Introduction to Computer Science and Programming for Astronomers
Lecture 5.

István Szapudi

Institute for Astronomy
University of Hawaii

February 12, 2014
Outline

1 Reminder

2 Sorting
   - Overview
   - Selection Sort
   - Heap Sort
   - Merge Sort
   - Quick Sort
Last class we went through the hierarchical index generation for KD-trees.

Along the way we learnt some advanced python

We looked at the two most basic searching algorithm.

One of them, binary search, was an example of the “Divide and Conquer” algorithmic paradigm.

Now we move on to sorting, the traditional vehicle to get into algorithms.
Comparison Sorting

- A fairly abstract model of sorting: it assumes that we can sort a pair of objects.
- These algorithms will work as long as we have a comparison function.
- In fact we don’t even have to write the algorithms: most standard libraries contain sort routines (but we have to understand them to choose between algorithms, and possibly modify them).
- In Python you can use `list.sort()` function will sort a list (adaptive mergesort). If it is a list of objects, you might have to write your own `__cmp__()` to overload standard comparison.
Overview of Sorting Algorithms

There are dozens of clever sorting algorithms. We cannot cover all of them. (Knuth has a whole volume on sorting!) Some of the most important ones are the following:

- Heap sort is the simplest modification of a naive (selection) sort.

- Merge sort and Quicksort are further examples of the Divide and Conquer paradigm I mentioned before.

- Randomization in quicksort represents yet another paradigm in algorithmic construction.

- Bucket sort is less abstract: it is an improved algorithm for numbers or strings, and uses their properties.
Overview of Sorting Algorithms: Scaling

- Heap/Merge and average Quick sort algorithms are all $O(n \log n)$.
- Worst case behaviour of Quick sort, as well as naive sorts are $O(n^2)$. The randomization in quicksort is for elimination of the worst case (paradoxically, when the elements are already sorted).
- The naive algorithms (insertion, selection, bubble, etc) are all $O(n^2)$
- If you want to sort huge numbers the scaling dominates. Remember, that for small $n$ the naive algorithms might fare better as they are simpler.
- You cannot do significantly better then $O(n \log n)$ with comparison sorting.
We could form an abstract picture of any comparison sorting algorithm as binary tree: each comparison corresponds to a branch of the tree.

When we arrive at the leaves of the tree, we know the order of the original sequence: therefore this abstract decision tree has $n!$ leaves.

The worst case behaviour of our algorithm is the $L$, longest path in this tree. For a tree with $k$ leaves $L(k) \geq \log(k)$ by induction. (Split the tree, one of the subtrees has at least half of the leaves, i.e. $L(k) \geq 1 + L(k/2)$, etc.)

Thus the worst case behaviour of any sort algorithm is at least $\log n! \approx \Omega(n \log n)$. The average case is more complicated to estimate, but it is of the same order.
Selection Sort
The simplest sort algorithm

def selSort(l):
    """
    selection sort algorithm
    """
    x = []
    while l != []:
        x.append(popSmallest(l))
    return x
def popSmallest(l):
    """
    pop the smallest element of a list l
    """
    smallInd = 0
    for i in range(len(l)):
        if(l[smallInd] < l[i]):
            smallInd = i
    return l.pop(smallInd)
Selection Sort: Analysis
A simple example of analysing scaling

- The main loop is executed $n$ times.
- The inner loop takes $O(len(l))$, which is becoming less by one in each loop.
- Thus the total time is $T \sim \sum i = O(N^2)$, corresponding to a quadratic scaling.
- This is not approaching our theoretical limit $O(N \log N)$.
$O(N^2)$ vs. $O(N \log N)$

The difference is huge!

