Introduction to Computer Science and Programming for Astronomers

Lecture 6.

István Szapudi

Institute for Astronomy
University of Hawaii

February 19, 2014
Outline

1. Reminder
2. Sorting Cont’d
3. Order Statistics
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.

Now we are moving on to bucket sort.
We have seen a few comparison sort algorithms with $\mathcal{O}(N \log N)$ scaling. Given the $\Omega(N \log N)$ lower bound these are pretty much optimal.

What if we are allowed other operations than just comparisons?

We are giving away the abstraction: the results will depend on the data type. E.g. integer, float, character, etc.

First example: how would you sort integers between 0 . . . $n$?
def bucketSort(A):
    """
    A is a list of positive integers between 0...max
    """
    max = A[0]
    for number in A:
        if number > max:
            max = number
bucketSort
Place the items into “buckets”

```python
bucket = [0]*(max+1)
for number in A:
    bucket[number] += 1
A = []
for i in range(max+1):
    for j in range(bucket[i]):
        A.append(i)
return A
```
The first and second cycles are $O(N)$.

The third cycle appears to be a nested cycle: however, the total number of objects is still $N$, therefore `A.append(i)` is executed $N$ times.

This is a linear sort $O(N)$.

For large $N$ this can be better than comparison sort.
Bucket Sort with Keys

- Typically we want more than just sorting numbers: we want to sort some items according to some keys calculated from the items.
- In this case it is not enough to simply sum the items.
- The solution is simply: the bucket will be a linked list or a list of lists.
- Since lists are so flexible in Python, I opted for the second solution.
def bucketSortKey(A, key, kMax):
    
    bucket sort a list A according to a key

    if a is an element of A, key(a)
    should calculate a key, a number from 0..kMax-1
    from it

    
    
    
    
    

bucket = [[] for i in range(kMax)]
for a in A:
    bucket[key(a)].append(a)
A = []
for i in range(kMax):
    for b in bucket[i]:
        A.append(b)
return A
A sort algorithm is called **stable** if two items which have the same key stay in their original order.

Stability becomes important when we want to sort a set of items by several keys (as we will do next).

- Bucket sort: stable.
- Heapsort: not stable
- Merge sort: if we are breaking ties in favour of the first element, it can become stable.
- Quick sort: depends on the implementation.
If we have a very large number of elements, bucket sort becomes inefficient.

Instead we could sort elements by several keys (e.g. sorting strings by different letters, etc.)

Or sorting integers by digits.
Just a set of bucket sorts...

def radixSortIntMod10(A, maxDigit):
    """
    radix sort integers with at most maxDigit digits by mod 10 keys
    """
    for d in range(maxDigit):
        A = bucketSortKey(A, lambda x: nthDigit(x, d), 10)
    return A
def nthDigit(i, n):
    """
    return the n-1-th digit of an integer
    """
    return i/10**n % 10
Radix Sort Analysis

- Subtleties: you want to sort by the most important digit last: this puts things into their final order.
- Correctness: follows from the stability of the bucketsort by induction.
- Scaling: for $K$ possible keys (= the max. number), we will have $\log_{10} K$ passes of bucketsort: $O(N \log K)$
- This is never optimal: if $K < N$ the bucket sort would take $O(N)$ time. If $K > N$, any good comparison sort is $O(N \log N)$, which is smaller.
- Possible improvement: use a large base, e.g. $2^{16} = 65536$. 
The lowest order description of a set of numbers is usually done by average, mode (most likely value), or median.

Median is a robust statistic, and it should be used when the distribution is non-Gaussian (e.g., has a long tail).

Median is defined as the middle value if the data are ordered: hence the name order statistics.

You can also calculate quartiles, percentiles, etc. as robust measure (instead of, say, variance).
The general computational problem derived from the previous examples is the following: given $N$ numbers, find the $k$-th value.

E.g. median is the special case when $k = N/2$. Similar definitions for quartiles, etc.

Obviously can be done by sorting. Q: can we do it without sorting a large data set?

