from Recursion to Iteration

QuickSort Revisited
  using numpy arrays
  partitioning arrays via scan and swap
  recursive QuickSort on arrays

Converting Recursion into Iteration
  an iterative version with a stack of parameters

Space Filling Curves
  Hilbert curves
  a GUI for Hilbert curves
  implementing the moves
  turtle graphics: a drawing method

Exercises

MCS 275 Lecture 15
Programming Tools and File Management
Jan Verschelde, 18 February 2008
from Recursion to Iteration

Quicksort Revisited
using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
Quicksort Revisited
using numpy arrays

Recall the idea of Quicksort:

1. choose $x$ and partition list in two:
   - left list: $\leq x$ and right list: $\geq x$

2. sort the lists left and right

Our first implementation of Lecture 13 is recursively functional.

→ Python's builtin lists handle all data

**pro:** convenient for programming

**con:** multiple copies of same data

Goals: 1. use arrays for data efficiency,
   2. turn recursion into iteration.
Quicksort Revisited
using numpy arrays

Recall the idea of Quicksort:

1. choose $x$ and partition list in two:
   - left list: $\leq x$ and right list: $\geq x$
2. sort the lists left and right

Our first implementation of Lecture 13 is recursively functional.

→ Python’s builtin lists handle all data

pro: convenient for programming
con: multiple copies of same data

Goals:
1. use arrays for data efficiency,
2. turn recursion into iteration.
Quicksort Revisited
using numpy arrays

Recall the idea of Quicksort:

1. choose \( x \) and partition list in two:
   - left list: \( \leq x \)
   - right list: \( \geq x \)

2. sort the lists left and right

Our first implementation of Lecture 13 is *recursively functional*.

\[ \rightarrow \text{Python's builtin lists handle all data} \]

- **pro**: convenient for programming
- **con**: multiple copies of same data

Goals:  
1. use arrays for data efficiency,  
2. turn recursion into iteration.
Quicksort Revisited
using numpy arrays

Recall the idea of Quicksort:

1. choose $x$ and partition list in two:
   left list: $\leq x$ and right list: $\geq x$

2. sort the lists left and right

Our first implementation of Lecture 13 is recursively functional.

→ Python’s builtin lists handle all data

pro: convenient for programming

con: multiple copies of same data

Goals: 1. use arrays for data efficiency,
        2. turn recursion into iteration.
Recall the idea of Quicksort:

1. choose $x$ and partition list in two:
   left list: $\leq x$ and right list: $\geq x$

2. sort the lists left and right

Our first implementation of Lecture 13 is *recursively functional*.

→ Python’s builtin lists handle all data

**pro:** convenient for programming

**con:** multiple copies of same data

Goals: 1. use arrays for data efficiency,
2. turn recursion into iteration.
Quicksort Revisited
using numpy arrays

Recall the idea of Quicksort:

1. choose \( x \) and partition list in two:
   left list: \( \leq x \) and right list: \( \geq x \)

2. sort the lists left and right

Our first implementation of Lecture 13 is *recursively functional*.

→ Python’s built-in lists handle all data

**pro:** convenient for programming
**con:** multiple copies of same data

Goals: 1. use arrays for data efficiency,
2. turn recursion into iteration.
Arrays of Random Integers
the setup in `main()`

from numpy import *

def main():
    """
    Generates a random array of integers and applies quick sort.
    """
    a = input('Give lower bound : ')
b = input('Give upper bound : ')
n = input('How many numbers ? ')
A = random.random_integers(a,b,n)
ans = raw_input('Trace ? (y/n) ')
print 'A =', A
QuickSort(A,0,n,ans==’y’)
print 'A =', A
Arrays of Random Integers
the setup in main()

from numpy import *

def main():
    """
    Generates a random array of integers and applies quick sort.
    """
    a = input('Give lower bound : ')
b = input('Give upper bound : ')
n = input('How many numbers ? ')
A = random.random_integers(a,b,n)
ans = raw_input('Trace ? (y/n) ')
print 'A =', A
QuickSort(A,0,n,ans=='y')
print 'A =', A
from Recursion to Iteration

Quicksort Revisited
using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
Partitioning Arrays

Take $x$ in the middle of the array.

