# Examples of an Analysis of Complexity of Algorithms

# Complexity of Algorithms

Some typical values of the size of an input $n$, for which an algorithm with the given time complexity usually computes the output on a "common PC" within a fraction of a second or at most in seconds.

(Of course, this depends on particular details. Moreover, it is assumed here that no big constants are hidden in the asymptotic notation)

| $O(n)$ | $O(n \log n)$ | $O(n^2)$ | $O(n^3)$ |
|---|---|---|---|
| $1\,000\,000 - 100\,000\,000$ | $100\,000 - 1\,000\,000$ | $1000 - 10\,000$ | $100 - 1000$ |

| $2^{O(n)}$ | $O(n!)$ |
|---|---|
| $20 - 30$ | $10 - 15$ |

# Complexity of Algorithms

When we use asymptotic estimations of the complexity of algorithms, we should be aware of some issues:

- Asymptotic estimations describe only how the running time grows with the growing size of input instance.
- They do not say anything about exact running time. Some big constants can be hidden in the asymptotic notation.
- An algorithm with better asymptotic complexity than some other algorithm can be in reality faster only for very big inputs.
- We usually analyze the time complexity in the worst case. For some algorithms, the running time in the worst case can be much higher than the running time on "typical" instances.

# Complexity of Algorithms

- This can be illustrated on algorithms for sorting.

| Algorithm | Worst-case | Average-case |
|---|---|---|
| Bubblesort | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Heapsort | $\Theta(n \log n)$ | $\Theta(n \log n)$ |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \log n)$ |

- Quicksort has a worse asymptotic complexity in the worst case than Heapsort and the same asymptotic complexity in an average case but it is usually faster in practice.

# Complexity of Algorithms

**Polynomial** — an expression of the form

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

where $a_0, a_1, \ldots, a_k$ are constants.

Examples of polynomials:

$$4n^3 - 2n^2 + 8n + 13 \qquad 2n + 1 \qquad n^{100}$$

Function $f$ is called **polynomial** if it is bounded from above by some polynomial, i.e., if there exists a constant $k$ such that $f \in O(n^k)$.

For example, the functions belonging to the following classes are polynomial:

$$O(n) \qquad O(n \log n) \qquad O(n^2) \qquad O(n^5) \qquad O(\sqrt{n}) \qquad O(n^{100})$$

# Complexity of Algorithms

Function such as $2^n$ or $n!$ are not polynomial — for arbitrarily big constant $k$ we have

$$2^n \in \Omega(n^k) \qquad\qquad n! \in \Omega(n^k)$$

**Polynomial algorithm** — an algorithm whose time complexity is polynomial (i.e., bounded from above by some polynomial)

Roughly we can say that:

- polynomial algorithms are effiecient algorithms that can be used in practice for inputs of considerable size
- algorithms, which are not polynomial, can be used in practice only for rather small inputs

# Complexity of Algorithms

The division of algorithms on polynomial and non-polynomial is very rough — we cannot claim that polynomial algorithms are always efficient and non-polynomial algorithms are not:

- an algorithm with the time complexity $\Theta(n^{100})$ is probably not very useful in practice,
- some algorithms, which are non-polynomial, can still work very efficiently for majority of inputs, and can have a time complexity bigger than polynomial only due to some problematic inputs, on which the computation takes long time.

**Remark:** Polynomial algorithms where the constant in the exponent is some big number (e.g., algorithms with complexity $\Theta(n^{100})$) almost never occur in practice as solutions of usual algorithmic problems.

# Complexity of Algorithms

For most of common algorithmic problems, one of the following three possibilities happens:

- A polynomial algorithm with time complexity $O(n^k)$ is known, where $k$ is some very small number (e.g., $5$ or more often $3$ or less).

- No polynomial algorithm is known and the best known algorithms have complexities such as $2^{\Theta(n)}$, $\Theta(n!)$, or some even bigger.

  In some cases, a proof is known that there does not exist a polynomial algorithm for the given problem (it cannot be constructed).

