### Content

<table>
<thead>
<tr>
<th>1</th>
<th>Understanding and analyzing algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>Time complexity, time as a function of the input size n, Big O, o, Big Ω, ω, Big Θ</td>
</tr>
<tr>
<td>1.3</td>
<td>Running time analysis for algorithms (Upper and Lower Bounds)</td>
</tr>
<tr>
<td>1.4</td>
<td>Recursive algorithms: Correctness (Induction), Time analysis (Guess and Check then Induction, Unraveling the Recurrence)</td>
</tr>
<tr>
<td>1.5</td>
<td>Divide-and-Conquer: MergeSort, Multiplication of two n-digit numbers</td>
</tr>
<tr>
<td>1.6</td>
<td>Worst-case analysis for scheduling algorithms: Scheduling algorithms, Worst-case examples of scheduling heuristics, Graham's results</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>Using graphs and graph algorithms</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>Using combinatorial reasoning to quantitatively analyze algorithms and systems</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Using probability to algorithm analysis</th>
</tr>
</thead>
</table>
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
Weak induction

**Lemma:** The sum of all integers from 1 to n is equal to \( n(n+1)/2 \).

**Proof (by weak induction).**

**Basis step:**

\( n = 1 \) is true (as the sum of the first \( n = 1 \) numbers is 1).

**Inductive step:** (If the Lemma is true for \( n-1 \), then it is also true for \( n \))

Using the inductive hypothesis that the sum of all integers from 1 to \( n-1 \) is equal to \( (n-1)n/2 \), we can conclude:

\[ (n-1)n/2 + n = (n^2 + n)/2 = n(n+1)/2. \]

This completes the inductive step.
Strong induction

**Lemma:** Every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

**Proof (by strong induction).**

**Basis step:** Postage of 12 cents (P(12)) can be formed using three 4-cent stamps. P(13) using one 5-cent and two 4-cent stamps, P(14) using two 5-cent and one 4-cent stamps, P(15) using three 5-cent stamps.

**Inductive step:** (If the Lemma is true for n-3, then it is also true for n+1) We use the inductive hypothesis that P(n) is true for any 12 ≤ j ≤ n, where n is an integer with n ≥ 15. Using the induction hypothesis, we can assume that P(n-3) is true because n-3≥12 (we can form postage of n-3 cents using just 4-cent and 5-cent stamps).

To form postage of n+1 cents, we need only add another 4-cent stamp to the stamps we used to form postage of n-3 cents.

This completes the inductive step.
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.

- Basis step: $k=1$ 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 $n-3$ or $n/2$ or $n/4$, we must use strong induction.

- Basis step: $k=1$ 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
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 `ExpBase2`.

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

$M(1)=0$
$M(2)=1$
$M(3)=2$
$M(4)=3$

...  
$M(n)=???
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 \( \text{ExpBase2} \).

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

\[
M(1)=0 \quad \text{Lemma: ExpBase2 needs } n-1 \text{ multiplications to compute } 2^n.
M(2)=1 \quad \text{Proof (by induction)}.
M(3)=2 \quad n=1: 0 \text{ multiplications to compute } 2^1=2.
M(4)=3 \quad n-1 \rightarrow n: \text{Inductive hypothesis: } n-2 \text{ multiplications to compute } 2^{n-1}.
\ldots
M(n)=n-1 \quad \text{There is one more multiplication with 2 to compute } 2^{n-1} \times 2 = 2^n, \text{ in total we have then } n-1 \text{ multiplications to compute } 2^n.
This completes the inductive step.
Example: Exponentiation (Time analysis)

