ST0898 Levelling Course in Computation
Master in Data Sciences and Analytics

Andrés Sicard-Ramírez

Universidad EAFIT

June 2019

Introduction

Administrative Information

Textbook

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. [1983] [1985]. Data Structures and Algorithms. Reprinted with corrections. Addison-Wesley.

Other books

- Brassard, G. and Bratley, P. [1996]. Fundamentals of Algorithmics. Prentice Hall.
- Parberry, I. and Gasarch, W. [1994] [2002]. Problems on Algorithms. 2nd ed. Prentice Hall.

Convention

The references to examples, exercises, figures, quotes or theorems correspond to those in the textbook.

Examination

The exam will be on Tuesday, 2nd July.

Course Content

- Elementary algorithms
- Analysis of algorithms
- Abstract data types (lists, stacks and queues)

Preliminaries

Notation and conventions for number sets

$\mathbb{N} = \{0, 1, 2, \ldots\}$	(natural numbers)
$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$	(integers)
$\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$	(positive integers)
$\mathbb{Q} = \{\, p/q \mid p, q \in \mathbb{Z} \text{ and } q \neq 0 \,\}$	(rational numbers)
$\mathbb{R} = (-\infty, \infty)$	(real numbers)
$\mathbb{R}^{\geq 0} = [0, \infty)$	(non-negative real numbers)
$\mathbb{R}^+ = (0, \infty)$	(positive real numbers)

Preliminaries

Convention

All the logarithms are base 2.

Appendix

See in the appendix:

- Floor and ceiling functions
- Summation properties

Elementary Algorithms

From Problems to Programs

Question

Can be any problem solved by a program?

From Problems to Programs

Question

Can be any problem solved by a program?

No!

- Limitations when specifying the problem (no precise specification)
- Computation limitations (theoretical or practical)
- Ethical considerations and regulations

From Problems to Programs

Quote

‘Half the battle is knowing what problem to solve.’ (p. 1)

From Problems to Programs

Quote

‘Half the battle is knowing what problem to solve.’ (p. 1)

Steps when writing a computer program to solve a problem

- Problem formulation and specification
- Design of the solution
- Implementation
- Testing
- Documentation
- Evaluation
- Maintenance

From Problems to Programs

Quote

‘Half the battle is knowing what problem to solve.’ (p. 1)

Steps when writing a computer program to solve a problem

- Problem formulation and specification
- Design of the solution
- Implementation
- Testing
- Documentation
- Evaluation
- Maintenance

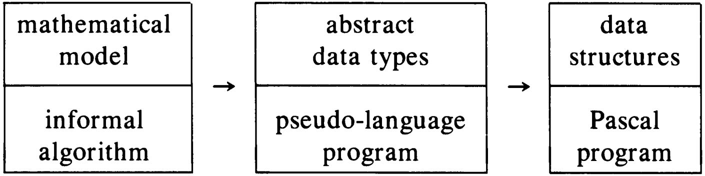
Remark

In software engineering the above steps are part of the software development life cycle.

From Problems to Programs

The problem solving process

Problem solving stages.*



---

*Figure source: Fig. 1.9.

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

Question

Are missing the computers on the above definition of algorithm?

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

Question

Are missing the computers on the above definition of algorithm? No!

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

Question

What is an instruction?

From Problems to Programs

Definition

Informally, an **algorithm** is 'a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.' (p. 2)

Question

What is an instruction?

Remark

Any informal definition of algorithm necessary will be imprecise (but the above definition is enough for our course).

From Problems to Programs

Definition

Informally, an **algorithm** is ‘a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.’ (p. 2)

Discussion

Is any computer program an algorithm?

From Problems to Programs

Correctness of an algorithm

$$\text{partial correctness} := \text{if an answer is returned it will be correct}$$

$$\text{total correctness} := \text{partial correctness} + \text{termination}$$

Programming Languages

Some paradigms of programming

- Imperative/object-oriented: Describe computation in terms of state-transforming operations such as assignment. Programming is done with statements.
- Logic: Predicate calculus as a programming language. Programming is done with sentences.
- Functional: Describe computation in terms of (mathematical) functions. Programming is done with expressions.

Examples

$$\text{Imperative/OO}\begin{cases}\text{C}\\ \text{C++}\\ \textsc{Java}\\ \textsc{Python}\end{cases}\qquad \text{Logic}\begin{cases}\text{CLP(R)}\\ \textsc{Prolog}\end{cases}\qquad \text{Functional}\begin{cases}\textsc{Erlang}\\ \textsc{Haskell}\\ \text{ML}\end{cases}$$

Programming Languages

Discussion

Does the algorithm for solving a problem depend of the programming language used for implementing it?

Pseudo-Code

We shall write algorithms using pseudo-code.

Example

See pseudo-code entry on Wikipedia.*

---

*https://en.wikipedia.org/wiki/Pseudocode.

Pseudo-Code

Conventions

Based on the conventions used in [Cormen, Leiserson, Rivest and Stein 2009, pp. 20-22]:

- Indentation indicates block (e.g. **for** loop, **while** loop, **if-else** statement) structure instead of conventional indicators such as **begin** and **end** statements.
- Assignation is denoted by the symbol ‘$:=$’.
- Basic data types (e.g. integers, reals, booleans and characters) parameters to a procedure are passed by value.
- Compound data types (e.g. arrays) and abstract data types (e.g. lists, stacks and queues) parameters to a procedure are passed by reference.
- The symbols ‘//’ and ‘$\triangleright$’ denote a commentary.

Pseudo-Code

Example

Pseudo-code for calculating the factorial of a number (recursive version).

```
$\textsc{factorialR}(n : \mathbb{N})$
if $n \le 1$
    return $1$
else
    return $n * \textsc{factorialR}(n - 1)$
```

Pseudo-Code

Example

Pseudo-code for calculating the factorial of a number (iterative version).

```
$\textsc{FactorialI}(n : \mathbb{N})$
$fac : \mathbb{Z}^+$
$fac := 1$
for $i := 1$ to $n$
    $fac := fac * i$
return $fac$
```

Pseudo-Code

Example

Pseudo-code for calculating the factorial of a number (iterative version).

```
FACTORIALI($n : \mathbb{N}$)
$fac : \mathbb{Z}^+$
$fac := 1$
for $i := 1$ to $n$
    $fac := fac * i$
return $fac$
```

Remark

Recursive algorithms versus iterative algorithms.

