# Lezione 12

## Bioinformatica

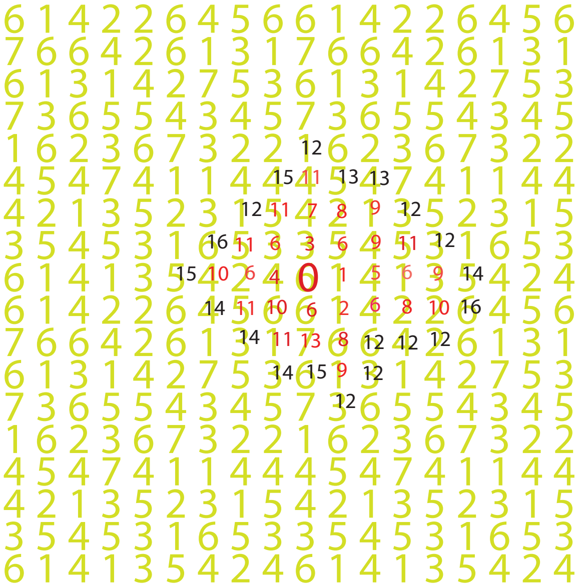
Mauro Ceccanti[‡] e Alberto Paoluzzi[†]

[†]Dip. Informatica e Automazione – Università “Roma Tre”
[‡]Dip. Medicina Clinica – Università “La Sapienza”

# Lezione 12: Algoritmo generazione dendritica

# Forward computation

## Example



# Basic tools

aaaaaaaaa

```
def colortable (n):
    """ To build a color lookup table with n elements
        Return a list of GLcolor
    """
    return [ GLcolor((array([1.0/n,0,0,0])*k).tolist()) for
        k in range(1,n+1) ]

colors = colortable(10)
size = 5
coords = [[i,j] for i in range(size) for j in range(size)]
table = STRUCT(map(item, coords))
numbers = STRUCT(map(mass, coords))
```

# Basic tools

aaaaaaaaa

```
def item((i,j)):
""" to generate a single colored cell of the table.
    Return a colored unit square translated in (i,j)."""
    transl = T([1,2])([i,j])
    if a[i,j] > 0: color = colors[a[i,j]]
    else: color = WHITE
    return COLOR(color)(transl(CUBOID([1,1])))

def mass ((i,j)):
""" To write the field value in position (i,j)
    Return a text suitably positioned in 2D space-"""
    transl = T([1,2])([i+0.5,j])
    m = TEXTWITHATTRIBUTES('centre',0.0,0.3,0.5,0.0) (
          str(abs(a[i,j])))
    return transl(T(2)(0.25)(m))
```

# Forward computation

## aaaaaaaaa

Algorithm FORWARD($A, h$)

**Require:** a (weighted) matrix $A$; a *goal* cell ($h$)
**Ensure:** $A$ filled with minimal distances from goal.

```
boundary = h
A[h] = 0
while boundary ≠ ∅ do
        pop(c, boundary)
        insort (boundary, CELLFORWARD(c))
end while
```

# aaaaaaaaa

## aaaaaaaaa

```
goal = (0,(2,2))
update (goal)
VIEW(show(table))

boundary = [goal]
while boundary != []:
    c = boundary.pop()[1]
    new = cellforward(c)
    for e in new: insort(boundary, e)
    VIEW(show(table))
```

# aaaaaaaaa

## side effect: $A$ is being updated with min $dist(k)$ from goal

Algorithm CELLFORWARD$(A, k)$

**Require:** a matrix $A$; a *current* boundary cell $(k)$
**Ensure:** a subset of boundary cells.

adjacent (k) = $\{i$ for $i \in$ NEIGHBOR$(k)$ if $a[i] \geq 0.0\}$
**for** $i \in$ adjacent (k) **do**
    $a[i] = -(a[i] + a[k])$
**end for**
**return** adjacent

# aaaaaaaaa

## aaaaaaaaa

```python
def cellforward(multiindex):
    adjacent = filter(lambda i: a[tuple(i)] > 0,
        neighbor(multiindex))
    for i in adjacent:
        a[tuple(i)] = a[multiindex] - a[tuple(i)]
    return [(a[tuple(i)],tuple(i)) for i in adjacent]
```

# aaaaaaaaa

a $d$-cell $\gamma$ is represented as a multi-index $k = (k_0, ..., k_{d-1}) \in \mathbb{Z}^d$

Algorithm NEIGHBOR($k$)

