Prof. Dr. rer. nat. habil. Martin O. Steinhauser

Frankfurt University of Applied Sciences, Germany

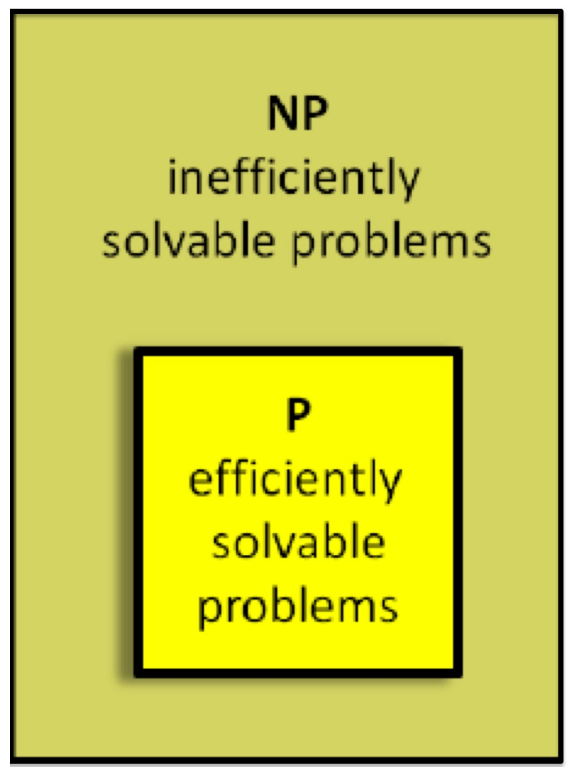
Faculty of Computer Science and Engineering

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Short Lecture Course:

Introduction to Computational Science with Applications in Molecular Dynamics

Sessions 5-6: Analysis of Algorithms and Asymptotic Analysis



Overview of this short course

Topics Covered (subject to change)

1st Session: Lec. 1-2		Introduction & Bits and Bytes
2nd Session: Lec 3	(2x)	Bits and Bytes continued
3rd Session: Lec 4-6		Molecular Dynamics
4th Session: Lec 7-8		MD continued / Algorithms
5th Session: Lec 9	(2x)	Algorithms/ Problem of Sorting
6th Session: Lec 10-11		Asymptotic Analysis of Algorithms
7th Session: Lec 12-13		Monte Carlo/Random Numbers

Session 5 / 6: Overview

OUTLINE OF LECTURES

- Short Review
- Asymptotic Analysis of Algorithms
- Some practical Examples

Overview of Lecture 7

Overview of Lecture 7

Short Review

- **Algorithm:** finite, deterministic, effective, efficient

- **Performance** is often at the edge of scientific innovation (simulating things that have never been done) and determines what is feasible and unfeasible.