Apply scan and swap, for $i < j$:


For example: $A = [31, 93, 49, 37, 56, 95, 74, 59]$

At the middle: $x = 56$ (== $A[4]$)

Start with $i = 0$ and $j = 8$.

Increase $i$ while $A[i] < x$, end at $i = 1$.

Decrease $j$ while $A[j] > x$, end at $j = 4$.


$A = [31, 93, 49, 37, 56, 95, 74, 59]$

$i = 4$, $j = 3$, $x = 56$

$A[0:4] = [31, 56, 49, 37] <= 56$

$A[4:8] = [93, 95, 74, 59] >= 56$
Partitioning Arrays

Take \( x \) in the middle of the array.
Apply scan and swap, for \( i < j \):

For example: \( A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59] \)
At the middle: \( x = 56 \) (== \( A[4] \))

Start with \( i = 0 \) and \( j = 8 \).
   Increase \( i \) while \( A[i] < x \), end at \( i = 1 \).
   Decrease \( j \) while \( A[j] > x \), end at \( j = 4 \).

\( A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59] \)
\( i = 4, \ j = 3, \ x = 56 \)
\( A[0:4] = [31 \ 56 \ 49 \ 37] <= 56 \)
\( A[4:8] = [93 \ 95 \ 74 \ 59] >= 56 \)
Partitioning Arrays

Take $x$ in the middle of the array.
Apply scan and swap, for $i < j$:

For example: $A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59]$
At the middle: $x = 56$ ($= A[4]$)

Start with $i = 0$ and $j = 8$.
  Increase $i$ while $A[i] < x$, end at $i = 1$.
  Decrease $j$ while $A[j] > x$, end at $j = 4$.

$A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59]$
i = 4, j = 3, x = 56
$A[0:4] = [31 \ 56 \ 49 \ 37] \leq 56$
$A[4:8] = [93 \ 95 \ 74 \ 59] \geq 56$
Partitioning Arrays

Take x in the middle of the array. Apply scan and swap, for \( i < j \):

\[
\]

For example: \( A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59] \)
At the middle: \( x = 56 \) (== \( A[4] \))

Start with \( i = 0 \) and \( j = 8 \).

Increase \( i \) while \( A[i] < x \), end at \( i = 1 \).
Decrease \( j \) while \( A[j] > x \), end at \( j = 4 \).

\begin{align*}
A &= [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59] \\
i &= 4, \ j = 3, \ x = 56 \\
A[0:4] &= [31 \ 56 \ 49 \ 37] \leq 56 \\
A[4:8] &= [93 \ 95 \ 74 \ 59] \geq 56
\end{align*}
from Recursion to Iteration

Quicksort Revisited
using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
the Function Partition()

```python
def Partition(A, first, last):
    """
    Partitions A[first:last] using as pivot the middle item x. On return is (i, j, x):
    i > j, all items in A[i:last] are >= x, all items in A[first:j+1] are <= x.
    """

    x = A[(first+last)/2]
    i = first; j = last-1
    while i <= j:
        while A[i] < x: i = i+1
        while A[j] > x: j = j-1
        if i <= j: i = i+1; j = j-1
    return (i, j, x)
```
the Function Partition()

def Partition(A, first, last):
    ""
    Partitions A[first:last] using as pivot the middle item x. On return is (i, j, x): i > j, all items in A[i:last] are >= x, all items in A[first:j+1] are <= x.
    ""

    x = A[(first+last)/2]
    i = first; j = last-1
    while i <= j:
        while A[i] < x: i = i+1
        while A[j] > x: j = j-1
        if i <= j: i = i+1; j = j-1
    return (i, j, x)
the Function Partition()

def Partition(A, first, last):
    """
    Partitions A[first:last] using as pivot the middle item x. On return is (i,j,x):
    i > j, all items in A[i:last] are \( \geq x \),
    all items in A[first:j+1] are \( \leq x \).
    """
    x = A[(first+last)/2]
    i = first; j = last-1
    while i <= j:
        while A[i] < x: i = i+1
        while A[j] > x: j = j-1
        if i <= j: i = i+1; j = j-1
    return (i, j, x)
the Function Partition()

def Partition(A, first, last):
    """
    Partitions A[first:last] using as pivot the middle item x. On return is (i,j,x): i > j, all items in A[i:last] are >= x, all items in A[first:j+1] are <= x.
    """

    x = A[(first+last)/2]
    i = first; j = last-1
    while i <= j:
        while A[i] < x: i = i+1
        while A[j] > x: j = j-1
        if i <= j: i = i+1; j = j-1
    return (i, j, x)
Checking Postconditions
check correctness of Partition()

