CSC321. Notes (DRAFT)

Massimo Di Pierro
School of Computer Science, Telecommunications and Information Systems
DePaul University, 243 S. Wabash Av, Chicago, IL 60604, USA

January 28, 2004

Abstract

These notes provide a coincise review to some of the material covered in the csc321 course. These notes are not intended as a substitution for the textbook. Attention: these notes are in a draft stage and may contain errors. Please report errors to mdipierro@cs.depaul.edu

Contents

1. Theory

1.1. Introduction

An algorithm is a step-by-step procedure for solving a problem, and it is typically developed before doing any programming. In fact, it is independent of any programming language. Efficient algorithms can have a dramatic effect on our problem-solving capabilities. The issues that will concern us when developing and analyzing algorithms are:

1. correctness: of the problem specification, of the proposed algorithm, and of its implementation in some programming language (we will not worry about the third one: program verification is another subject altogether).

2. amount of work done: i.e., running time of the algorithm in terms of the input size (independent of hardware and programming language).

3. amount of space used: here we mean the amount of extra space beyond the size of the input (independent of hardware and programming langueage). We will say that an algorithn is *in place* if the amount of extra space is constant with respect to input size.

4. simplicity, clarity: unfortunately the simplest is not always the best in other ways.

5. optimality: can we prove that it does the best of any algorithm?

1.1.1. Pseudocodes vs Java vs Python:

	Book pseudo-code	**Java/C++**	**Python**
assignment	$a \leftarrow b$ (*ambigous!*)	$a = b$;	$a = b$
comparison	if $a = b$	if $(a == b)$	if $a == b$:
loops	for $a \leftarrow 0$ to $n - 1$	for$(a = 0; a < n; a + +)$	for a in $range(0, n)$:
block	indentation	$\{...\}$	indentation
function	function $F(a)$	void F(int $a[]$) $\{$	def $F(a)$:
function call	$F[a]$	$F(a)$	$F(a)$
arrays/lists	A_i	$A[i]$	$A[i]$
member	? member$[A]$	A.member	A.member
nothing	0	*null* / *void*$*$	*None*

1.2. Math Review

1.2.1. Symbols

∞	infinity
$\wedge$	and
$\vee$	or
$\cap$	intersection
$\cup$	union
$\in$	element or In
$\forall$	for each
$\exists$	exists
$\Rightarrow$	implies
:	such that
iff	if and only if

(1.1)

1.2.2. Set Theory Review

Important Sets

$\mathbf{0}$	empty set
$\mathbb{N}$	natural numbers {0,1,2,3,...}
$\mathbb{N}^+$	positive natural numbers {1,2,3,...}
$\mathbb{Z}$	all integers {...,-3,-2,-1,0,1,2,3,...}
$\mathbb{R}$	all real numbers
$\mathbb{R}^+$	positive real numbers (not including 0)
$\mathbb{R}^*$	positive numbers including 0

(1.2)

Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**

$$\mathcal{A} \cap \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{1.3}$$

- **Union**

$$\mathcal{A} \cup \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{1.4}$$

- **Difference**

$$\mathcal{A} - \mathcal{B} \stackrel{def}{=} \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{1.5}$$

Corresponding Python functions:

```
def Intersection(A,B):
    C=[]
    for element in A:
        if element in B:
            C=C+[element]
    return C

def Union(A,B):
    C=[]
    for element in A+B:
        if not element in C:
            C=C+[element]
```

```
    return C

def Difference(A,B):
    C=[]
    for element in A:
        if not element in B:
            C=C+[element]
    return C
```

Set laws

- Empty set laws

$$\begin{aligned} \mathcal{A} \cup \mathbf{0} &= \mathcal{A} \\ \mathcal{A} \cap \mathbf{0} &= \mathbf{0} \end{aligned}$$

- Idempotency laws

$$\begin{aligned} \mathcal{A} \cup \mathcal{A} &= \mathcal{A} \\ \mathcal{A} \cap \mathcal{A} &= \mathcal{A} \end{aligned}$$

- Commutative laws

$$\begin{aligned} \mathcal{A} \cup \mathcal{B} &= \mathcal{B} \cup \mathcal{A} \\ \mathcal{A} \cap \mathcal{B} &= \mathcal{B} \cap \mathcal{A} \end{aligned}$$

- Associative laws

$$\begin{aligned} \mathcal{A} \cup (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} \cup \mathcal{B}) \cup \mathcal{C} \\ \mathcal{A} \cap (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} \cap \mathcal{B}) \cap \mathcal{C} \end{aligned}$$

- Distributive laws

$$\begin{aligned} \mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C}) \\ \mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C}) \end{aligned}$$

- Absorption laws

$$\begin{aligned} \mathcal{A} \cap (\mathcal{A} \cup \mathcal{B}) &= \mathcal{A} \\ \mathcal{A} \cup (\mathcal{A} \cap \mathcal{B}) &= \mathcal{A} \end{aligned}$$

- DeMorgan laws

$$\begin{aligned} \mathcal{A} - (\mathcal{B} \cup \mathcal{C}) &= (\mathcal{A} - \mathcal{B}) \cap (\mathcal{A} - \mathcal{C}) \\ \mathcal{A} - (\mathcal{B} \cap \mathcal{C}) &= (\mathcal{A} - \mathcal{B}) \cup (\mathcal{A} - \mathcal{C}) \end{aligned}$$

More set definitions

- $\mathcal{A}$ is a **subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$
- $\mathcal{A}$ is a **proper subset** of $\mathcal{B}$ iff $\forall x \in \mathcal{A}, x \in \mathcal{B}$ and $\exists x \in \mathcal{B}, x \notin \mathcal{A}$
- $P = \{S_i, i = 1, ..., N\}$ (a set of sets S_i) is a **partition** of $\mathcal{A}$ iff $S_1 \cup S_2 \cup ... \cup S_N = \mathcal{A}$ and $\forall i, j, S_i \cap S_j = \mathbf{0}$
- The number of elements in a set $\mathcal{A}$ is called the **cardinality** of set $\mathcal{A}$.
- cardinality($\mathbb{N}$)=countable infinite (∞)
- cardinality($\mathbb{R}$)=uncountable infinite (∞) !!!

Cantor's argument

Cantor proved that the real numbers in any interval (for example in $[0, 1)$) are more then the integer numbers, therefore real numbers are uncountable. The proof proceeds as follow:

1. Consider the real numbers in the interval $[0, 1)$ not including 1.
2. Assume that these real numbers are countable. Therefore is it possible to associate each of them to an integer

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 2 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 3 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 4 & \longleftrightarrow & 0.xxxxxxxxxx... \\ 5 & \longleftrightarrow & 0.xxxxxxxxxx... \\ ... & ... & ... \end{array} \tag{1.6}$$

(here the x represent a decimal digits of a real numbers)

3. Now construct a number $\alpha = 0.yyyyyyyy....$ where the first decimal digit differs from the first decimal digit of the first real number of table 1.6, the second decimal digit differs from the second decimal digit of the second real number of table 1.6 and so on and on for all the infinite decimal digits:

$$\begin{array}{lll} 1 & \longleftrightarrow & 0.\overline{x}xxxxxxxxxx... \\ 2 & \longleftrightarrow & 0.x\overline{x}xxxxxxxxx... \\ 3 & \longleftrightarrow & 0.xx\overline{x}xxxxxxxx... \\ 4 & \longleftrightarrow & 0.xxx\overline{x}xxxxxxx... \\ 5 & \longleftrightarrow & 0.xxxx\overline{x}xxxxxx... \\ ... & ... & ... \end{array} \tag{1.7}$$

4. The new number α is a real number and by construction it is not in the table. In fact it differs with each item for at least one decimal digit. Therefore the existence of α disproves the assumption that all real numbers in the interval $[0, 1)$ are listed in the table.

Godel's Theorem

Godel used a similar diagonal argument to prove that there are as many problems (or theorems) as real numbers and as many algorithms (or proofs) as natural numbers. Since there is more of the former than the latter it follows that thare are problems for which there is no corresponding solving algorithm. Another interpretation of Goedel's theorem is that, in any formal language, for example mathematics, there are theorems that cannot be proved.

Proposition 1. *It is impossible to write a computer program to test if a given algorithm stops or enters into an infinite loop.*

Example 2. *Just as an example of the above statement, it is not known if the following program stops:*

```
def verify(i):
    s=str(i*i)
    d=[]
    for c in s:
```

```
        if not c in d:
            d.append(c)
    if len(d)==2:
        return 1
    else:
        return 0

def next(i):
    i=long(i)
    while 1:
        i=i+2
        if verify(i):
            print i, i*i
            break


next(81619)
```

While one day this problem may be solved there are many other problems there are still unsolved, actually there is an infinite number of them!

Relations

- A **Cartesian Product** is defined as

$$\mathcal{A} \times \mathcal{B} = \{(a, b) : a \in \mathcal{A} \text{ and } b \in \mathcal{B}\} \tag{1.8}$$

- A **binary relation** R between two sets $\mathcal{A}$ and $\mathcal{B}$ if a subset of their Cartesian product.
- A binary relation is **transitive** is aRb and bRc implies aRc
- A binary relation is **symmetric** if aRb implies bRa
- A binary relation is **reflexive** if aRa if always true for each a.

Example 3. *$a < b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)*

Example 4. *$a > b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive)*

Example 5. *$a = b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, symmetric and reflexive)*

Example 6. *$a \leq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)*

Example 7. *$a \geq b$ for $a \in \mathcal{A}$ and $b \in \mathcal{B}$ is a relation (transitive, and reflexive)*

- A relation R that is transitive, symmetric and reflexive is called an **equivalence relation** and is often indicated with the notation $a \sim b$.

Theorem 8. *An equivalence relation is the same as a partition.*

Corresponding Python functions:

```
def CartesianProduct(A,B):
    C=[]
    for a in A:
        for b in B:
            C=C+[(a,b)]
    return C
```

Functions

- A **function** between two sets $\mathcal{A}$ and $\mathcal{B}$ is a binary relation on $\mathcal{A} \times \mathcal{B}$ and is usually indicated with the notation $f : \mathcal{A} \longmapsto \mathcal{B}$
- The set $\mathcal{A}$ is called **domain** of the function.
- The set $\mathcal{B}$ is called **codomain** of the function.
- A function **maps** each element $x \in \mathcal{A}$ into an element $f(x) = y \in \mathcal{B}$
- The **image** of a function $f : \mathcal{A} \longmapsto \mathcal{B}$ is the set $\mathcal{B}' = \{y \in \mathcal{B} : \exists x \in \mathcal{A}, f(x) = y\} \subseteq \mathcal{B}$
- If $\mathcal{B}'$ is $\mathcal{B}$ then a function is said to be **surjective**.
- If for each x and x' in $\mathcal{A}$ where $x \neq x'$ implies that $f(x) \neq f(x')$ (i.e. if not two different elements of $\mathcal{A}$ are mapped into the same element in $\mathcal{B}$) the function is said to be a **bijection**.

- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is invertible if it exists a function $g : \mathcal{B} \longmapsto \mathcal{A}$ such that for each $x \in \mathcal{A}, g(f(x)) = x$ and $y \in \mathcal{B}, f(g(y)) = y$. The function g is indicated with f^{-1}.
- A function $f : \mathcal{A} \longmapsto \mathcal{B}$ is a surjection and a bijection iff f is an invertible function.

Example 9. $f(n) \stackrel{def}{=} n \bmod 2$ *with domain* $\mathbb{N}$ *and codomain* $\mathbb{N}$ *is not a surjection nor a bijection.*

Example 10. $f(n) \stackrel{def}{=} n \bmod 2$ *with domain* $\mathbb{N}$ *and codomain* $\{0, 1\}$ *is a surjection but not a bijection*

Example 11. $f(x) \stackrel{def}{=} 2x$ *with domain* $\mathbb{N}$ *and codomain* $\mathbb{N}$ *is not a surjection but is a bijection (in fact it is not invertible on odd numbers)*

Example 12. $f(x) \stackrel{def}{=} 2x$ *with domain* $\mathbb{R}$ *and codomain* $\mathbb{R}$ *is not a surjection and is a bijection (in fact it is invertible)*

1.2.3. Logarithms

If $x = a^y$ with $a > 0$ then $y = \log_a x$ with domain $x \in (0, \infty)$ and co-domain $y = (-\infty, \infty)$. If the base a is not indicated the natural $\log a = e = 2.7183...$ is assumed.

Useful properties of logarithms:

$$\begin{aligned} \log_a x &= \frac{\log x}{\log a} \\ \log xy &= (\log x) + (\log y) \\ \log \frac{x}{y} &= (\log x) - (\log y) \\ \log x^n &= n \log x \end{aligned}$$

Note that $\log^n x = (\log x)^n$ is not the same as $\log x^n = \log(x^n)$, therefore the former is not equal to $n \log x$.

1.2.4. Finite sums

Definition

$$\sum_{i=0}^{i<n} f(i) \stackrel{def}{=} f(0) + f(1) + ... + f(n-1) \tag{1.9}$$

Corresponding Python functions:

```
def Sum(f,min_i,max_i):
    a=0
    for i in range(min_i,max_i):
        a=a+f(i)
    return a
```

Properties

- **Linearity I**

$$\begin{aligned} \sum_{i=0}^{i\leq n} f(i) &= \sum_{i=0}^{i<n} f(i) + f(n) \\ \sum_{i=a}^{i\leq b} f(i) &= \sum_{i=0}^{i\leq b} f(i) - \sum_{i=0}^{i<a} f(i) \end{aligned}$$

- **Linearity II**

$$\sum_{i=0}^{i<n} af(i) + bg(i) = a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \tag{1.10}$$

Proof:

$$\begin{aligned} \sum_{i=0}^{i<n} af(i) + bg(i) &= (af(0) + bg(0)) + ... + (af(n-1) + bg(n-1)) \\ &= af(0) + ... + af(n-1) + bg(0) + ... + bg(n-1) \\ &= a\left(f(0) + ... + f(n-1)\right) + b\left(g(0) + ... + g(n-1)\right) \\ &= a\left(\sum_{i=0}^{i<n} f(i)\right) + b\left(\sum_{i=0}^{i<n} g(i)\right) \end{aligned} \tag{1.11}$$

Example 13.

$$\begin{aligned}
\sum_{i=0}^{i<n} c &= cn \text{ for any constant } c \\
\sum_{i=0}^{i<n} i &= \frac{1}{2}n(n-1) \\
\sum_{i=0}^{i<n} i^2 &= \frac{1}{6}n(n-1)(2n-1) \\
\sum_{i=0}^{i<n} i^3 &= \frac{1}{4}n^2(n-1)^2 \\
\sum_{i=0}^{i<n} x^i &= \frac{x^n-1}{x-1} \text{ (geometric sum)} \\
\sum_{i=0}^{i<n} \frac{1}{i(i+1)} &= 1-\frac{1}{n} \text{ (telescopic sum)}
\end{aligned}$$

1.2.5. Limits ($x \to \infty$)

In these section we will only deal with limits ($x \to \infty$) of positive functions.

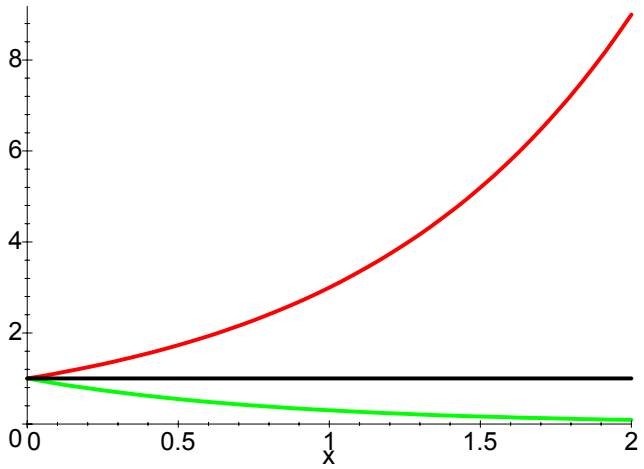
Example 14. *Example of limits:*

$$\lim_{x\to\infty} a^x = 0 \text{ if } (a<1) \tag{1.12}$$

$$\lim_{x\to\infty} a^x = 1 \text{ if } (a=1) \tag{1.13}$$

$$\lim_{x\to\infty} a^x = \infty \text{ if } (a>1) \tag{1.14}$$

As we can see from the following plot



Practical rule

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = ? \tag{1.15}$$

First compute limits of numerator and denominator separately

$$\begin{aligned} \lim_{x\to\infty} f(x) &= a \\ \lim_{x\to\infty} g(x) &= b \end{aligned}$$

- If $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \frac{a}{b} \tag{1.16}$$

- If $a \in \mathbb{R}$ and $b = \infty$ then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{1.17}$$

- If ($a \in \mathbb{R}^+$ and $b = 0$) or ($a = \infty$ and $b \in \mathbb{R}$)

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty \tag{1.18}$$

- If ($a = 0$ and $b = 0$) or ($a = \infty$ and $b = \infty$) use de l'Hopital rule

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)} \tag{1.19}$$

  and start again!

- Else ... the limit does not exist (typically oscillating functions or non-analytic functions).

Theorem 15. *For any $a \in \mathbb{R}$ or $a = \infty$*

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = a \Rightarrow \lim_{x\to\infty} \frac{g(x)}{f(x)} = 1/a$$

Table of Derivatives

$f(x)$	$f'(x)$
c	0
ax^n	anx^{n-1}
$\log x$	$\frac{1}{x}$
e^x	e^x
a^x	$a^x \log a$
$x^n \log x, n > 0$	$x^{n-1}(n \log x + 1)$

(1.20)

Practical rules to compute derivatives

$$\begin{aligned}
\frac{d}{dx}\left(f(x) + g(x)\right) &= f'(x) + g'(x) \\
\frac{d}{dx}\left(f(x) - g(x)\right) &= f'(x) - g'(x) \\
\frac{d}{dx}\left(f(x)g(x)\right) &= f'(x)g(x) + f(x)g'(x) \\
\frac{d}{dx}\left(\frac{1}{f(x)}\right) &= -\frac{f'(x)}{f(x)^2} \\
\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) &= \frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2} \\
\frac{d}{dx} f(g(x)) &= f'(g(x))g'(x)
\end{aligned}$$

Useful limits (computed using above rules)

Example 16.

$$\lim_{x\to\infty} \frac{x-3}{2x+5} = \frac{1}{2} \tag{1.21}$$

Example 17.

$$\lim_{x\to\infty} \frac{a_n x^n + a_{n-1}x^{n-1} + ... + a_0}{b_m x^m + b_{m-1}x^{m-1} + ... + b_0} = \left\{ \begin{array}{c} 0 \text{ is } n < m \\ \frac{a_n}{b_m} \text{ if } n = m \\ \infty \text{ if } n > m \end{array} \right\} \tag{1.22}$$

Example 18.

$$\lim_{x\to\infty} \frac{(\log x)^n}{P_m(x)} = 0 \tag{1.23}$$

where $P_m(x)$ indicates any polynomial in x.

Example 19.

$$\lim_{x\to\infty} \frac{e^{nx}}{P_m(x)} = \infty \tag{1.24}$$

1.3. Sorting functions

The limit

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{1.25}$$

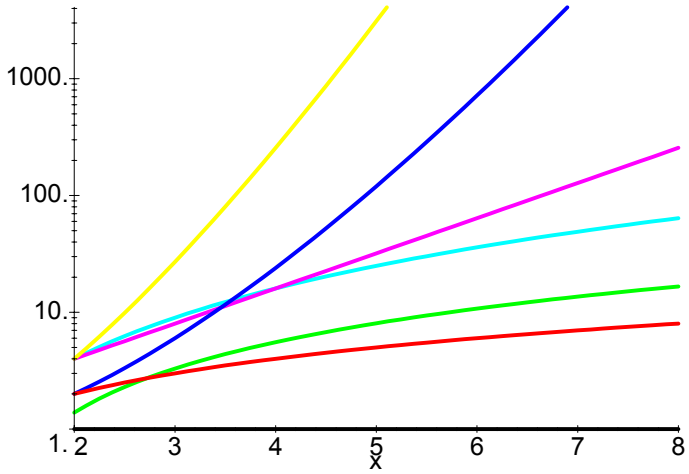
means that $g(x)$ grows more than $g(x)$ when $x \to \infty$. This induces a relation between $f(x)$ and $g(x)$:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \Longrightarrow f(x) \text{ “grows less than” } g(x) \tag{1.26}$$

We can then sort functions according with their behavior at infinity for example:

$$1 \prec \log x \prec x \prec x \log x \prec x^2 \prec 2^x \prec x! \prec x^x \tag{1.27}$$

where the symbol $\prec$ in this context reads “grow less than”.