Pseudo-Code

Exercise

Assume the parameter n in the function below is a positive power of 2, i.e. $n = 2, 4, 8, 16, \ldots$. Give the formula that expresses the value of the variable $count$ in terms of the value of n when the function terminates (Exercise 1.17).

```
MISTERY($n : \mathbb{N}$)
$count, x : \mathbb{N}$
$count := 0$
$x := 2$
while $x < n$
    $x := 2 * x$
    $count := count + 1$
return $count$
```

Pseudo-Code

Exercise

Which is the value returned by the MYSTERY function? Hint: Keep y fixed. From [Parberry and Gasarch 2002, Exercise 257].

```
MYSTERY($y : \mathbb{R}, z : \mathbb{N}$)
    $x : \mathbb{R}$
    $x := 1$
    while $z > 0$
        if $z$ is odd
            $x := x * y$
        $z := \lfloor z/2 \rfloor$
        $y := y^2$
    return $x$
```

Pseudo-Code

Example

Brassard and Bratley [1996] describe an algorithm for multiplying two positive integers which does not use any multiplication tables. The algorithm is called multiplication *a la russe*.[*]

```
RUSSE($m : \mathbb{Z}^+, n : \mathbb{Z}^+$)
1  $result : \mathbb{N}$
2  $result := 0$
3  repeat
4          if $m$ is odd
5                  $result := result + n$
6          $m := \lfloor m/2 \rfloor$
7          $n := n + n$
8  until $m == 0$
9  return $result$
```

---

[*] In Brassard and Bratley [1996], the condition in Line 8 is $m == 1$, which is wrong.

Pseudo-Code

Exercise

Test the RUSSE algorithm on some inputs.

Sorting

Introduction

A sorting algorithm is an algorithm that puts elements of list according to some linear (total) order. Sorting algorithms are fundamental in Computer Science.

Sorting

Bubble sort (first version)

```
bubbleSort($A$ : Array $[1 \mathinner{..} n]$)
▷ Sorts array $A$ into increasing order.
for $i := 1$ to $n - 1$
    for $j := n$ downto $i + 1$
        if $A[j-1] > A[j]$
            ▷ Swap $A[j-1]$ and $A[j]$.
            $temp := A[j-1]$
            $A[j-1] := A[j]$
            $A[j] := temp$
```

Sorting

Bubble sort (second version)

Since a procedure swap is very common in sorting algorithms, we rewrite the bubble sort algorithm calling this procedure.

```
SWAP($A$ : Array, $i : \mathbb{N}$, $j : \mathbb{N}$)
▷ Exchanges $A[i]$ and $A[j]$.
$temp := A[i]$
$A[i] := A[j]$
$A[j] := temp$
```

```
BUBBLESORT($A$ : Array $[1\,..\,n]$)
▷ Sorts array $A$ into increasing order.
for $i := 1$ to $n - 1$
    for $j := n$ downto $i + 1$
        if $A[j-1] > A[j]$
            SWAP($A, j - 1, j$)
```

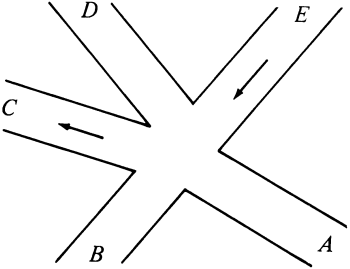
Problem: Setting Traffic Light Cycles

Problem

To design an optimal traffic light for an intersection of roads (Example 1.1).

An instance of the problem

In the figure, roads C and E are one-way, the others two way.*



---

*Figure source: Fig. 1.1.

Problem: Setting Traffic Light Cycles

Mathematical model for the problem

We can model the problem via a graph of incompatible turns.

Problem: Setting Traffic Light Cycles

Mathematical model for the problem

We can model the problem via a graph of incompatible turns.

Some questions about the model

What is a graph? Is the graph directed or undirected in our model? Do we need graphs with or without loops? What about parallel edges?

Problem: Setting Traffic Light Cycles

Notation

Let A be a set. We denote the set of all k-subsets of A by $[A]^k$.

Problem: Setting Traffic Light Cycles

Notation

Let A be a set. We denote the set of all k-subsets of A by $[A]^k$.

Definition

A **graph** is an order pair $G = (V, E)$ of disjoint sets such that $E \subseteq [V]^2$ [Diestel 2017].

Definition

The **vertices** and the **edges** (*aristas*) of a graph $G = (V, E)$ are the elements of V and E, respectively.

Problem: Setting Traffic Light Cycles

Notation

Let A be a set. We denote the set of all k-subsets of A by $[A]^k$.

Definition

A **graph** is an order pair $G = (V, E)$ of disjoint sets such that $E \subseteq [V]^2$ [Diestel 2017].

Definition

The **vertices** and the **edges** (*aristas*) of a graph $G = (V, E)$ are the elements of V and E, respectively.

Notation

Let A be a set. The cardinality of A is denoted by $|A|$.

Problem: Setting Traffic Light Cycles

Some remarks on our definition of graph

- The edges in our graphs are undirected because $\{v, w\} = \{w, v\}$.

Problem: Setting Traffic Light Cycles

Some remarks on our definition of graph

- The edges in our graphs are undirected because $\{v, w\} = \{w, v\}$.
- Since $\{v, v\} \notin [V]^2$ because $|\{v, v\}| = |\{v\}| = 1$, our graphs have no loops.

Problem: Setting Traffic Light Cycles

Some remarks on our definition of graph

- The edges in our graphs are undirected because $\{v, w\} = \{w, v\}$.
- Since $\{v, v\} \notin [V]^2$ because $|\{v, v\}| = |\{v\}| = 1$, our graphs have no loops.
- Since the multiplicity of an element in a set is one, our graphs have no parallel edges.

Problem: Setting Traffic Light Cycles

Some remarks on our definition of graph

- The edges in our graphs are undirected because $\{v, w\} = \{w, v\}$.
- Since $\{v, v\} \notin [V]^2$ because $|\{v, v\}| = |\{v\}| = 1$, our graphs have no loops.
- Since the multiplicity of an element in a set is one, our graphs have no parallel edges.
- A graph with undirected egdes, without loops and without parallel edges is also called a **simple** graph in the literature. E.g. [Bondy and Murty 2008].

Problem: Setting Traffic Light Cycles

Some remarks on our definition of graph

- The edges in our graphs are undirected because $\{v, w\} = \{w, v\}$.
- Since $\{v, v\} \notin [V]^2$ because $|\{v, v\}| = |\{v\}| = 1$, our graphs have no loops.
- Since the multiplicity of an element in a set is one, our graphs have no parallel edges.
- A graph with undirected egdes, without loops and without parallel edges is also called a **simple** graph in the literature. E.g. [Bondy and Murty 2008].
- The sets V and E must be disjoint for ruling out 'graphs' like $V = \{a, b, \{a, b\}\}$ and $E = \{\{a, b\}\}$.

Problem: Setting Traffic Light Cycles

Definition

Two vertices x, y of a graph G are **adjacent** or **neighbours**, iff $\{x, y\}$ is an edge of G.