Short Review
Insertion Sort

- Pseudo-Code of *Insertion Sort*:

```
for j = 2 to n
  do key = A[j]
  // Remark: Insert A[j] into the sorted sequence A[1,...,j-1]
  i = j -1
  while i > 0 and A[j] > key
      do A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

Short Review
What does the running time $T(n)$ of *Insertion Sort* *depend on?*

- Input size (sorting more elements takes more time)

- Input itself (whether it is already sorted)

- Running time $T(n)$ of an algorithm means the number of elementary operations (or steps) executed.

- We usually want to know *upper bounds* on the running time !

$$\left.\begin{array}{l}\text{“Worst-Case”}\\ \text{“Average Case”}\\ \text{“Best Case”}\end{array}\right\}\ \text{analysis of an algorithm}$$

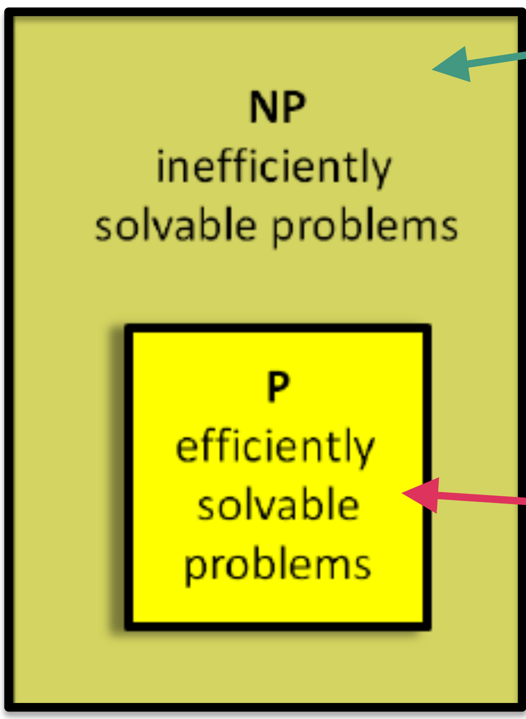
Overview of Lecture 7

Classification of Algorithms

algorithm	runtime	N=10	N=20	N=50	N=100
A_1	N	10 ES $10^{-8}\,s$			
A_2	N^2				
A_3	N^3				
A_4	2^N				
A_5	$N!$				

Classification of Algorithms
P and NP Problems

- P: Polynomial time solvable