- No algorithm solving the given problem is known (and it is possibly proved that there does not exist such algorithm)

# Complexity of Algorithms

A typical example of polynomial algorithm — matrix multiplication with time complexity $\Theta(n^3)$ and space complexity $\Theta(n^2)$:

**Algorithm:** Matrix multiplication

```
MATRIX-MULT (A, B, C, n):
    for i := 1 to n do
        for j := 1 to n do
            x := 0
            for k := 1 to n do
                x := x + A[i][k] * B[k][j]
            C[i][j] := x
```

# Complexity of Algorithms

- For a rough estimation of complexity, it is often sufficient to count the number of nested loops — this number then gives the degree of the polynomial

  **Example:** Three nested loops in the matrix multiplication — the time complexity of the algorithm is $O(n^3)$.

- If it is not the case that all the loops go from $0$ to $n$ but the number of iterations of inner loops are different for different iterations of an outer loops, a more precise analysis can be more complicated.

  It is often the case, that the sum of some sequence (e.g., the sum of arithmetic or geometric progression) is then computed in the analysis.

  The results of such more detailed analysis often does not differ from the results of a rough analysis but in many cases the time complexity resulting from a more detailed analysis can be considerably smaller than the time complexity following from the rough analysis.

# Arithmetic Progression

**Arithmetic progression** — a sequence of numbers $a_0, a_1, \ldots, a_{n-1}$, where

$$a_i = a_0 + i \cdot d,$$

where $d$ is some constant independent on $i$.

So in an arithmetic progression, we have $a_{i+1} = a_i + d$ for each $i$.

**Example:** The arithmetic progression where $a_0 = 1$, $d = 1$, and $n = 100$:

$$1, 2, 3, 4, 5, 6, \ldots, 96, 97, 98, 99, 100$$

**The sum of an arithmetic progression**:

$$\sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1} = \frac{1}{2} n \left(a_0 + a_{n-1}\right)$$

# Arithmetic Progression

**Example:**

$$1 + 2 + \cdots + n \;=\; \frac{1}{2}n(n+1) \;=\; \frac{1}{2}n^2 + \frac{1}{2}n \;=\; \Theta(n^2)$$

For example, for $n = 100$ we have

$$1 + 2 + \cdots + 100 \;=\; 50 \cdot 101 \;=\; 5050.$$

# Arithmetic Progression

**Proof:** Let us denote

$$s = \sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1}$$

$$\begin{aligned}
2s &= s + s \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_0 + a_1 + \cdots + a_{n-1}) \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_{n-1} + a_{n-2} + \cdots + a_0) \\
&= (a_0 + a_{n-1}) + (a_1 + a_{n-2}) + \cdots + (a_{n-1} + a_0) \\
&= ((a_0 + 0 \cdot d) + (a_0 + (n-1) \cdot d)) + ((a_0 + 1 \cdot d) + (a_0 + (n-2) \cdot d)) + \\
&\quad \cdots + ((a_0 + (n-1) \cdot d) + (a_0 + 0 \cdot d)) \\
&= n \cdot (a_0 + a_0 + (n-1) \cdot d) \\
&= n \cdot (a_0 + a_{n-1})
\end{aligned}$$

# Aritmetická posloupnost

**Example:** $s = 1 + 2 + 3 + \cdots + 99 + 100$

$$\begin{aligned} 2s &= s + s \\ &= (1 + 2 + \cdots + 100) + (1 + 2 + \cdots + 100) \\ &= (1 + 2 + \cdots + 100) + (100 + 99 + \cdots + 1) \\ &= (1 + 100) + (2 + 99) + (3 + 98) + \cdots + (99 + 2) + (100 + 1) \\ &= 100 \cdot (1 + 100) = 10100 \end{aligned}$$

So

$$s = \frac{1}{2} \cdot 10100 = 5050$$

# Geometric Progression

**Geometric progression** — a sequence of numbers $a_0, a_1, \ldots, a_n$, where

$$a_i = a_0 \cdot q^i,$$

where $q$ is some constant independent on $i$.

So in a geometric progression we have $a_{i+1} = a_i \cdot q$ for each $i$.