Problem: Setting Traffic Light Cycles

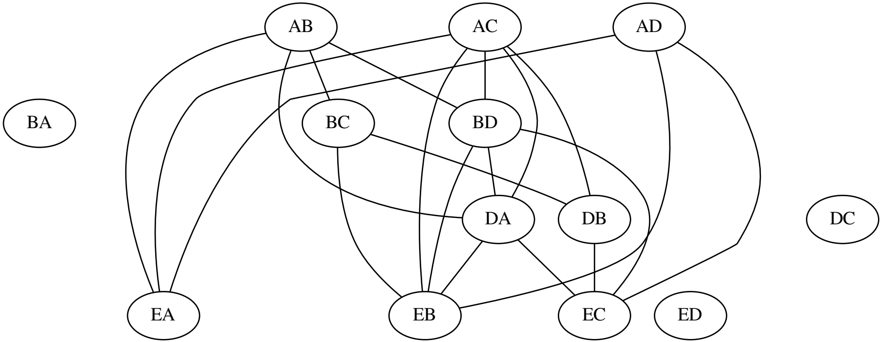
Example

Graph of incompatible turns for the instance of our problem where the vertices represent turns and whose edges represent turns cannot be performed simultaneously.

Problem: Setting Traffic Light Cycles

Example

Graph of incompatible turns for the instance of our problem where the vertices represent turns and whose edges represent turns cannot be performed simultaneously.



Problem: Setting Traffic Light Cycles

Definition

Let $G = (V, E)$ be a graph and S be a set whose elements are the available **colours**.

A **colouring** of G is a function

$$c : V \to S$$

such that $c(v) \neq c(w)$ whenever v and w are adjacent.

Problem: Setting Traffic Light Cycles

Problem equivalence (initial version)

Our problem is equivalent to the problem of colouring the graph of incompatible turns using as few colours as possible.

Problem: Setting Traffic Light Cycles

Problem equivalence (initial version)

Our problem is equivalent to the problem of colouring the graph of incompatible turns using as few colours as possible.

Definition

A k-**colouring** of a graph G is a colouring of G using at most k colours.

Problem: Setting Traffic Light Cycles

Problem equivalence (initial version)

Our problem is equivalent to the problem of colouring the graph of incompatible turns using as few colours as possible.

Definition

A k-**colouring** of a graph G is a colouring of G using at most k colours.

Definition

Let G be a graph and k be the smallest integer such that G has a k-colouring. This number k is the **chromatic number** of G.

Problem: Setting Traffic Light Cycles

Problem equivalence (initial version)

Our problem is equivalent to the problem of colouring the graph of incompatible turns using as few colours as possible.

Definition

A k-**colouring** of a graph G is a colouring of G using at most k colours.

Definition

Let G be a graph and k be the smallest integer such that G has a k-colouring. This number k is the **chromatic number** of G.

Problem equivalence (final version)

Our problem is equivalent to the problem of finding the chromatic number of the graph of incompatible turns.

Problem: Setting Traffic Light Cycles

Some remarks about our new problem

- The problem is a NP-complete problem.

Problem: Setting Traffic Light Cycles

Some remarks about our new problem

- The problem is a NP-complete problem.
- Is a good (no necessarily optimal) solution enough? If so, we could use a heuristic approach.

Problem: Setting Traffic Light Cycles

Greedy heuristic

Description from p. 5:

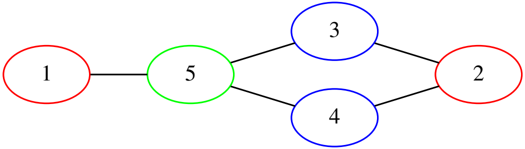
One reasonable heuristic for graph coloring is the following ‘greedy’ algorithm. Initially we try to color as many vertices as possible with the first color, then as many as possible of the uncolored vertices with the second color, and so on. To color vertices with a new color, we perform the following steps.

1. Select some uncoloured vertex and colour it with the new colour.
2. Scan the list of uncoloured vertices. For each uncoloured vertex, determine whether it has an edge to any vertex already coloured with the new colour. If there is no such edge, colour the present vertex with the new colour.

Problem: Setting Traffic Light Cycles

Example (Our greedy heuristic can fail)

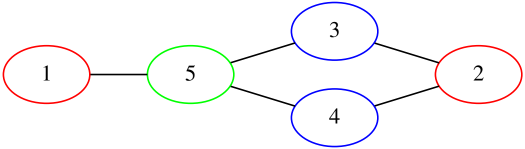
Colouring using the heuristic.



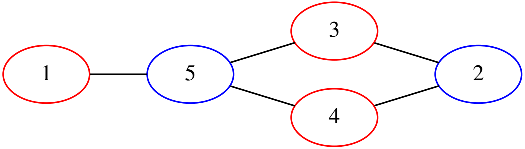
Problem: Setting Traffic Light Cycles

Example (Our greedy heuristic can fail)

Colouring using the heuristic.



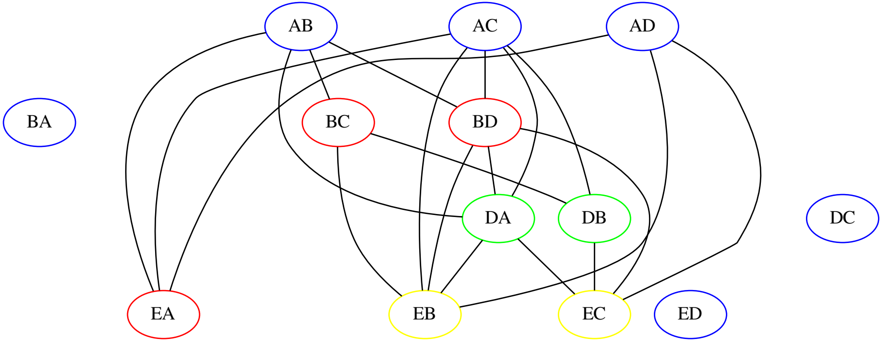
The chromatic number of graph is two.



Problem: Setting Traffic Light Cycles

A solution using the greedy heuristic

We can solve our original problem using a traffic light controller with four phases (one by each colour).



Problem: Setting Traffic Light Cycles

Discussion

From the previous solution to a real program.

Problem: Setting Traffic Light Cycles

Graph colouring applications

The graph colouring problem has an important number of applications. For example:

- Scheduling problems (e.g. assigning jobs to time slots, assigning aircraft to flights, sports scheduling, exams time table)
- Register allocation
- Sudoku puzzles

Analysis of Algorithms

The Running Time of a Program

Discussion

My program is better than yours. What does this means?

The Running Time of a Program

Discussion

My program is better than yours. What does this means?

Programs

Two contradictory goals: maintainability versus efficiency

The Running Time of a Program

Discussion

My program is better than yours. What does this means?

Programs

Two contradictory goals: maintainability versus efficiency

Quote

‘Programmers must not only be aware of ways of making programs run fast, but must know when to apply these techniques and when not to bother.’ (p. 16)

The Running Time of a Program

Dependency

The running time of a program depends on factors as:

1. The input to the program.
2. The compiler used to create the program.
3. Features of set of instructions of machine.
4. The time complexity of the algorithm underlying the program.

