Theory of Computing

Lecture 1
MAS 714
2019
Hartmut Klauck

Organization:

- Lectures:
  Mon 10:30-11:30 TR+12
  Tue 10:30-12:30
- Tutorial:
  Tue 11:30-12:30
- Exceptions: This week no tutorial, next week no tutorial

Organization

- Final: 60%, Midterm: 20%, Homework: 20%

- There will be 4 sets of homework
  - First homework on September 2, to be handed in on September 10
  - Each set of homework is 5%
- http://www.ntu.edu.sg/home/hklauck/MAS714.htm

Books:

- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms
- Sipser: Introduction to the Theory of Computation
- Arora, Barak: Computational Complexity - A Modern Approach

Theory of Computing

ALGORITHMS

CRYPTOGRAPHY

DATA STRUCTURES

COMPLEXITY

MACHINE MODELS

UNCOMPUTABLE

FORMAL LANGUAGES

UNIVERSALITY

PROOF SYSTEMS

Overview

- First Half: Efficient Algorithms
  - Sorting
  - Graph Algorithms
  - Data Structures
  - Linear Programming
- Second Half: Theory of Computing
  - Computational Complexity
  - Computability
  - Formal Languages

Computation

- Computation: Mapping inputs to outputs in a prescribed way, by small, easy steps

- Example: Multiplication
  - Mult(a,b)=c such that a*b=c
    - How to find c?
    - School method

Algorithms

- An algorithm is a procedure for performing a computation
- Algorithms consist of elementary steps/instructions
- Elementary steps depend on the model of computation
  - Example: C++ commands
  - Models Like Turing machines allow very simple steps only

Algorithms: Example

- Gaussian Elimination
  - Input: Matrix M
  - Output: Triangular Matrix that is row-equivalent to M
  - Elementary operations: row operations
    - swap, scale, add

Algorithms

- Algorithms are named after Al-Khwārizmī
  (Abū ʿAbdallāh Muḥammad ibn Mūsā al-Khwārizmī)
  c. 780-850 ce
  Persian mathematician and astronomer
- (Algebra is also named after his work)
- His works (later) brought the positional system of numbers to the attention of Europeans

Algorithms: Example

- Addition via the school method:
  - Write numbers under each other
  - Add number position by position moving a „carry“ forward
- Elementary operations:
  - Add two numbers between 0 and 9 (memorized)
  - Read and Write
- Can deal with arbitrarily long numbers!

Datastructure

- The addition algorithm uses (implicitly) an *array* as datastructure
  - An array is a fixed length vector of cells that can each store a number/digit
  - Note that when we add x and y then x+y is at most 1 digit longer than max{x,y}
  - So the result can be stored in an array of length n+1 (where n allows to store x or y)

Multiplication

- The school multiplication algorithm is an example of a *reduction*
- First we learn how to add n numbers with n digits each
- To multiply x and y we generate n numbers $x_i \cdot y \cdot 10^i$ and add them up
- Reduction from Multiplication to Addition

Complexity

- We usually analyze algorithms to grade the performance
- The most important (but not the only) parameters are time and space
- Time refers to the number of elementary steps
- Space refers to the storage needed during the computation

Example: Addition

- Assume we add two numbers x,y with n decimal digits
- Clearly the number of elementary steps (adding digits etc) grows linearly with n
- Space is also $\approx$n

- Typical: asymptotic analysis

Example: Multiplication

- We generate n numbers with at most 2n digits, add them
- Number of steps is $O(n^2)$
- Space is $O(n^2)$
  - Easy to reduce to $O(n)$

- Much faster algorithms exist
  - (but not easy to do with pen and paper)
- Question: Is multiplication harder than addition?
  - Answer: we don‘t know...

Our Model of Computation

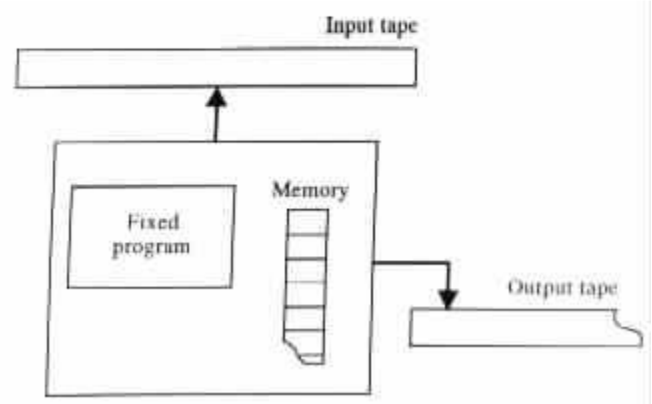
- We could use Turing machines...
- Will consider algorithms in a richer model, a RAM
- RAM:
  - random access machine
- Basically we will just use pseudocode/informal language

RAM

- Random Access Machine
  - Storage is made of registers that can hold a number (an unlimited amount of registers is available)
  - The machine is controlled by a finite program
  - Instructions are from a finite set that includes
    - Fetching data from a register into a special register
    - Arithmetic operations on registers
    - Writing into a register
    - Indirect addressing

RAM

- Random Access Machines and Turing Machines can simulate each other
- There is a universal RAM
  - Can simulate all other RAMs when given their program
- RAM's are very similar to actual computers
  - machine language



Computation Costs

- The time cost of a RAM step involving 1 or 2 registers is the logarithm of the numbers stored
  - logarithmic cost measure
  - adding numbers with n bits takes time n etc.
- The time cost of a RAM program is the sum of the time costs of its steps
  - For a fixed input
- Space used is the sum (over all registers used) of the logarithms of the maximum numbers stored in the register

Other Machine models

- Turing Machines (we will define them later)
- Circuits
- Many more!

