## Content

| 1 Understanding and analyzing algorithms | 1.1 Analyzing algorithms: 1. What? 2. How? 3. Why? → Correctness: Loop invariant, Induction on the number of times through the loop 4. When? → Time analysis: Number of comparisons, worst-, best-, average case 1.2 Time complexity, time as a function of the input size $n$, Big $O$, $o$, Big $\Omega$, $\omega$, Big $\Theta$ 1.3 Running time analysis for algorithms (Upper and Lower Bounds) 1.4 Recursive algorithms (Exponentiation, Towers of Hanoi, Merging sorted arrays, Binary strings avoiding 00, Ternary strings avoiding a 2 after a 0): Correctness: Induction Time analysis: Guess and Check then Induction, Unraveling the Recurrence, Master theorem 1.5 Divide-and-Conquer (MergeSort, Multiplication of two $n$-digit numbers) 1.6 Worst-case analysis of algorithms: Scheduling algorithms, Graham's results |
| 2 Using graphs and graph algorithms | |
| 3 Using combinatorial reasoning and probability to quantitatively analyze algorithms and systems | |
Recursion
Recursive algorithms

A recursive problem is one that can be defined in terms of smaller instances of the same problem. Searching in a sorted array with Binary Search is such an example.

Noticing that a problem has a recursive structure can help you write an algorithm for solving it.

Some benefits to recursive algorithms are that they are easy to write down and easy to prove correct - just use induction.
Recursive algorithms: Correctness

Induction and recursion go hand in hand:

**Induction:** a proof strategy where we show
- a base case
- how to prove a statement about \( n \), assuming it is true of \( n-1 \)

**Recursion:** a way of defining a problem where we must state
- a base case
- how to solve a problem of size \( n \), assuming we can solve a problem of some smaller size
When weak and when strong induction?

**Weak induction:**
If a recursive algorithm solves a problem of size $n$ by turning it into a smaller problem of size $n-1$, we use regular (weak) induction.

- **Base case:** $k=0$ or $k=1$ is true.
- **Inductive step:** If the inductive hypothesis is true for $k=n-1$, then it is also true for $k=n$.

**Strong induction:**
If a recursive algorithm solves a problem of size $n$ by turning it into a problem of some other smaller size, like $n/2$, we must use strong induction.

- **Base cases:** Cases $k=1$ is true.
- **Inductive step:** If the inductive hypothesis is true for $k=1,...,n-1$ (this is a stronger assumption than above!), then it is also true for $k=n$.

Because we can (not have to, though!) use all $n-1$ statements ($k=1,...,n-1$ is true) to prove $k=n$ is true, strong induction is a more flexible proof technique than weak induction.
Example 1: Exponentiation
Example: Exponentiation (Correctness Proof)

The following algorithm, given a positive integer \( n \), computes \( 2^n \).

```c
int ExpBase2(int n)
{
    if(n==1) then return(2);
    return(ExpBase2(n-1)*2);
}
```

To prove recursive algorithms correct, we use induction on \( n \), the input size.

- **Base Case:**
  When \( n=1 \), the algorithm returns \( 2=2^1 \).

- **Inductive Hypothesis:**
  Assume that for \( n \geq 1 \), \( \text{ExpBase2}(n-1) \) returns \( 2^{n-1} \).

- **Inductive Step:**
  Then \( \text{ExpBase2}(n) \) returns \( \text{ExpBase2}(n-1) \times 2 = 2^{n-1} \times 2 = 2^n \).
  This completes the inductive step.
Example: Exponentiation (Time analysis)

The following algorithm, given a positive integer \( n \), computes \( 2^n \).

```c
int ExpBase2(int n)
if (n==1) then return (2);
return (ExpBase2(n-1)*2);
```

Let \( M(n) \) be the number of multiplications needed to compute \( 2^n \) with \texttt{ExpBase2}.

**Time analysis strategy 1: “Guess and Check, then Induction”**

\[
\begin{align*}
M(1)&=0 \\
M(2)&=1 \\
M(3)&=2 \\
M(4)&=3 \\
\cdots \\
M(n)&=???
\end{align*}
\]
Example: Exponentiation (Time analysis)

The following algorithm, given a positive integer $n$, computes $2^n$.

```c
int ExpBase2(int n)
if (n==1) then return (2);
return (ExpBase2(n-1) * 2);
```

Let $M(n)$ be the number of multiplications needed to compute $2^n$ with `ExpBase2`.