(1.28)

1.3.1. Order of growth of functions

Definitions

$$O(g(x)) \stackrel{def}{=} \{f(x) : \exists x_0, c_0,\ \forall x > x_0,\ 0 \leq f(x) < c_0 g(x)\}$$
$$\Omega(g(x)) \stackrel{def}{=} \{f(x) : \exists x_0, c_0,\ \forall x > x_0,\ 0 \leq c_0 g(x) < f(x)\}$$
$$\Theta(g(x)) \stackrel{def}{=} O(g(x)) \cap \Omega(g(x))$$
$$o(g(x)) \stackrel{def}{=} O(g(x)) - \Omega(g(x))$$
$$\omega(g(x)) \stackrel{def}{=} \Omega(g(x)) - O(g(x))$$

Intuitive meaning

- $O(g(x))$ = Functions that grow no faster than $g(x)$ when $x \to \infty$
- $\Omega(g(x))$ = Functions that grow no slower than $g(x)$ when $x \to \infty$
- $\Theta(g(x))$ = Functions that grow at the same rate as $g(x)$ when $x \to \infty$
- $o(g(x))$ = Functions that grow slower than $g(x)$ when $x \to \infty$
- $\omega(g(x))$ = Functions that grow faster than $g(x)$ when $x \to \infty$

Practical rules

1. Compute the limit
$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = a \tag{1.29}$$
2. Then look it up on the table
$$\begin{array}{lcl} a \text{ is positive or zero} & \Longrightarrow & f(x) \in O(g(x)) \Leftrightarrow f \preceq g \\ a \text{ is positive or infinity} & \Longrightarrow & f(x) \in \Omega(g(x)) \Leftrightarrow f \succeq g \\ a \text{ is positive} & \Longrightarrow & f(x) \in \Theta(g(x)) \Leftrightarrow f \sim g \\ a \text{ is zero} & \Longrightarrow & f(x) \in o(g(x)) \Leftrightarrow f \prec g \\ a \text{ is infinity} & \Longrightarrow & f(x) \in \omega(g(x)) \Leftrightarrow f \succ g \end{array} \tag{1.30}$$

The rules assume the limits exist. The inverse is not true. For example:
$$f(x) \in \Theta(g(x)) \not\Rightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} \text{ is positive} \tag{1.31}$$

Theorem 20. *Any polynomial $T(n) = P_m(n)$ of degree m is $T(n) \in \Theta(n^m), \in O(n^m)$*

Theorem 21. *$T_1(n) \in O(f(n))$ and $T_2 \in O(g(n)) \Rightarrow T_1(n)+T_2(n) \in O(\max(f(n), g(n)))$*

Theorem 22. *$T_1(n) \in O(f(n))$ and $T_2 \in O(g(n)) \Rightarrow T_1(n)T_2(n) \in O(f(n)g(n))$*

Example 23. *Is $x \log x + 3x$ in $O(x^2)$?*
$$\lim_{x\to\infty} \frac{x\log x + 3x}{x^2} \overset{l'Hopital}{\longrightarrow} \lim_{x\to\infty} \frac{1/x}{2} = 0 \tag{1.32}$$
answer: Yes

Example 24. *Is $x \log x$ in $\Omega(x^3)$?*
$$\lim_{x\to\infty} \frac{x\log x}{x^3} \overset{l'Hopital^2}{\longrightarrow} \lim_{x\to\infty} \frac{1/x}{6x} = 0 \tag{1.33}$$
answer: No

1.4. Introduction to study of algorithms

1.4.1. Maximum and Minimum

Consider the following algorithms which determine respectively the minimum and maximum elements in the input array/list A:

```
def Minimum(A):
    j=0
    for i in range(1,len(A)):
        if A[i]<A[j]:
            j=i
    return  A[j]

def Maximum(A):
    j=0
    for i in range(1,len(A)):
        if A[i]>A[j]:
            j=i
    return  A[j]
```

Each of these algorithms perform:

- One assignment (line 02)
- $n =$`len(A)` comparisons (line 03)
- $n - 1$ comparisons (line 04)
- line 05 is executed only if line 05 condition is true. In the worst case line 05 is executed $n - 1$ times.

Therefore in the worst case this program performs $2n - 1$ comparisons and n assignments.

We say this program running time goes like

$$T(n) = c_1(2n - 1) + c_2n = (2c_1 + c_2)n - c_1 \tag{1.34}$$

And it is obvious to prove that $T(n) \in \Theta(n)$.

Example: loop0

```
def loop0(n):
    for i in range(0,n):
        print i
```

$$T(n) = \sum_{i=0}^{i<n} 1 = n \in \Theta(n) \Rightarrow \texttt{loop0} \in \Theta(n)$$

Example: loop1

```
def loop1(n):
    for i in range(0,n*n):
        print i
```

$$T(n) = \sum_{i=0}^{i<n^2} 1 = n^2 \in \Theta(n^2) \Rightarrow \texttt{loop1} \in \Theta(n^2)$$

Example: loop2

```
def loop2(n):
    for i in range(0,n):
        for j in range(0,n):
            print i,j
```

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<n} 1 = \sum_{i=0}^{i<n} n = n^2 + ... \in \Theta(n^2) \Rightarrow \texttt{loop2} \in \Theta(n^2)$$

Example: loop3

```
def loop3(n):
    for i in range(0,n):
        for j in range(0,i):
            print i,j
```

$$T(n) = \sum_{i=0}^{i<n} \sum_{j=0}^{j<i} 1 = \sum_{i=0}^{i<n} i = \frac{1}{2}n(n-1) \in \Theta(n^2) \Rightarrow \texttt{loop3} \in \Theta(n^2)$$

Example: loop4

```
def loop4(n):
    for i in range(0,n):
        for j in range(0,i*i):
            print i,j
```

$$\begin{aligned} T(n) &= \sum_{i=0}^{i<n} \sum_{j=0}^{j<i^2} 1 = \sum_{i=0}^{i<n} i^2 = \frac{1}{6} n(n-1)(2n-1) \in \Theta(n^3) \\ &\Rightarrow \texttt{loop4} \in \Theta(n^3) \end{aligned}$$

Example: factorial0

```
def factorial0(n):
    prod=1
    for i in range(2,n+1):
        prod=prod*i
    return prod
```

$$T(n) = \sum_{i=1}^{i \leq n} 1 = n \in \Theta(n) \Rightarrow \texttt{factorial0} \in \Theta(n)$$

Example: concatenate0

```
def concatenate0(n):
    for i in range(n*n):
        print i
    for j in range(n*n*n):
        print j
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate0} \in \Theta(n^3)$$

Example: concatenate1

```
def concatenate1(n):
    if a<0:
        for i in range(n*n):
            print i
    else:
        for j in range(n*n*n):
            print j
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in \Theta(n^3)$$

1.4.2. Insertion Sort

Consider the following algorithm the sorts the elements in the input array/list A:

```
def InsertionSort(A):
    for i in range(1,len(A)):
        j=i-1
        while(j>=0):
            if A[j]>A[j+1]:
                (A[j], A[j+1]) = (A[j+1], A[j])
                j=j-1
            else:
                break
            pass
```

Prove that $T(n) \in \Theta(n^2)$.

1.4.3. Recurrence relations

The running time $T(n)$ of recursive algorithms is determined by:

1. Writing the recurrence relation in $T(n)$ for the algorithm.
2. Solving the recurrence relation or, at least, put a bound on $T(n)$.

In these notes we assume $f(n)$ is a positive monotonic increasing function, $a \geq 1$ and $b \geq 2$.

Recurrence Relations solvable by recursion (type I)

$$T(n) = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ aT(n-1) + f(n) & \text{if } n > 1 \end{array} \right\} \tag{1.35}$$

The result is:

$$T(n) = a^{n-1}\left[c - f(1)\right] + \sum_{i=1}^{i \leq n} a^{n-i} f(i) \tag{1.36}$$

The order of growth of $T(n)$ can be determined by substituting in the value of a and the function $f(n)$.

Here are two cases of interest:

Theorem 25. *If $P_m(n)$ is a polynomial of degree m*

$$T(n) = T(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(n^{m+1})$$

Theorem 26. *If $P_m(n)$ is a polynomial of degree m and $a > 1$*

$$T(n) = aT(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(a^n)$$

Theorem 27. *If $P_m(n)$ is a polynomial of degree m and $a < 1$*

$$T(n) = aT(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(n^m)$$

Your are not allowed to use the above three theorems in solving problems. I want to see proofs using eq.(1.36).

Recurrence Relations solvable using Master Theorem (type II)

$$T(n) = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ aT(n/b) + f(n) & \text{if } n > 1 \end{array} \right\} \tag{1.37}$$

We do not solve it in $T(n)$ but we can determine the order of growth of $T(n)$ by using the following theorem:

Master Theorem

Given a recurrence relation of the form (1.37) find the best polynomial bound on the function $f(n)$ in the following way:

Determine the lowest value of $k \in \mathbb{R}^*$ (0 or positive but not ∞) so that

$$\lim_{n\to\infty} \frac{f(n)}{n^k} = \alpha \in \mathbb{R}^* \text{ (0 or positive real number but not } \infty)$$

- If there is not such a k because $f(n)$ grows more than any polynomial then

$$\lim_{n\to\infty} \frac{af(n/b)}{f(n)} < 1 \Rightarrow T(n) \in \Theta(f(n)) \tag{1.38}$$

$$\lim_{n\to\infty} \frac{af(n/b)}{f(n)} \geq 1 \Rightarrow T(n) \in \Omega(f(n)) \tag{1.39}$$

  else ...

- If $\alpha = 0$ for every k in some range then $f(n) \in O(n^k)$ and we look up the tighter upper bound in table (1.40) for k in the range. Else...

- If $\alpha > 0$ and finite then $f(n) \in \Theta(n^k)$ and we look up solution in table (1.41).

recurrence type	$T(n) \leq aT(n/b) + f(n)$
case HAVE ONLY UPPER BOUND:	$f(n) \in O(n^k)$
if $a < b^k$ then	$T(n) \in O(n^k)$
if $a = b^k$ then	$T(n) \in O(n^k \log n)$
if $a > b^k$ then	$T(n) \in O(n^{\log_b a})$

(1.40)

recurrence type	$T(n) = aT(n/b) + f(n)$
case HAVE ORDER OF GROWTH:	$f(n) \in \Theta(n^k)$
if $a < b^k$ then	$T(n) \in \Theta(n^k)$
if $a = b^k$ then	$T(n) \in \Theta(n^k \log n)$
if $a > b^k$ then	$T(n) \in \Theta(n^{\log_b a})$

(1.41)

recurrence type	$T(n) \geq aT(n/b) + f(n)$
case HAVE ONLY LOWER BOUND:	$f(n) \in \Omega(n^k)$
if $a < b^k$ then	$T(n) \in \Omega(n^k)$
if $a = b^k$ then	$T(n) \in \Omega(n^k \log n)$
if $a > b^k$ then	$T(n) \in \Omega(n^{\log_b a})$

(1.42)

Note that the only difficult part is finding a polynomial bound on $f(n)$, i.e. a value of k.

We look-up table (1.42) if we are interested in lower bounds.

Example 28. *Consider the following recurrence relation:*

$$T(n) = 2T(n/2) + n$$

from which we read

$$a = b = 2 \text{ and } f(n) = n \in \Theta(n) \Rightarrow k = 1$$

In the table (1.41), corresponding to case $f(n) \in \Theta(n^k)$, row 4, corresponding to case $a = b^k$, we find the solution:

$$T(n) \in \Theta(n \log n)$$

This is the order of growth of the MergeSort!

Example 29. *Consider the following recurrence relation:*

$$T(n) = T(n/2) + e^n$$

from which we read

$$a = 1,\ b = 2 \text{ and } f(n) = e^n$$

since $f(n)$ grows more than polynomially then from (1.38)

$$\lim_{n\to\infty} \frac{af(n/b)}{f(n)} = \lim_{n\to\infty} \frac{e^{n/2}}{e^n} = \lim_{n\to\infty} \left(\frac{1}{\sqrt{e}}\right)^n = \lim_{n\to\infty} (0.606...)^n = 0 < 1$$

we conclude that $T(n) \in \Theta(e^n)$.