**Example:** The geometric progression where $a_0 = 1$, $q = 2$, and $n = 14$:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

**The sum of a geometic progression** (where $q \neq 1$):

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n = a_0 \frac{q^{n+1} - 1}{q - 1}$$

# Geometric Progression

**Example:**

$$1 + q + q^2 + \cdots + q^n = \frac{q^{n+1} - 1}{q - 1}$$

In particular, for $q = 2$:

$$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2 \cdot 2^n - 1 = \Theta(2^n)$$

# Geometric Progression

**Proof:** Let us denote

$$s = \sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n$$

$$\begin{aligned}
s &= a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n \\
s \cdot q &= (a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n) \cdot q \\
&= a_0 \cdot q^1 + a_0 \cdot q^2 + \cdots + a_0 \cdot q^{n+1} \\
s \cdot q - s &= a_0 \cdot q^{n+1} - a_0 \cdot q^0 \\
s \cdot (q-1) &= a_0 \cdot (q^{n+1} - 1) \\
s &= a_0 \cdot \frac{q^{n+1} - 1}{q - 1}
\end{aligned}$$

# Complexity of Algorithms

An **exponential** function: a function of the form $c^n$, where $c$ is a constant — e.g., function $2^n$

**Logarithm** — the inverse function to an exponential function: for a given $n$,

$$\log_c n$$

is the value $x$ such that $c^x = n$.

# Complexity of Algorithms

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |
| 11 | 2048 |
| 12 | 4096 |
| 13 | 8192 |
| 14 | 16384 |
| 15 | 32768 |
| 16 | 65536 |
| 17 | 131072 |
| 18 | 262144 |
| 19 | 524288 |
| 20 | 1048576 |

| $n$ | $\lceil \log_2 n \rceil$ |
|---|---|
| 0 | — |
| 1 | 0 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 3 |
| 8 | 3 |
| 9 | 4 |
| 10 | 4 |
| 11 | 4 |
| 12 | 4 |
| 13 | 4 |
| 14 | 4 |
| 15 | 4 |
| 16 | 4 |
| 17 | 5 |
| 18 | 5 |
| 19 | 5 |
| 20 | 5 |

| $n$ | $\log_2 n$ |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 8 | 3 |
| 16 | 4 |
| 32 | 5 |
| 64 | 6 |
| 128 | 7 |
| 256 | 8 |
| 512 | 9 |
| 1024 | 10 |
| 2048 | 11 |
| 4096 | 12 |
| 8192 | 13 |
| 16384 | 14 |
| 32768 | 15 |
| 65536 | 16 |
| 131072 | 17 |
| 262144 | 18 |
| 524288 | 19 |
| 1048576 | 20 |

# Complexity of Algorithms

Examples where exponential functions and logarithms can appear in an analysis of algorithms:

- Some value is repeatedly decreased to one half or is repeatedly doubled.

  For example, in the **binary search**, the size of an interval halves in every iteration of the loop.

  Let us assume that an array has size $n$.

  What is the minimal size of an array $n$, for which the algorithm performs at least $k$ iterations?

  The answer: $2^k$

  So we have $k = \log_2(n)$. The time complexity of the algorithm is then $\Theta(\log n)$.