For $k = 1$ the problem is trivial and $\mathcal{O}(N)$: you simply have to cycle through the data once to find first ranked (e.g. largest) element.
def findMin(A):
    """
    find the smallest element in a list
    """
    min = A[0]
    for a in A:
        if min > a:
            min = a
    return min
Finding the Second Smallest Item

- We can use the same general strategy as before.
- Instead of storing one value, we could store the smallest, and second smallest element.
- It is enough to test for the second smallest element.
- If the second smallest needs replacing, we need to check whether the new second smallest is smaller then the min.
def findMin2(A):
    """
    find the two smallest element in a list
    """
    min, min2 = A[0], A[1]
    if min2 < min:
        min, min2 = min2, min
    for a in A[2:]:
        if min2 > a:
            min2 = a
        if min2 < min:
            min, min2 = min2, min
    return min, min2
Analysis of \texttt{findMin2}

- Worst case: if the items are already in descending order, we need 2 comparisons for each of the $N - 2$ items: $2N - 3$ total comparisons (including the first one).
- Average case analysis: if any permutation can come up.
- Then the first comparison will still happen $N - 1$.
- The second comparison happens when $A[i]$ is in the top 2 of the list $A[0..i]$. The probability of this is $2/(i + 1)$ (since $A[i]$ can be either of the top two).
- The total of extra comparisons is then $\sum_i 2/i \simeq 2 \log N$.
- The final result is $N + \mathcal{O}(\log N)$. 
This is encouraging: find the second element took only \( \mathcal{O}(\log N) \) comparisons.

Can we modify our previous search algorithms to do selection?

A: yes. I show you one, \texttt{quickSelect}.

If we look at the \texttt{quickSort} algorithm, we could use it for selecting the \( k \)-th element, but we would really only use the \texttt{length} of the lists.
def quickSelect(L,k):
    """
    simple quick select algorithm
    """
    i = random.randrange(len(L))
    L1 = [x for x in L if x < L[i]]
    L2 = [x for x in L if x == L[i]]
    L3 = [x for x in L if x > L[i]]
quickSelect Cont’d

if len(L1) >= k:
    return quickSelect(L1,k)
elif len(L1)+len(L2) < k:
    return quickSelect(L3,k-len(L1)-len(L2))

return L[i]
**quickSelect Analysis**

- The worst case of the algorithm $T(N) = \mathcal{O}(N) + T(N - 1)$ which gives $\mathcal{O}(N^2)$ like with quickSort.

- The average case is a lot better. In principle we would have to write a recurrence with two parameters, $N$, $k$, but use the worst case $k = N/2$ for simplicity.

- At each level we eliminate $\text{len}(L2) + \min(\text{len}(L1, L3))$ items from the list, the $\min$ is equally likely to be between 1..$N/2$, so the list size at the next level will be between $N/2 - N$.

- Note that $k! = N/2$ would increase the likelihood of smaller lists, which shows that this is the worst case of $k$. 


Thus for $k = N/2$ the average case recursion is

$$T(N) = N - 1 + \sum_{i \geq N/2} 2T(i)/N.$$ 

It is intuitively clear that this is $O(N)$ due to the $1/N$ factor.

Let’s assume that $T(N) \leq cN$, and prove it by induction.

$$T(N) \leq N - 1 + \sum_{i \geq N/2} 2ci/N = N - 1 + 2c/N(\Sigma(N) - \Sigma(N/2 - 1)),$$

where

$$\Sigma(N) = \sum_{i=1}^{N} i = N(N+1)/2 = \mathcal{O}(N^2).$$

$$T(N) \leq N - 1 + 2c/N(N^2/2 + (N/2)^2/2 + \mathcal{O}(N)) = N(1 + c(1/2 + 1/4)) + \mathcal{O}(1)$$

This will satisfy or induction with $1 + 3/4c \geq c$ or $4 \geq c$. This means that the rough behaviour of our algorithm (worst case in $k$) is $\approx 4N$. 

735 Computational Astronomy
Comments on Order Selection Algorithms

- This is a simple linear, randomized, algorithm which can select an element of any rank.
- It turns out that with more careful selection of the pivot element (such that it is already close to the $k$-th rank, one can do a bit better $N + \min(K, N - K) + \mathcal{O}((N \log N)^{2/3})$ (Floyd and Rivest algorithm, it is bit complicated).
- One can also construct clever algorithms, without randomization, which use about $3N$ comparisons.
Summary

- We have seen that using properties of the data one can do a faster sort: e.g. bucket sort.
- We have defined order statistics, and concluded that selection of the $k$-th element is needed.
- We have found that we can do that without sorting, with a linear, randomized algorithm, quickSelect.
Homework # 6

(E1) Write a bucket sort algorithm, which uses linked lists instead of a list of list I used in class.

(E2) Check each sorting algorithms we have looked at so far if they are stable. Modify (one of) those which are not stable to become stable.
(E3) Write a radix sort which will use $2^{16}$ instead of 10 for base. Compare the performance with the radix sort with 10 base. How would you use radix sort to sort floating point numbers?

(E4) Modify the quickSelect algorithm, such that it will yield the $k$-th largest (as opposed to smallest) element of the list.