Important to verify the correctness:

\[ i = 4, \ j = 3, \ x = 56 \]
\[ A[0:4] = [31, 56, 49, 37] \leq 56 \]
\[ A[4:8] = [93, 95, 74, 59] \geq 56 \]

```python
def CheckPartition(A, first, last, i, j, x):
    
    
    Prints the result of the Partition
    in order to check the postconditions.
    
    print 'i = %d, j = %d, x = %d' % (i, j, x)
print 'A[%d:%d] =' % (first, j+1), \  A[first:j+1], '<=', x
print 'A[%d:%d] =' % (i, last), \  A[i:last], '>=', x
```
Checking Postconditions

check correctness of Partition()

Important to verify the correctness:

\[ i = 4, \quad j = 3, \quad x = 56 \]
\[ A[0:4] = [31, 56, 49, 37] \leq 56 \]
\[ A[4:8] = [93, 95, 74, 59] \geq 56 \]

```python
def CheckPartition(A, first, last, i, j, x):
    """
    Prints the result of the Partition in order to check the postconditions.
    """
    print 'i = %d, j = %d, x = %d' % (i, j, x)
    print 'A[%d:%d] =< %d' % (first, j+1), A[first:j+1], '<=', x
    print 'A[%d:%d] >= %d' % (i, last), A[i:last], '>=', x
```
A Recursive Quicksort
divide and conquer

def QuickSort(A, first, last, opt):
    """
    Sorts the array in increasing order. If opt, then extra output is written.
    """
    (i, j, x) = Partition(A, first, last)
    if opt: CheckPartition(A, first, last, i, j, x)
    if j > first: QuickSort(A, first, j+1, opt)
    if i < last-1: QuickSort(A, i, last, opt)

Important: first sort A[first:j+1].
A Recursive Quicksort
divide and conquer

def QuickSort(A, first, last, opt):
    """
    Sorts the array in increasing order.
    If opt, then extra output is written.
    """
    (i, j, x) = Partition(A, first, last)
    if opt: CheckPartition(A, first, last, i, j, x)
    if j > first: QuickSort(A, first, j+1, opt)
    if i < last-1: QuickSort(A, i, last, opt)

Important: first sort A[first:j+1].
A Recursive Quicksort
divide and conquer

```python
def QuickSort(A, first, last, opt):
    
    """
    Sorts the array in increasing order. If opt, then extra output is written.
    """

    (i, j, x) = Partition(A, first, last)
    if opt: CheckPartition(A, first, last, i, j, x)
    if j > first: QuickSort(A, first, j+1, opt)
    if i < last-1: QuickSort(A, i, last, opt)

Important: **first** sort A[first:j+1].
```
from Recursion to Iteration

Quicksort Revisited
using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
Converting Recursion into Iteration
a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push \((\text{first}, \text{last})\) on the stack.

As long as the stack of indices is not empty:
1. pop the indices \((\text{first}, \text{last})\) from the stack
2. we partition the array \(A[\text{first} : \text{last}]\)
3. push \((i, \text{last})\) and then \((\text{first}, j+1)\)
Converting Recursion into Iteration

a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push (first, last) on the stack.

As long as the stack of indices is not empty:

1. pop the indices (first, last) from the stack
2. we partition the array A[first:last]
3. push (i, last) and then (first, j+1)
Converting Recursion into Iteration
a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push \((\text{first}, \text{last})\) on the stack.

As long as the stack of indices is not empty:
1. pop the indices \((\text{first}, \text{last})\) from the stack
2. we partition the array \(A[\text{first:} \text{last}]\)
3. push \((i, \text{last})\) and then \((\text{first}, j+1)\)
Converting Recursion into Iteration

a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push \((\text{first, last})\) on the stack.

As long as the stack of indices is not empty:

1. pop the indices \((\text{first, last})\) from the stack
2. we partition the array \(A[\text{first: last}]\)
3. push \((i, \text{last})\) and then \((\text{first, j+1})\)
Converting Recursion into Iteration
a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push \((first, last)\) on the stack.