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

```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 2: “Unraveling the recurrence”**

\[
\begin{align*}
M(1) & = 0 \\
M(n) & = M(n-1) + 1 \\
& = M(n-2) + 2 \\
& = \ldots \\
& = M(n-k) + k \\
& = \ldots \\
& = M(1) + n-1 = n-1
\end{align*}
\]
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.”
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 \( n \)

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

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

Inductive Step:

\[
\begin{align*}
T(n) &= 2T(n-1) + 1 \\
T(n) &= 2(2^{n-1} - 1) + 1 \\
&= 2^n - 1
\end{align*}
\]

"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

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

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 \(C[t] \leftarrow B[j], j++\)
5. 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

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

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 \( C[1, \ldots, n] \).

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 $v \odot C[1, \ldots, m]$ denote an array of length $m + 1$ whose first element is $v$ and the rest is $C[1, \ldots, m]$.

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

1. IF $k = 0$ return $B[1, \ldots, \ell]$
2. IF $\ell = 0$ return $A[1, \ldots, k]$
4. ELSE return $B[1] \odot R\text{Merge}(A[1, \ldots, k], B[2, \ldots, \ell])$
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 \( A[1, \ldots, k] \)
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 $1\text{_______}$ OR $01\text{_______}$

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 $1\underline{\hspace{1cm}}$ OR $01\underline{\hspace{1cm}}$

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

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

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

<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. Formulate the algorithm to compute $B(n)$.
2. Prove the correctness by induction.
Example: Binary strings avoiding 00

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

Fibonacci Numbers
(Bernoulli, Euler)

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

<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>

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

(leave as an Exercise)
## Content

1.2 Time complexity, time as a function of the input size n, Big O, o, Big Ω, ω, Big Θ  
1.3 Running time analysis for algorithms (Upper and Lower Bounds)  
1.4 Recursive algorithms: Correctness (Induction), Time analysis (Guess and Check then Induction, Unraveling the Recurrence)  
1.5 **Divide-and-Conquer:** MergeSort, Multiplication of two n-digit numbers  
1.6 Worst-case analysis for scheduling algorithms: Scheduling algorithms, Worst-case examples of scheduling heuristics, Graham's results |
| 2  | Using graphs and graph algorithms | |
| 3  | Using combinatorial reasoning to quantitatively analyze algorithms and systems | |
| 4  | Using probability to algorithm analysis | |

---

CSE 21  
SS2, 2016  
Prof. Dr. Oliver Braun  

1 Analyzing Algorithms  | 2 Graphs and graph algorithms  | 3 Combinatorial reasoning  | 4 Probability  
1.1 What?How?Why?When?  | 1.2 Time Complexity  | 1.3 Running Time  | 1.4 Recursion  | 1.5 Divide-And-Conquer
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

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

11, 9, 7, 2

11, 9

11

9

7, 2

7

2

13, 12, 6

13, 12

13

12

2, 7

2

6, 12, 13

6

12

13

2, 7, 9, 11

2, 7

2

6, 12, 13

6

9, 11

9

11

2, 7, 9, 11

2, 7

2

6, 12, 13

6

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

2, 6

2

6

Divide-And-Conquer: MergeSort

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

11, 9, 7, 2

11, 9

11

9

7, 2

7

2

13, 12, 6

13, 12

13

12

2, 7

2

6, 12, 13

6

9, 11

9

11

2, 7, 9, 11

2, 7

2

6, 12, 13

6

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

2, 6
MergeSort

\[ \text{MergeSort}(A[1, \ldots, n]) \]

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

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

(Remember:)

- In strong induction, you assume that the statement you want to show holds for all integers $n'$ with $k \leq n' \leq n$.
- 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 $n-1$.

(Left as an Exercise)
MergeSort: Time analysis