Time analysis strategy 1: “Guess and Check, then Induction”

<table>
<thead>
<tr>
<th>M(1)</th>
<th>M(2)</th>
<th>M(3)</th>
<th>M(4)</th>
<th>...</th>
<th>M(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>n-1</td>
</tr>
</tbody>
</table>

Lemma: `ExpBase2` needs $n-1$ multiplications to compute $2^n$.

Proof (by induction).

- $n=1$: 0 multiplications to compute $2^1=2$.
- $n-1 \rightarrow n$: Inductive hypothesis: $n-2$ multiplications to compute $2^{n-1}$.
  - There is one more multiplication with 2 to compute $2^{n-1} \times 2 = 2^n$,
  - in total we have then $n-1$ multiplications to compute $2^n$.

This completes the inductive step.
Example: Exponentiation (Time analysis)

The following algorithm, given a positive integer \( n \), computes \( 2^n \).

```plaintext
int ExpBase2(int n)
if(n==1) then return(2);
return(ExpBase2(n-1)*2);
```

Let \( M(n) \) be the number of multiplications needed to compute \( 2^n \) with \texttt{ExpBase2}.

**Time analysis strategy 2: “Unraveling the recurrence”**

\[
M(1) = 0 \\
M(n) = M(n-1) + 1 \\
    = M(n-2) + 2 \\
    = \ldots \\
    = M(n-k) + k \\
    = \ldots \\
    = M(1) + n - 1 = n - 1
\]
The fun part

- It can be fun to try to find the recursive structure in problems. Once we have the recurrence, the rest is just induction proofs and unraveling the recurrence.

- Let's look at some more problems and see if we can identify the recursive structure.
Example 2: Towers of Hanoi
Example: Towers of Hanoi

“In the Temple of Brahma in Hanoi there is a brass platform with three diamond needles and 64 golden discs all of different sizes. At the beginning of time the discs were placed on the first needle in a pile from largest up to smallest. The priests of the temple are transferring the discs to another needle one at a time so that no disc ever rests on a smaller disc. When they finish, time and the world will end.”

Disclaimer: These slides were made by Berteun Damman and Martin Hofmann, and found at http://www.texample.net/tikz/examples/towers-of-hanoi/.
Example: Towers of Hanoi

**The solution has three steps:**

1. Move the stack of the smallest \( n-1 \) disks to an empty pole.
2. Move the largest disk to an empty pole.
3. Move the stack of the smallest \( n-1 \) disks to the pole with the largest disk.

If \( T(n) \) is the number of moves required to solve the puzzle with \( n \) disks, we have:

\[
T(n-1) \text{ moves}
\]

\[
1 \text{ move}
\]

\[
T(n-1) \text{ moves}
\]

Therefore, \( T(n) = 2T(n-1) + 1 \).
Example: Towers of Hanoi (Time Analysis 1)

\[ T(n) = 2T(n-1) + 1 \]

<table>
<thead>
<tr>
<th>n</th>
<th>T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>n</td>
<td>???</td>
</tr>
</tbody>
</table>

„Guess and Check …“
Try plugging in small values of n and guessing a pattern. Confirm your guess with a proof by induction.
Example: Towers of Hanoi (Time Analysis 1)

\[ T(n) = 2T(n-1) + 1 \]

Proof by induction on n

<table>
<thead>
<tr>
<th>n</th>
<th>T(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

Base case: \( n=1, T(1)=1 \). Correct.

Inductive Hypothesis: \( T(n-1) = 2^{n-1} - 1 \)

Inductive Step:
\[
T(n) = 2 \, T(n-1) + 1 \\
T(n) = 2 \, (2^{n-1} - 1) + 1 \\
= 2^n - 1
\]

„Guess and Check, then proof by induction“
Try plugging in small values of \( n \) and guessing a pattern. Confirm your guess with a proof by induction.
Example: Towers of Hanoi (Time Analysis 2)

\[ T(n) = 2T(n-1) + 1 \]
\[ = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \]
\[ = 4(2T(n-3) + 1) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 \]
\[ \vdots \]
\[ = 2^k T(n-k) + (2^k - 1) \]
\[ \vdots \]
\[ = 2^{n-1} T(1) + (2^{n-1} - 1) \]
\[ = 2^n - 1 \text{ since } T(1) = 1 \]