As long as the stack of indices is not empty:

1. pop the indices \((first, last)\) from the stack
2. we partition the array \(A[first:last]\)
3. push \((i, last)\) and then \((first, j+1)\)
Converting Recursion into Iteration
a stack for the parameters of the calls

Recursion is executed via a stack.

For Quicksort, we store first and last index of the array to sort.

With every call we push \( (first, last) \) on the stack.

As long as the stack of indices is not empty:

1. pop the indices \( (first, last) \) from the stack
2. we partition the array \( A[first:last] \)
3. push \( (i, last) \) and then \( (first, j+1) \)
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]
...

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
Running the iterative Code

\[ A = [31 \ 93 \ 49 \ 37 \ 56 \ 95 \ 74 \ 59] \]
\[ S = [(0, 8)] \]
\[ i = 4, \ j = 3, \ x = 56 \]
\[ A[0:4] = [31 \ 56 \ 49 \ 37] \leq 56 \]
\[ A[4:8] = [93 \ 95 \ 74 \ 59] \geq 56 \]
\[ S = [(0, 4), (4, 8)] \]
\[ i = 3, \ j = 1, \ x = 49 \]
\[ A[0:2] = [31 \ 37] \leq 49 \]
\[ A[3:4] = [56] \geq 49 \]
\[ S = [(0, 2), (4, 8)] \]
\[ i = 2, \ j = 0, \ x = 37 \]
\[ A[0:1] = [31] \leq 37 \]
\[ A[2:2] = [] \geq 37 \]
\[ S = [(4, 8)] \]
...
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]
...
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]
...

Exercises
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]
...

Quicksort Revisited using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves implementing the moves
turtle graphics: a drawing method

Exercises
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]

...
Running the iterative Code

A = [31 93 49 37 56 95 74 59]
S = [(0, 8)]
i = 4, j = 3, x = 56
A[0:4] = [31 56 49 37] <= 56
A[4:8] = [93 95 74 59] >= 56
S = [(0, 4), (4, 8)]
i = 3, j = 1, x = 49
A[0:2] = [31 37] <= 49
A[3:4] = [56] >= 49
S = [(0, 2), (4, 8)]
i = 2, j = 0, x = 37
A[0:1] = [31] <= 37
S = [(4, 8)]
...

Space Filling Curves

Hilbert curves
a GUI for Hilbert curves
 implementing the moves
turtle graphics: a drawing method

Exercises
An Iterative Quicksort

def ItrQuickSort(A,opt):
    
    The iterative version of QuickSort uses a stack of indices in A.
    
    S = []
    S.insert(0,(0,len(A)))
    while S != []:
        if opt: print 'S =', S
        (first,last) = S.pop(0)
        (i,j,x) = Partition(A,first,last)
        if opt: CheckPartition(A,first,last,i,j,x)
        if i < last-1: S.insert(0,(i,last))
        if j > first: S.insert(0,(first,j+1))
def ItrQuickSort(A,opt):
    
    The iterative version of QuickSort uses a stack of indices in A.
    
    S = []
    S.insert(0,(0,len(A)))
    while S != []:
        if opt: print 'S =', S
        (first,last) = S.pop(0)
        (i,j,x) = Partition(A,first,last)
        if opt: CheckPartition(A,first,last,i,j,x)
        if i < last-1: S.insert(0,(i,last))
        if j > first: S.insert(0,(first,j+1))
An Iterative Quicksort

```python
def ItrQuickSort(A,opt):
    """
The iterative version of QuickSort uses a stack of indices in A.
    """
    S = []
    S.insert(0,(0,len(A)))
    while S != []:
        if opt: print 'S =', S
        (first,last) = S.pop(0)
        (i,j,x) = Partition(A,first,last)
        if opt: CheckPartition(A,first,last,i,j,x)
        if i < last-1: S.insert(0,(i,last))
        if j > first: S.insert(0,(first,j+1))
```
from Recursion to Iteration

Quicksort Revisited
using numpy arrays
partitioning arrays via scan and swap
recursive Quicksort on arrays

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves
Hilbert curves
a GUI for Hilbert curves
implementing the moves
turtle graphics: a drawing method