- NP: Nondeterministic polynomial time solvable



Only *exponential* algorithms are known

Examples: Travelling Salesman Problem,
Three-Color Problem,
Knapsack Problem,
… and many more

Polynomial algorithms are known.
Sometimes, ***Recursion*** is an elegant way to find an algorithmic solution

Example: Towers of Hanoi

Why are Non-Polynomial Algorithms Inefficient?

Assume a technology jump with a speedup factor of 10 or 100

algorithm	runtime	efficiency	speedup factor 10	speedup factor 100
A_1	N	N_1		
A_2	N^2	N_2		
A_3	N^3	N_3		
A_4	2^N	N_4		
A_5	$N!$	N_5		

Why are Non-Polynomial Algorithms Inefficient?

Assume a technology jump with a speedup factor of 10 or 100

algorithm	runtime	efficiency	speedup factor 10	speedup factor 100
A_1	N	N_1		
A_2	N^2	N_2		
A_3	N^3	N_3		
A_4	2^N	N_4		
A_5	$N!$	N_5		

- Polynomial algorithms are shifted by a <u>constant factor</u>
- Non-Polynomial algorithms are shifted by an <u>additive constant</u>

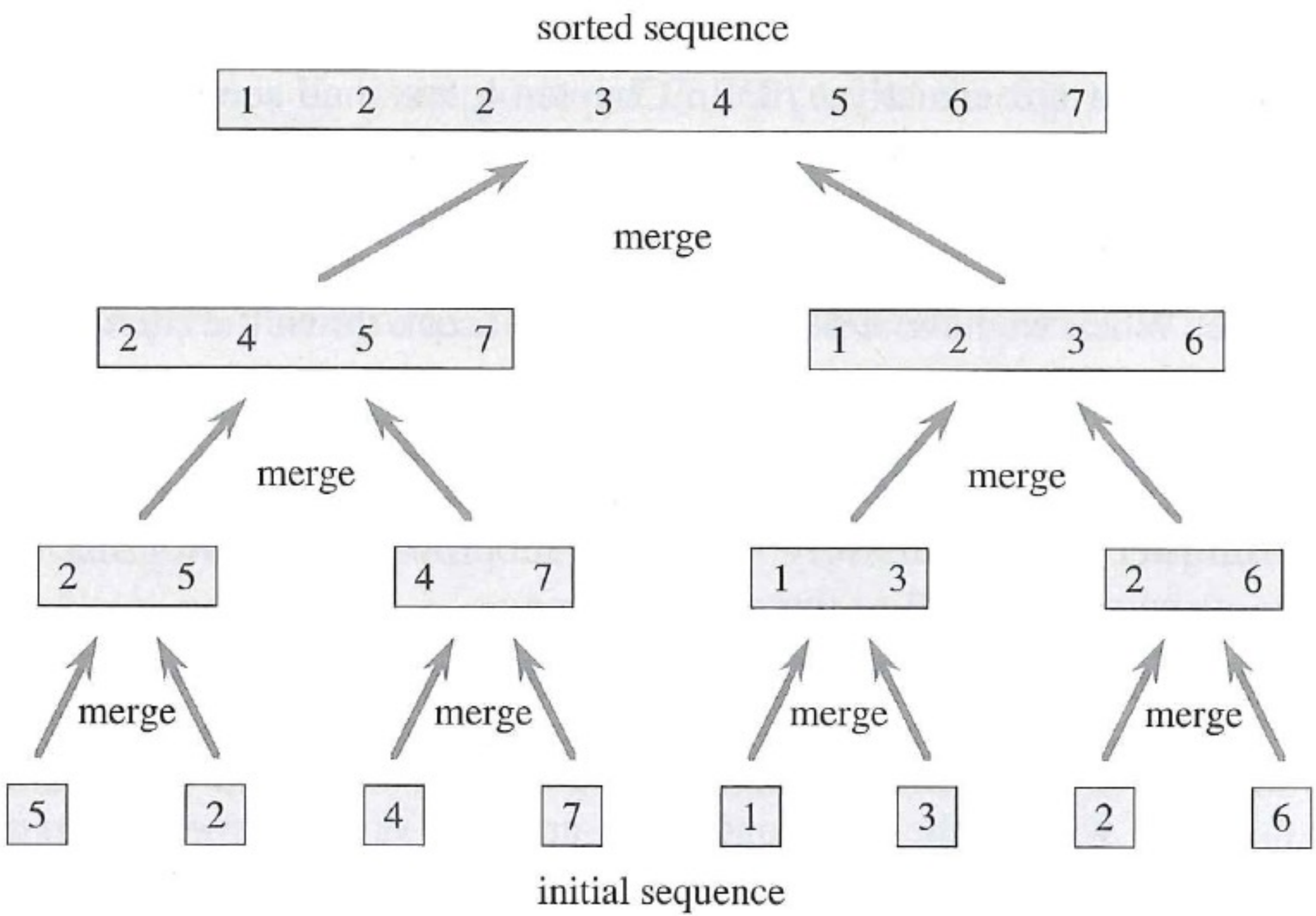
So, is insertion sort fast?