„Unraveling the recurrence“
Start with the general recurrence, and keep replacing to get the formula in terms of smaller input values. Keep unraveling until you reach the base case.
Example 3: Merging sorted arrays
Merging sorted arrays

In the merge problem, we are given two sorted arrays $A[1..n]$ and $B[1..m]$ and want to produce a sorted array containing the union of both lists. While this is interesting in its own right, it will also be a key sub-procedure in the recursive sorting algorithm MergeSort.

\[
A = \begin{bmatrix} 2 & 7 & 9 & 11 \end{bmatrix} \quad B = \begin{bmatrix} 6 & 12 & 13 \end{bmatrix}
\]

\[
C = \begin{bmatrix} 2 & 6 & 7 & 9 & 11 & 12 & 13 \end{bmatrix}
\]

We will present the merge algorithm first as an iterative algorithm and then show how to describe the same algorithm recursively.
Iterative Merge

\[ IMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \]

1. \[ n \leftarrow k + \ell \]
2. Initialize array \( C[1, \ldots, n] \)
3. \( i \leftarrow 1, j \leftarrow 1 \)
4. FOR \( t = 1 \) TO \( n \) DO:
   5. IF \( i > k \) THEN \( C[t] \leftarrow B[j], j++ \)
6. IF \( j > \ell \) THEN \( C[t] \leftarrow A[i], i++ \)
7. IF \( A[i] \leq B[j] \) THEN \( C[t] \leftarrow A[i], i++ \)
8. ELSE \( C[t] \leftarrow B[j], j++ \)
9. Return \( C[1, \ldots, n] \).
Iterative Merge: Correctness

IMALmerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays})

1. \( n \leftarrow k + \ell \)
2. Initialize array \( C[1, \ldots, n] \)
3. \( i \leftarrow 1, j \leftarrow 1 \)
4. FOR \( t = 1 \) TO \( n \) DO:
5. \( \quad \text{IF} \ i > k \ \text{THEN} \ C[t] \leftarrow B[j], j++ \)
6. \( \quad \text{IF} \ j > \ell \ \text{THEN} \ C[t] \leftarrow A[i], i++ \)
7. \( \quad \text{IF} \ A[i] \leq B[j] \ \text{THEN} \ C[t] \leftarrow A[i], i++ \)
8. \( \quad \text{ELSE} \ C[t] \leftarrow B[j], j++ \)
9. Return \( C[1, \ldots, n] \).

Loop invariant: After \( t \) iterations, \( C[1, \ldots, t] \) are the \( t \) smallest elements of the union, they are sorted, and they contain all elements in \( A[1, \ldots, i - 1] \) and \( B[1, \ldots, j - 1] \).

(Left as an Exercise)
Iterative Merge: Time analysis

\[ IMerge(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \]

1. \( n \gets k + \ell \)
2. Initialize array \( C[1, \ldots, n] \)
3. \( i \gets 1, j \gets 1 \)
4. FOR \( t = 1 \) TO \( n \) DO:
   5. IF \( i > k \) THEN \( C[t] \gets B[j], j++ \)
   6. IF \( j > \ell \) THEN \( C[t] \gets A[i], i++ \)
   7. IF \( A[i] \leq B[j] \) THEN \( C[t] \gets A[i], i++ \)
   8. ELSE \( C[t] \gets B[j], j++ \)
9. Return \( C[1, \ldots, n] \).

Lines 5-8: \( O(1) \)     Inside loop in line 4: \( O(n) \)

Lines 1-3, 9: \( O(1) \)     Total: \( O(1 + n + 1) = O(n) \)
Recursive Merge

Definition
Let \( v \circ C[1, \ldots, m] \) denote an array of length \( m + 1 \) whose first element is \( v \) and the rest is \( C[1, \ldots, m] \).

\[
R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell] \colon \text{sorted arrays})
\]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)
Recursive Merge: Correctness

\[
R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]): \text{sorted arrays}
\]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
   \( A[1] \circ R\text{Merge}(A[2, \ldots, k], B[1, \ldots, \ell]) \)
4. ELSE return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

We want to show that \( R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]) \) is a sorted array containing all elements from either array. We'll prove this by induction on \( n = k + \ell \), the total input size. (left as an Exercise)
Recursive Merge: Time analysis

\[ R\text{Merge}(A[1, \ldots, k], B[1, \ldots, \ell]: \text{sorted arrays}) \]

1. IF \( k = 0 \) return \( B[1, \ldots, \ell] \)
2. IF \( \ell = 0 \) return \( A[1, \ldots, k] \)
4. ELSE return \( B[1] \circ R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell]) \)

Every step is constant time, except that we make one recursive call in either line 3 or line 4. Thus,

\[ T(1) = c \text{ for some constant } c \]
\[ T(n) = T(n - 1) + c' \text{ for some constant } c'. \]

This is of the same form as the same recurrence for base 2 exponentiation, so we already know \( T(n) \in O(n) \).
Example 4: Binary Strings avoiding 00
Example: Binary strings avoiding 00

How many binary strings of length $n$ are there with no two consecutive 0's?