Example 30. *Consider the following recurrence relation:*

$$T(n) = 3T(n/3) + n\log n$$

from which we read

$$a = b = 3 \text{ and } f(n) = n\log n$$

the best bounds we can put on $f(x)$ are: $f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$. From them and tables (1.40) and (1.42) respectively we conclude: $T(n) \in O(n^2)$ and $T(n) \in \Omega(n\log n)$. (The true solution is, in fact, $T(n) \in \Theta(n\log^2 n)$ as stated in the one of the theorems below!)

Theorem 31. *For $m \geq 0$, $p \geq 0$ and $q > 1$, from the Master Theorem (in its most general form) we conclude that:*

$$\begin{aligned}
T(n) &= aT(n/b) + \Theta(n^m) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m) \\
T(n) &= aT(n/b) + \Theta(n^m) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log n) \\
T(n) &= aT(n/b) + \Theta(n^m) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m \log^p n) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log^{p+1} n) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \text{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \\
T(n) &= aT(n/b) + \Theta(q^n) \Rightarrow T(n) \in \Theta(q^n)
\end{aligned}$$

Recurrence Relations solved by substitution (type III)

For example

$$T(n) = T(b) + T(n-b-a) + c$$

can be solved by guessing $T(n) = n + a - c$.

$$\begin{aligned}
T(n) &= T(b) + T(n-b-a) + c \\
&= [b + a - c] + [n - b - a + a - c] + c \\
&= n + a - c
\end{aligned}$$

from which we conclude $T(n) \in \Theta(n)$.

Sometimes our guess is only an inequality for example

$$T(n) = T(n/a - b) + c$$

with $a > 1, b > 0$, can be solved by guessing $T(n) \leq c \log_a n$ in fact

$$\begin{aligned} T(n) &= T(n/a - b) + c \\ &\leq c \log_a (n/a - b) + c \\ &\leq c \log_a (n/a) + a \\ &\leq c \log_a n - c \log_a a + c \\ &= c \log_a n \end{aligned}$$

from which we conclude $T(n) \in O(\log n)$.

Note that when making a guess:

$$\begin{aligned} \text{“} \ = \ \text{”} &\rightarrow \text{“}\Theta\text{”} \\ \text{“} \ \leq \ \text{”} &\rightarrow \text{“}O\text{”} \\ \text{“} \ \geq \ \text{”} &\rightarrow \text{“}\Omega\text{”} \end{aligned}$$

Recurrence Relations that can be reduced (type IV)

Recurrence relations can be very difficult and a general technique for solving them does not exist. Sometimes we try a **substitution method**.

Example 32. *Consider the following recurrence relation:*

$$T(n) = \left\{ \begin{array}{ll} c & \textit{if } n \leq 1 \\ 2T(\sqrt{n}) + \log n & \textit{if } n > 1 \end{array} \right\} \tag{1.43}$$

we can replace n with $e^k = n$ in eq.(1.43) and obtain:

$$T(e^k) = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ 2T(e^{k/2}) + k & \text{if } n > 1 \end{array} \right\} \tag{1.44}$$

If we also replace $T(e^k)$ with $S(k) = T(e^k)$ we obtain:

$$\underbrace{S(k)}_{T(e^k)} = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ 2 \underbrace{S(k/2)}_{T(e^{k/2})} + k & \text{if } n > 1 \end{array} \right\} \tag{1.45}$$

so that we can now apply the Master Theorem to S. We obtain that $S(k) \in \Theta(k \log k)$. Once we have the order of growth of S we can determine the order of growth of $T(n)$ by substitution:

$$T(n) = S(\log n) \in \Theta(\underbrace{\log n}_{k} \log \underbrace{\log n}_{k}) \tag{1.46}$$

Note that there are recurrence relations that cannot be solved with any of the methods described above.

Example 33. *Consider the following recurrence relation:*

$$T(n) = \left\{ \begin{array}{ll} c & \textit{if } n \leq 1 \\ T(n-b) + n^2 & \textit{if } n > 1 \end{array} \right\} \tag{1.47}$$

we can replace n with $bk = n$ in eq.(1.43) and obtain:

$$T(bk) = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ T(bk-b) + b^2k^2 & \text{if } n > 1 \end{array} \right\} \tag{1.48}$$

If we also replace $T(bk)$ with $S(k) = T(bk)$ we obtain:

$$\underbrace{S(k)}_{T(bk)} = \left\{ \begin{array}{ll} c & \text{if } n \leq 1 \\ \underbrace{S(k-1)}_{T(bk-b)} + b^2k^2 & \text{if } n > 1 \end{array} \right\} \tag{1.49}$$

which has solution

$$S(k) = (c - b^2) + \sum_{i=1}^{i \leq k} b^2 i^2 = (c - b^2) + \frac{1}{6} b^2 k(k+1)(2k+1) \in \Theta(k^3)$$

therefore

$$T(n) = T(bk) = S(k) \in \Theta(k^3) = \Theta(n^3/b^3) = \Theta(n^3)$$

Other types of recurrence relations (type V)

There are many types of recurrence relations and many of them do not find in the categories listed above. They are not covered in this class.

Useful theorems about recurrence relations

$$\begin{aligned} O(f(n)) + \Theta(g(n)) &= \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \\ \Theta(f(n)) + \Theta(g(n)) &= \Theta(g(n)) \text{ iff } f(n) \in O(g(n)) \\ \Omega(f(n)) + \Theta(g(n)) &= \Omega(f(n)) \text{ iff } f(n) \in \Omega(g(n)) \end{aligned}$$

How to apply:

$$T(n) = [\underbrace{n^2 + n + 3}_{\Theta(n^2)} + \underbrace{e^n - \log n}_{\Theta(e^n)}] \in \Theta(e^n) \text{ because } n^2 \in O(e^n)$$

Example: factorial1

```
def factorial1(n):
    if n==0:
        return 1
    else:
        return n*factorial1(n-1)
```

$$T(n) = T(n-1) + 1 \Rightarrow T(n) \in \Theta(n) \Rightarrow \texttt{factorial1} \in \Theta(n)$$

Example: recursive0

```
def recursive0(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*n*recursive0(n-1)
```

$$T(n) = T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(n^2) \Rightarrow \texttt{recursive0} \in \Theta(n^3)$$

Example: recursive1

```
def recursive1(n):
    if n==0:
        return 1
    else:
        loop3(n)
        return n*recursive1(n-1)*recursive1(n-1)
```

$$T(n) = 2T(n-1) + P_2(n) \Rightarrow T(n) \in \Theta(2^n) \Rightarrow \texttt{recursive1} \in \Theta(2^n)$$

Example: recursive2

```
def recursive2(n):
    if n==0:
        return 1
    else:
        a=factorial0(n)
        return a*recursive2(n/2)*recursive1(n/2)
```

$$T(n) = 2T(n/2) + P_1(n) \Rightarrow T(n) \in \Theta(n \log n) \Rightarrow \texttt{recursive2} \in \Theta(n \log n)$$

Example: binarySearch

```
def binary_search(array,x):
    low=0
    high=len(array)-1
    while high>=low:
        mid=(high=low)/2
        if array[mid]<x:
            low=mid+1
        elif array[mid]>x:
            high=mid-1
        else:
            return mid
    return None
```

$$T(n) = T(n/2) + 1 \Rightarrow \texttt{binarySearch} \in O(\log n)$$

1.4.4. MergeSort

Consider the following sorting algorithm:

```
def Merge(A,p,q,r):
    B=[]
    i=p
    j=q
    while true:
        if A[i]<=A[j]:
            B.append(A[i])
            i=i+1
        else:
            B.append(A[j])
            j=j+1
        if i==q:
            while j<r:
                B.append(A[j])
                j=j+1
            break
        if j==r:
            while i<q:
                B.append(A[i])
                i=i+1
            break
    A[p:r]=B

def MergeSort(A, p=0, r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=int((p+r)/2)
        MergeSort(A,p,q)
        MergeSort(A,q,r)
        Merge(A,p,q,r)
```

The MergeSort follows a **Divide-and-Conquer** approach (see definition below). A Divide-and-conquer algorithm is a recursive algorithm and running time

of recursive algorithms is determined by solving the corresponding recurrence relation.

Let assume that the running time of the MergeSort is $T(n)$ where $n = \texttt{len(A)}$. It is easy to prove that $T_{\text{Merge}}(n) = cn$ since the Merge function contains only one loop. Therefore at each step

$$T(n) = \left\{ \begin{array}{ll} 1 & \text{if } n \leq 1 \\ 2T(n/2) + T_{\text{Merge}}(n) & \text{if } n > 1 \end{array} \right\} \tag{1.50}$$

1.4.5. Order of growth of algorithms

Given an algorithm, A, we can determine three characteristics functions:

- $T_{worst}(n)$: the running time in the worst case
- $T_{best}(n)$: the running time in the best case
- $T_{average}(n)$: the running time in the average case

If $T_{worst}(n) \in O(f(n))$ or $T_{worst}(n) \in \Theta(f(n))$ we say that the algorithms A is $O(f(n))$.

If $T_{best}(n) \in \Omega(f(n))$ or $T_{best}(n) \in \Theta(f(n))$ we say that the algorithms A is $\Omega(f(n))$.

If the algorithm A is $O(f(n))$ and is $\Omega(f(n))$ we say that A is $\Theta(f(n))$.

The order of growth in the average case is a very important measure of the efficiency of the algorithm but it does enter in the definitions above.

2. Data Structures and Algorithms

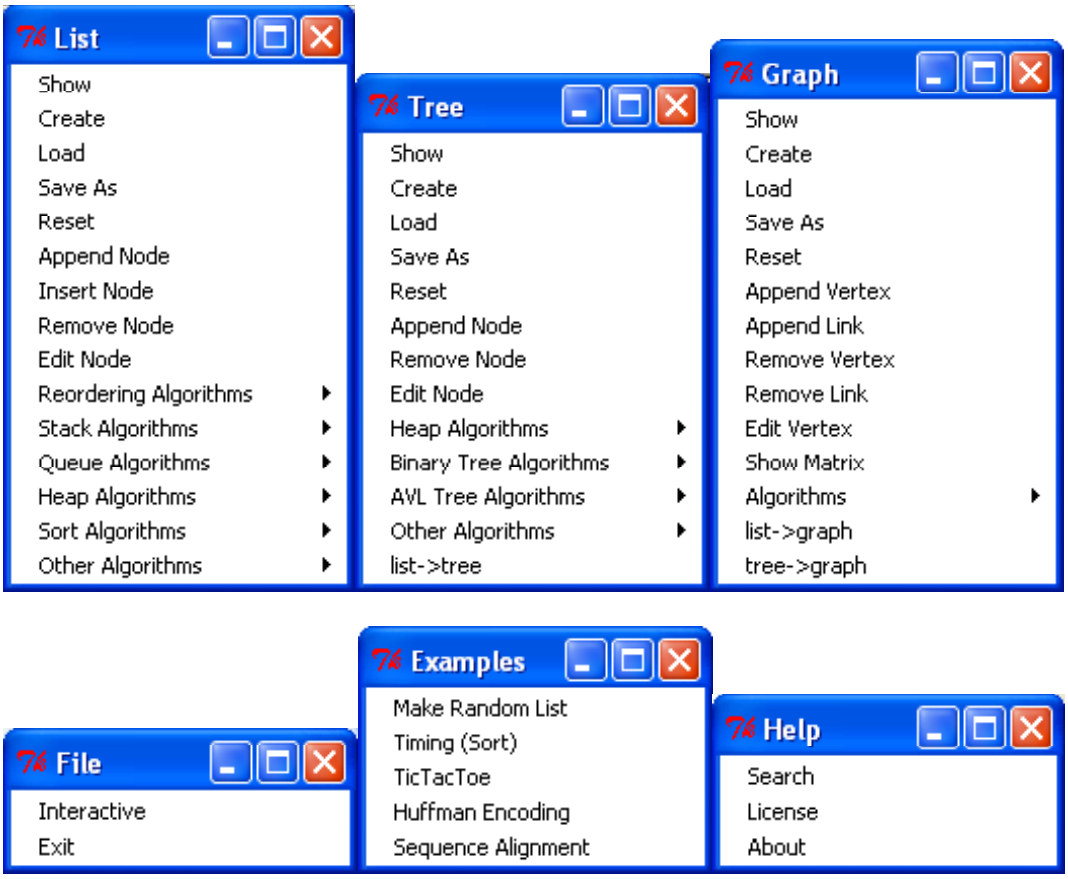
2.1. CSC321 program

The CSC321 program can be downloaded from:

http://www.phoenixcollective.org/mdp/csc321.exe

The CSC321 program is implemented in Python and based on the Tkinter graphic library and the Python Mega Widgets (Pmw). Python is an interpreted language invented by Guido van Rossum. This language was chosen because its syntax closely resembles the syntax commonly used to write pseudo-codes. Moreover Python includes, as basic types, structures such as tuples, lists and dictionaries. The version of the program distributed with these notes is packaged for Windows. Versions for other platforms are available upon request.

The main menus are



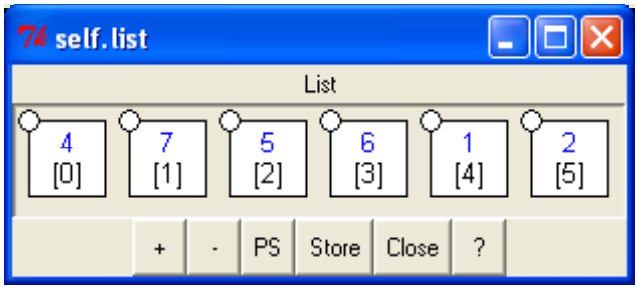
(2.1)

2.1.1. Lists

To start creating a list choose [**List**][**Create**] and type a Python list, for example:

$$[4, 7, 5, 6, 1, 2]$$

which is displayed as



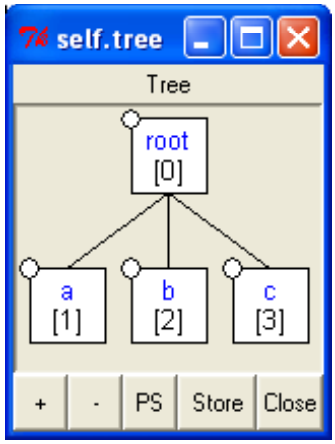
Note that each Node (in a list or a tree or a graph) can have different attributes: a **value**, a **name**, a **color**, etc. By default, when a list, a tree or a graph is created, only the value of the Nodes is specified ('root', 'a', 'b', 'c', 3, 4, 5). One should not confuse the value of a Node (for example 4 in the example) with its **location** ([0] in the example). The location is an index number which is automatically generated and is unique for each Node in a list, tree or graph. While the value of a node is not necessarily unique, the location index is unique.

2.1.2. Trees

In CSC321 a tree is implemented as a list of lists. For example, create the following list:

$$['root','a','b','c']$$

which is displayed as:

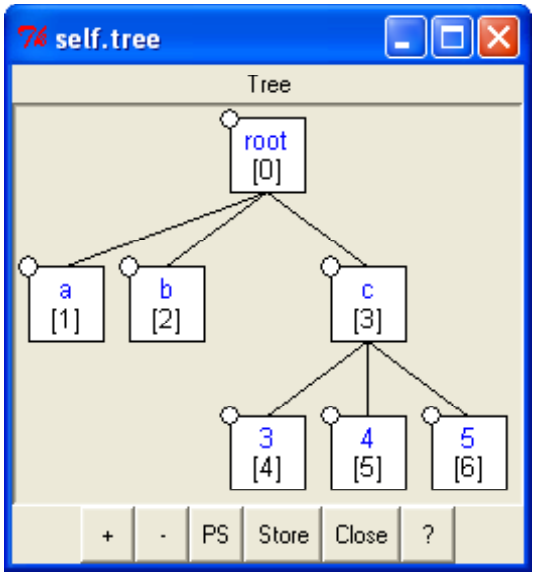


(2.2)

To add children Nodes to the Node c, one may create a new list where Node c itself is replaced by a subtree. For example, create the following list:

$$['root','a','b',['c',3,4,5]]$$

which is displayed as:



(2.3)

In general each Node in a tree can have an arbitrary number of children. It is possible to specialize a tree by imposing some constraints on it. In this course we examine three types of special trees:

- **Heaps**
- **Binary Trees**
- **AVL Tree**

2.1.3. Graphs

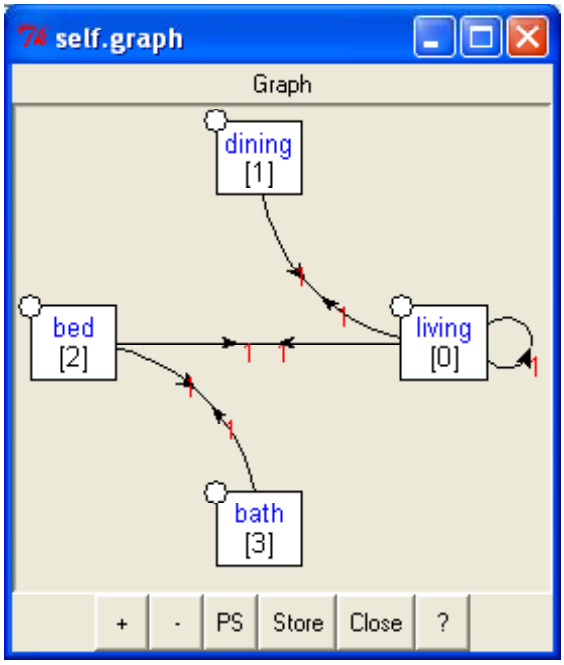
A graph is a collection of Nodes (with the same attributes as List Nodes and Tree Nodes) plus a collection of links between Nodes. In general links may have directions (for example Node [0] may be linked to Node [1] but not vice versa). Moreover each Node may be linked to itself (Node [0] may be linked to Node[0]).

In CSC321 a graph is implemented as a List of Lists where the element 0 of each sub-list is the value of the corresponding Node. The other elements of each sub-list are the location indices of the Nodes connected to the Node itself (**Adjacency List Representation**).

To start creating a graph choose [**Graph**][**Create**] and type a graph, for example:

$$[['living','dining','bed','bath'],[[0,0],[0,1],[1,0],[0,2],[2,0],[2,3],[3,2]]]$$

that represents the topology of a one bedroom apartment. This graph is displayed as:



(2.4)

A graphs is represented as two lists: a list of vertices/nodes (for example: $V = ['living','dining','bed','bath']$) and a list of links.

- The living room (Node [0]) is connected to itself [0,0] (has an island kitchen?) to the dining room [0,1] and to the bedroom [0,2].
- The dining room (Node [1]) is connected only to the living room [1,0].

- The bedroom (Node [2]) is connected to the living room [2,0] and the bathroom [2,3].
- The bathroom (Node [3]) is connected to the bedroom only [3,2].

In the representation of the graph the number 1 attached to each link represents the length of the link (this is one by default).

2.2. Algorithms

2.2.1. Algorithm types: defintions

Definition 34. *The* **Divide-and-Conquer** *is a method of designing algorithms that (informally) proceeds as follows: Given an instance of the problem to be solved, split this into several, smaller, sub-instances (of the same problem) independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance. This description raises the question: By what methods are the sub-instances to be independently solved? The answer to this question is central to the concept of the Divide-and-Conquer algorithm and is a key factor in gauging their efficiency. The solution depends on the problem.*

Definition 35. Dynamic Programming *is a paradigm that is most often applied in the construction of algorithms to solve a certain class of optimization problems. That is problems which require the minimization or maximization of some measure. One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since identical sub-instances may arise. The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice. The method usually accomplishes this by maintaining a table of sub-instance results. We say that Dynamic Programming is a Bottom-Up technique in which the smallest sub-instances are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances. In contrast, we say that the Divide-and-Conquer is a Top-Down technique. One case of Top-Down approach, called* **memoization,** *that is normally included under the umbrella of Dynamic Programming. The reason is that the memoization approch uses tables to store intermediate results.*

Definition 36. Greedy algorithms *work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. This 'take what you can get now' strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the global optimum. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a suboptimal solution. If the best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer. Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.*

There are other types of algorithms that do not follow in any of the above categories. One is, for example, backtracking. Backtracking is not covered in this course.

2.2.2. Sorting algorithms

The T_{memory} in the following paragraph is the size of the extra memory required by the algorithm (i.e. space taken by local variables) and it does no include stape for the input.

InsertionSort (divide and conquer)

$$\begin{aligned} T_{best} &\in \Theta(n) \\ T_{average} &\in \Theta(n^2) \\ T_{worst} &\in \Theta(n^2) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

MergeSort (divide and conquer)

$$\begin{aligned} T_{best} &\in \Theta(n \log n) \\ T_{average} &\in \Theta(n \log n) \\ T_{worst} &\in \Theta(n \log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

MergeSortDP (dynamic programming)

```
def MergeSortDP(A):
    blocksize=1
    listsize=len(A)
    while blocksize<listsize:
        for p in range(0, listsize, 2*blocksize):
            q=p+blocksize
            r=min(q+blocksize, listsize)
            if r>q:
                Merge(A,p,q,r)
        blocksize=2*blocksize
```

$$\begin{aligned} T_{best} &\in \Theta(n \log n) \\ T_{average} &\in \Theta(n \log n) \\ T_{worst} &\in \Theta(n \log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

QuickSort (divide and conquer)

There are at least 2 possible implementations of partition and quicksort. The following:

```
def Partition(A,i,j):
    x=A[i]
    while true:
        j=j-1
        while A[j]>x:
            j=j-1
        while A[i]<x:
            i=i+1
        if i<j:
            (A[i],A[j])=(A[j],A[i])
        else:
            return j+1
```

```
def QuickSort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=Partition(A,p,r)
        QuickSort(A,p,q)
        QuickSort(A,q,r)
```

and the following, based on the so called Lomuto partition:

```
def Partition(A,i,j):
    x=A[i]
    h=i
    for k in range(i+1,j):
        if A[k]<x:
            h=h+1
            A[h],A[k]=A[k],A[h]
    A[h],A[i]=A[i],A[h]
    return h

def QuickSort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=Partition(A,p,r)
        QuickSort(A,p,q)
        QuickSort(A,q+1,r)
```

