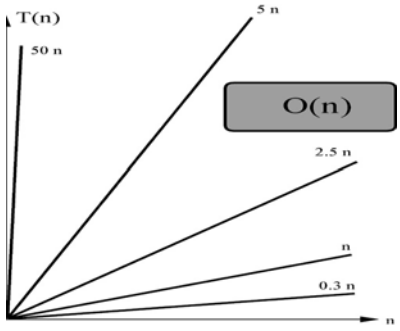
$g(n) \le (a + |b|)\, n \quad \forall\ n \ge 1$

All following examples have time complexity O(n):

$T(n) = 50\, n; \qquad T(n) = n/4 + 100$

$T(n) = 10^{8} + n; \qquad T(n) = 50 + 10^{-8} n$

“Big-Oh” O(…): Linear Complexity



“Big-Oh” tool – Examples-2

Polynomial function:

$P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$a_k > 0, \quad is\ O(n^k)$

All following examples have time complexity $O(n^k)$:

$T(n) = 3n^2 + 5n + 1 \quad is \quad O(n^2);$

$T(n) = 10^{-8} n^3 + 10^{8} n^2 + 3 \quad is \quad O(n^3);$

$T(n) = 100\ n^8 + 5n^6 + 1 \quad is \quad O(n^8)$

“Big-Oh” tool – Examples-3

Exponential functions:

$g(n) = 2^{n+k}\ is\ O(2^n),\ 2^{n+k} = 2^k 2^n \quad \forall\ n$

$g(n) = m^{n+k}\ is\ O(l^n),\ l \ge m > 1:$

$m^{n+k} \le l^{n+k} = l^k l^n \quad \forall\ n, k$

“Big-Oh” tool – Examples-4

Logarithmic functions:

$g(n) = \log_m n\ is\ O(\log_2 n);$

$\log_m n = \log_m 2 \cdot \log_2 n \quad \forall\ n, m > 0$

Following examples have logarithmic complexity :

$T(n) = 5 \log_{10} n + 100\ ;$

$T(n) = \log n + \log_5 \log_5 n$

Logarithmic “Big-Oh” Complexity