The Running Time of a Program

Remark

That the running time depends of the input usually means it depends of the size of the input:

So, we shall use a function

$$T(n) : \mathbb{N} \to \mathbb{R}^{\geq 0}$$

which will denote the running time of a program on inputs of size n.

The Running Time of a Program

Question

Have all the inputs of size n the same running time?

The Running Time of a Program

Question

Have all the inputs of size n the same running time?

No! The function $T(n)$ is the worst-case running time of a program on inputs of size n.

The Running Time of a Program

Question

Why not use a function

$$T_{\text{avg}}(n) : \mathbb{N} \to \mathbb{R}^{\geq 0}$$

which will denote the average running time of a program on inputs of size n?

The Running Time of a Program

Question

Why not use a function

$$T_{\text{avg}}(n) : \mathbb{N} \to \mathbb{R}^{\geq 0}$$

which will denote the average running time of a program on inputs of size n?

Can you defined what is an average input for problem? (e.g. which is an average graph for the graph colouring problem?)

The Running Time of a Program

Remark

- The function $T(n)$ has no measure units because it depends of a compiler and a set of instructions of machine.
- We can think in $T(n)$ as the number of instructions executed on an idealised computer.
- If $T(n) = n^2$ for some algorithm/program, we should talk about that the running time of the algorithm/program is proportional to n^2.

Asymptotic Notations

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big-oh of $\boldsymbol{g(n)}$**, denoted by $O(g(n))$, by

$$O(g(n)) := \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \begin{array}{l}\text{there exist positive constants } c \in \mathbb{R}^{+} \\ \text{and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \leq cg(n) \\ \text{for all } n \geq n_0 \,\}.\end{array}$$

Asymptotic Notations

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big-oh of $\boldsymbol{g(n)}$**, denoted by $O(g(n))$, by

$$O(g(n)) := \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exist positive constants } c \in \mathbb{R}^{+} \text{ and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0 \,\}.$$

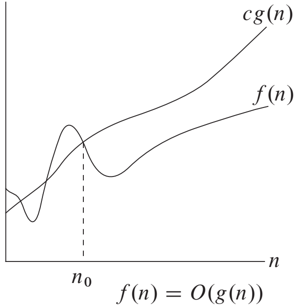
Notation

Both '$f(n) = O(g(n))$' and '$f(n)$ is $O(g(n))$' mean that $f(n) \in O(g(n))$.

Asymptotic Notations

Remark

If $f(n) \in O(g(n))$ then function $g(n)$ is an upper bound on the growth rate of the function $f(n)$.*



---

*Figure source: Cormen, Leiserson, Rivest and Stein [2009, Fig. 3.1b].

Asymptotic Notations

Exercise

Let $T(n) = (n + 1)^2$. To prove that $T(n) \in O(n^2)$. Hint: Choose $n_0 = 1$ and $c = 4$. (Example 1.4).

Asymptotic Notations

Exercise

Let $T(n) = (n+1)^2$. To prove that $T(n) \in O(n^2)$. Hint: Choose $n_0 = 1$ and $c = 4$. (Example 1.4).

Question

If $T(n) \in O(n^2)$ then $T(n) \in O(n^3)$?

Asymptotic Notations

Example

See `http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/`.

Asymptotic Notations

Example

See `http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/`.

Example

Note that

$$\begin{aligned} O(\log n) &\subseteq O(\sqrt{n}) \\ &\subseteq O(n) \\ &\subseteq O(n \log n) \\ &\subseteq O(n^2) \\ &\subseteq O(n^3) \\ &\subseteq O(2^n). \end{aligned}$$

Asymptotic Notations

Exercise

To prove that $6n^2$ is not $O(n)$. Hint: Use proof by contradiction.

Asymptotic Notations

Theorem

Let d be a natural number and $T(n)$ a polynomial function of degree d, that is,

$$T : \mathbb{N} \to \mathbb{R}$$

$$T(n) = \sum_{i=0}^{d} c_i n^i, \quad \text{with } c_i \in \mathbb{R} \text{ and } c_d \neq 0.$$

If $c_d > 0$ then $T(n) \in O(n^d)$.*

---

*See, e.g. [Cormen, Leiserson, Rivest and Stein 2009].

Asymptotic Notations

Theorem

Let d be a natural number and $T(n)$ a polynomial function of degree d, that is,

$$T : \mathbb{N} \to \mathbb{R}$$

$$T(n) = \sum_{i=0}^{d} c_i n^i, \quad \text{with } c_i \in \mathbb{R} \text{ and } c_d \neq 0.$$

If $c_d > 0$ then $T(n) \in O(n^d)$.[*]

Example

$T(n) = 42n^3 + 1523n^2 + 45728n$ is $O(n^3)$.

---

[*] See, e.g. [Cormen, Leiserson, Rivest and Stein 2009].

Asymptotic Notations

Example

Since any constant is a polynomial of degree 0, any constant function is $O(n^0)$, i.e. $O(1)$.

Remark

Note the missing variable in $O(1)$.[*]

---

[*] We could use the λ-calculus notation, i.e. $O(\lambda n.1)$.

Asymptotic Notations

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big-omega of $\boldsymbol{g(n)}$**, denoted by $\Omega(g(n))$, by

$$\Omega(g(n)) := \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \begin{array}{l} \text{there exist positive constants } c \in \mathbb{R}^{+} \\ \text{and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \geq cg(n) \\ \text{for all } n \geq n_0 \,\}. \end{array}$$

Asymptotic Notations

Definition

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. We define the set of functions **big-omega of $\boldsymbol{g(n)}$**, denoted by $\Omega(g(n))$, by

$$\Omega(g(n)) := \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \begin{array}{l} \text{there exist positive constants } c \in \mathbb{R}^{+} \\ \text{and } n_0 \in \mathbb{Z}^{+} \text{ such that } f(n) \geq cg(n) \\ \text{for all } n \geq n_0 \,\}. \end{array}$$

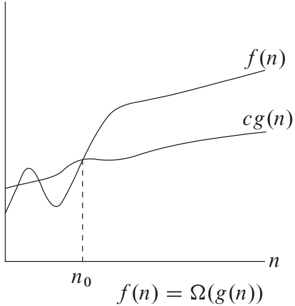
Notation

Both '$f(n) = \Omega(g(n))$' and '$f(n)$ is $\Omega(g(n))$' mean that $f(n) \in \Omega(g(n))$.

Asymptotic Notations

Remark

If $f(n) \in \Omega(g(n))$ then function $g(n)$ is a lower bound on the growth rate of the function $f(n)$.*



---

*Figure source: Cormen, Leiserson, Rivest and Stein [2009, Fig. 3.1c].