In both cases:

$$\begin{aligned} T_{best} &\in \Theta(n \log n) \\ T_{average} &\in \Theta(n \log n) \\ T_{worst} &\in \Theta(n^2) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

RandomizedQuickSort

The Randomized partition is similar to partition but uses a random element of the array $A[p \leq i < r]$ as pivot.

```
def RandomizedPartition(A,p,r):
    i=randint(p,r-1)
    (A[p],A[i])=(A[i],A[p])
    return Partition(A,p,r)

def RandomizedQuickSort(A,p=0,r=-1):
    if r is -1:
        r=len(A)
    if p<r-1:
        q=RandomizedPartition(A,p,r)
        RandomizedQuickSort(A,p,q)
        RandomizedQuickSort(A,q+1,r)
```

$$\begin{aligned} T_{best} &\in \Theta(n \log n) \\ T_{average} &\in \Theta(n \log n) \\ T_{worst} &\in \Theta(n^2) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

HeapSort

Definition 37. *A* **leaf** *is a node of a tree with no children nodes.*

Definition 38. *A* **binary tree** *is a tree where each node has no more than two children.*

Definition 39. *A* **heap** *is a special kind of binary tree. It has two properties that are not generally true for other trees:*

Completeness: *The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.*

Heapness: *The item in the tree with the highest value is at the top of the tree, and the same is true for every subtree.*

A Heap is defined by the above relations:

```
def Parent(i):
    return int((i-1)/2)

def Left(i):
    return 2*i+1

def Right(i):
    return 2*i+2
```

Heaps have two important applications: HeapSort and priority queues (see later)

```
def Heapify(A,i,heapsize=-1):
    if heapsize<0:
        heapsize=len(A)
    left=2*i+1
    right=2*i+2
    if left<heapsize and A[left]>A[i]:
        largest=left
    else:
        largest=i
    if right<heapsize and A[right]>A[largest]:
        largest=right
    if largest!=i:
        (A[i], A[largest])=(A[largest], A[i])
        Heapify(A,largest,heapsize)

def BuildHeap(A):
    heapsize=len(A)
    i=int(heapsize/2)
    while i>0:
        i=i-1
        Heapify(A,i)

def HeapSort(A):
    BuildHeap(A)
```

```
    heapsize=len(A)
    i=heapsize
    while i>1:
        i=i-1
        (A[0],A[i])=(A[i],A[0])
        heapsize=heapsize-1
        Heapify(A,0,heapsize)
```

$$\begin{aligned} T_{best} &\in \Theta(n) \\ T_{average} &\in \Theta(n \log n) \\ T_{worst} &\in \Theta(n \log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

CountingSort

he CountingSort algorithm is the only algorithm, among the ones considered here, that uses the array values `A[i]` as index for another array `C[A[i]]`. For this reason this algorithm only works if the elements to be sorted are integers but it presents the advantage of running in linear time in the range spanned by the array/list elements k. The implementation below assumes elements are positive integers.

```
def CountingSortSmall(A):
    if Minimum(A)<0:
        raise 'CountingSort List Unbound'
    n=len(A)
    C=[]
    k=Maximum(A)+1
    for j in range(k):
        C.append(0)
    for j in range(n):
        C[A[j]]=C[A[j]]+1
    i=0
    for j in range(k):
        while C[j]>0:
            A[i]=j
```

```
            C[j]=C[j]-1
            i=i+1
```

Another alternative implementation is the following:

```
def CountingSort(A):
    if Minimum(A)<0:
        raise 'CountingSort List Unbound'
    B=[]
    C=[]
    k=Maximum(A)+1
    for j in range(k):
        C.append(0)
    for j in range(len(A)):
        B.append(0)
        C[A[j]]=C[A[j]]+1
    for i in range(1,k):
        C[i]=C[i]+C[i-1]
    j=len(A)-1
    while j>=0:
        B[C[A[j]]-1]=A[j]
        C[A[j]]=C[A[j]]-1
        j=j-1
    return B
```

If we define $k = max(A) - min(A) + 1$ and $n = len(A)$ we see:

$$\begin{aligned} T_{best} &\in \Theta(k+n) \\ T_{average} &\in \Theta(k+n) \\ T_{worst} &\in \Theta(k+n) \\ T_{memory} &\in \Theta(k+n) \end{aligned}$$

2.2.3. Searching algorithms

Priorities queues

A priority queue is a cointainer implemented as a Heap. We can put items in the proprity queue and we can easily extract the item with maximum value (the top of the heap).

```
def HeapExtractMax(A):
    if len(A)<1:
        raise 'Heap Underflow'
    max=A[0]
    A[0]=A[len(A)-1]
    del A[len(A)-1]
    Heapify(A,0)
    return max
```

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(\log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

```
def HeapInsert(A,node):
    A.append(node)
    i=len(A)-1
    while i>0 and A[Parent(i)]<A[i]:
        (A[i],A[Parent(i)])=(A[Parent(i)],A[i])
        i=Parent(i)
```

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(\log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

Binary search trees

A binary search tree is a binary tree and, for each node, the values of left descendant nodes are lower than the value of parent node and the values of the right descendant nodes are higher than the value of the parent node. A binary search tree is a container that allows us to insert items and seach them rapidly, $O(\log n)$, accordingly with their values.

```
def InorderTreeWalk(tree,list=[]):
    if isNullTree(tree):
        return
    InorderTreeWalk(tree[leftchild],list)
    list.append(tree[rootnode])
    InorderTreeWalk(tree[rightchild],list)
    return list

def TreeSearch(tree, k):
    if isNullTree(tree):
        return
    if k < tree[rootnode]:
        return TreeSearch(tree[leftchild],k)
    elif k > tree[rootnode]:
        return TreeSearch(tree[rightchild],k)
    else:
        return tree[rootnode]

def TreeMinimum(tree):
    while not isNullTree(tree[leftchild]):
        tree=tree[leftchild]
    return tree

def TreeMaximum(tree):
    while not isNullTree(tree[rightchild]):
        tree=tree[rightchild]
    return tree
```

Each of these algorithms goes like:

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

```
def TreeInsertLeaf(x, k):
    if isNullTree(x):
```

```
        k.parent=None
        x[:]=BinaryTree(k)
    else:
        k.parent=x
        if k<x[rootnode]:
            x[leftchild]=BinaryTree(k)
        else:
            x[rightchild]=BinaryTree(k)

def TreeInsert(tree, k):
    if isNullTree(tree):
        TreeInsertLeaf(tree,k)
        return tree
    x=tree
    while not isNullTree(x):
        y=x
        if k < x[rootnode]:
            x=x[leftchild]
        elif k > x[rootnode]:
            x=x[rightchild]
        else:
            return tree
    x=y
    TreeInsertLeaf(x, k)
    return tree
```

The TreeInsert algorithms grows like:

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

```
def TreeDelete(z):
    if isNullTree(z[leftchild]) and isNullTree(z[rightchild]):
        TreeLeafDelete(z)
    elif isNullTree(z[leftchild]):
```

```
    x=z[rightchild]
    TreeNodeReplaceWithChildren(z,x)
elif isNullTree(z[rightchild]):
    x=z[leftchild]
    TreeNodeReplaceWithChildren(z,x)
else:
    x=TreeMinimum(z[rightchild])
    TreeNodeReplaceWithoutChildren(z,x)
```

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

AVL trees (balanced binary trees)

AVL trees are binary search trees that are rebalanced after each insertion/deletion. They are rebalanced in such a way that for each node the height of the left subtree minus height of the right subtree is -1,0 or +1. The rebalance operation can be done in $O(\log n)$.

$$\begin{aligned} T_{best} &\in \Theta(1) \\ T_{average} &\in \Theta(\log n) \\ T_{worst} &\in \Theta(\log n) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

k-trees, B-trees and Red-black trees

Until now we have considered binary trees (each node has 2 childern and store 1 value). One can generalize this to $k$trees where each node has k children and store more than one value.

2.2.4. Graph algorithms

Definition 40. *A* **graph** *G is a pair of sets (V, E) where V is a set of vertices (or Nodes) and E a set of links between the couple of vertices.*

Definition 41. *In general a link, indicated with the notation e_{ij}, connecting vertex i with vertex j is called a* **directed link**. *If the link has no direction $e_{ij} = e_{ji}$ it is called an undirected link. A graph that contains only undirected links is an* **undirected graph**, *otherwise it is a* **directed graph**.

Definition 42. *A* **walk** *is an alternating sequence of vertices and links, with each link being incident to the vertices immediately preceding and succeeding it in the sequence. A* **trail** *is a walk with no repeated links.*

Definition 43. *A* **path** *is a walk with no repeated vertices. A walk is closed if the initial vertex is also the terminal vertex.*

Definition 44. *A* **cycle** *is a closed trail with at least one edge and with no repeated vertices except that the initial vertex is the terminal vertex.*

Definition 45. *A graph that contains no cycles is an* **acyclic graph**. *Any connected acyclic undirected graph is also a* **tree**.

Definition 46. *A* **loop** *is a one link path connecting a vertex with itself.*

Definition 47. *A non-null graph is* **connected** *if, for every pair of vertices, there is a walk whose ends are the given vertices. Let us write $i \tilde{} j$ if there is path from i to j. Then $\tilde{}$ is an equivalence relation. The equivalence classes under $\tilde{}$ are the vertex sets of the connected components of G. A connected graph is therefore a graph with exactly one connected component.*

Definition 48. *A graph is called* **complete** *when every pair of vertices is connected by a link (or edge).*

Definition 49. *A* **clique** *of a graph is a subset of vertices in which every pair is an edge.*

Definition 50. *The* **degree** *of a vertex of a graph is the number of edges incident to it.*

Definition 51. *If i and j are vertices, the* **distance** *from i to j, written d_{ij}, is the minimum length of any path from i to j. In an undirected graph, this induces a metric.*

Definition 52. *The* **eccentricity**, *$e(i)$, of the vertex i is the maximum value of d_{ij}, where j is allowed to range over all of the vertices of the graph.*

Definition 53. *The* **subgraph** *of G induced by a subset W of its vertices V ($W \subseteq V$) is the graph formed by the vertices in W and all edges whose two endpoints are in W.*

Breadth first search

```
def Adjacents(u,graph):
    vertices=graph[0]
    links=graph[1]
    a=[]
    for link in links:
        if vertices[link.source] is u:
            a.append(link)
    return a

def BreadthFirstSearch(graph,start):
    vertices=graph[0]
    links=graph[1]
    blacknodes=[]
    graynodes=[start]
    t=0
    vertices[start].time_discovered=t
    while len(graynodes)>0:
        k=Dequeue(graynodes)
        for link in Adjacents(vertices[k],graph):
            j=link.destination
            if not j in blacknodes and not j in graynodes:
                Enqueue(graynodes,j)
                vertices[j].time_discovered=t
        blacknodes.append(k)
```

```
        vertices[k].time_found=t
        t=t+1
    return blacknodes
```