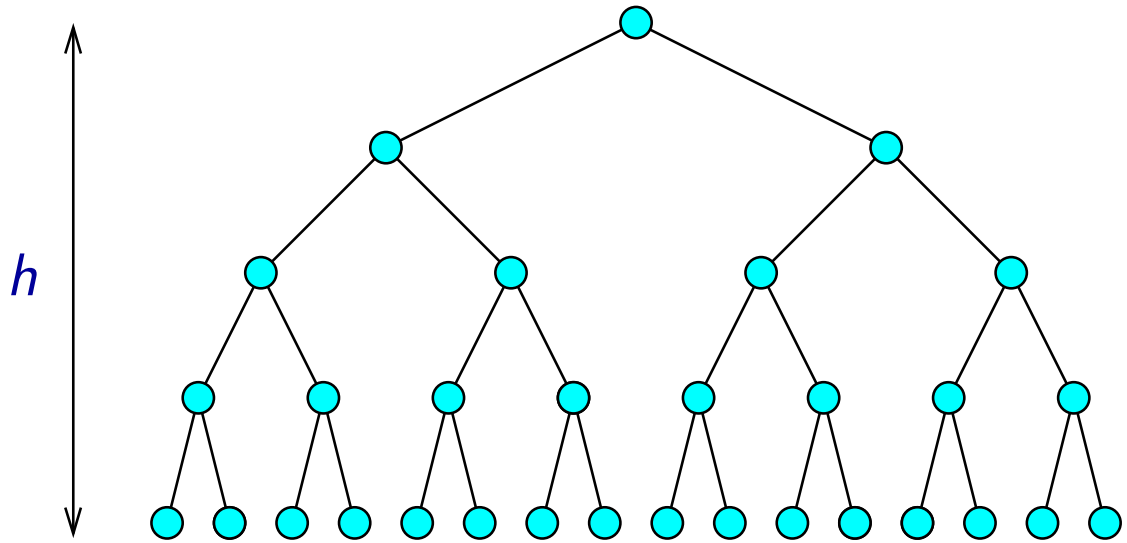
# Complexity of Algorithms

- Using $n$ bits we can represent numbers from $0$ to $2^n - 1$.
- The minimal numbers of bits, which are sufficient for representing a natural number $x$ in binary is
$$\lceil \log_2(x+1) \rceil.$$
- A perfectly balanced tree of height $h$ has $2^{h+1} - 1$ nodes, and $2^h$ of these nodes are leaves.
- The height of a perfectly balanced binary tree with $n$ nodes is $\log_2 n$.

  An illustrating example: If we would draw a balanced tree with $n = 1\,000\,000$ nodes in such a way that the distance between neighbouring nodes would be 1 cm and the height of each layer of nodes would be also 1 cm, the width of the tree would be 10 km and its height would be approximately 20 cm.

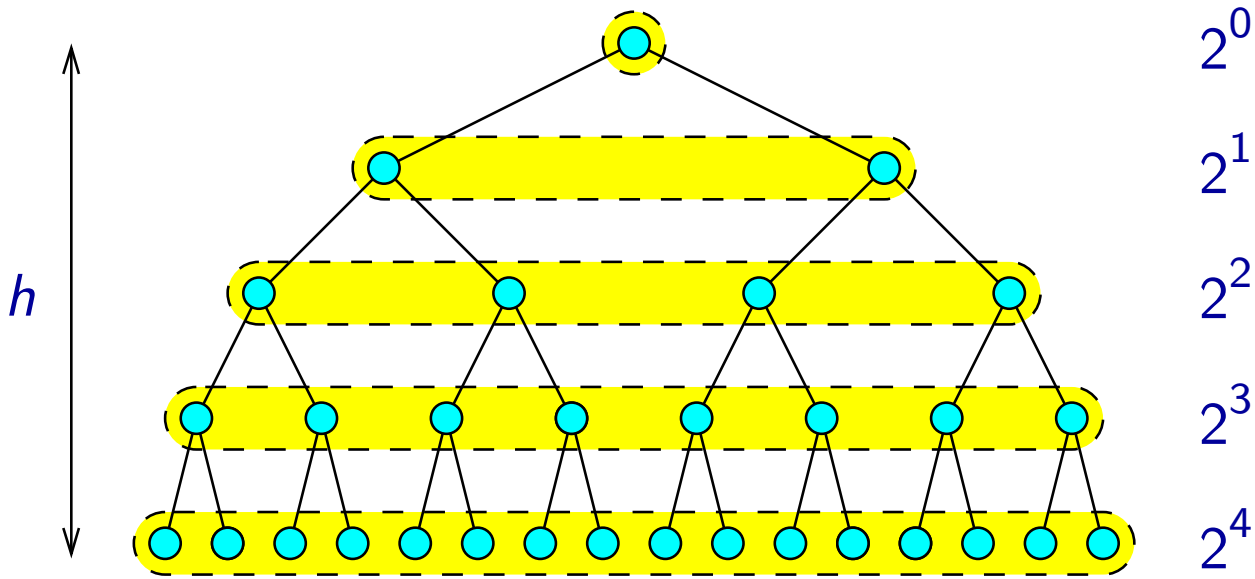
# Complexity of Algorithms

A perfectly balanced binary tree of height $h$:



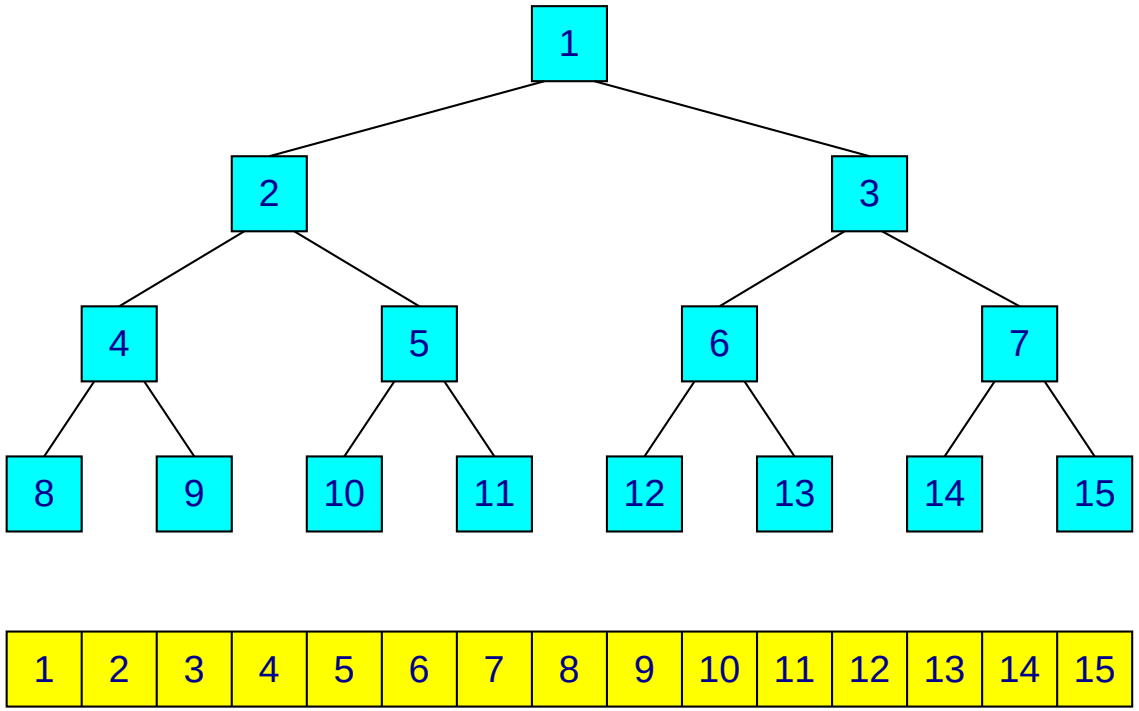
# Complexity of Algorithms

A perfectly balanced binary tree of height $h$:



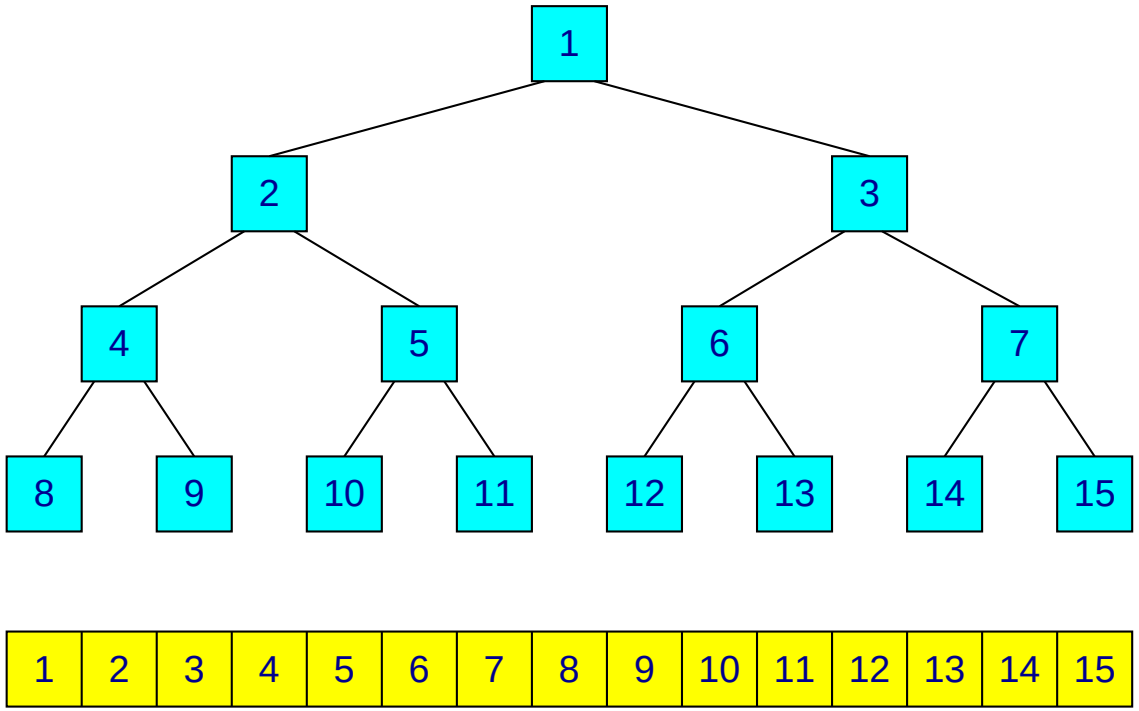
# Complexity of Algorithms

An efficient way to store a complete binary tree in an array:



# Complexity of Algorithms

An efficient way to store a complete binary tree in an array:




Children of a node with index $i$ have indexes $2i$ and $2i + 1$.
The parent of a node with index $i$ has index $\lfloor i/2 \rfloor$.

# Complexity of Algorithms

Heap — a complete binary tree stored in an array $A$ in way described on the previous slide, where moreover the following invariant holds for each $i = 1, 2, \ldots, n$:

- if $2i \leq n$ then $A[i] \leq A[2i]$
- if $2i + 1 \leq n$ then $A[i] \leq A[2i + 1]$

Examples of a usage of a heap:

- sorting algorithm **HeapSort**
- an efficient implementation of a **priority queue** — this allows to perform most operations on this queue with time complexity in $O(\log n)$ where $n$ is the number of elements currently in the queue

# Complexity of Algorithms

**Algorithm:** Construction of a heap from an unsorted array

```
CREATE-HEAP (A, n):
    i := ⌊n/2⌋
    while i ≥ 1 do
        j := i
        x := A[j]
        while 2 * j ≤ n do
            k := 2 * j
            if k + 1 ≤ n and A[k + 1] < A[k] then
                k := k + 1
            if x ≤ A[k] then break
            A[j] := A[k]
            j := k
        A[j] := x
        i := i − 1
```

# Complexity of Algorithms

**Time complexity** of CREATE-HEAP:

- By a quick and rough analysis, we can easily determine that this complexity is in $O(n \log n)$ and in $\Omega(n)$:
  - The outer cycle is executed always $\lfloor n/2 \rfloor$ times — so the number of its iterations is in $\Theta(n)$.
  - The number of iterations of the inner cycle (in one iteration of the outer cycle) is obviously in $O(\log n)$.
- It is much less obvious that the total number of iterations of the inner cycle (i.e., over all iterations of the outer cycle) is in fact in $O(n)$.

So together we obtain:

The time complexity of CREATE-HEAP is in $\Theta(n)$.

# Complexity of Algorithms

Justification that the total number of iterations of the inner cycle is in $O(n)$:

Let us assume for simplicity that all branches of the tree are of the same length and that their length is $h$ — so we have $n = 2^{h+1} - 1$.

Let $C_i$, where $0 \leq i < h$, be the total number of iterations of the inner cycle where at the beginning of the cycle the node with index $j$ is in $i$-th layer of the tree (the layers are numbered top to bottom as $0, 1, 2, \ldots$).

It is obvious that the total number of iterations $s$ is

$$s = C_{h-1} + C_{h-2} + \cdots + C_0 = \sum_{i=0}^{h-1} C_i$$

The value of $C_i$ can be computed as the total number of nodes in the layers $0, 1, \ldots, i$:

$$C_i = 2^0 + 2^1 + \cdots + 2^i = \sum_{k=0}^{i} 2^k = \frac{2^{i+1} - 1}{2 - 1} = 2^{i+1} - 1$$

# Complexity of Algorithms

The total sum then can be computed as follows:

$$s = \sum_{i=0}^{h-1} C_i = \sum_{i=0}^{h-1} (2^{i+1} - 1) = 2 \cdot \left( \sum_{i=0}^{h-1} 2^i \right) - \left( \sum_{i=0}^{h-1} 1 \right)$$

$$= 2 \cdot \frac{2^h - 1}{2 - 1} - h = 2^{h+1} - 2 - h = n - 1 - h = O(n)$$