Asymptotic Notations

Example

To prove that the function

$$T : \mathbb{N} \to \mathbb{R}^{\geq 0}$$

$$T(n) = \begin{cases} n, & \text{if } n \text{ is odd;} \\ n^2/100, & \text{if } n \text{ is even;} \end{cases}$$

is $\Omega(n)$. Hint: Choose $c = 1/100$ and $n_0 = 1$.

Asymptotic Notations

Theorem (the duality rule)

$T(n) \in \Omega(g(n))$ iff $g(n) \in O(T(n))$.

Asymptotic Notations

Remark

Note that the textbook uses an alternative definition for the big-omega notation.

Let $g : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be a function. The set of functions **big-omega of $\boldsymbol{g(n)}$**, denoted by $\Omega(g(n))$, is defined by

$$\Omega(g(n)) := \{\, f : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid \text{there exists a positive constant } c \in \mathbb{R}^{+} \text{ such that } f(n) \geq cg(n) \text{ for an infinite number of values of } n \,\}.$$

We prefer the previous definition introduced instead of the definition in the textbook because it is easier to work with it (e.g. it is transitive and it satisfies the duality rule).*

---

*See, e.g. Knuth [1976], Brassard and Bratley [1996] and Cormen, Leiserson, Rivest and Stein [2009].

Asymptotic Notations

Theorem (rule for sums)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n) + T_2(n) \quad \text{is} \quad O(\max(g_1(n), g_2(n))).$$

Asymptotic Notations

Theorem (rule for sums)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n) + T_2(n) \quad \text{is} \quad O(\max(g_1(n), g_2(n))).$$

Example

The function $2^n + 3n^2 + 15n$ is $O(2^n)$.

Asymptotic Notations

Theorem (rule for sums)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n) + T_2(n) \quad \text{is} \quad O(\max(g_1(n), g_2(n))).$$

Example

The function $2^n + 3n^2 + 15n$ is $O(2^n)$.

Exercise

To prove the rule for sums.

Asymptotic Notations

Theorem (rule for products)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n)T_2(n) \quad \text{is} \quad O(g_1(n)g_2(n)).$$

Asymptotic Notations

Theorem (rule for products)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n)T_2(n) \quad \text{is} \quad O(g_1(n)g_2(n)).$$

Example

Whiteboard.

Asymptotic Notations

Theorem (rule for products)

Let $T_1(n)$ and $T_2(n)$ be $O(g_1(n))$ and $O(g_2(n))$, respectively. Then

$$T_1(n)T_2(n) \quad \text{is} \quad O(g_1(n)g_2(n)).$$

Example

Whiteboard.

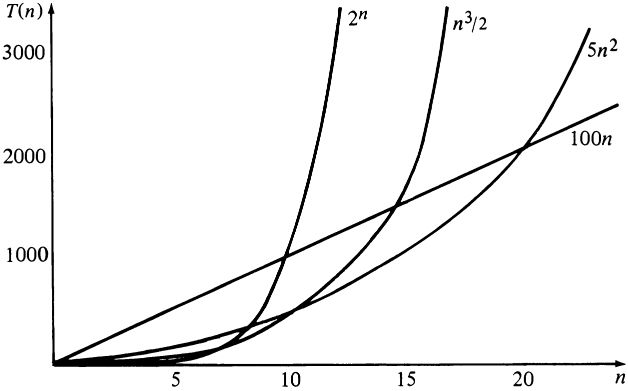
Exercise

To prove the rule for products.

The Tyranny of Growth Rate

Example

Running times of four programs.*



---

*Figure source: Fig. 1.11.

The Tyranny of Growth Rate

Example

Comparison of several running time functions (supposing that one instruction runs in one microsecond).

$T(n)$	$n = 10$	$n = 50$	$n = 100$	$n = 1000$
$\log n$	3.3 μs	5.6 μs	6.4 μs	9.9 μs
n	10.0 μs	50.0 μs	100.0 μs	1.0 ms
n^2	100.0 μs	2.5 ms	10.0 ms	1.0 s
2^n	1.0 ms	35.8 y	4.0e16 y	3.4e287 y
3^n	59.0 ms	2.3e10 y	1.6e34 y	4.2e463 y
$n!$	3.6 s	9.7e50 y	3.0e144 y	1.3e2554 y

The Tyranny of Growth Rate

Definition

A **tractable problem** is a problem than can be solved by a computer algorithm that runs in polynomial-time.

The Tyranny of Growth Rate

Definition

A **literal** is an atomic formula (propositional variable) or the negation of an atomic formula.

Definition

A (propositional logic) formula F is in **conjunctive normal form** iff

$$F \text{ has the form } F_1 \wedge \cdots \wedge F_n,$$

where each $F_1, \ldots, F_n$ is a disjunction of literals.

The Tyranny of Growth Rate

Example (3-SAT: An intractable problem)

To determine the satisfiability of a propositional formula in conjunctive normal form where each disjunction of literals is limited to at most three literals.

The problem was proposed in Karp's 21 NP-complete problems [Karp 1972].

The Tyranny of Growth Rate

Improvements on 3-SAT deterministic algorithmic complexity*

$O(1.32793^n)$	Liu [2018]
$O(1.3303^n)$	Makino, Tamaki and Yamamoto [2011, 2013]
$O(1.3334^n)$	Moser and Scheder [2011]
$O(1.439^n)$	Kutzkov and Scheder [2010]
$O(1.465^n)$	Scheder [2008]
$O(1.473^n)$	Brueggemann and Kern [2004]
$O(1.481^n)$	Dantsin, Goerdt, Hirsch, Kannan, Kleinberg, Papadimitriou, Raghavan and Schöning [2002]
$O(1.497^n)$	Schiermeyer [1996]
$O(1.505^n)$	Kullmann [1999]
$O(1.6181^n)$	Monien and Speckenmeyer [1979, 1985]
$O(2^n)$	Brute-force search

---

*Main sources: Hertli [2011, 2015]. Last updated: June 2019.

The Tyranny of Growth Rate

Supercomputers

Machines from: `www.top500.org`*
PetaFLOP (PFLOP): 10^{15} floating-point operations per second

Date	Machine	PFLOPs
2019-06	Summit	148.60
2018-11	Summit	143.50
2018-06	Summit	122.30
2016-06	Sunway TaihuLight	93.01
2013-06	Tianhe-2	33.86
2012-06	Blue Gene/Q	16.32
2011-06	K computer	8.16

---

*Last updated: TOP500 List - June 2019.

The Tyranny of Growth Rate

Simulation

Running 3-SAT times on different supercomputers using the faster deterministic algorithm, i.e. $T(1.32793^n)$.