```
for j = 2 to n
  do key = A[j]
  // Remark: Insert A[j] into the sorted sequence A[1,...,j-1]
  i = j -1
  while i > 0 and A[j] > key
      do A[i+1] = A[i]
      i = i - 1
  A[i+1] = key
```

Is *Insertion Sort* FAST ?

- Moderately so, for *small* N: $O(N)$
- Not at all for *large* N! $O(N^2)$

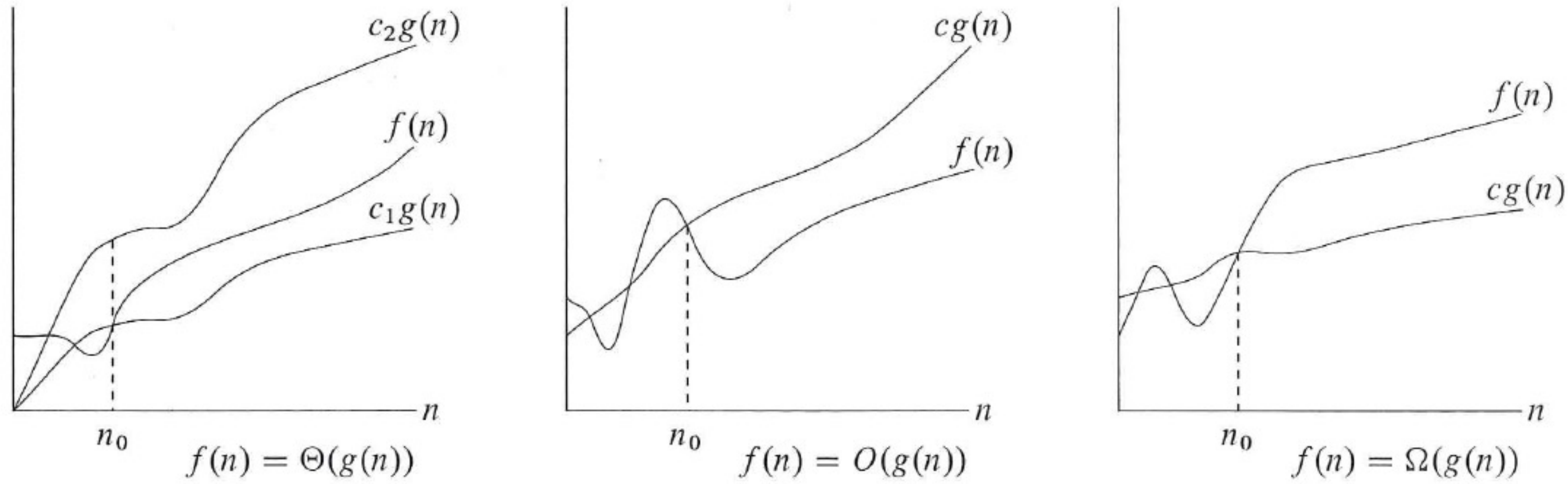
An Algorithm that is fast than IS
Merge-Sort: A Divide-and-Conquer Approach



Asymptotic Notations: Overview
$\theta-$, Big-O and Ω-Notation

$g(n)$ is:

asymtotically tight bound an asymptotic upper bound an asymptotic lower bound



Overview of Lecture 7

Example: Maximum Subarray Problem

Find the maximum partial sum of consecutive numbers in a given array:

Example: -59, 52, 46, 14, -50, 58, -87, -77, 34, 15

The maximum partial sum would be:

$$52 + 46 + 14 + -50 + 58 = 120$$

Used for e.g.:

- Detection of praphical patterns
- Analysis of share prices at the stock market

Maximum Subarray Problem
Cubic Algorithm