**Require:** an $d$-cell $(k)$
**Ensure:** the set of $d$-cells $\delta\partial k - \{k\}$.

1: **return** $\delta\partial k - \{k\}$

# Laplace-Beltrami operator $\delta\partial$

Compute the coboundary of the boundary of a cell. Return a list of multiindices of cells

```
def neighbor(multiindex):
    def pred(k): return k-1
    def succ(k): return k+1
    def close (f,sequence,k):
        local = list(sequence)
        local[k] = f(local[k])
        return local
    def adj(f,index):
        ret = []
        for k in range(len(index)):
            ret += [close(f,index,k)]
        return ret
    def lowfilter(seq):
        zeros = [0]*len(shape(a))
        return AND(AA(ISLE)(TRANS([seq,zeros])))
    def highfilter(seq):
        return AND(AA(ISGT)(TRANS([seq,shape(a)])))
    lower = filter(lowfilter, adj(pred, multiindex))
    upper = filter(highfilter, adj(succ, multiindex))
    return lower + upper
```

# Laplace-Beltrami operator $\delta\partial$

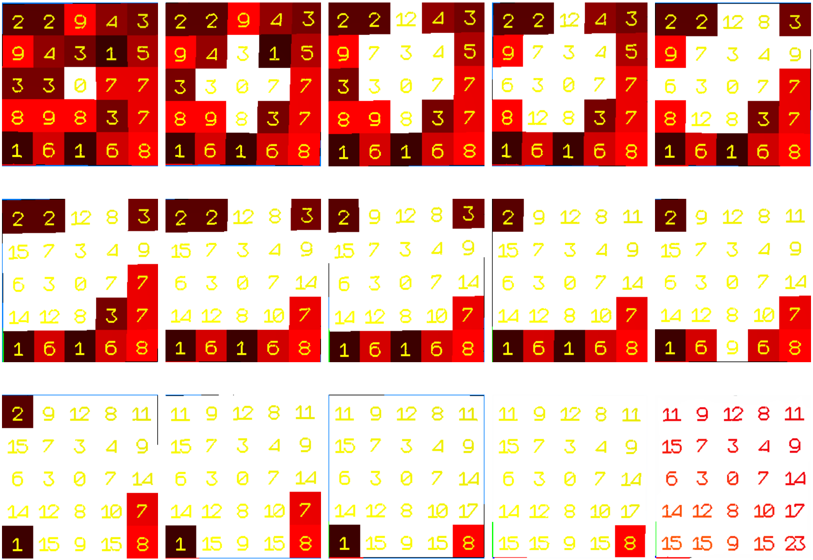
## Examples in several dimensions

```
>>> print neighbor([8,7])
[[7, 7], [8, 6], [9, 7], [8, 8]]
>>> print neighbor([7])
[[6], [8]]
>>> print neighbor([1,2,3])
[[0, 2, 3], [1, 1, 3], [1, 2, 2], [2, 2, 3], [1, 3, 3],
    [1, 2, 4]]
>>> print neighbor([1,2,3,4])
[[0, 2, 3, 4], [1, 1, 3, 4], [1, 2, 2, 4], [1, 2, 3, 3],
     [2, 2, 3, 4], [1, 3, 3, 4], [1, 2, 4, 4], [1, 2, 3,
    5]]
>>> print neighbor([0,0])
[[1, 0], [0, 1]]
```

The output is bounded by the 0-values of the indices, and by the actual shape of the underlying array, named a.

