Stackoverflow-interview questions

I was asked the following question in an interview:

How to get top 100 numbers out of a large set of numbers (can't fit in memory)

And this is what I said:

Divide the numbers in batches of 1000 each. Sort each batch in "O(1)" time. Total time taken is O(n) up till now. Now take 1st 100 numbers from 1st and 2nd batch (in O(1)). Take 1st 100 from the above computed nos and the 3rd batch and so on. This will take O(n) in total - so it is an O(n) algorithm.

The interviewer replies that sorting a batch of 1000 nos. won't take O(1) time and so won't picking out 1st 100 out of a batch and after a lot of discussion he said, he doesn't have problem with the algo taking O(n) time, he just has a problem with me saying that sorting the batch takes O(1) time.

My explanation was that 1000 doesn't depend on the input (n). Irrespective of what n is, I'll always make batches of 1000 nos. and if you have to calculate, the sorting takes O(1000\*log 1000)) which is essentially O(1).

If you have to make proper calculations, it would be

**1000\*log 1000 to sort one batch sort (n/1000) such batches takes 1000 \* log 1000 \* n/1000 = O(n\*log(1000)) time = O(n) time**

I asked a lot of my friends also about this and although they agreed with me but partially. So I wan't to know if my reasoning is 100% accurate (please criticize even if it is 99% correct).

Answer:

The interviewer is wrong, but it's useful to consider why. What you're saying is correct, but there is an unstated assumptions that you depend on. Possibly, the interviewer is making a different assumption.

If we say that sorting 1000 numbers is O(1), we're being a bit informal. Specifically, what we mean is that, in the limit as N goes to infinity, there is a constant greater than or equal to the cost of sorting the 1000 numbers. Since the cost of sorting the fixed-size set is independent of N, the limit isn't going to depend on N, either. Thus, it's O(1) as N goes to infinity.

A generous interpretation is that the interviewer wanted you to treat the sorting step differently. You could be more precise and say that it was O(M\*log(M)) as M goes to infinity (or M goes to N, if you prefer), with M representing the size of the batches of numbers. That would make an overall O(N\*log(M)) for your approach, as N and M both approach infinity. Of course, that wasn't the limit you described.

Strictly speaking, it's meaningless to say that something is O(1) without specifying the limit. One usually doesn't need to bother for algorithms, because it's clear from the context: the limit commonly taken is as a single parameter approaches infinity. Your description is correct when considering only N, but you could consider more than just N.

One of my friend been asked with a question, Retrieving the max top 100 numbers from one hundred million of numbers, in a recent job interview. Do you have any idea to come up with an efficient way to solve it?

Answer:

Run them all through a [min-heap](http://en.wikipedia.org/wiki/Heap_%28data_structure%29) of size 100: for each input number k, replace the current min m withmax(k, m). Afterwards the heap holds the 100 largest inputs.

A search engine like Lucene can use this method, with refinements, to choose the most-relevant search answers.

**Edit:** I fail the interview -- I got the details wrong twice (after having done this before, in production). Here's code to check it; it's almost the same as Python's standard heapq.nlargest():

import heapq

def funnel(n, numbers):

if n == 0: return []

heap = numbers[:n]

heapq.heapify(heap)

for k in numbers[n:]:

if heap[0] < k:

heapq.heapreplace(heap, k)

return heap

>>> funnel(4, [3,1,4,1,5,9,2,6,5,3,5,8])

[5, 8, 6, 9]

A point in 3-d is defined by (x,y,z). Distance d between any two points (X,Y,Z) and (x,y,z) is d= Sqrt[(X-x)^2 + (Y-y)^2 + (Z-z)^2]. Now there are a million entries in a file, each entry is some point in space, in no specific order. Given any point (a,b,c) find the nearest 10 points to it. How would you store the million points and how would you retrieve those 10 points from that data structure.

Million points is a small number. The most straightforward approach works here (code based on KDTree is slower (for querying only one point)).