The function `Adjacents(vertex,graph)` returns a list of links starting from vertex. The present implementation of the Adjacents function is not very efficient (sorry!). The BFS algorithm goes like:

$$\begin{aligned} T_{best} &\in \Theta(n_E + n_V) \\ T_{average} &\in \Theta(n_E + n_V) \\ T_{worst} &\in \Theta(n_E + n_V) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

Depth first search

```
def DepthFirstSearch(graph,start):
    vertices=graph[0]
    links=graph[1]
    blacknodes=[]
    graynodes=[start]
    t=0
    vertices[start].time_discovered=t
    while len(graynodes)>0:
        k=Pop(graynodes)
        for link in Adjacents(vertices[k],graph):
            j=link.destination
            if not j in blacknodes and not j in graynodes:
                Push(graynodes,j)
                vertices[j].time_discovered=t
        blacknodes.append(k)
        vertices[k].time_found=t
        t=t+1
    return blacknodes
```

The DFS algorithm goes like:

$$T_{best} \in \Theta(n_E + n_V)$$

$$
\begin{aligned}
T_{average} &\in \Theta(n_E + n_V) \\
T_{worst} &\in \Theta(n_E + n_V) \\
T_{memory} &\in \Theta(1)
\end{aligned}
$$

Minimum spanning tree: Kruskal

```
def FindSet(S,i):
    for j in range(len(S)):
        if i in S[j]:
            return j
    return None

def UnionSets(S,i,j):
    for k in range(len(S)):
        if i in S[k]:
            A=S[k]
            break
    del S[k]
    for k in range(len(S)):
        if j in S[k]:
            S[k]=S[k]+A
            break
    return

def MSTKruskal(graph):
    vertices=graph[0]
    links=graph[1]
    A=[]
    S=[]
    for i in range(len(vertices)):
        S.append([i])
    links.sort()
    for link in links:
        i=link.source
        j=link.destination
        if FindSet(S,i)!=FindSet(S,j):
```

```
            A=A+[(i,j)]
            UnionSets(S,i,j)
    return A
```

Functions FindSet and UnionsSets are described in the book (section 21.1)
The Kruskal algorithm goes like:

$$\begin{aligned} T_{worst} &\in \Theta(n_E \log n_V) \\ T_{memory} &\in \Theta(n_E) \end{aligned}$$

Minimum spenning tree: Prim

```
def ExtractMin(Q):
    i=0
    for j in range(1,len(Q)):
        if Q[j].d<Q[i].d or Q[i].d==Infinity:
            i=j
    u=Q[i]
    del Q[i]
    return u

def MSTPrim(graph, r):
    vertices=graph[0]
    links=graph[1]
    Q=[]
    for i in range(len(vertices)):
        vertices[i].d=Infinity
        Q.append(vertices[i])
    Q[r].parent=None
    Q[r].d=0

    while len(Q)>0:
        u=ExtractMin(Q)
        for link in Adjacents(u,graph):
            v=vertices[link.destination]
            w=link.length
            if v in Q and w<v.d:
```

```
            v.parent=u
            v.d=w
```

For maximum speed Q should be implemented as a priority queue or, better, as a Fibonacci heap (not covered). Here adopted a slower implementation.

The Prim algorithm, when using a priority queue for Q, goes like:

$$\begin{aligned} T_{worst} &\in \Theta(n_E + n_V \log n_V) \\ T_{memory} &\in \Theta(n_E) \end{aligned}$$

Single source shortest path: Dijkstra

```
def Dijkstra(graph, r):
    vertices=graph[0]
    links=graph[1]
    Q=[]
    for i in range(len(vertices)):
        vertices[i].d=Infinity
        Q.append(vertices[i])
    Q[r].parent=None
    Q[r].d=0

    while len(Q)>0:
        u=ExtractMin(Q)
        for link in Adjacents(u,graph):
            v=vertices[link.destination]
            w=link.length
            if v in Q and w+u.d<v.d:
                v.parent=u
                v.d=w+u.d
```

The Dijkstra algorithm goes like:

$$\begin{aligned} T_{worst} &\in \Theta(n_E + n_V \log n_V) \\ T_{memory} &\in \Theta(n_E) \end{aligned}$$

Single source shortest path: Bellman-Ford

```
def InitializeSingleSource(graph,s):
    vertices=graph[0]
    for vertex in vertices:
        vertex.d=Infinity
        vertex.parent=None
    vertices[s].d=0

def Relax(u,v,graph):
    vertices=graph[0]
    links=graph[1]
    for link in Adjacents(u,graph):
        if v is vertices[link.destination]:
            w=link.length
            if not u.d is Infinity:
                if v.d>u.d+w:
                    v.d=u.d+w
                    v.parent=u
            return

def BellmanFord(graph, s):
    vertices=graph[0]
    links=graph[1]
    InitializeSingleSource(graph,s)
    for i in range(1,len(vertices)-1):
        for link in links:
            u=vertices[link.source]
            v=vertices[link.destination]
            Relax(u,v,graph)
    for link in links:
        u=vertices[link.source]
        v=vertices[link.destination]
        w=link.length
        if u.d is Infinity:
            d=1e10
        else:
```

```
            d=u.d+w
        if v.d>d:
            return false
    return true
```

The Bellman-Ford algorithm goes like:

$$\begin{aligned} T_{worst} &\in \Theta(n_E n_V) \\ T_{memory} &\in \Theta(1) \end{aligned}$$

2.2.5. More examples

Fibonacci number and memoization

Let's consider a divide-and-conquer approach (recursive, top-down) to the probelm of determining fibonacci numbers:

```
def FibonacciRecursive(n):
    print 'calling FibonacciRecursive(%i)' % (n)
    if n is 0 or n is 1:
        return 1
    return FibonacciRecursive(n-1)+FibonacciRecursive(n-2)
```

This solution works but it is very slow because the function `FibonacciRecursive` calls itself twice and the two branches contain common subproblems. Therefore the same subproblems are solved multiple times. The fact that a problem, for example `FibonaccyRecursive(3)`, contains solution of subproblems, for example `FibonacciRecursive(2)`, is referred to as **optimal substructure**. If a problem has optimal substracture then we can use a dynamic programming approach:

```
def FibonacciDynamicProgramming(n):
    print 'calling FibonacciDynamicProgramming(%i)' % (n)
    if n is 0 or n is 1:
        return 1
    a=1
    b=1;
    for i in range(1,n):
        (a,b) = (b, a+b)
        print '    fibonacci(',i+1,')=', b
    return b
```

Sometime it is convenient, easier, to mantain a top-down recursive approach and store in a table the subproblems that have been solved already. So that if they appear again one can look them up in the table instead of recomputing their solution. This approach is called **memoization**:

```
def FibonacciMemoize(n, table={0:1, 1:1}):
    print 'calling FibonacciMemoize(%i)' % (n)
    if table.has_key(n):
        print '    value is in table'
        return table[n]
    b=FibonacciMemoize(n-1,table)+FibonacciMemoize(n-2,table)
    table[n]=b
    print '    computing fibonacci(',n,')=', b, ' and storing in table'
    return b
```

Huffman encoding

Definition 54. Shannon-Fano encoding *(also known as* **minimal prefix code***).* *In this encoding each character in a string is mapped into a sequence of bits so characters that appear with less frequency are encoded with in a longer sequence of bits while characters that appear with more frequency are encoded with a shorter sequence.*

Definition 55. Huffman Encoding. *The optimal choice for a minimal prefix code). The choice is built in the following way. We associate a tree to each character in the string to compress. Each tree is a trivial tree containing only one node: the root node. We then associate to the root node the frequency of the character representing the tree. We then extract from the list of trees the two trees with lowest frequency: t1 and t2. We form a new tree t3, we attach t1 and t2 to t3 and we associate a frequency to t3 equal to the sum of the frequencies of t1 and t2. We repeat this operation until the list of trees containing only one tree. At this point we associate a sequence of bits to each node of the tree. Each bit corresponds to one level on the tree. The more frequent characters end up being closer to the root and are encoded with few bits, while rare characters are far from the root and encoded with more bits.*

PKZIP, ARJ, ARC, JPEG, MPEG3 (mp3), MPEG4 and a other programs and compressed file formats all use the Huffman coding algorithm for compressing

strings. Note that Huffman is a compression algorithm with no-information-loss. In the JPEG and MPEG compression algorithms Huffman if combined with some form of cut of the Fourier specturm (for example MP3 is an audio compression format where frequencies below 2KHz are dumped and not compressed because they are not audible). Therefore the JPEG and MPEG formats are referred to as compression with information-loss.

Longest common subsequence

Given two sequencies of characters, S_1 and S_2, this is the problem of determing the length of the longest common sub-sequence, LCS, that is a sub-suence of both S_1 and S_2.

Why might we want to solve the longest common subsequence problem? There are several motivating applications.

- **Molecular biology**. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four submolecules forming DNA. When biologists find a new sequences, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

- **File comparison**. The Unix program `diff` is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files and displays the set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character.

- **Spelling Correction**. If a text contains a word, w, that is not in the dictionary, a 'close' word, i.e. one with a small edit distance to w, may be suggested as a correction. Transposition errors are common in written text. A transposition can be treated as a deletion plus an insertion, but a simple variation on the algorithm can treat a transposition as a single point mutation.

- **Speech Recognition**. Algorithms similar to the LCS are used in some speech recognition systems: find a close match between a new utterance

and one in a library of classified utterances.

Let's start with some simple observations about the LCS problem. If we have two strings, say "ATGGCACTACGAT" and "ATCGAGC", we can represent a subsequence as a way of writing the two so that certain letters line up:

```
ATGGCACTACGAT
||  |  |   |
ATCG AG  C
```

From this we can observe the following simple fact: if the two strings start with the same letter, it's always safe to choose that starting letter as the first character of the subsequence. This is because, if you have some other subsequence, represented as a collection of lines as drawn above, you can "push" the leftmost line to the start of the two strings, without causing any other crossings, and get a representation of an equally-long subsequence that does start this way.

On the other hand, suppose that, like the example above, the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed.

Finally, observe that once we've decided what to do with the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. Therefore we can solve it recursively.

Rather than finding the subsequence itself, it turns out to be more efficient to find directly the length of the longest subsequence. Then in the case where the first characters differ, we can determine which subproblem gives the correct solution by solving both and taking the max of the resulting subsequence lengths. Once we turn this into a dynamic programing algorithm we get the following:

```
def LCS(X, Y)
    m=len(X)
    n=len(Y)
    c=[]
    for i in range(0,m):
        c.append([])
        for j in range(0,n):
            c[i].append(0)
            if X[i] == Y[j]:
               if i==0 or j==0:
```

```
                c[i][j]=1
            else:
                c[i][j] = c[i-1][j-1] + 1
        else:
            if i==0 or j==0:
                c[i][j]=0
            else:
                c[i][j] = max(c[i][j-1],c[i-1][j])
    return c[m-1][n-1]
```

The algorithms can be shown to be $O(nm)$ (where m =len(X) and n =len(Y)).