Machine	PFLOPs	$n = 150$	$n = 200$	$n = 400$
Summit (2019-06)	148.60	20.1 `s`	336.1 `d`	4.0e24 `y`
Summit (2018-11)	143.50	20.8 `s`	348.1 `d`	4.1e24 `y`
Summit (2018-06)	122.30	24.5 `s`	1.1 `y`	4.8e24 `y`
Sunway TaihuLight	93.01	32.2 `s`	1.5 `y`	6.4e24 `y`
Tianhe-2	33.86	1.5 `m`	4.1 `y`	1.7e25 `y`
Blue Gene/Q	16.32	3.1 `m`	8.4 `y`	3.6e25 `y`
K computer	8.16	6.1 `m`	16.8 `y`	7.3e25 `y`

The Tyranny of Growth Rate

Simulation

Running 3-SAT times for different deterministic algorithms using the faster supercomputer, i.e. 148.60 PFLOPs.

Complexity	$n = 150$	$n = 200$	$n = 400$
$T(1.32793^n)$	20.1 `s`	336.1 `d`	$4.0e24$ `y`
$T(1.3303^n)$	26.3 `s`	1.3 `y`	$8.1e24$ `y`
$T(1.3334^n)$	37.3 `s`	2.1 `y`	$2.1e25$ `y`
$T(1.439^n)$	39.9 `d`	$8.7e6$ `y`	$3.6e38$ `y`
$T(1.465^n)$	1.6 `y`	$3.1e8$ `y`	$4.6e41$ `y`
$T(2^n)$	$3.1e20$ `y`	$3.4e35$ `y`	$5.5e95$ `y`

Calculating the Running Time of a Program

General rules for the analysis of programs

The running time of

1. each assignment, read, and write statement is $O(1)$,
2. a sequence of statements is the largest running time of any statement in the sequence (rule for sums),
3. evaluate conditions is $O(1)$,
4. an if-statement is the cost of evaluate the condition plus the running time of the body of the if-statement (worst case running time).
5. an if-then-else construct is the cost of evaluate the condition plus the larger running time of the true-body and the else-body (worst case running time).
6. a loop is the sum, over all times around the loop, of the running time of the body plus the cost of evaluate the termination condition.

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of first version of bubble sort.

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of the MYSTERY function (Exercise 1.12b).

```
MYSTERY($n : \mathbb{N}$)
for $i := 1$ to $n - 1$
    for $j := i + 1$ to $n$
        for $k := 1$ to $j$
            Some statement requiring $O(1)$ time.
```

Calculating the Running Time of a Program

General rules for the analysis of programs (continuation)

7. For calculating the running time of programs which call non-recursive procedures/functions, we calculate first the running time of these non-recursive procedures/functions.

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of second version of bubble sort.

Calculating the Running Time of a Program

General rules for the analysis of programs (continuation)

8. For calculating the running time of recursive programs, we get a recurrence for $T(n)$ (i.e. an equation for $T(n)$) and we solve the recurrence.

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of the FACTORIALR function.

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of the BUGGY function (Exercise 1.12d).

```
BUGGY($n : \mathbb{N}$)
if $n \leq 1$
    return $1$
else
    return $(\text{BUGGY}(n-1) + \text{BUGGY}(n-1))$
```

Calculating the Running Time of a Program

Example (whiteboard)

Worst case running time of the BUGGY function (Exercise 1.12d).

```
BUGGY($n : \mathbb{N}$)
if $n \leq 1$
    return $1$
else
    return (BUGGY($n - 1$) + BUGGY($n - 1$))
```

Question

Why is the function buggy?

Calculating the Running Time of a Program

Exercise

The $\textsc{max}(i : \mathbb{N}, n : \mathbb{N})$ function returns the largest element in positions i through $i + n - 1$ of an integer array A. You may assume for convenience that n is a power of 2. Let $T(n)$ be the worst-case time taken by the $\textsc{max}$ function with second argument n. That is, n is the number of elements of which the largest is found. Give, using the big-oh notation the worst case running time of the $\textsc{max}$ function (Exercise 1.18).

(continued on next slide)

Calculating the Running Time of a Program

Exercise (continuation)

```
$\text{MAX}(i : \mathbb{N}, n : \mathbb{N})$
$m_1, m_2 : \mathbb{Z}$
**if** $n == 1$
    **return** $A[\,i\,]$
**else**
    $m_1 := \text{MAX}(i, \lfloor n/2 \rfloor)$
    $m_2 := \text{MAX}(i + \lfloor n/2 \rfloor, \lfloor n/2 \rfloor)$
    **if** $m_1 < m_2$
        **return** $m_2$
    **else**
        **return** $m_1$
```

Abstract Data Types

Abstract Data Types

Definition

‘We can think of an **abstract data type** (ADT) as a mathematical model with a collection of operations defined on that model.’ (p. 13)

Definition

‘The **data type** of a variable is the set of values that the variable may assume.’ (p. 13)

Definition

Data structures ‘are collections of variables, possibly of several different data types, connected in various ways.’ (p. 13)

Abstract Data Types

Some remarks

- Abstract data type is a theoretical concept (design and analisis of algorithms).
- Data structures are concrete representations of data (implementation of algorithms).
- ADTs are implemented by data structures.

Abstract Data Types

Some remarks

- Abstract data type is a theoretical concept (design and analisis of algorithms).
- Data structures are concrete representations of data (implementation of algorithms).
- ADTs are implemented by data structures.

Example

Data types: Bool, char, integer, float and double

Data structures: Arrays and records

ADTs: Graphs, lists, queues, sets, stacks and trees

Abstract Data Types

Advantages of using abstract data types

- Generalisation

  ‘ADT’s are generalizations of primitive data types (integer, real, and so on), just as procedures are generalizations of primitive operations ($+$, $-$, and so on).’ (p. 11).

- Encapsulation

  ‘The ADT encapsulates a data type in the sense that the definition of the type and all operations on that type can be localized to one section of the program.’ (p. 11).

Lists

Definition

A **list** is a sequence of zero or more elements of a given type

$$a_1, a_2, \ldots, a_n$$

where,

n: **length** of the list, if $n == 0$ then the list is **empty**,

a_1: **first element** of the list,

a_n: **last element** of the list,

the **element** a_i is in the **position** i, and

elements are linearly ordered according to their position on the list.

Lists

Operations on lists

- $\textsc{End}(L)$

  Returns the position following position n in an n-element list L.

Lists

Operations on lists

- $\textsc{end}(L)$

  Returns the position following position n in an n-element list L.

- $\textsc{insert}(x, p, L)$

  Inserts x at position p in list L:

  $$a_1, a_2, \ldots, a_n \rightarrow a_1, a_2, \ldots, a_{p-1}, x, a_{p+1}, \ldots, a_n$$

  If p is $\textsc{end}(L)$, then

  $$a_1, a_2, \ldots, a_n \rightarrow a_1, a_2, \ldots, a_n, x$$

  If list L has no position p, the result is undefined.