**Brute-force approach (time ~1 second)**

#!/usr/bin/env python  
import numpy  
  
NDIM = 3 # number of dimensions  
  
# read points into array  
a = numpy.fromfile('million\_3D\_points.txt', sep=' ')  
a.shape = a.size / NDIM, NDIM  
  
point = numpy.random.uniform(0, 100, NDIM) # choose random point  
print 'point:', point  
d = ((a-point)\*\*2).sum(axis=1)  # compute distances  
ndx = d.argsort() # indirect sort   
  
# print 10 nearest points to the chosen one  
import pprint  
pprint.pprint(zip(a[ndx[:10]], d[ndx[:10]]))

Run it:

$ time python nearest.py

point: [ 69.06310224 2.23409409 50.41979143]

[(array([ 69., 2., 50.]), 0.23500677815852947),

(array([ 69., 2., 51.]), 0.39542392750839772),

(array([ 69., 3., 50.]), 0.76681859086988302),

(array([ 69., 3., 50.]), 0.76681859086988302),

(array([ 69., 3., 51.]), 0.9272357402197513),

(array([ 70., 2., 50.]), 1.1088022980015722),

(array([ 70., 2., 51.]), 1.2692194473514404),

(array([ 70., 2., 51.]), 1.2692194473514404),

(array([ 70., 3., 51.]), 1.801031260062794),

(array([ 69., 1., 51.]), 1.8636121147970444)]

real 0m1.122s

user 0m1.010s

sys 0m0.120s

Here's the script that generates million 3D points:

#!/usr/bin/env python

import random

for \_ in xrange(10\*\*6):

print ' '.join(str(random.randrange(100)) for \_ in range(3))

Output:

$ head million\_3D\_points.txt

18 56 26

19 35 74

47 43 71

82 63 28

43 82 0

34 40 16

75 85 69

88 58 3

0 63 90

81 78 98

You could use that code to test more complex data structures and algorithms (for example, whether they actually consume less memory or faster then the above simplest approach). It is worth noting that at the moment it is the only answer that contains working code.

**Solution based on**[**KDTree**](http://docs.scipy.org/doc/scipy/reference/spatial.html#spatial-data-structures-and-algorithms)**(time ~1.4 seconds)**

#!/usr/bin/env python  
import numpy  
  
NDIM = 3 # number of dimensions  
  
# read points into array  
a = numpy.fromfile('million\_3D\_points.txt', sep=' ')  
a.shape = a.size / NDIM, NDIM  
  
point =  [ 69.06310224,   2.23409409,  50.41979143] # use the same point as above  
print 'point:', point  
  
  
from scipy.spatial import KDTree  
  
# find 10 nearest points  
tree = KDTree(a, leafsize=a.shape[0]+1)  
distances, ndx = tree.query([point], k=10)  
  
# print 10 nearest points to the chosen one  
print a[ndx]

Run it:

$ time python nearest\_kdtree.py

point: [69.063102240000006, 2.2340940900000001, 50.419791429999997]

[[[ 69. 2. 50.]

[ 69. 2. 51.]

[ 69. 3. 50.]

[ 69. 3. 50.]

[ 69. 3. 51.]

[ 70. 2. 50.]

[ 70. 2. 51.]

[ 70. 2. 51.]

[ 70. 3. 51.]

[ 69. 1. 51.]]]

real 0m1.359s

user 0m1.280s

sys 0m0.080s

**Partial sort in C++ (time ~1.1 seconds)**

// $ g++ nearest.cc && (time ./a.out < million\_3D\_points.txt )  
#include <algorithm>  
#include <iostream>  
#include <vector>  
  
#include <boost/lambda/lambda.hpp>  // \_1  
#include <boost/lambda/bind.hpp>    // bind()  
#include <boost/tuple/tuple\_io.hpp>  
  
namespace {  
  typedef double coord\_t;  
  typedef boost::tuple<coord\_t,coord\_t,coord\_t> point\_t;  
  
  coord\_t distance\_sq(const point\_t& a, const point\_t& b) { // or boost::geometry::distance  
    coord\_t x = a.get<0>() - b.get<0>();  
    coord\_t y = a.get<1>() - b.get<1>();  
    coord\_t z = a.get<2>() - b.get<2>();  
    return x\*x + y\*y + z\*z;  
  }  
}  
  
int main() {  
  using namespace std;  
  using namespace boost::lambda; // \_1, \_2, bind()  
  
  // read array from stdin  
  vector<point\_t> points;  
  cin.exceptions(ios::badbit); // throw exception on bad input  
  while(cin) {  
    coord\_t x,y,z;  
    cin >> x >> y >> z;      
    points.push\_back(boost::make\_tuple(x,y,z));  
  }  
  