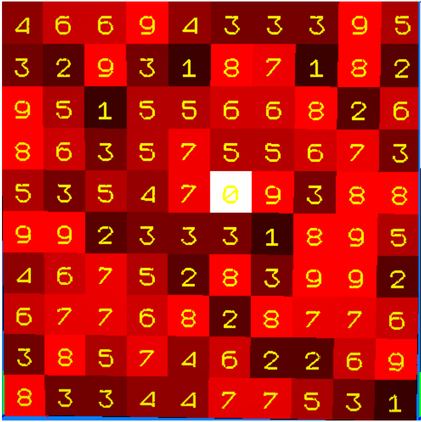
# aaaaaaaaa

## aaaaaaaaa



# aaaaaaaaa

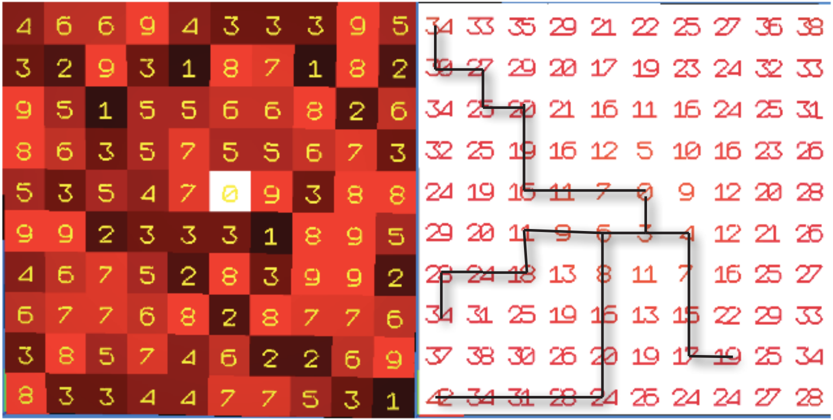
## aaaaaaaaa




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 34 | 33 | 35 | 29 | 21 | 22 | 25 | 27 | 36 | 38 |
| 30 | 27 | 29 | 20 | 17 | 19 | 23 | 24 | 32 | 33 |
| 34 | 25 | 20 | 21 | 16 | 11 | 16 | 24 | 25 | 31 |
| 32 | 25 | 19 | 16 | 12 | 5 | 10 | 16 | 23 | 26 |
| 24 | 19 | 16 | 11 | 7 | 0 | 9 | 12 | 20 | 28 |
| 29 | 20 | 11 | 9 | 6 | 3 | 4 | 12 | 21 | 26 |
| 28 | 24 | 18 | 13 | 8 | 11 | 7 | 16 | 25 | 27 |
| 34 | 31 | 25 | 19 | 16 | 13 | 15 | 22 | 29 | 33 |
| 37 | 38 | 30 | 26 | 20 | 19 | 17 | 19 | 25 | 34 |
| 42 | 34 | 31 | 28 | 24 | 26 | 24 | 24 | 27 | 28 |

# aaaaaaaaa

## aaaaaaaaa



# aaaaaaaaa

## aaaaaaaaa

```
def maxClimb(a, element):
    coboundary = neighbor(element)
    maxelement = coboundary.pop()
    while coboundary != []:
        temp = coboundary.pop()
        if a[tuple(temp)] > a[tuple(maxelement)]:
            maxelement = temp
    return maxelement
```

# aaaaaaaaa

## aaaaaaaaa

```python
def backPath(a, List):
    path = List
    if a[tuple(path[-1])] != 0:
        path += [maxClimb(a, path[-1])]
        backPath(a, path)
    return map(tuple, path)
```

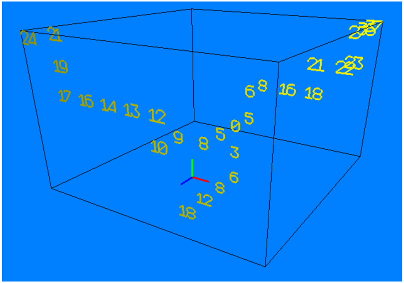
# aaaaaaaaa

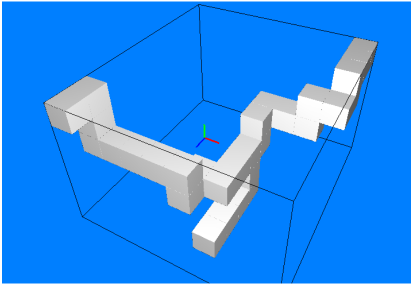
## aaaaaaaaa

```
def backward(a, points):
    tree = set([])
    for point in points:
        tree = tree.union(backPath(a, [point]))
    return list(tree)
```

# aaaaaaaaa

## aaaaaaaaa



# Multidimensional update

## aaaaaaaaa

```python
def item((i,j)):
""" to generate a single colored cell of the table.
    Return a colored unit square translated in (i,j)."""
    transl = T([1,2])([i,j])
    if a[i,j] > 0: color = colors[a[i,j]]
    else: color = WHITE
    return COLOR(color)(transl(CUBOID([1,1])))
```

```python
def item (index):
""" to generate a single colored d-cell of the table
    Return a colored cuboid translated in (i,j,...,k)"""
    coords = range(1,len(index)+1)
    transl = T(coords)(index)
    if a[tuple(index)] > 0:
        color = colors[a[tuple(index)]]
    else: color = WHITE
    return COLOR(color)(transl(CUBOID([1]*len(index))))
```

# Multidimensional update

aaaaaaaaa

```
def mass ((i,j)):
""" To write the field value in position (i,j)
    Return a text suitably positioned in 2D space−"""
    transl = T([1,2])([i+0.5,j])
    m = TEXTWITHATTRIBUTES('centre',0.0,0.3,0.5,0.0) (
        str(abs(a[i,j])))
    return transl(T(2)(0.25)(m))
```

```
def mass (index):
""" To write the field value in position (i,j)
    Return a text suitably positioned in space"""
    coords = range(1,len(index)+1)
    tpar = [0]*len(index)
    tpar[0] = 0.5
    transl = T(coords)(array(index) + array(tpar))
    m = TEXTWITHATTRIBUTES('centre', 0.0, 0.3, 0.5, 0.0)
        (str(abs(a[tuple(index)])))
    return transl(T(2)(0.25)(m))
```