```
int maxsum1(int z[], int n){

  int i,j,k, sum, max=-10000000;

    for (i = 0; i < n; i++)
      for (j = i; j < n; j++){
         sum = 0;
         for (k = i; k <= j; k++){
           sum += z[k];
          }
          if (sum > max)
          max = sum;
       }
    return(max);
}
```

3 nested loops!
Time required is proportional to n^3

Maximum Subarray Problem
Quadratic Algorithm

```
int maxsum2(int z[], int n){

    int i,j,k, sum, max=-10000000;

      for (i = 0; i < n; i++){
         sum = 0;
         for (j = i; j < n; j++){
            sum += z[j];
           if (sum > max)
            max = sum;
         }
      }
      return(max);
}
```

Access to the sum already calculated:
S(i,j-1) --->S(i,j) = S(i,j-1) + z[j]
saves the third loop

2 nested loops!
Time required is proportional to n^2

Maximum Subarray Problem
Kinear Algorithm = Optimum !

```
int maxsum3(int z[], int n){

   int i,s, totalmax=-10000000, endsum = 0;
     for (i = 0; i < n; i++){
        endsum = ((s = endsum + z[i]) > 0) ? s : 0;
       if (endsum > totalmax)
           totalmax = endsum;
      }
    return (totalmax);
}
```

No nested loops!
Each element is accessed only once!
Time required is proportional to n

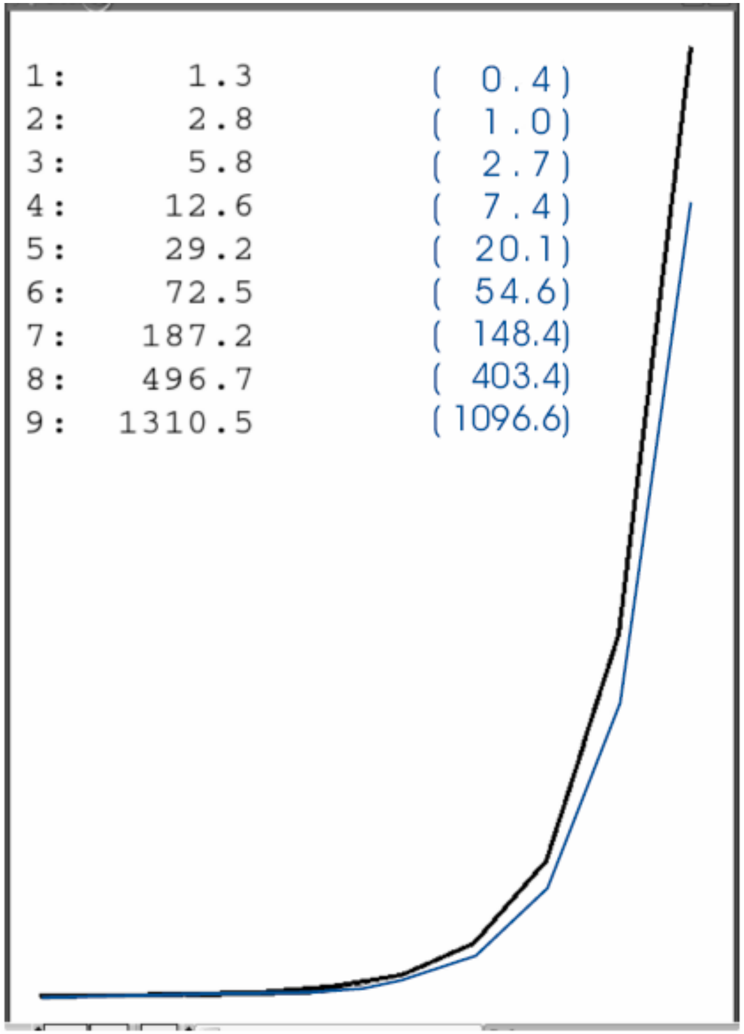
Checking for Primes: Exponential Growth

This algorithm checks recursively, whether number is a prime

```
int prim(int number, int divisor, int *z) {
   (*z)++;
   if (number < 2 || number%2 == 0 || number%divisor == 0)
      return 0;
   else if (divisor*divisor > number)
      return 1;
   return prim(number, divisor+1, z);
}
```

Checking for Primes: Exponential Growth

The number of recursive calls depends on the number of digits; here: 1-9



Complexity of Algorithms
Memory is also Important

Besides the complexity with respect to time, the complexity with respect to memory is also important!

Example: Two algorithms that solve the same problem but with different memory usage

Array of size n stores integers [0, n−1] .
The problem is now, to check, whether there are numbers in the array that are doubly listed.

Complexity of Algorithms
Quadratic Complexity of Memory Storage

```
int doppcheck1(int z[]) {
    for (int i=0; i<MAX-1; i++)
        for (int j=i+1; j<MAX; j++)
            if (z[i] == z[j])
                return 1;
    return 0;
}
```

Just an array with n numbers
BUT:
Two nested loops: complexity of running time $\sim n^2$
Complexity of memory storage: $\sim n^2$

Complexity of Algorithms
Exponential Complexity of Memory Storage

```
int help [ MAXNUMBER] = {0} ;
 ....
int doppcheck2(int z[]) {
   for (int i=0; i<MAX; i++)
       if ( help z[i ]] != 0)
          return 1;
       else
          help [z[i ]] = 1;
   return 0;
}
```

Prize for linear complexity in time:
Exponential complexity in memory

Just one loop: complexity of running time ~ n (perfect!)
BUT: Complexity of memory storage is: $\sim 2^m$ (worst!)
Problem size: m is the number of bit-positions needed to store the largest number

Example: Sequential Search
O(N) Complexity

Is a certain number element of an array?