  // use point value from previous examples  
  point\_t point(69.06310224, 2.23409409, 50.41979143);  
  cout << "point: " << point << endl;  // 1.14s  
  
  // find 10 nearest points using partial\_sort()   
  // Complexity: O(N)\*log(m) comparisons (O(N)\*log(N) worst case for the implementation)  
  const size\_t m = 10;  
  partial\_sort(points.begin(), points.begin() + m, points.end(),   
               bind(less<coord\_t>(), // compare by distance to the point  
                    bind(distance\_sq, \_1, point),   
                    bind(distance\_sq, \_2, point)));  
  for\_each(points.begin(), points.begin() + m, cout << \_1 << "\n"); // 1.16s  
}

Run it:

g++ -O3 nearest.cc && (time ./a.out < million\_3D\_points.txt )

point: (69.0631 2.23409 50.4198)

(69 2 50)

(69 2 51)

(69 3 50)

(69 3 50)

(69 3 51)

(70 2 50)

(70 2 51)

(70 2 51)

(70 3 51)

(69 1 51)

real 0m1.152s

user 0m1.140s

sys 0m0.010s

**Priority Queue in C++ (time ~1.2 seconds)**

#include <algorithm>           // make\_heap  
#include <functional>          // binary\_function<>  
#include <iostream>  
  
#include <boost/range.hpp>     // boost::begin(), boost::end()  
#include <boost/tr1/tuple.hpp> // get<>, tuple<>, cout <<  
  
namespace {  
  typedef double coord\_t;  
  typedef std::tr1::tuple<coord\_t,coord\_t,coord\_t> point\_t;  
  
  // calculate distance (squared) between points `a` & `b`  
  coord\_t distance\_sq(const point\_t& a, const point\_t& b) {   
    // boost::geometry::distance() squared  
    using std::tr1::get;  
    coord\_t x = get<0>(a) - get<0>(b);  
    coord\_t y = get<1>(a) - get<1>(b);  
    coord\_t z = get<2>(a) - get<2>(b);  
    return x\*x + y\*y + z\*z;  
  }  
  
  // read from input stream `in` to the point `point\_out`  
  std::istream& getpoint(std::istream& in, point\_t& point\_out) {      
    using std::tr1::get;  
    return (in >> get<0>(point\_out) >> get<1>(point\_out) >> get<2>(point\_out));  
  }  
  
  // Adaptable binary predicate that defines whether the first  
  // argument is nearer than the second one to given reference point  
  template<class T>  
  class less\_distance : public std::binary\_function<T, T, bool> {  
    const T& point;  
  public:  
    less\_distance(const T& reference\_point) : point(reference\_point) {}  
  
    bool operator () (const T& a, const T& b) const {  
      return distance\_sq(a, point) < distance\_sq(b, point);  
    }   
  };  
}  
  
int main() {  
  using namespace std;  
  
  // use point value from previous examples  
  point\_t point(69.06310224, 2.23409409, 50.41979143);  
  cout << "point: " << point << endl;  
  
  const size\_t nneighbours = 10; // number of nearest neighbours to find  
  point\_t points[nneighbours+1];  
  
  // populate `points`  
  for (size\_t i = 0; getpoint(cin, points[i]) && i < nneighbours; ++i)  
    ;  
  
  less\_distance<point\_t> less\_distance\_point(point);  
  make\_heap  (boost::begin(points), boost::end(points), less\_distance\_point);  
  
  // Complexity: O(N\*log(m))  
  while(getpoint(cin, points[nneighbours])) {  
    // add points[-1] to the heap; O(log(m))  
    push\_heap(boost::begin(points), boost::end(points), less\_distance\_point);   
    // remove (move to last position) the most distant from the  
    // `point` point; O(log(m))  
    pop\_heap (boost::begin(points), boost::end(points), less\_distance\_point);  
  }  
  
  // print results  
  push\_heap  (boost::begin(points), boost::end(points), less\_distance\_point);  
  //   O(m\*log(m))  
  sort\_heap  (boost::begin(points), boost::end(points), less\_distance\_point);  
  for (size\_t i = 0; i < nneighbours; ++i) {  
    cout << points[i] << ' ' << distance\_sq(points[i], point) << '\n';    
  }  
}

Run it:

$ g++ -O3 nearest.cc && (time ./a.out < million\_3D\_points.txt )

point: (69.0631 2.23409 50.4198)

(69 2 50) 0.235007

(69 2 51) 0.395424

(69 3 50) 0.766819

(69 3 50) 0.766819

(69 3 51) 0.927236

(70 2 50) 1.1088

(70 2 51) 1.26922

(70 2 51) 1.26922

(70 3 51) 1.80103

(69 1 51) 1.86361

real 0m1.174s

user 0m1.180s

sys 0m0.000s

**Linear Search -based approach (time ~1.15 seconds)**

// $ g++ -O3 nearest.cc && (time ./a.out < million\_3D\_points.txt )  
#include <algorithm>           // sort  
#include <functional>          // binary\_function<>  
#include <iostream>  
  
#include <boost/foreach.hpp>  
#include <boost/range.hpp>     // begin(), end()  
#include <boost/tr1/tuple.hpp> // get<>, tuple<>, cout <<  
  
#define foreach BOOST\_FOREACH  
  
namespace {  
  typedef double coord\_t;  
  typedef std::tr1::tuple<coord\_t,coord\_t,coord\_t> point\_t;  
  
  // calculate distance (squared) between points `a` & `b`  
  coord\_t distance\_sq(const point\_t& a, const point\_t& b);  
  
  // read from input stream `in` to the point `point\_out`  
  std::istream& getpoint(std::istream& in, point\_t& point\_out);      
  
  // Adaptable binary predicate that defines whether the first  
  // argument is nearer than the second one to given reference point  
  class less\_distance : public std::binary\_function<point\_t, point\_t, bool> {  
    const point\_t& point;  
  public:  
    explicit less\_distance(const point\_t& reference\_point)   
        : point(reference\_point) {}  
    bool operator () (const point\_t& a, const point\_t& b) const {  
      return distance\_sq(a, point) < distance\_sq(b, point);  
    }   
  };  
}  
  
int main() {  
  using namespace std;  
  
  // use point value from previous examples  
  point\_t point(69.06310224, 2.23409409, 50.41979143);  
  cout << "point: " << point << endl;  
  less\_distance nearer(point);  
  
  const size\_t nneighbours = 10; // number of nearest neighbours to find  
  point\_t points[nneighbours];  
  
  // populate `points`  
  foreach (point\_t& p, points)  
    if (! getpoint(cin, p))  
      break;  
  
  // Complexity: O(N\*m)  
  point\_t current\_point;  
  while(cin) {  
    getpoint(cin, current\_point); //NOTE: `cin` fails after the last  
                                  //point, so one can't lift it up to  
                                  //the while condition  
  
    // move to the last position the most distant from the  
    // `point` point; O(m)  
    foreach (point\_t& p, points)  
      if (nearer(current\_point, p))   
        // found point that is nearer to the `point`   
  
        //NOTE: could use insert (on sorted sequence) & break instead  
        //of swap but in that case it might be better to use  
        //heap-based algorithm altogether  
        std::swap(current\_point, p);  
  }  
  
  // print results;  O(m\*log(m))  
  sort(boost::begin(points), boost::end(points), nearer);  
  foreach (point\_t p, points)  
    cout << p << ' ' << distance\_sq(p, point) << '\n';    
}  
  
namespace {  
  coord\_t distance\_sq(const point\_t& a, const point\_t& b) {   
    // boost::geometry::distance() squared  
    using std::tr1::get;  
    coord\_t x = get<0>(a) - get<0>(b);  
    coord\_t y = get<1>(a) - get<1>(b);  
    coord\_t z = get<2>(a) - get<2>(b);  
    return x\*x + y\*y + z\*z;  
  }  
  
  std::istream& getpoint(std::istream& in, point\_t& point\_out) {      
    using std::tr1::get;  
    return (in >> get<0>(point\_out) >> get<1>(point\_out) >> get<2>(point\_out));  
  }  
}

Measurements shows that most of the time is spent reading array from the file, actual computations take on order of magnitude less time.

Answer 2

If the million entries are already in a file, there's no need to load them all into a data structure in memory. Just keep an array with the top-ten points found so far, and scan over the million points, updating your top-ten list as you go.

This is O(n) in the number of points.

I think this is a tricky question that tests if you don't try to overdo things.