Continuum Knapsack

The continuum Knapsack problem can be formulated as the problem of maximizing:

$$f(x) = a_0x_0 + a_1x_1 + ... + a_nx_n$$

given the contraint

$$b_0x_0 + b_1x_1 + ... + b_nx_n \leq c$$

where coefficients a_i, b_i and c are provided and $x_i \in [0, 1]$ are to be determined.

Using financial terms we can say that

- The set $\{x_0, x_1, ..., x_n\}$ forms a portfolio
- b_i is the cost of investment i
- c is the total investment capital available
- a_i is the expected return of investment for investment i
- $f(x)$ is the expected value of our portfolio $\{x_0, x_1, ..., x_n\}$

Here is the solving algorithm:

```
def ContinuumKnapsack(a,b,c):
    n=len(a)
    table=[]
    f=0.0
```

```
for i in range(n):
    y=a[i]/b[i]
    table.append((y,i))
table.sort()
table.reverse()
for (y,i) in table:
    quantity=min(c/b[i],1)
    x.append((i,quantity))
    c=c-b[i]*quantity
    f=f+a[i]*quantity
return (f,x)
```

This algorithm is dominated by the sort therefore

$$T_{worst}(x) \in O(n \log n)$$

Discrete Knapsack

The discrete Knapsack problem is very similar to the contiuum knapsack problem but $x_i \in \{0, 1\}$ (can only zero or one).

In this case a greedy approck does not apply and the problem is, in general NP complete. If we assume that c and b_i are all multiple of a finite factor ε then it is possible to solve the problem in $O(c/\varepsilon)$. Even when there is not a finite factor ε, we can always round c and b_i to some finite precision ε and we can conclude that, for any finite precision ε, we can solve the problem in linear time. The algorithm that solves this problem follows a dynamic programming approach.

We can re-formulate the problem in terms of simple capital budgeting problem. We have to invest \$5M. We assume ε =\$1M. We are in contact with 3 investment firms. Each of them offers a number of investment opportunites characterized by an investment cost $c[i, j]$ and an expected return of investment $r[i, j]$. The index i labels the investment firm and the index j label the different investment opportunities offered by the firm. We have to build a portfolio that maximizes the return of investment. We cannot select more than one investment for each firm and we cannot select fractions of investements.

Without loss of generality we will assume that

$$c[i, j] \leq c[i, j + 1] \text{ and } r[i, j] \leq r[i, j + 1]$$

which means that investment opportunites for each firm are sorted according with their cost.

Let's consider the following explicit case:

	Firm	$i=0$	Firm	$i=1$	Firm	$i=2$
proposal	$c[0,j]$	$r[0,j]$	$c[1,j]$	$r[1,j]$	$c[2,j]$	$r[2,j]$
$j=0$	0	0	0	0	0	0
$j=1$	1	5	2	8	1	4
$j=2$	2	6	3	9	-	-
$j=3$	-	-	4	12	-	-

(Table 1)

(table values are always multiple of ε =\$1M).

Notice that we can label each possible portfolio by a triplet $\{j_0, j_1, j_2\}$.

A straightforward way to solve this is to try all possibilities and choose the best. In this case, there are only $3 \times 4 \times 2 = 24$ possible portfolios. Many of these are infeasible (for instance, portfolio $\{2, 3, 0\}$ costs \$6M and we cannot afford it). Other portfolios are feasible, but very poor (like portfolio $\{0, 0, 1\}$ which is feasible but returns only \$4M)

Here are some disadvantages of total enumeration:

- For larger problems the enumeration of all possible solutions may not be computationally feasible.
- Infeasible combinations may not be detectable a priori, leading to inefficiency.
- Information about previously investigated combinations is not used to eliminate inferior, or infeasible, combinations (unless we use memoization but in this case the algorithm would not grow polynomially in memory space).

We can, instead, use a dynamic programming approach.

We break the problem into three stages and, at each stage we fill a table of optimal investments for each discerete amount on money. At each stage i we only consinder investements from firm i and the table during the previous stage.

So stage 0 represents the money allocated to firm 0, stage 1 the money to firm 1, and stage 2 the money to firm 2.

STAGE ZERO: we maximize the return of investement considering only offers from firm 0. We fill a table $f[0,k]$ with the maximum return of investment if we invest k million dollars on firm 0:

$$f[0,k] = \max_{j|c[0,j]<k} r[0,j] \tag{2.5}$$

k	$f[0,k]$
0	0
1	5
2*	6*
3	6
4	6
5	6

(2.6)

STAGE TWO: we maximize the retun of investment considering offers from firm 1 and the above table. We fill a table $f[1,k]$ with the maximum return of investment if we invest k million dollars on firm 0 and firm 1:

$$f[1,k] = \max_{j|c[1,j]<k} r[1,j] + f[0,k-c[0,j]] \tag{2.7}$$

k	$c[2,j]$	$f[0,k-c[0,j]]$	$f[1,k]$
0	0	0	0
1	0	1	5
2	2	0	8
3	2	1	9
4	3	1	13
5*	4*	1*	18*

(2.8)

STAGE THREE: we maximize the retun of investment considering offers from firm 2 and the above table. We fill a table $f[2,k]$ with the maximum return of investment if we invest k million dollars on firm 0, firm 1 and firm 2:

$$f[2,k] = \max_{j|c[2,j]<k} r[2,j] + f[1,k-c[1,j]] \tag{2.9}$$

k	$c[2,j]$	$f[1,k-c[1,j]]$	$f[2,k]$
0	0	0	0
1	0	1	5
2	2	0	8
3	2	1	9
4	1	3	13
5^*	2^*	3^*	18^*

(2.10)

The maximum return of investment with \$5M can is therefore \$18M. It can be achieved by investing \$2M on firm 2 and \$3M on firms 0 and 1. The optimal choice is marked with a star in each table. Note that in order to deteremine how much money have to be allocated in order to maximize the return of investment requires storing past tables in order to be able to lookup the solution to subproblems.

We can generalize eq.(2.7) and eq.(2.9) for any number of investment firms (decision stages):

$$f[i,k] = \max_{j|c[i,j]<k} r[i,j] + f[i-1, k-c[i-1,j]]$$

2.3. NP and NPC

Definition 56. *We say a problem is in P if it can be solved in polinomial time:* $T_{worst} \in O(n^\alpha)$ *for some* α.

Definition 57. *We say a problem is in NP if an input string can be verified to be a solution in polynomial time:* $T_{worst} \in O(n^\alpha)$ *for some* α.

Definition 58. *We say a problem is in co-NP if an input string can be verified not to be a solution in polynomial time:* $T_{worst} \in O(n^\alpha)$ *for some* α.

Definition 59. *We say a problem is in NPH (NP Hard) if it is harder than any other problem in NP.*

Definition 60. *We say a problem is in NPC (NP Complete) if it is in NP and in NPH. Consequences:*

$$\text{if } \exists x \mid x \in NPC \text{ and } x \in P \Rightarrow \forall y \in NP, y \in P$$

Open problems:

- co-NP $\stackrel{?}{=}$ NP
- co-NP $\cap$ NP $\stackrel{?}{=}$ P
- NPC $\stackrel{?}{=}$ NP

Part I

Appendices

2.4. Programs in Java

Example: loop0

```
public static void loop0(int n) {
   int i;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
   }
}
```

Example: loop1

```
public static void loop1(int n) {
   int i;
   for(i=0; i<n*n; i++) {
      System.out.println("i="+i);
   }
}
```

Example: loop2

```
public static void loop2(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
      for(j=0; j<n; j++) {
         System.out.println("j="+j);
      }
   }
}
```

Example: loop3

```
public static void loop3(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
```

```
      for(j=0; j<i; j++) {
         System.out.println("j="+j);
      }
   }
}
```

Example: loop4

```
public static void loop4(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
      for(j=0; j<i*i; j++) {
         System.out.println("j="+j);
      }
   }
}
```

Example: factorial0

```
public static int factorial0(int n) {
   int i, prod=1;
   for(i=1; i<=n; i++) prod=prod*n;
   return n;
}
```

Example: concatenate0

```
public static void concatenate0(int n) {
   for(int i=0; i<n*n; i++) System.out.println(""+i);
   for(int i=0; i<n*n*n; i++) System.out.println(""+i);
}
```

Example: concatenate1

```
public static void concatenate1(int n, int a) {
   if(a<0)
```

```
      for(int i=0; i<n*n; i++) System.out.println(""+i);
   else
      for(int i=0; i<n*n*n; i++) System.out.println(""+i);
}
```

Example: factorial1

```
public static int factorial1(int n) {
   if(n==0) return 1;
   else return n*factorial1(n-1);
}
```

Example: recursive0

```
public static int recursive0(int n) {
   if(n==0) return 1;
   else {
      loop3(n)
      return n*n*recursive0(n-1);
   }
}
```

Example: recursive1

```
public static int recursive1(int n) {
   if(n==0) return 1;
   else {
      loop3(n)
      return n*recursive1(n-1)*recursive1(n-1);
   }
}
```

Example: recursive2

```
public static int recursive2(int n) {
   if(n==0) return 1;
   else {
```

```
      int a;
      a=factorial0(n)
      return a*recursive2(n/2)*recursive2(n/2);
   }
}
```

Example: binarySearch

```
public static int binarySearch(Comparable[] a, Comparable x) {
   int low=0, high=a.length-1;
   while(high>=low) {
      int mid=(high-low)/2;
      if(a[mid].compareTo(x)<0) low=mid+1;
      else if(a[mid].compareTo(x)>0) high=mid-1;
      else return mid;
   }
   return -1;
}
```

3. Practice Exercises

3.1. Limits

$$\begin{aligned}
\lim_{n\to\infty} \frac{\sqrt{n^2+2}}{\sqrt{2n^2-1}} &= \frac{1}{\sqrt{2}} \\
\lim_{n\to\infty} \frac{2n^3+n^5}{5n^7+n^2} &= 0 \\
\lim_{n\to\infty} \frac{n^{1/3}+n^{5/9}}{2n^{1/2}-n^{1/10}} &= \infty \\
\lim_{n\to\infty} \frac{3n+\sqrt{n}\log n}{2n+\sqrt{n}e^{-n}} &= \frac{3}{2} \\
\lim_{n\to\infty} \frac{\sqrt{n+1}-1}{\sqrt{n+2}} &= 1 \\
\lim_{n\to\infty} \frac{n}{3^n-2^n} &= 0 \\
\lim_{n\to\infty} \frac{n^2}{n!} &= 0 \\
\lim_{n\to\infty} \frac{n^2+2^n+1}{n^2+5^n+2^n} &= 0 \\
\lim_{n\to\infty} \frac{n!}{n^{n/2}} &= \infty \\
\lim_{n\to\infty} \frac{\log(n^2+1)+\log(n^3+2)}{3\log(n+1)+\log(n)} &= \frac{5}{4} \\
\lim_{n\to\infty} \frac{\sqrt[2n]{n!}}{n} &= 0 \\
\lim_{n\to\infty} \frac{2^{n+8}}{\log(n)+2^n} &= 256
\end{aligned}$$

For some of these limits you may need

$$n! = \sqrt{2\pi n} n^n e^{-n} (1 + O(1/n))$$