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N \log N$</th>
<th>$N^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>33</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>665</td>
<td>10000</td>
</tr>
<tr>
<td>1000</td>
<td>$10^4$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$2 \times 10^7$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>$10^9$</td>
<td>$3 \times 10^{10}$</td>
<td>$10^{18}$</td>
</tr>
</tbody>
</table>

Table: This difference justifies some more effort on sorting to get closer to the theoretical scaling.
Heapsort: a Generalization of Selection Sort

It uses a heap data structure to perform the main operation of the selection sort quickly. It has the following basic steps:

- Take the initial list and turn it into the heap data structure.
- Find the largest element.
- Remove it from the heap and restore heap property for the rest of the elements.
The binary heap we will be using has the following properties:

- The elements are arranged in a tree data structure.
- The tree is approximately balanced in the sense that all paths are $O(\log N)$.
- Heap property: each parent element is larger than the child.
- We will represent the binary tree in an array, so this algorithm would work in any language.
Auxiliary Functions for Tree Indexing:

```python
# tree indexing in 1D!
def parent(i): return i//2
def left(i): return 2*i
def right(i): return 2*i+1
def heapLength(A): return len(A)-1
```

The way the algorithm works is that we will only swap values in the tree, but the tree structure itself is fixed once for all by this indexing. To make this work, I pad the array `A[0] = None` to simulate indexing from 1.
def heapSort(A):
    heapSize = heapLength(A)
    buildHeap(A, heapSize)
    for i in range(heapSize, 1, -1):
        # A[1]: largest, put at the end
        heapSize = heapSize - 1
        heapify(A, heapSize)
def heapify(A, n, i=1):
    l = left(i)
    r = right(i)
    if l <= n and A[l] > A[i]:
        largest = l
    else:
        largest = i
    if r <= n and A[r] > A[largest]:
        largest = r
    if largest != i:
    heapify(A, n, largest)
A divide and conquer algorithm...

def buildHeap(A, n, root=1):
    if root >= n:
        return

    buildHeap(A, n, left(root))
    buildHeap(A, n, right(root))
    heapify(A, n, root)
Note that you can build the heap non-recursively as well:

```python
def buildHeapNonRec(A, n):
    """
    build a heap A from an unsorted array
    non-recursive way
    """
    for i in range(n/2, 0, -1):
        heapify(A, n, i)
```
Each call to `heapify` (fix the heap) takes about \( \log N \) steps (the depth of the tree).

The main loop is therefore \( \mathcal{O}(N \log N) \).

Building the tree recursively takes

\[
T(N) = 2T(N/2) + 2 \log N
\]

In the homework you will see that the solution to this recursion is \( 4N \).
Merge Sort

- One of the first sorting algorithms ever written for a computer (by János Neumann) in 1945, and it is the best of the pack.
- The idea is very simple: we can easily merge two sorted lists.
- Therefore you can sort two half recursively of the list first with `mergesort`, and then merge them.
- This magic is very simple to program, and gives one of the most efficient sorts!
def merge(A, B):
    Out = []
    iA = iB = 0
    while len(A) + len(B) > iA + iB:
        if len(B) <= iB or (len(A) > iA and A[iA] < B[iB]):
            Out.append(A[iA])
            iA = iA + 1
        else:
            Out.append(B[iB])
            iB = iB + 1
    return Out
mergeSort

It cannot get much simpler...

def mergeSort(L):
    ""
    recursive merge sort
    ""
    if len(L) < 2: return L
    return merge(mergeSort(L[:len(L)/2]), mergeSort(L[len(L)/2:]))
Analysis of Merge Sort

- The number of comparisons is \( C(N) \leq N - 1 + 2C(N/2) \).
- The solution of this recursion is \( N \log N - N + 1 \).

