Introduction to Computer Science and Programming for Astronomers
Lecture 4.

István Szapudi

Institute for Astronomy
University of Hawaii

February 5, 2014
Outline

1. Reminder
2. KDTree
3. Searching and Sorting
Summary

- We finished basic data structures
- Started to study analysis of algorithms
- The most important measure of algorithmic efficiency is expressed by the $\mathcal{O}$ notation.
- We have looked at assorted Fibonacci algorithms
hier.py
A pythonic digression

This will be a bit technical, but if you have solved the previous homework it should be understandable.

- How to represent (KD)-trees with arrays (this works even in Fortran)
- The use of doctest module in python
- List comprehensions
- Looping over arbitrary dimensions.
- Advanced list operations (functional programming) in python, including map, filter, reduce, and lambda.
- Then we continue with searching and sorting algorithms.
KD-tree pixellization

- The homework contained two-dimensional definitions, (quad-tree or 2D-tree). In K-dimensions (e.g. galaxy positions+ colors) we have KD-trees.
- There is an array representation.
- We can re-index a rectangular array with hierarchical indices.
- These are defined recursively: The first K bits tell you which K-drant (quadrant, octant, etc) the pixel is. Then replace the full cube with the K-drant.
- Spatially close pixels now have indices which are also close. Searching in this tree takes $O(\log N)$ time.
The 2d Case: Quad Tree

3 1
11₂ 01₂

2 0
10₂ 00₂

Prograde

15 13
11 10 11 01₂
01 11₂ 01 01₂

14 12
11 10 11 00₂
01 10₂ 01 00₂

7 5
11 9 3 1
10 11₂ 00 01₂
00 11₂ 00 01₂

6 4
10 8 2 0
10 10₂ 10 00₂
00 10₂ 00 00₂

Degrade
A KDTree Class

For the time-being we are only concerned about calculating hierarchical indices. Normally, you would store something, e.g. galaxy number counts.

class KDTree:
    ""
    builds a spatial KDtree structure for an array of `buckets`
    ""
def __init__(self,*log2n):
    from operator import mul
    self.dim = [2**l for l in log2n]
    self.nsize = reduce(mul,self.dim)
    self.hierInd = []
    indTup = [0]*len(self.dim)
    for i in range(self.nsize):
        bits = self.linInd2HierBits(self.dim,indTup)
        self.hierInd.append(self.bits2HierInd(bits))
    self.increaseIndTup(indTup)
for i in xIndices:
    for j in yIndices:
        do_something_with(i, j)
One advanced way would be to create a string which contains the program, then run the program with the `exec` statement.

```python
## rough outline only
def createCommand(n=2):
    ...
    return
    ""
    for i in xIndices:
        for j in yIndices:
            do_something_with(i, j)
    ""
    str=createCommand(2)
    exec(str)
```
Looping Through N-dim Array
This is simple and would work in any language.

Note the `try: except ...Error:` statement

def increaseIndTup(self, indTup):
...

    i = 0
    indTup[i] += 1
while indTup[i] == self.dim[i]:
    indTup[i] = 0
    i += 1
try:
    indTup[i] += 1
except IndexError:
    break
List Comprehension
A short hand to avoid for loops

```python
self.dim = [2**l for l in log2n]

## same as
self.dim = []
for l in log2n:
    self.dim.append(2**l)
```
self.nsize = reduce(mul,self.dim)
## same as
self.nsize = 1
for d in self.dim:
    self.nsize *= d
From Bits To Hierarchical Index

Watch out for the `lambda`!

```python
def bits2HierInd(self, bits):
    """
    the hierarchical index is created from the bits
    as \( b_{\text{max}} + 2*(b_{\text{max-1}}+2*...\) \)
    """
    return reduce(lambda x,y:2*x+y, bits)

## reduce(lambda x,y:2*x+y, bits) same as
def yp2x(x,y):
    return 2*x + y
reduce(y2px, bits)
```
Bits About Bits

- In a lower level languages, like C, we would represent bits simply by an integer.
- Adding new bits is simply a left shift (≪) and addition: very fast operations.
- This can be done in Python as well, but it is unlikely to be faster.
- Given the hierarchical index at a particular level, a lower (coarser) level can be calculated simply by a series of right shifts (or division by 2).
def linInd2HierBits(self, dim, indTup):
    """
    calculate a bit sequence of hierarchical index from a tuple of indices (i1,i2,...)
    """
    from operator import div
    locInd = indTup[:]  # create deep copy
    halfDim = map(lambda x: x/2, dim)
    ## same as
    ## halfDim = [0]*len(dim)
    ## for i in range(len(dim)):
    ##     halfDim[i] = dim[i]/2
linInd2HierBits: Cont’d

Some list operations.

```
zipped = filter(lambda (x,y): x > 0, zip(halfDim, locInd))
halfDim = map(lambda (x,y): x, zipped)
if halfDim == []:
    return None

locInd = map(lambda (x,y): y, zipped)
```
**zip A Useful Function**

Creates List of Tuples from Lists

Example:

```python
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> zip(a,b)
[(1, 4), (2, 5), (3, 6)]
>>> 
```
**filter** Another Useful Function

Example:

```python
c = a+b
>>> c
[1, 2, 3, 4, 5, 6]
>>> filter(lambda x: x > 3, c)
[4, 5, 6]
```
Yet Another ... Useful Function

```python
>>> c
[1, 2, 3, 4, 5, 6]
>>> map(lambda x: x**2, c)
[1, 4, 9, 16, 25, 36]
## same as
>>> [x**2 for x in c]
[1, 4, 9, 16, 25, 36]
```
At this point we have created `locInd`, `halfDim`.

```python
bits = map(div, locInd, halfDim)
locInd = map(lambda x, y, z: x - y * z, locInd, bits, halfDim)
newbits = self.linInd2HierBits(halfDim, locInd)
if newbits != None:
    for b in newbits:
        bits.append(b)
return bits
```
def test2d():
    