- A machine model is universal, if it can simulate any computation of a Turing machine
- RAM's are universal
  - Vice versa, Turing machines can simulate RAM's

Types of Algorithm Analysis

- Usually we will use asymptotic analysis
  - Reason: next year‘s computer will be faster, so constant factors don‘t matter (usually)
  - Understand the inherent complexity of a problem (can you multiply in linear time?)
  - Usually gives the right answer in practice
- Worst case analysis
  - The running time of an algorithm is the maximum time used over all inputs
  - On the safe side for all inputs…

Other Types of Analysis

- Average case:
  - Average under which distribution?
  - Often the uniform distribution, but may be unrealistic
- Amortized analysis:
  - Sometimes after some costly preparations we can solve many problem instances cheaply
  - Count the average cost of an instance (preparation costs are spread between instances)
  - Often used in datastructure analysis

Asymptotic Analysis

- Different models of computation lead to different running times
  - E.g. depending on the instruction set
- Also, real computers become faster through faster processors
  - Same sequence of operations performed faster
- Therefore we generally are not interested in constant factors in the running time
  - Unless they are very bad

O, Ω, Θ

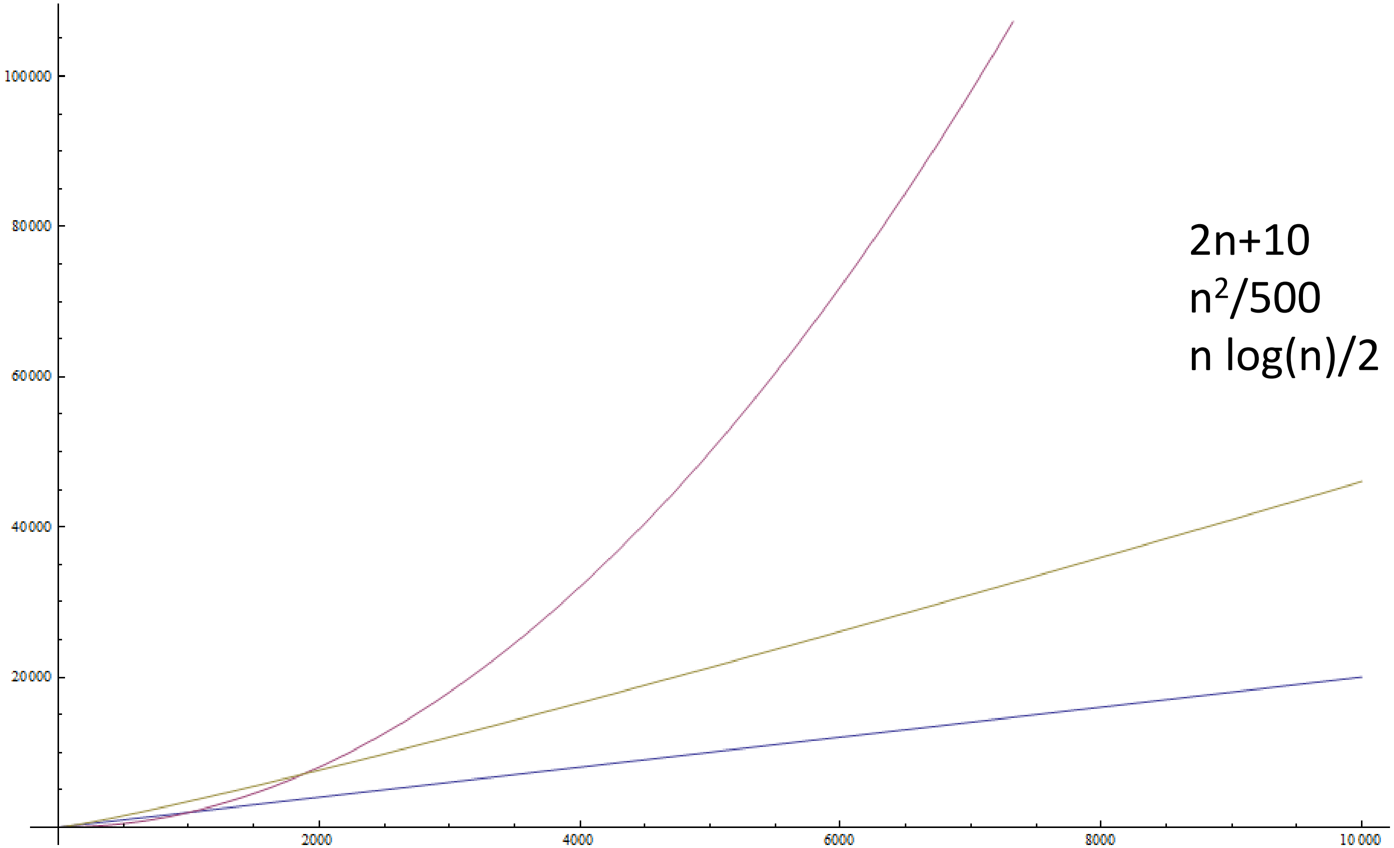
- Let f,g be two monotonically increasing functions that send **N** to **R**$^+$
- f=O(g) if $\exists\, n_0, c\ \forall\, n > n_0$: $f(n) \leq c\, g(n)$
- Example:
  f(n)=n, g(n)=1000n+100 $\Rightarrow$ g(n)=O(f(n))
  - Set c=1001 and n_0=100
- Example:
  f(n)=n log n, g(n)=n^2