Any such binary string looks like $1\ldots$ OR $01\ldots$

What goes in the blanks? A binary string of shorter length that also avoids 00. Let $B(n)$ be the number of such length $n$ strings.

Recurrence: $B(n)=???
Example: Binary strings avoiding 00

How many binary strings of length $n$ are there with no two consecutive 0's?

Any such binary string looks like $1\________$ OR $01\________$

What goes in the blanks? A binary string of shorter length that also avoids 00. Let $B(n)$ be the number of such length $n$ strings.

Recurrence: $B(n) = B(n-1) + B(n-2)$

To start off this recurrence, we must know two base cases:

$B(0) = 1$ (the empty binary string)
$B(1) = 2$ (the string 1 and the string 0)
Example: Binary strings avoiding 00

Recurrence: \( B(0) = 1 \)
\( B(1) = 2 \)
\( B(n) = B(n-1) + B(n-2) \)

\[
\begin{array}{|c|c|}
\hline
n & B(n) \\
\hline
0 & 1 \\
1 & 2 \\
2 & 3 \\
3 & 5 \\
4 & 8 \\
5 & 13 \\
6 & 21 \\
7 & 34 \\
\hline
\end{array}
\]

Fibonacci Numbers
(Bernoulli, Euler)

\[
s = \frac{1 + \sqrt{5}}{2}, \quad t = \frac{1 - \sqrt{5}}{2}
\]

\[
B(n) = \frac{s^{n+2} - t^{n+2}}{\sqrt{5}}
\]

Exercise:
1. Formulate the algorithm to compute \( B(n) \).

2. Prove the correctness by induction.
Example: Binary strings avoiding 00

Recurrence:  
\[ B(0) = 1 \]
\[ B(1) = 2 \]
\[ B(n) = B(n-1) + B(n-2) \]

\[ s = \frac{1 + \sqrt{5}}{2}, \quad t = \frac{1 - \sqrt{5}}{2} \]
\[ B(n) = \frac{s^{n+2} - t^{n+2}}{\sqrt{5}} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>( B(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>34</td>
</tr>
</tbody>
</table>

Fibonacci Numbers
(Bernoulli, Euler)

Exercise:
1. „Guess and Check, then Induction“.
2. Unraveling the recurrence.

(left as an Exercise)
Example 5: Ternary Strings avoiding a 2 after a 0
Ternary Strings

A ternary string is like a binary string except it uses three symbols, 0, 1, and 2. For example,

12210021

is a ternary string of length 8.

Let \( T(n) \) be the number of ternary strings of length \( n \) with the property that there is never a 2 appearing anywhere after a 0. For example,

12120110

has this property, but

10120012

does not. Write a recurrence for \( T(n) \).
| Content |
|-----------------|-----------------|
| **1 Understanding and analyzing algorithms** | **1.1 Iterative algorithms (Minsort, Bubble Sort, Insertion Sort, Linear Search, Binary Search):**
|  | Correctness: Loop invariant, Induction |
|  | Time analysis: Number of comparisons, worst-, best-, average case |
|  | **1.1.2 Time complexity:** Time as a function of the input size n, Big O, o, Big Ω, ω, Big Θ |
|  | **1.1.3 Time analysis (Upper and Lower Bounds):** Minsort, Summing Triple, Intersection |
|  | **1.1.4 Recursive algorithms (Exponentiation, Towers of Hanoi, Merging sorted arrays, Binary strings avoiding 00, Ternary strings avoiding a 2 after a 0):**
|  | Correctness: Induction |
|  | Time analysis: Guess and Check then Induction, Unraveling the Recurrence, Master theorem |
| **1.5 Divide-and-Conquer (MergeSort, Multiplication of two n-digit numbers)** | **1.6 Worst-case analysis of algorithms:** Scheduling algorithms, Graham's results |
| **2 Using graphs and graph algorithms** | **2.1** | **2.2** |
| **3 Using combinatorial reasoning and probability to quantitatively analyze algorithms and systems** | **3.1** | **3.2** |
Divide-And-Conquer
Divide-and-conquer is a form of recursive strategy for designing algorithms.