Exercises
Space Filling Curves

Hilbert curves

$H_1$

$H_2$

$H_3$

moves

recursion

| A : D ← A | A → B |
| B : C ↑ B | B ↓ A |
| C : B → C | C ↑ B |
| D : A ↓ D | D ↑ C |
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

moves

A: D ← A ↓ A → B
B: C ↑ B → B ↓ A
C: B → C ↑ C ← B
D: A ↓ D ← D ↑ C

recursion

Converting Recursion into Iteration
an iterative version with a stack of parameters

Space Filling Curves

Hilbert curves
a GUI for Hilbert curves implementing the moves
turtle graphics: a drawing method

Exercises
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

<table>
<thead>
<tr>
<th>Moves</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: D</td>
<td>A \downarrow A \rightarrow B</td>
</tr>
<tr>
<td>B: C</td>
<td>B \rightarrow B \downarrow A</td>
</tr>
<tr>
<td>C: B</td>
<td>C \uparrow C \leftarrow B</td>
</tr>
<tr>
<td>D: A</td>
<td>D \downarrow D \leftarrow C</td>
</tr>
</tbody>
</table>
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

moves

recursion

\[ A : D \leftarrow A \downarrow \quad A \rightarrow B \]

\[ B : C \uparrow B \rightarrow B \downarrow A \]

\[ C : B \rightarrow C \uparrow C \leftarrow B \]

\[ D : A \downarrow D \leftarrow D \uparrow C \]

Exercises
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

Moves

\[ A : \quad D \leftarrow A \downarrow A \rightarrow B \]

\[ B : \quad C \uparrow B \rightarrow B \downarrow A \]

\[ C : \quad B \rightarrow C \uparrow C \leftarrow B \]

\[ D : \quad A \downarrow D \leftarrow D \uparrow C \]

Recursion
Space Filling Curves

Hilbert curves

$H_1$

$H_2$

$H_3$

moves

recursion

\[
\begin{align*}
&\rightarrow A : D \leftarrow A \downarrow A \rightarrow B \\
&\square B : C \uparrow B \rightarrow B \downarrow A \\
&\rightarrow C : B \rightarrow C \uparrow C \leftarrow B \\
&\downarrow D : A \downarrow D \leftarrow D \uparrow C
\end{align*}
\]
Space Filling Curves

Hilbert curves

\( H_1 \)

\( H_2 \)

\( H_3 \)

moves

recursion

\[ \begin{align*}
A & : \quad D \leftarrow A \downarrow A \rightarrow B \\
B & : \quad C \uparrow B \rightarrow B \downarrow A \\
C & : \quad B \rightarrow C \uparrow C \leftarrow B \\
D & : \quad A \downarrow D \leftarrow D \uparrow C
\end{align*} \]
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

moves

recursion

\[ \text{A} : \quad \text{D} \quad \rightarrow \quad \text{A} \quad \downarrow \quad \text{A} \quad \rightarrow \quad \text{B} \]

\[ \text{B} : \quad \text{C} \quad \uparrow \quad \text{B} \quad \rightarrow \quad \text{B} \quad \downarrow \quad \text{A} \]

\[ \text{C} : \quad \text{B} \quad \rightarrow \quad \text{C} \quad \uparrow \quad \text{C} \quad \leftarrow \quad \text{B} \]

\[ \text{D} : \quad \text{A} \quad \downarrow \quad \text{D} \quad \leftarrow \quad \text{D} \quad \uparrow \quad \text{C} \]
Space Filling Curves

Hilbert curves

H_1

H_2

H_3

moves

recursion

\[ \begin{align*}
A &: D \rightarrow A \downarrow A \rightarrow B \\
B &: C \uparrow B \rightarrow B \downarrow A \\
C &: B \rightarrow C \uparrow C \leftarrow B \\
D &: A \downarrow D \leftarrow D \uparrow C
\end{align*} \]
Space Filling Curves