```
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])
```

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

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

Here we have six 6-digit multiplications.

Time analysis

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

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

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

Here we have six 6-digit multiplications.

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

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

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

\[
= (467 \cdot 319) \cdot 10^6
\]

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

\[
+ (322 \cdot 269)
\]

\[
T(1) = c'
\]

\[
T(n) = 4 \cdot T \left( \frac{n}{2} \right) + c \cdot n
\]

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

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

\[
n = 2^k
\]

\[
k = \log_2 n
\]

\[
\frac{4 \cdot \log_2 n}{n \log_2 4}
\]

\[
= n^2
\]
Multiplication: Divide-And-Conquer

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

Time analysis

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

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

\[ \begin{align*}
\frac{n}{2^k} &= 1 \\
n &= 2^k \\
k &= \log_2 n \\
4^{\log_2 n} &= n^{\log_2 4} \\
&= n^2
\end{align*} \]

Time: O(n^2)
Multiplication: Karatsuba

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

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

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

But how could that work?
Multiplication: Karatsuba

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

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

But how could that work?

\[
T(1) = c' \\
T(n) = 3 \times T\left(\frac{n}{2}\right) + cn \\
= 3^2 \times T\left(\frac{n}{4}\right) + \frac{3}{2} \times cn \\
= \ldots \\
= 3^k \times T\left(\frac{n}{2^k}\right) + 3 \times \left(\frac{3}{2}\right)^k \times c \times n \\
= n^{\log_2 3} \times T(1) + 3 \times \left(\frac{3}{2}\right)^{\log_2 n} \times c \times n \\
\in O(n^{\log_2 3}) = O(n^{1.58\ldots})
\]

Time analysis:

Here we have six 6-digit multiplications.

Here we have four 3-digit multiplications.

\[
\begin{align*}
n &= 2^k \\
k &= \log_2 n \\
\frac{n}{2^k} &= 1 \\
n^{\log_2 3} &= n^{\log_2 3}
\end{align*}
\]

Time: \(O(n^{1.58})\)
| Content |
|-----------------|--------------------------------------------------|
| 1  Understanding and analyzing algorithms |
| 1.2  Time complexity, time as a function of the input size n, Big O, o, Big Ω, ω, Big Θ |
| 1.3  Running time analysis for algorithms (Upper and Lower Bounds) |
| 1.4  Recursive algorithms: Correctness (Induction), Time analysis (Guess and Check then Induction, Unraveling the Recurrence) |
| 1.5  Divide-and-Conquer |
| 1.6  Worst-case analysis for scheduling algorithms: Scheduling algorithms, Worst-case examples of scheduling heuristics, Graham’s results |

| 2  Using graphs and graph algorithms |
| 3  Using combinatorial reasoning to quantitatively analyze algorithms and systems |
| 4  Using probability to algorithm analysis |
Scheduling algorithms
Scheduling

Jobs
• preemptive
• offline
• online

Scheduling

Processors
parallel, identical processors

minimizing the makespan

Objectives

1.1 What? How? Why? When?
1.2 Time Complexity
1.3 Running Time
1.4 Recursion
1.5 Divide-And-Conquer
1.6 Worst-case analysis
### Scheduling-Models and 3-Fields-Notation

<table>
<thead>
<tr>
<th>Processors</th>
<th>Optimization Goal</th>
<th>Algorithm</th>
<th>Running Time</th>
<th>Worst-case (*=optimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (One Processor)</td>
<td>Minimize maximum Lateness ( L_{\text{max}} )</td>
<td>Earliest Due Date</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Minimize number of delayed jobs ( \sum U_j )</td>
<td>Moore</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td>( P ) (parallel, identical processors)</td>
<td>Minimize Makespan (preemptions allowed) ( C_{\text{max}} )</td>