1. **Divide** an instance into several smaller instances of the same problem
2. **Recursively solve** each smaller instance.
3. **Conquer** by combining the solutions into the solution for the original instance.

Because Divide-And-Conquer creates at least two subproblems, a Divide-And-Conquer algorithm makes multiple recursive calls.
Example 1: MergeSort
Divide-And-Conquer: MergeSort

Divide-And-Conquer:

11, 9, 7, 2, 13, 12, 6

11, 9, 7, 2

11, 9

11

9

divide

divide

divide

divide

9, 11

2, 7

combine

combine

combine

combine

2, 7, 9, 11

2, 6, 7, 9, 11, 12, 13

6, 12, 13
MergeSort

MergeSort(A[1, ... , n])

1. IF n = 1 Return A
2. B[1, ... , n/2] ← MergeSort(A[1, ... , n/2])
3. C[1, ... , n/2] ← MergeSort(A[n/2 + 1, ... , n])
4. Return Merge(B[1, ... , n/2], C[1, ... , n/2])
For **MergeSort** to be correct, it should return a sorted array, and that array should contain exactly the elements A[1],…,A[n].

**Prove (strong induction on n).**

(Remember:)

- In strong induction, you assume that the statement you want to show holds for all integers n’ with \( k < n' \leq n \).
- Then you must show that under this inductive hypothesis your statement is also true for n.
- We use strong induction whenever a recursive algorithm acting on an input of size n makes calls with inputs of size other than n-1.

(lei as an Exercise)
MergeSort: Time analysis

MergeSort(A[1, ..., n])

1. IF \( n = 1 \) Return A
2. \( B[1, ..., n/2] \leftarrow \text{MergeSort}(A[1, ..., n/2]) \)
3. \( C[1, ..., n/2] \leftarrow \text{MergeSort}(A[n/2 + 1, ..., n]) \)
4. Return \( \text{Merge}(B[1, ..., n/2], C[1, ..., n/2]) \)

\[
T(1) = c' \\
T(n) = 2 \cdot T\left( \frac{n}{2} \right) + cn \\
= 2^2 \cdot T\left( \frac{n}{2^2} \right) + 2 \cdot cn \\
= \ldots \\
= 2^k \cdot T\left( \frac{n}{2^k} \right) + k \cdot cn \\
= n \cdot T(1) + \log n \cdot c \cdot n \\
\in O(n \log n)
\]

\[
\frac{n}{2^k} = 1 \\
n = 2^k \\
k = \log_2 n
\]
Example 2: Multiplication of two $n$-digit numbers
School Multiplication

467 322 • 319 269

1401966
0467322
4205898
0934644
2803932
4205898

149201427618

Time analysis

Here we have six 6-digit multiplications.

Time: $O(n^2)$
Multiplication: Divide-And-Conquer

\[
\begin{array}{cc}
467 & 322 \\
\cdot & 319 \ 269 \\
\end{array}
\]

\[
= (467 \cdot 10^3 + 322) \cdot (319 \cdot 10^3 + 269) \\
= (467 \cdot 319) \ 10^6 \\
+ (467 \cdot 269) \ 10^3 + (322 \cdot 319) \ 10^3 \\
+ (322 \cdot 269)
\]

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.
Multiplication: Divide-And-Conquer

\[
\begin{align*}
467 &\, 322 \quad \cdot \quad 319 &\, 269 \\
= (467 \cdot 10^3 + 322) \cdot (319 \cdot 10^3 + 269) \\
= (467 \cdot 319) \cdot 10^6 \\
+ (467 \cdot 269) \cdot 10^3 + (322 \cdot 319) \cdot 10^3 \\
+ (322 \cdot 269)
\end{align*}
\]