Hilbert curves

\[ H_1 \]

\[ H_2 \]

\[ H_3 \]

moves

recursion

\[ A : D \leftarrow A \downarrow A \rightarrow B \]

\[ B : C \uparrow B \rightarrow B \downarrow A \]

\[ C : B \rightarrow C \uparrow C \leftarrow B \]

\[ D : A \downarrow D \leftarrow D \uparrow C \]
Space Filling Curves

Hilbert curves

$H_1$

$H_2$

$H_3$

moves

A : D ↔ A  ↓  A → B
B : C ↑ B → B  ↓  A
C : B → C ↑  C ← B
D : A ↓ D ← D ↑ C

recursion

Exercises
from Recursion to Iteration

Quicksort Revisited
  using numpy arrays
  partitioning arrays via scan and swap
  recursive Quicksort on arrays

Converting Recursion into Iteration
  an iterative version with a stack of parameters

Space Filling Curves
  Hilbert curves
  a GUI for Hilbert curves
  implementing the moves
turtle graphics: a drawing method

Exercises
A GUI for Hilbert Curves
A GUI for Hilbert Curves

class Hilbert:
    """
    Hilbert's space filling curves
    """
    def __init__(self, wdw, dimension):
        """
        Defines a canvas, buttons, and scale.
        """
    def Plot(self, nx, ny):
        """
        Plots a line for the current position to the new coordinates (nx, ny), and sets the current position to (nx, ny).
        """
    def Draw(self, v):
        """Draws the Hilbert curve on canvas."
    def Clear(self):
        """Clears the canvas."""
The Moves in the GUI

def A(self, k):
    """
    Turns counterclockwise from up right to down right.
    """

def B(self, k):
    """
    Turns clockwise from down left to down right.
    """

def C(self, k):
    """
    Turns counterclockwise from down left to up left.
    """

def D(self, k):
    """
    Turns clockwise from up right to up left.
    """
The Functions `Plot()` and `Draw()`

```python
def Plot(self, nx, ny):
    """
    Plots a line for the current position to the new coordinates (nx, ny), and sets the current position to (nx, ny).
    """
    self.c.create_line(self.x, self.y, nx, ny, width=2)
    self.x = nx; self.y = ny

def Draw(self, v):
    "Draws the Hilbert curve on canvas."
    n = int(v)
    self.h = self.dim/2**n
    self.y = 5 + self.dim - self.h/2
    self.x = self.y
    self.A(n)
```
The Functions \texttt{Plot()} and \texttt{Draw()}

```python
def Plot(self, nx, ny):
    """
    Plots a line for the current position to the new coordinates (nx, ny), and sets the
current position to (nx, ny).
    """
    self.c.create_line(self.x, self.y, nx, ny, width=2)
    self.x = nx; self.y = ny

def Draw(self, v):
    "Draws the Hilbert curve on canvas."
    n = int(v)
    self.h = self.dim/2**n
    self.y = 5 + self.dim - self.h/2
    self.x = self.y
    self.A(n)
```
from Recursion to Iteration

Quicksort Revisited
  using numpy arrays
  partitioning arrays via scan and swap
  recursive Quicksort on arrays

Converting Recursion into Iteration
  an iterative version with a stack of parameters

Space Filling Curves
  Hilbert curves
  a GUI for Hilbert curves
  implementing the moves
  turtle graphics: a drawing method

Exercises
the Move A

- A: D ← A ↓ A → B

```python
def A(self, k):
    """
    Turns counterclockwise from up right to down right.
    """
    if k > 0:
        self.D(k-1); self.Plot(self.x-self.h,self.y)
        self.A(k-1); self.Plot(self.x,self.y-self.h)
        self.A(k-1); self.Plot(self.x+self.h,self.y)
        self.B(k-1)
```
the Move A

```
def A(self, k):
    """
    Turns counterclockwise from up right to down right.
    """
    if k > 0:
        self.D(k-1); self.Plot(self.x-self.h,self.y)
        self.A(k-1); self.Plot(self.x,self.y-self.h)
        self.A(k-1); self.Plot(self.x+self.h,self.y)
        self.B(k-1)
```
the Move B

```python
def B(self, k):
    """
    Turns clockwise from down left to down right.
    ""

    if k > 0:
        self.C(k-1); self.Plot(self.x, self.y+self.h);
        self.B(k-1); self.Plot(self.x+self.h, self.y);
        self.B(k-1); self.Plot(self.x, self.y-self.h);
        self.A(k-1)
```

the Move B

def B(self,k):
    ""
    Turns clockwise from down left to down right.
    ""
    if k > 0:
        self.C(k-1); self.Plot(self.x,self.y+self.h)
        self.B(k-1); self.Plot(self.x+self.h,self.y)
        self.B(k-1); self.Plot(self.x,self.y-self.h)
        self.A(k-1)
the Move C

```
def C(self,k):
    """
    Turns counterclockwise from down left to up left.
    """
    if k > 0:
        self.B(k-1); self.Plot(self.x+self.h,self.y)
        self.C(k-1); self.Plot(self.x,self.y+self.h)
        self.C(k-1); self.Plot(self.x-self.h,self.y)
        self.D(k-1)
```
def C(self,k):
    