This means that it is not only simpler than \texttt{heapSort}, but the number of comparison is (marginally) less. The scaling is still \( \mathcal{O}(N \log N) \).
Quick Sort

- It is based on the same Divide and Conquer idea.
- Partion a list, like in merge sort.
- It would be simple to merge, if the any element on the first list would be smaller then any element of the right list.
- To put it together: we pick an element \( x \) and create three lists: those which are smaller, equal, and larger then \( x \).
- We sort the two lists (with Quick Sort), and merge is now a simple concatenation.
def quickSort(L):
    if len(L) < 2: return L
    i = random.randrange(len(L))
    return quickSort([x for x in L if x < L[i]]) + \
    [x for x in L if x == L[i]] + \
    quickSort([x for x in L if x > L[i]])
Worst Case Analysis of Quick Sort

The partition step takes \( N - 1 \) comparisons, thus the total comparisons is
\[
C(N) = N - 1 + C(N_s) + C(N_l),
\]
where \( N_s + N_l \leq N - 1 \) the size of the two lists to be sorted.

The problem is that the worst case, \( N_s = 0 \) and \( N_l = l - 1 \) is terrible:
\[
C(N) = N - 1 + C(N - 1),
\]
from which \( \mathcal{O}(N^2) \) follows.

This happens when the list is ordered: to avoid that, we randomize the list at each step. This introduces the third algorithmic paradigm: stochastic methods.
Average Case Analysis of Quick Sort

- We can assume that any element in the array can be the pivot element due to randomization with $1/N$ probability. It follows that $C(N) = \frac{1}{N} \sum_{i=0}^{N-1} (N - 1 + C(i) + C(N - i - 1))$.

- $C(N) = N - 1 + \frac{2}{N} \sum_{i=0}^{N-1} C(i)$, using the fact that each term appears twice with equal probability.

- It can be shown by induction, that this is $O(N \log N)$.
Divide and Conquer Recursions

Let’s assume that in general that we solve a problems of size $N/b$ instead of our original (1) problem with the divide and conquer paradigm.

For simplicity let’s assume that $N = b^m$ and that the recursion stops when the size of the problem is 1.

At the bottom level $N/b^m = 1$ therefore $m = \log N / \log b$.

At level $i$ the size of the problem is $N/b^i$ and the let’s assume that the time to execute the program is $f(N/b^i)$, there is $a^i$ programs to be executed at this level.

Putting it all together our final result is:

$$T(N) = \sum_{i=0}^{\log N / \log b} a^i f(N/b^i)$$

(1)
The above equation has logarithmically many terms.
It is often dominated by either the $f(N) \ (i = 0)$ term
Or the $N^{\log a/\log b} \ (i = \log N/ \log b)$ term.
Generally you are logarithmically close to the solution using $O(f(N) + N^{\log a/\log b})$.
E.g., $heapSort \ a = b = 2, f(N) = \log N$
Summary

- We have studied three Divide and Conquer searching algorithms: Heap, Merge, and Quick sort.
- Each of them are comparison sorts.
- We analyzed their properties (basically each of them are $N \log N$).
- In real life (in Python programs) you should use the built-in sort as much as you can.
- We analyzed Divide and Conquer algorithms in general.
Homework # 5

(E1) Analyse the recursions we have written for the Heap, Merge and Quicksort algorithms, and prove the scaling I quoted in class.

(E2) Solve the general equation for Divide and Conquer scaling for assuming that $f(N) = N^k$. Distinguish three cases: $a > b^k$, $a = b^k$, and $a < b^k$. 
(E3) Run all the search algorithms I provided in `lecture5.py` and measure their scaling. Compare with your calculation in (E1) and with the built in `sort` of `python`.

(E4) Modify `selSort` and `heapSort` in such a way that they provide the list sorted in opposite order. In the first case, achieve this by using a stack, while in the second case, actually modify the algorithm.
(P1) Write a program, which reads galaxy/star objects represented by their \((x, y)\) coordinates, and links them up in a tree based on their position on the sky. (Hint: first link up all galaxies linearly, then link them to two links according to the \(x\) coordinate being bigger or smaller than half the range in \(x\). Then do the same in \(y\) coordinates. Then divide the size by two, and continue recursively until there is one galaxy at the leaf. This is the method which is often used in \(N\)-body simulations and for spatial searching (e.g., of neighbours).