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

\[
\begin{align*}
T(1) &= c' \\
T(n) &= 4 \cdot T\left(\frac{n}{2}\right) + c \cdot n
\end{align*}
\]

\[
\frac{n}{2^k} = 1 \\
n = 2^k \\
k = \log_2 n
\]

\[
\begin{align*}
4^{\log_2 n} \\
= n^{\log_2 4} \\
= n^2
\end{align*}
\]
### Multiplication: Divide-And-Conquer

\[
\begin{align*}
467 & \quad \cdot \quad 319 \\
322 & \quad \cdot \quad 269
\end{align*}
\]

\[
= (467 \cdot 10^3 + 322) \cdot (319 \cdot 10^3 + 269) \\
= (467 \cdot 319) \cdot 10^6 \\
+ (467 \cdot 269) \cdot 10^3 \\
+ (322 \cdot 319) \cdot 10^3 \\
+ (322 \cdot 269)
\]

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

**Time analysis**

- \(T(1) = c'
- T(n) = 4 \cdot T\left(\frac{n}{2}\right) + c \cdot n
- = 4^2 \cdot T\left(\frac{n}{2^2}\right) + (1 + 2) \cdot c \cdot n
- = \ldots
- = 4^k \cdot T\left(\frac{n}{2^k}\right) + (2^k - 1) \cdot c \cdot n
- = n^2 \cdot T(1) + (n - 1) \cdot c \cdot n
- \in O(n^2)

\[
\frac{n}{2^k} = 1 \\
n = 2^k \\
k = \log_2 n
\]

\[
4^{\log_2 n} \\
= n^{\log_2 4} \\
= n^2
\]

**Time:** \(O(n^2)\)
**Time analysis**

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

Karatsuba’s idea:
Compute the same with only three 3-digit multiplications.
*But how could that work?*

\[
\begin{align*}
467 \ 322 & \cdot 319 \ 269 \\
= (467 \cdot 10^3 + 322) \cdot (319 \cdot 10^3 + 269) \\
= (467 \cdot 319) \ 10^6 \\
+ (467 \cdot 269) \ 10^3 + (322 \cdot 319) \ 10^3 \\
+ (322 \cdot 269)
\end{align*}
\]
Karatsuba’s idea: Compute the same with only three 3-digit multiplications. 
But how could that work?
Multiplication: Karatsuba

\[
\begin{array}{c}
467 \times 322 \\
467 \times 269 + 322 \times 219 \\
322 \times 269
\end{array}
\quad \begin{array}{c}
319 \times 269 \\
(467-322)(269-319) + (467\times 319 + 322 \times 269) \\
322 \times 269
\end{array}
\]

### Karatsuba’s idea:
Compute the same with only three 3-digit multiplications.

*But how could that work?*
## Multiplication: Karatsuba

**467 322** \( \cdot \) **319 269**

\[
\begin{array}{l}
467 \times 319 \\
467 \times 269 + 322 \times 219 \\
322 \times 269
\end{array}
\]

\[
\begin{array}{l}
467 \times 319 \\
(467-322) \times (269-319) + (467 \times 319 + 322 \times 269) \\
322 \times 269
\end{array}
\]

\[
T(1) = c \\
T(n) = 3 \cdot T\left(\frac{n}{2}\right) + cn \\
= 3^2 \cdot T\left(\frac{n}{2^2}\right) + \frac{3}{2} \cdot cn \\
= \ldots \\
= 3^k \cdot T\left(\frac{n}{2^k}\right) + 2c \cdot n \cdot \left((\frac{3}{2})^k - 1\right) \\
= n^{\log_2 3} \cdot T(1) + 2c \cdot n \cdot \left((\frac{3}{2})^{\log_2 n} - 1\right) \\
= n^{\log_2 3} \cdot (3c) - 2c \cdot n \\
\in O(n^{\log_2 3}) = O(n^{1.58...})
\]

\[
\begin{array}{l}
\frac{n}{2^k} = 1 \\
n = 2^k \\
k = \log_2 n \\
\frac{3^{\log_2 n}}{3^{\log_2 3}} = n^{\log_2 3}
\end{array}
\]

**Time:** \( O(n^{1.58}) \)
## Content

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Understanding and analyzing algorithms</td>
<td>1.1 Iterative algorithms (Minsort, Bubble Sort, Insertion Sort, Linear Search, Binary Search): Correctness: Loop invariant, Induction Time analysis: Number of comparisons, worst-, best-, average case 1.2 Time complexity: Time as a function of the input size n, Big O, o, Big Ω, ω, Big Θ 1.3 Time analysis (Upper and Lower Bounds): Minsort, Summing Triple, Intersection 1.4 Recursive algorithms (Exponentiation, Towers of Hanoi, Merging sorted arrays, Binary strings avoiding00, Ternary strings avoiding a 2 after a 0): Correctness: Induction Time analysis: Guess and Check then Induction, Unraveling the Recurrence, Master theorem 1.5 Divide-and-Conquer (MergeSort, Multiplication of two n-digit numbers) 1.6 Worst-case analysis of algorithms: Scheduling algorithms, Graham’s results</td>
</tr>
<tr>
<td>2</td>
<td>Using graphs and graph algorithms</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Using combinatorial reasoning and probability to quantitatively analyze algorithms and systems</td>
<td></td>
</tr>
</tbody>
</table>