<td>Mc Naughton</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Minimize Sum of processing times ( \sum C_j )</td>
<td>SPT</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Minimize Makespan (no preemptions allowed) ( C_{\text{max}} )</td>
<td>n! permutations</td>
<td>exponential</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>List Scheduling</td>
<td>polynomial</td>
<td>2 - 1/( m ) ( \text{(Graham)} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LPT</td>
<td>polynomial</td>
<td>4/3 - 1/(3( m )) ( \text{(Graham)} )</td>
</tr>
<tr>
<td>( F ) (Flow-Shop)</td>
<td>Minimize Makespan (with 2 processors) ( C_{\text{max}} )</td>
<td>Johnson</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Minimize Makespan (( \geq 3 ) processors) ( C_{\text{max}} )</td>
<td>n! permutations</td>
<td>exponential</td>
<td>*</td>
</tr>
<tr>
<td>( J ) (Job-Shop)</td>
<td>Minimize Makespan (with 2 processors and max. 2 stations) ( C_{\text{max}} )</td>
<td>Jackson</td>
<td>polynomial</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Minimize Makespan (( \geq 3 ) processors) ( C_{\text{max}} )</td>
<td>n! permutations</td>
<td>exponential</td>
<td>*</td>
</tr>
</tbody>
</table>
Heuristics 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).

**List Scheduling**

- **A**
- **B**
- **C**
- **D**
- **E**

$p_A = 2$

$p_B = 2$

$p_C = 1$

$p_D = 1$

$p_E = 3$

**Example Diagrams**

- **LPT**
- **SPT**

**List Scheduling**

- **$P_1$**
- **$P_2$**
- **$P_3$**

**LPT**

- **$P_1$**
- **$P_2$**
- **$P_3$**

**SPT**

- **$P_1$**
- **$P_2$**
- **$P_3$**
C\(_{\text{max}}^p\) := \(\max\left\{ \frac{\sum p_j}{m}, p_{\text{max}} \right\}\)

- job processing of the jobs can be interrupted
- continue to work on the jobs after an interruption is possible
  - ... on any processor (including the same)
  - ... at any time
  - ... without any time penalty (like setup costs)
Exercise

The following \( n=8 \) jobs have to be scheduled on \( m=3 \) parallel and identical processors with the objective of minimizing the makespan.

\[
\begin{array}{cccc}
A & \text{3 TU} & E & \text{5 TU} \\
B & \text{4 TU} & F & \text{3 TU} \\
C & \text{7 TU} & G & \text{9 TU} \\
D & \text{3 TU} & H & \text{2 TU}
\end{array}
\]

a) Solve the problem with McNaughton's algorithm. How much is the makespan \( C_{max} \) and how much is \( \Sigma C_j \)? Analyze the number of preemptions.

b) Draw the LPT-schedule as a Gantt-Chart. How much is \( C_{max} \) and \( \Sigma C_j \)?

c) Draw the SPT-schedule as a Gantt-Chart. How much is \( C_{max} \) and \( \Sigma C_j \)?

d) Find the optimal and the worst list concerning the objective of minimizing the makespan.

e) Find the optimal and the worst list concerning the objective of minimizing the sum of the processing times.
Worst-case examples of scheduling heuristics
Worst-case analysis of List Scheduling 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).
List Scheduling for P\|C_{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).

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

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.)

Schedule I

Schedule II

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.)
LPT for P\|C_{\text{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.)

A bad LPT-Schedule can be roughly up to $\frac{4}{3}$ times longer than an optimal schedule.

Ron Graham and his results
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*

---

**Magical Mathematics**

*The Mathematical Ideas That Animate Great Magic Tricks*

**Persi Diaconis and Ron Graham**

Foreword by Martin Gardner

---

1.1 What? How? Why? When? | 1.2 Time Complexity | 1.3 Running Time | 1.4 Recursion | 1.5 Divide-And-Conquer | 1.6 **Worst-case analysis**
Ron Graham

What is Graham's Number? (feat. Ron Graham)
Worst-case analysis of List Scheduling


**Theorem:** \( C_{max}^{LS} \leq \left( 2 - \frac{1}{m} \right) C_{max}^* \)
Worst-case analysis of LPT


\[ \text{Theorem: } C^{LPT}_{\text{max}} \leq \left( \frac{4}{3} - \frac{1}{3m} \right) C^*_{\text{max}} \]