Consider the simplest algorithm people already have given above: keep a table of ten best-so-far candidates and go through all the points one by one. If you find a closer point than any of the ten best-so-far, replace it. What's the complexity? Well, we have to look at each point from the file once, calculate it's distance (or square of the distance actually) and compare with the 10th closest point. If it's better, insert it in the appropriate place in the 10-best-so-far table.

So what's the complexity? We look at each point once, so it's n computations of the distance and n comparisons. If the point is better, we need to insert it in the right position, this requires some more comparisons, but it's a constant factor since the table of best candidates is of a constant size 10.

We end up with an algorithm that runs in linear time, O(n) in the number of points.

But now consider what is the **lower bound** on such an algorithm? If there is no order in the input data, we have to look at each point to see if it's not one of the closest ones. So as far as I can see, the lower bound is Omega(n) and thus the above algorithm is optimal.

WHY is std::map implemented as a red-black tree? There are several balanced BST out there, what were design trade-offs in choosing red-black tree?

Answer:

It really depends on the usage. AVL tree has higher complexity of rebalancing. So if your application doesn't have too many insertion and deletion operations, but weights heavily on searching, then AVL tree probably is a good choice.

std::map uses BR-tree as it gets a reasonable trade-off between the complexity of node insertion/deletion and searching.

Probably the two most common self balancing tree algorithms are [Red-Black trees](http://en.wikipedia.org/wiki/Red-black_tree) and [AVL trees](http://en.wikipedia.org/wiki/AVL_tree). To balance the tree after an insertion/update both algorithms use the notion of rotations where the nodes of the tree are rotated to perform the re-balancing.

While in both algorithms the insert/delete operations are O(log n), in the case of Red-Black tree re-balancing rotation is an [O(1)](http://en.wikipedia.org/wiki/Big_O_notation) operation while with AVL this is a [O(log n)](http://en.wikipedia.org/wiki/Big_O_notation) operation, making the Red-Black tree more efficient in this aspect of the re-balancing stage and one of the possible reasons that it is more commonly used.

Red-Black trees are used in most collection libraries, including the offerings from Java and Microsoft .NET Framework.

AVL trees have a maximum height of 1.44logn, while RB trees have a maximum of 2logn. Inserting an element in a AVL may imply a rebalance at one point in the tree. The rebalancing finishes the insertion. After insertion of a new leaf, updating the ancestors of that leaf has to be done up to the root, or up to a point where the two subtrees are of equal depth. The probability of having to update k nodes is 1/3^k. Rebalancing is O(1). Removing an element may imply more than one rebalancing (up to half the depth of the tree).

RB-trees are B-trees of order 4 represented as binary search trees. A 4-node in the B-tree results in two levels in the equivalent BST. In the worst case, all the nodes of the tree are 2-nodes, with only one chain of 3-nodes down to a leaf. That leaf will be at a distance of 2logn from the root.

Going down from the root to the insertion point, one has to chnage 4-nodes into 2-nodes, to make sure any insertion will not saturate a leaf. Coming back from the insertion, all these nodes have to be analysed to make sure they correctly represent 4-nodes. This can also be done going down in the tree. The global cost will be the same. There is no free lunch! Removing an element from the tree is of the same order.

All these trees require that nodes carry information on height, weight, color, etc. Only Splay trees are free from such additional info. But most people are afraid of Splay trees, because of the ramdomness of their structure!

Finally, trees can also carry weight information in the nodes, permitting weight balancing. Various schemes can be applied. One should rebalance when a subtree contains more than 3 times the number of elements of the other subtree. Rebalancing is again done either throuh a single or double rotation. This means a worst case of 2.4logn. One can get away with 2 times instead of 3, a much better ratio, but it may mean leaving a little less thant 1% of the subtrees unbalanced here and there. Tricky!

Which type of tree is the best? AVL for sure. They are the simplest to code, and have their worst height nearest to logn. For a tree of 1000000 elements, an AVL will be at most of height 29, a RB 40, and a weight balanced 36 or 50 depending on the ratio.

There are a lot of other variables: randomness, ratio of adds, deletes, searches, etc.

<http://stackoverflow.com/questions/4168002/object-oriented-design-for-a-chess-game>

<http://stackoverflow.com/questions/764933/amazon-interview-question-design-an-oo-parking-lot>