(continued on next slide)

Lists

Operations on lists (continuation)

- $\textsc{locate}(x, L)$

  Returns the position of x on list L.

  If x appears more than once, then the position of the first occurrence is returned. If x does not appear at all, then $\textsc{end}(L)$ is returned.

Lists

Operations on lists (continuation)

- $\textsc{locate}(x, L)$

  Returns the position of x on list L.

  If x appears more than once, then the position of the first occurrence is returned. If x does not appear at all, then $\textsc{end}(L)$ is returned.

- $\textsc{retrieve}(p, L)$

  Returns the element at position p on list L.

  The result is undefined if $p == \textsc{end}(L)$ or if L has no position p.

(continued on next slide)

Lists

Operations on lists (continuation)

- $\textsc{next}(p, L)$ and $\textsc{previous}(p, L)$

  Return the positions following and preceding position p on list L.

  If p is the last position on L, then $\textsc{next}(p, L) = \textsc{end}(L)$. $\textsc{next}$ is undefined if p is $\textsc{end}(L)$. $\textsc{previous}$ is undefined if p is 1. Both functions are undefined if L has no position p.

Lists

Operations on lists (continuation)

- NEXT(p, L) and PREVIOUS(p, L)

  Return the positions following and preceding position p on list L.

  If p is the last position on L, then NEXT$(p, L) =$ END(L). NEXT is undefined if p is END(L). PREVIOUS is undefined if p is 1. Both functions are undefined if L has no position p.

- DELETE(p, L)

  Deletes the element at position p of list L:

  $$a_1, a_2, \ldots, a_n \to a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots a_{n-1}$$

  The result is undefined if L has no position p or if $p =$ END(L).

(continued on next slide)

Lists

Operations on lists

- $\textsc{makeNull}(L)$

  Causes L to become an empty list and returns position $\textsc{end}(L)$.

Lists

Operations on lists

- $\textsc{makeNull}(L)$

  Causes L to become an empty list and returns position $\textsc{end}(L)$.

- $\textsc{first}(L)$

  Returns the first position on list L.

  If L is empty, the position returned is $\textsc{end}(L)$.

Lists

Operations on lists

- $\textsc{makeNull}(L)$

  Causes L to become an empty list and returns position $\textsc{end}(L)$.

- $\textsc{first}(L)$

  Returns the first position on list L.

  If L is empty, the position returned is $\textsc{end}(L)$.

- $\textsc{printList}(L)$

  Prints the elements of L in the order of occurrence.

Lists

Example

A procedure for removing all duplicates of a list (from Fig 2.1).

```
PURGE($L$ : List)
▷ Removes duplicate elements from list $L$.
$p$ := FIRST($L$)
while $p <> $ END($L$)
        $q$ := NEXT($p, L$)
        while $q <> $ END($L$)
                if SAME(RETRIEVE($p, L$), RETRIEVE($q, L$))
                        DELETE($q, L$)
                else
                        $q$ := NEXT($q, L$)
        $p$ := NEXT($p, L$)
```

Lists

Exercise

Suppose that the list operations and the SAME function are $O(1)$. To give the worst case running time of the PURGE procedure. Hint: To suppose that the list has n elements.

Lists

Exercise

‘The following procedure was intended to remove all occurrences of element x from list L. Explain why it doesn’t always work and suggest a way to repair the procedure so it performs its intended task.’ (Exercise 2.9)

```
$\textsc{Delete}(x : \text{ElementType}, L : \text{List})$
$p : \text{ElementType}$
$p := \textsc{First}(L)$
while $p <> \textsc{End}(L)$
    if $\textsc{Retrieve}(p, L) == x$
        $\textsc{Delete}(p, L)$
    $p := \textsc{Next}(p, L)$
```

Stacks

Definition

‘A **stack** is a special kind of list in which all insertions and deletions take place at one end, called the **top**.’ (p. 53).

Stacks

Definition

‘A **stack** is a special kind of list in which all insertions and deletions take place at one end, called the **top**.’ (p. 53).

Remark

Stacks are also named LIFO (last-input-first-output) lists.

Stacks

Definition

‘A **stack** is a special kind of list in which all insertions and deletions take place at one end, called the **top**.’ (p. 53).

Remark

Stacks are also named LIFO (last-input-first-output) lists.

Example

Whiteboard.

Stacks

Operations on stacks

- MAKENULL(S). Makes stack S be an empty stack.
- TOP(S). Returns the element at the top of stack S.
- POP(S). Deletes the top element of the stack.
- PUSH(x, S). Inserts the element x at the top of stack S.
- EMPTY(S). Returns true if S is an empty stack; return false otherwise.

Stacks

Example

Program for processing a line by a text editor using a stack (Example 2.2).

Special characters:

- The character '#' is the erase character (back-space key) which cancel the previous uncanceled character, e.g.,

  abc#d##e   is ae.

- The character '@' is the kill character which cancel all previous characters on the current line.

(continued on next slide)

Stacks

Example (continuation)

```
EDIT()
S : Stack
c : Char
MAKENULL(S)
while not eoln
    read(c)
    if c == '#'
        POP(S)
    elseif c == '@'
        MAKENULL(S)
    else
        ▷ The character c is an ordinary character.
        PUSH(c, S)
print S in reverse order
```

Queues

Definition

‘A **queue** is another special kind of list, where items are inserted at one end (the **rear**) and deleted at the other end (the **front**).’ (p. 56)

Queues

Definition

‘A **queue** is another special kind of list, where items are inserted at one end (the **rear**) and deleted at the other end (the **front**).’ (p. 56)

Remark

Queues are also named FIFO (first-input-first-output) lists.

Queues

Definition

‘A **queue** is another special kind of list, where items are inserted at one end (the **rear**) and deleted at the other end (the **front**).’ (p. 56)

Remark

Queues are also named FIFO (first-input-first-output) lists.

Example

Whiteboard.

Queues

Operations on queues

- MAKENULL(Q). Makes queue Q an empty list.
- FRONT(Q). Returns the first element on queue Q.
- ENQUEUE(x, Q). Inserts element x at the end of queue Q.
- DEQUEUE(Q). Deletes the first element of Q
- EMPTY(Q). Returns true iff Q is an empty queue.