### Chapter 1: Analyzing Algorithms

#### 1.1 Iterative algorithms
- Correctness: Loop invariant, Induction
- Time analysis: Number of comparisons, worst-, best-, average case

#### 1.2 Time Complexity
- Time as a function of the input size $n$, Big O, o, Big Ω, ω, Big Θ

#### 1.3 Time Analysis
- Upper and Lower Bounds: Minsort, Summing Triple, Intersection

#### 1.4 Recursive Algorithms
- Exponentiation, Towers of Hanoi, Merging sorted arrays, Binary strings avoiding 00, Ternary strings avoiding a 2 after a 0:
  - Correctness: Induction
  - Time analysis: Guess and Check then Induction, Unraveling the Recurrence, Master theorem

#### 1.5 Divide-and-Conquer
- MergeSort, Multiplication of two n-digit numbers

#### 1.6 Worst-case Analysis of Algorithms
- Scheduling algorithms, Graham’s results
Scheduling

Objectives
- preemptive
- offline
- online

minimizing the makespan

Jobs

Scheduling

Processors
- parallel, identical processors

Parallel

\[ J_1 \]

\[ J_1 \]

\[ J_2 + J_3 \]

\[ p_1 \]

\[ p_2 \]

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad t \]
Scheduling models: 3-fields-notation

**Processor models:**

1. **Single Processor Scheduling**
2. **Parallel Processor Scheduling** (parallel, identical processors)
3. **Flow Shop Scheduling**
4. **Job Shop Scheduling**

**Jobs:**

- **pmtn** preemptive (a job can be interrupted at any time on any processor and resumed later on the same or any other processor without any additional (setup) time costs)

**Optimization goals (here: all are minimization goals):**

- **C_{max}** makespan = maximum completion time of a job
- **\( \sum C_j \)** sum of the completion times
- **L_{max}** maximum lateness of a job
- **\( \sum U_j \)** number of delayed jobs
- **\( \sum T_j \)** sum of tardiness of all jobs
## Content

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>Algorithms</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>Assignment and Knapsack problem</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>Bin Packing</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>Scheduling</td>
</tr>
<tr>
<td></td>
<td>4.1 Basics and 3-fields-notation</td>
</tr>
<tr>
<td></td>
<td><strong>4.2 Parallel processor scheduling: Worst-case analysis of approximation algorithms, Ron Graham</strong></td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>Graph algorithms</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>Linear Programming</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>Heuristics</td>
</tr>
</tbody>
</table>
Approximation algorithms for $P||C_{\text{max}}$

List Scheduling: Take the next job from the list, schedule this job on that processor that has done the least amount of work so far (if there are more than one of such processors, take that with the smallest index) – „utilization principle“.

LPT: sort before in non-increasing order

SPT: sort before in non-decreasing order
Example

Schedule the following \( n=8 \) jobs on \( m=3 \) parallel processors.

\begin{itemize}
  \item A
  \item B
  \item C
  \item D
  \item E
  \item F
  \item G
  \item H
\end{itemize}

\begin{itemize}
  \item 3 ZE
  \item 4 ZE
  \item 7 ZE
  \item 3 ZE
  \item 3 ZE
  \item 5 ZE
  \item 9 ZE
  \item 2 ZE
\end{itemize}

In the following there are no preemptions allowed, obey the utilization principle.

a) Draw the LPT schedule as a Gantt-Chart. What is \( C_{\text{max}} \) and what is \( \sum C_j \) ?
b) Draw the SPT schedule as a Gantt-Chart. What is \( C_{\text{max}} \) and what is \( \sum C_j \) ?
c) Find the list that leads to the schedule with minimum makespan. Draw the Gantt-Chart.
Example

a) Draw the LPT schedule as a Gantt-Chart. What is $C_{max}$ and what is $\sum C_j$?