    >>> test2d()
    4
    [0, 2, 8, 10, 1, 3, 9, 11, 4,
     6, 12, 14, 5, 7, 13, 15]
    
    log2n = 2
    print 2**log2n
    kd = KDTree(log2n,log2n)
    print kd.hierInd
doctest
The doctest module searches for pieces of text that look like interactive Python sessions ...

```python
def increaseIndTup(self, indTup):
    """..."
    ...  
    >>> kd = KDTree(2,1,2)
    >>> a = [0,0,0]
    >>> for i in range(kd.nsize+1):
    ...    print a
    ...    kd.increaseIndTup(a)
    ...
    [0, 0, 0]
    [1, 0, 0]
    [2, 0, 0]
    [3, 0, 0]
    [0, 1, 0]
```
You can include the following standard line at the end (substitute `hier` with your module name)

```python
def _test():
    import doctest, hier
    return doctest.testmod(hier)

if(__name__=='__main__ '):
    _test()
```
doctest Output

[szapudi@joshu lecture4] ./hier.py -v
...
Trying:
for i in range(kd.nsize+1):
    print a
    kd.increaseIndTup(a)
Expecting:
[0, 0, 0]
[1, 0, 0]
...
[3, 1, 3]
[0, 0, 0]
ok
hier Passed!

8 items had no tests:
...
3 items passed all tests:
  3 tests in hier.KDTree.increaseIndTup
  1 tests in hier.test2d
  1 tests in hier.test3d
5 tests in 11 items.
5 passed and 0 failed.
Test passed.
Notes on **doctest**

- If you run it without the -v option, it will not produce output (as long as your tests passed)
- It is a simple, informal way to run unit tests (for larger projects unittest is recommended)
- You can directly cut and paste live python sessions into your program
Sequential Search
The simplest possible search algorithm

def seqSearch(list, item):
    i = 0
    for l in list:
        if l == item:
            return i
        i += 1
    return (-1)
The number of comparisons is equal to the position of the item. Let $I$ be the set of possible inputs, say size $n$.

- Worst case analysis: $\max_I(t(I))$
- Best case analysis: $\min_I(t(I))$ (usually not too useful)
- Average case analysis $\sum P(I)t(I)$

The average case is the most important, but it is often difficult to estimate probabilities.

Instead we often do worst case.
Average Case Analysis of seqSearch

If the item is on the list, and it could be at any position with equal $1/n$ probability, the average case would be $1/n \sum_{i=1}^{n} i = (n + 1)/2$, as expected naively.

If we assume that each permutation of the list is equally likely, we get the same result, even though it appears to be a stronger assumption.

If, however, the item is not on the list, the time is always $n$.

Randomization is used in many algorithms to avoid the worst case scenario. One can also calculate variance, etc. in addition to average case.
Binary Search

- In a sorted list (a list in which the elements are in order) we can do a much better then linear search
- We can first compare with the middle element, and see if it is bigger, equal, or smaller.
- The repeat for the half in which it will be.
def binarySearch(sortedList, item):
    n = len(sortedList)
    i = n/2
    if n == 0:
        return (-1)
    if sortedList[i] == item:
        return i
    elif sortedList[i] > item:
        ind = binarySearch(sortedList[0:i], item)
    else:
        ind = binarySearch(sortedList[i+1:n], item)
    if ind >= 0: ind += i + 1
    return ind
Clearly

\[ t_n = \mathcal{O}(1) + \frac{t_n}{2} \approx \mathcal{O}(\log(n)) \]  \hspace{1cm} (1)

For large \( n \gg \log(n) \) binary search is much more efficient

There must be a property according to which the items can be sorted \(<=\>\).

The list must be presorted.
Divide and Conquer

- Binary Search and KDTree are examples of the “Divide and Conquer” algorithmic paradigm.
- Similar to (although distinct from) the Top Down approach to programming: this has a much narrower meaning.
- The idea is that instead of solving the problem, we focus on solving a natural subproblem.
- Recursion is the most natural outcome of such thinking, and optionally, we can resolve it.
def binarySearchNonRec(sortedList, item):
    a = 0
    b = len(sortedList) - 1
    i = (a + b) / 2
    while sortedList[i] != item:
        if sortedList[i] > item:
            b = i - 1
        else:
            a = i + 1
        if a > b:
            return -1
        i = (a + b) / 2
    return i
Summary

- We have gone through the KDTree indexing
- We have learnt some advanced python
- We have analysed two basic searching algorithms, sequential and binary search.
(E1) Read a set of objects (e.g. galaxies or stars) represented by their $x y$ coordinates from a file. Using the hierarchical indexing you created in the last homework, create a set of arrays with sizes $2^k \times 2^k$ for $k = 0, \ldots, N$, and all of them indexed hierarchically. Each pixel of each array contains the total number of objects in that pixel. Each array represents the bounding box $L \times L$ of the observation, but on different scales.

(E2) Real observations have cut-out masks due to CCD problems, bright objects, etc. How would you keep track of those issues (just write down your ideas).
Homework # 4
continued

(E3) Measure the speed of the hierarchical indexing algorithm you created in the last homework. Comment on the scaling with size of the array. Instead of creating a lookup table to begin with we could apply the principles of dynamic programming, and write a function which calculates hierarchical indices. How would you go about doing that?

(E4) Write a modified stack class, which stores numbers with a push method but returns always the smallest number with a pop method.