Queues

Queue operations on terms of list operations

We can use list operations for defining queue operations.

$$\begin{aligned}\textsc{front}(Q) &:= \textsc{retrieve}(\textsc{first}(Q), Q),\\ \textsc{enqueue}(x, Q) &:= \textsc{insert}(x, \textsc{end}(Q), Q),\\ \textsc{dequeue}(Q) &:= \textsc{delete}(\textsc{first}(Q), Q).\end{aligned}$$

Appendix

Floor and Ceiling Functions

Definition

The **floor** function is defined by

$$\begin{aligned} \lfloor \cdot \rfloor &: \mathbb{R} \to \mathbb{Z} \\ \lfloor x \rfloor &:= \text{that unique integer } n \text{ such that } n \leq x < n+1. \end{aligned}$$

Definition

The **ceiling** function is defined by

$$\begin{aligned} \lceil \cdot \rceil &: \mathbb{R} \to \mathbb{Z} \\ \lceil x \rceil &:= \text{that unique integer } n \text{ such that } n-1 < x \leq n. \end{aligned}$$

Summation Properties

Definition

Let $a_1, a_2, \ldots, a_n$ be a sequence of numbers, where n is a positive integer. Recall the inductive definition of the **summation notation**:

$$\begin{aligned}
\sum_{k=1}^{1} a_k &:= a_1, \\
\sum_{k=1}^{n} a_k &:= \left(\sum_{k=1}^{n-1} a_k \right) + a_n \\
&= a_1 + a_2 + \cdots + a_{n-1} + a_n.
\end{aligned}$$

Summation Properties

Properties

$$\sum_{k=1}^{n}(a_k + b_k) = \sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k \qquad \text{(additive property)},$$

$$\sum_{k=1}^{n} c a_k = c \sum_{k=1}^{n} a_k \qquad \text{(homogeneous property)},$$

$$\sum_{k=1}^{n}(\alpha a_k + \beta b_k) = \alpha \sum_{k=1}^{n} a_k + \beta \sum_{k=1}^{n} b_k \qquad \text{(linearity property)}.$$

Summation Properties

Properties

$$\sum_{k=1}^{n} f(n) = nf(n),$$

$$\sum_{k=1}^{n} a_k = \sum_{k=1}^{i} a_k + \sum_{k=i+1}^{n} a_k.$$

Summation Properties

Properties

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$

$$\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2.$$

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. [1983] (1985). Data Structures and Algorithms. Reprinted with corrections. Addison-Wesley (cit. on p. 3).

Bondy, J. A. and Murty, U. S. R. (2008). Graph Theory. Springer-Verlag (cit. on pp. 42–46).

Brassard, G. and Bratley, P. (1996). Fundamentals of Algorithmics. Prentice Hall (cit. on pp. 3, 31, 90).

Brueggemann, T. and Kern, W. (2004). An Improved Deterministic Local Search Algorithm for 3-SAT. Theoretical Computer Science 329.1–3, pp. 303–313. DOI: 10.1016/j.tcs.2004.08.002 (cit. on p. 102).

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. [1990] (2009). Introduction to Algorithms. 3rd ed. MIT Press (cit. on pp. 25, 76, 82, 83, 87, 90).

Dantsin, E. et al. (2002). A Deterministic $(2-2/(k+1))^n$ Algorithm for k-SAT Based on Local Search. Theoretical Computer Science 289.1, pp. 69–83. DOI: 10.1016/S0304-3975(01)00174-8 (cit. on p. 102).

Diestel, R. [1997] (2017). Graph Theory. 5th ed. Springer. DOI: 10.1007/978-3-662-53622-3 (cit. on pp. 39–41).

References

Hertli, T. (2011). 3-SAT Faster and Simpler - Unique-SAT Bounds for PPSZ Hold in General. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011). IEEE, pp. 277–284. DOI: 10.1109/FOCS.2011.22 (cit. on p. 102).

— (2015). Improved Exponential Algorithms for SAT and CISP. PhD thesis. ETH Zurich. DOI: 10.3929/ethz-a-010512781 (cit. on p. 102).

Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations. Ed. by Miller, R. E. and Thatcher, J. W. Plenum Press, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9 (cit. on p. 101).

Knuth, D. E. (1976). Big Omicron and Big Omega and Big Theta. SIGACT News 8.2, pp. 18–24. DOI: 10.1145/1008328.1008329 (cit. on p. 90).

Kullmann, O. (1999). New Methods for 3-SAT Decision and Worst-Case Analysis. Theoretical Computer Science 223.1–2, pp. 1–72. DOI: 10.1016/S0304-3975(98)00017-6 (cit. on p. 102).

Kutzkov, K. and Scheder, D. (2010). Using CSP to Improve Deterministic 3-SAT. CoRR abs/1007.1166

URL: https://arxiv.org/abs/1007.1166 (cit. on p. 102).

References

Liu, S. (2018). Chain, Generalization of Covering Code, and Deterministic Algorithm for k-SAT. In: 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018). Ed. by Chatzigiannakis, I., Kaklamanis, C., Marx, D. and Sannella, D. Vol. 107. Leibniz International Proceedings in Informatics (LIPIcs), 88:1–88:13. DOI: `10.4230/LIPIcs.ICALP.2018.88` (cit. on p. 102).

Makino, K., Tamaki, S. and Yamamoto, M. (2011). Derandomizing HSSW Algorithm for 3-SAT. In: Computing and Combinatorics (COCOON 2011). Ed. by Fu, B. and Du, D.-Z. Vol. 6842. Lecture Notes in Computer Science. Springer, pp. 1–12. DOI: `10.1007/978-3-642-22685-4_1` (cit. on p. 102).

— (2013). Derandomizing HSSW Algorithm for 3-SAT. Algorithmica 67.2, pp. 112–124. DOI: `10.1007/s00453-012-9741-4` (cit. on p. 102).

Monien, B. and Speckenmeyer, E. (1979). 3-Satisfiability is Testable in $O(1.62^r)$ Steps. Tech. rep. 3/1979. Reihe Theoretische Informatik, Universität Gesamthochschule Paderborn (cit. on p. 102).

— (1985). Solving Satisfiability in less than 2^n Steps. Discrete Applied Mathematics 10.3, pp. 287–295. DOI: `10.1016/0166-218X(85)90050-2` (cit. on p. 102).

References

Moser, R. A. and Scheder, D. (2011). A Full Derandomization of Schöning's k-SAT Algorithm. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing (STOC 2011), pp. 245–252. DOI: 10.1145/1993636.1993670 (cit. on p. 102).

Parberry, I. and Gasarch, W. [1994] (2002). Problems on Algorithms. 2nd ed. Prentice Hall (cit. on pp. 3, 30).

Scheder, D. (2008). Guided Search and a Faster Deterministic Algorithm for 3-SAT. In: Proc. of the 8th Latin American Symposium on Theoretical Informatic (LATIN 2008). Ed. by Laber, E. S., Bornstein, C., Nogueira, T. L. and Faria, L. Vol. 4957. Lecture Notes in Computer Science. Springer, pp. 60–71. DOI: 10.1007/978-3-540-78773-0_6 (cit. on p. 102).

Schiermeyer, I. (1996). Pure Literal Look Ahead: An $O(1.497^n)$ 3-Satisfability Algorithm (Extended Abstract). Workshop on the Satisfability Problem, Siena 1996. URL: http://gauss.ececs.uc.edu/franco_files/SAT96/sat-workshop-abstracts.html (cit. on p. 102).