![Gantt-Chart]

$P_1$

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]

$P_2$

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]

$P_3$

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\end{array}
\]
Solution

a) Draw the LPT schedule as a Gantt-Chart. What is $C_{\text{max}}$ and what is $\Sigma C_j$?

![Gantt-Charts for tasks A to H]

Lower Bound: $\max\{9, 36/3\} = 12 \rightarrow \text{LPT is optimal.}$

$\Sigma C_j = 76$
b) Draw the SPT schedule as a Gantt-Chart. What is $C_{max}$ and what is $\Sigma C_j$?
b) Draw the SPT schedule as a Gantt-Chart. What is $C_{max}$ and what is $\sum C_j$?

\[
\begin{align*}
\sum C_j &= 19 + 26 + 11 = 56 \\
\end{align*}
\]
c) Find the list that leads to the schedule with minimum makespan. Draw the Gantt-Chart.
c) Find the list that leads to the schedule with minimum makespan. Draw the Gantt-Chart.

Lower Bound: \( \max\{9, \frac{36}{3}\} = 12 \)

The LPT schedule is the optimal schedule in this specific case.

List Scheduling for P||Cmax

List Scheduling: Take the next job from the list, schedule this job on that processor that has done the least amount of work so far (if there are more than one of such processors, take that with the smallest index).

Schedule I

Schedule II

List Scheduling: Take the next job from the list, schedule this job on that processor that has done the least amount of work so far (if there are more than one of such processors, take that with the smallest index).
List Scheduling for P||Cmax

**List Scheduling**: Take the next job from the list, schedule this job on that processor that has done the least amount of work so far (if there are more than one of such processors, take that with the smallest index).

A bad **List-Schedule** can roughly be up to 2 times longer than an optimal schedule.

Worst-case analysis of List Scheduling


Theorem: $C_{max}^{LS} \leq \left(2 - \frac{1}{m}\right) C_{max}^*$
**LPT for P||C_{max}**

**LPT:** Longest Processing Time first (d.h. sort the jobs from large to small – largest first, smallest last. If two jobs are equal schedule that with the smaller index first.)

**Table:**

<table>
<thead>
<tr>
<th>Schedule I</th>
<th>Schedule II</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ P_1 ]</td>
<td>[ P_1 ]</td>
</tr>
<tr>
<td>[ P_2 ]</td>
<td>[ P_2 ]</td>
</tr>
<tr>
<td>[ P_3 ]</td>
<td>[ P_3 ]</td>
</tr>
</tbody>
</table>

**Diagram:**

- **A**, **B**, **C**
- **D**, **E**, **F**, **G**

**Legend:**

- **A**
- **B**
- **C**
- **D**
- **E**
- **F**
- **G**
A bad **LPT-Schedule** can be roughly up to \( \frac{4}{3} \) times longer than an optimal schedule.


**Theorem:** \( C_{\text{LPT}} \leq \left( \frac{4}{3} - \frac{1}{3m} \right) C_{\text{max}} \)
Ron Graham

since 1999 Professor at UCSD: CSE & Math Department

Chief Scientist: California Institute for Telecommunications and Information Technology (Calit2, Qualcomm Institute)

Check out these websites:
- http://www.math.ucsd.edu/~fan/ron/
- https://vimeo.com/136044050
- https://www.simonsfoundation.org/science_lives_video/ronald-graham/
Ron Graham

- * 31.10.1935
- 1962 PhD in Mathematics (UC Berkeley)

- Chief Scientist Bell Labs (AT&T) (37 years)

- married to Fan Chung (Professor in the Math department at UCSD) since 1983

- Euler Medal (1994)
- Steele Prize (2003)
Ron Graham

- President of the International Jugglers Association
- Show im Cirque Du Soleil
Ron Graham

What Is Graham's Number? (feat Ron Graham)

CSE 21
SS2, 2018
Prof. Dr. Oliver Braun

1 Analyzing Algorithms | 2 Graphs and graph algorithms | 3 Combinatorial reasoning and Probability
1.1 Iterative algorithms | 1.2 Time Complexity | 1.3 Running Time | 1.4 Recursive algorithms | 1.5 Divide-And-Conquer | 1.6 Worst-case analysis
Analyzing Algorithms

1. Iterative algorithms
2. Time Complexity
3. Running Time
4. Recursive algorithms
5. Divide-And-Conquer
6. Worst-case analysis