```
int sequentialSearch(int z[], int n, int number){
    int i;
      for (i = 0; i < n; i++)
        if (z[i] == number)
          return i;              /* Number found */
     return (-1);                /* Number not found */
}
```

Example: Quick Exponentiation
O log N Complexity

Legendre algorithm for quick exponentiation of a number (pseudo code)

```
x = a; y = b; z = 1;
while (y < 0){
  if (y „uneven“)
    z = z * x;
y = y / 2;
x = x * x;
}
/* z = a to the power of b */
```

Complexity of Algorithms
Program Demonstrations

Some simple, practical tricks of the trade for program performance optimizations!

Constant Propagation

Operator Strength Reduction

Copy Propagation

Loop Strength Reduction

Invariant Code Motion

Loop Jamming

Constant Propagation

Use constants, not variables in expressions

Without Optimization

```
for (i=1; i < 100000000; i++){
  x = 2;
  y = x + 5;
  a = x;
  b = 0;
  c = a / x;
  d = x * c * c * c * c;
  e = x + b +b + b;
}
```

With Optimization

```
for (i=1; i < 100000000; i++){
  x = 2;
  y = 7;

  b = 0;
  c = 1;

  e = d = e = x;
}
```

Operator Strength Reduction

Avoid unnecessary function calls

Without Optimization

```
for (i=1; i < 100000000; i++){
  x = ceil(pow(2,17))
  x = a / 8;
  c = b * 16,
  if (d%2 != 0)
    e = x + 3;
}
```

With Optimization

```
for (i=1; i < 100000000; i++){
  y = 2.17 * 2.17;
  x = y + 0.5
  x = a >> 3
  c = b << 4
if (d&1)
    e = x + 3;
}
```

Copy Propagation

Avoid unnecessary or double calculations

Without Optimization

```
for (i=1; i < 100000000;
i++){
  a = x * y;
  b = x;
  c = b * y;
  d = x * y;
}
```

With Optimization

```
for (i=1; i < 100000000;
i++){
  b = x;
  a = b = d = x * y ;
}
```

Loop Strength Reduction

Avoid unnecessary loops

Without Optimization

```
for (i=1; i <= 100000; i++){
  for (j=0; i <= 1000; j++){
    if (j%10 ==0)
      array[j] = j;
```

With Optimization

```
for (i=1; i <= 100000; i++){
  for (j=0; i <= 1000; j+=10){
    array[j] = j;
```

Invariant Code Motion

Remove constant terms from loops

Without Optimization

```
for (i=1; i <= 10000; i++){
  for (j=1; i <= 1000; j++)
    array[j] = a + b * sin(2.33)

    for (j = 1; j < strlen(string) – sqrt(h); j++)
      string[j] = ‚-‘;
}
```

With Optimization

```
x = a + b * sin (2.33);
for (i=1; i <= 10000; i++){
  for (j=1; i <= 1000; j++)
    array[j] = x;
    l = strlen(string) – sqrt(h);
      for (j=1; i < l; j++)
      string[j] = ‚-‘;
}
```

Loop Jamming

Combine loops of the same size

Without Optimization

```
for (i=1; i <= 10000000; i++){
  for (j=0; j <= 1000; j++)
    a[j] = j;
  for (j=0; j <= 1000; j++)
    b[j] = a[j] + x;
  for (j=0; j <= 1000; j++)
    c[j] = a[j];
}
```

With Optimization

```
for (i=1; i <= 10000000; i++){
  for (j=0; j <= 1000; j++){
    a[j] = j;
    b[j] = j + x;
    c[j] = j;
}
```

Learning Objectives

- Understand the Importance of Asymptotic Analysis
- Understand the Concept of Divide and Conquer
- Be aware of Optimization Techniques

My University Research Page: https://www.frankfurt-university.de/steinhauser

Contact Me: martin.steinhauser@fb2.fra-uas.de

Research Gate: https://www.researchgate.net/profile/Martin-Steinhauser