O, Ω, Θ

- Let f,g be two monotonically increasing functions that send **N** to $\mathbf{R}^+$
- f = Ω(g)   iff g=O(f)
  - Definition by Knuth

- f = Θ(g)   iff [f=O(g) and g=O(f)]
- o, ω: asymptotically smaller/larger
- E.g., $n=o(n^2)$
- But $2n^2 + 100\,n=\Theta(n^2)$

Some functions



2n+10
$n^2/500$
n log(n)/2

Sorting

- Computers spend a lot of time sorting!
- Assume we have a list of numbers $x_1,\ldots,x_n$ from a universe U
- For simplicity assume the x_i are distinct
- The goal is to compute a permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$
- Think of a deck of cards
- Often the numbers have additional information attached
  - E.g. Telephone book

InsertionSort

- An intuitive sorting algorithm
- The input is provided in A[1…n]
- Code:

```
for i = 2 to n,
        for (k = i; k > 1 and A[k] < A[k-1]; k--)
                swap A[k,k-1]

                → invariant: A[1..i] is sorted
end
```

- Clear: $O(n^2)$ comparisons and $O(n^2)$ swaps
- Algorithm works in place, i.e., uses linear space

Correctness

- By induction
- Base case: n=2:
  We have one conditional swap operation, hence the output is sorted
- Induction Hypothesis:
  After iteration i the elements A[1]…A[i] are sorted
- Induction Step:
  Consider Step i+1. A[1]…A[i] are sorted already. Inner loop starts from A[i+1] and moves it to the correct position. After this A[1]…A[i+1] are sorted.

Best Case

- In the worst case Insertion Sort takes time $\Omega(n^2)$
- If the input sequence is already sorted the algorithms takes time $O(n)$
- The same is true if the input is almost sorted
  - important in practice
- Algorithm is simple and fast for small n

Worst Case

- On some inputs InsertionSort takes $\Omega(n^2)$ steps
- Proof: consider a sequence that is decreasing, e.g., n,n-1,n-2,…,2,1
- Each element is moved from position i to position 1
- Hence the running time is at least $\sum_{i=1,\ldots,n} i = \Omega(n^2)$

Can we do better?

- Attempt 1:
  Searching for the position to insert the next element is inefficient, employ *binary search*

- Ordered search:
  - Given an array A with n numbers in a sorted sequence, and a number x, find the smallest i such that A[i] >= x

- Use $A[n+1]=\infty$

Linear Search

- Simplest way to search
- Run through A (from 1 to n) and compare A[i] with x until A[i] >= x is found, output i
- Time: Θ(n)
- Can also be used to search unsorted Arrays

Binary Search

- If the array A is sorted already, we can find an item much faster!
- Assume we search for a x among A[1]<...<A[n]
- **Algorithm** (to be first called with l=1 and r=n):
  - BinSearch(x,A,l,r]
    - If r-l=0 test if A[l]=x, end
  - Compare A[(r-l)/2+l] with x
  - If A[(r-l)/2+l]=x output (r-l)/2+l, end
  - If A[(r-l)/2+l]> x BinSearch(x,A,l,(r-l)/2+l)
  - If A[(r-l)/2+l]< x Bin Search(x,A,(r-l)/2+l,r)

Time of Binary Search

- Define T(n) as the time/number of comparisons needed on Arrays of length n
- T(2)=1
- T(n)=T(n/2)+1

- Solution: T(n)=log(n)

- logs have base 2 usually
  - In asymptotic analysis the base of the logarithms does not matter (as long as it's constant)

Recursion

- We just described an algorithm via recursion: a procedure that calls itself
- This is often convenient but we must make sure that the recursion eventually terminates
  - Have a base case (here r=l)
  - Reduce some parameter in each call (here r-l)

Binary Insertion Sort

- Using binary search in InsertionSort reduces the number of comparisons to O(n log n)
  - The outer loop is executed n times, each inner loop now uses log n comparisons
- Unfortunately the number of swaps does not decrease:
  - To insert an element we need to shift the remaining array to the right!