    """
    Turns counterclockwise from down left to up left.
    """
    if k > 0:
        self.B(k-1); self.Plot(self.x+self.h,self.y+
        self.C(k-1); self.Plot(self.x,self.y+self.h)
        self.C(k-1); self.Plot(self.x-self.h,self.y)
        self.D(k-1)
the Move D

def D(self,k):
    """
    Turns clockwise from up right to up left.
    """
    if k > 0:
        self.A(k-1); self.Plot(self.x, self.y-self.h)
        self.D(k-1); self.Plot(self.x-self.h, self.y)
        self.D(k-1); self.Plot(self.x, self.y+self.h)
        self.C(k-1)
the Move D

\[ D: A \rightarrow D \rightarrow D \uparrow C \]

def D(self, k):
    
    """
    Turns clockwise from up right to up left.
    """
    if k > 0:
        self.A(k-1); self.Plot(self.x, self.y-self.h)
        self.D(k-1); self.Plot(self.x-self.h, self.y)
        self.D(k-1); self.Plot(self.x, self.y+self.h)
        self.C(k-1)
from Recursion to Iteration

Quicksort Revisited
  using numpy arrays
  partitioning arrays via scan and swap
  recursive Quicksort on arrays

Converting Recursion into Iteration
  an iterative version with a stack of parameters

Space Filling Curves
  Hilbert curves
  a GUI for Hilbert curves
  implementing the moves
  turtle graphics: a drawing method

Exercises
Turtle Graphics
a general method to draw

The way the space filling curves were drawn is an example of “turtle graphics”.

Identify the pen to plot with a turtle.

Represent the turtle by

1. its current location on the canvas
2. the direction it will be going

The drawing is defined by connecting the current and the new position of a turtle by a line.
Turtle Graphics
a general method to draw

The way the space filling curves were drawn is an example of “turtle graphics”.

Identify the pen to plot with a turtle.

Represent the turtle by

1. its current location on the canvas
2. the direction it will be going

The drawing is defined by connecting the current and the new position of a turtle by a line.
Turtle Graphics
a general method to draw

The way the space filling curves were drawn is an example of “turtle graphics”.

Identify the pen to plot with a turtle.

Represent the turtle by

1. its current location on the canvas
2. the direction it will be going

The drawing is defined by connecting the current and the new position of a turtle by a line.
Turtle Graphics

a general method to draw

The way the space filling curves were drawn is an example of “turtle graphics”.

Identify the pen to plot with a turtle.

Represent the turtle by

1. its current location on the canvas
2. the direction it will be going

The drawing is defined by connecting the current and the new position of a turtle by a line.
Turtle Graphics
a general method to draw

The way the space filling curves were drawn is an example of “turtle graphics”.

Identify the pen to plot with a turtle.

Represent the turtle by

1. its current location on the canvas
2. the direction it will be going

The drawing is defined by connecting the current and the new position of a turtle by a line.
Exercises

1. Give Python code to enumerate all permutations of an array without making a copy of the array.

2. Two natural numbers \( m \) and \( n \) are input to the Ackermann function \( A \). For \( m = 0 \): \( A(0, n) = n + 1 \), for \( m > 0 \): \( A(m, 0) = A(m - 1, 1) \), and for \( m > 0, n > 0 \): \( A(m, n) = A(m - 1, A(m, n - 1)) \).
   2.1 Give a recursive Python function for \( A \).
   2.2 Turn the recursive function into an iterative one.

3. Design a GUI to implement turtle graphics. In addition to the canvas, the GUI should have buttons to turn and move the turtle. With entry fields the user can adjust the angle increment and the distance of each move of the turtle.

4. Write an iterative version of the GUI to draw Hilbert’s space filling curves.