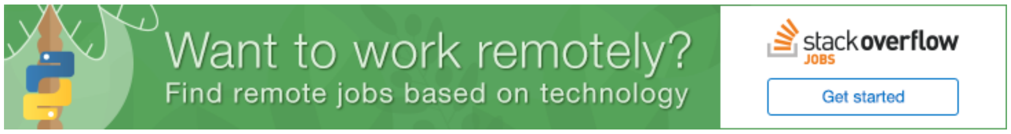
Quicksort: Choosing the pivot



When implementing Quicksort, one of the things you have to do is to choose a pivot. But when I look at pseudocode like the one below, it is not clear how I should choose the pivot. First element of list? Something else?

```
function quicksort(array)
    var list less, greater
    if length(array) ≤ 1
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x ≤ pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

Can someone help me grasp the concept of choosing a pivot and whether or not different scenarios call for different strategies.

algorithm sorting pseudocode quicksort

edited Nov 30 '13 at 16:28 Saurin 19 2

asked Oct 2 '08 at 19:37 Jacob T. Nielsen 1,610 5 19 29

stackoverflow.com/questions/1688264/improving-the-quick-sort – Pale Blue Dot Nov 28 '09 at 16:09

12 Answers

Choosing a random pivot minimizes the chance that you will encounter worst-case O(n^2) performance (always choosing first or last would cause worst-case performance for nearly-sorted or nearly-reverse-sorted data). Choosing the middle element would also be acceptable in the majority of cases.

Also, if you are implementing this yourself, there are versions of the algorithm that work in-place (i.e. without creating two new lists and then concatenating them).

edited Oct 2 '08 at 19:46

answered Oct 2 '08 at 19:41 Kip 54.8k 72 178 223

7 I would second the notion that implementing a search yourself might not be worth the effort. Also, be careful how you're picking random numbers, since random number generators are kinda slow sometimes. – PeterAllenWebb Oct 3 '08 at 18:56

It depends on your requirements. Choosing a pivot at random makes it harder to create a data set that generates O(N^2) performance. 'Median-of-three' (first, last, middle) is also a way of avoiding problems. Beware of relative performance of comparisons, though; if your comparisons are costly, then Mo3 does more comparisons than choosing (a single pivot value) at random. Database records can be costly to compare.

Update: Pulling comments into answer.

mdkess asserted:

> 'Median of 3' is NOT first last middle. Choose three random indexes, and take the middle value of this. The whole point is to make sure that your choice of pivots is not deterministic - if it is, worst case data can be quite easily generated.

To which I responded:

- Analysis Of Hoare's Find Algorithm With Median-Of-Three Partition (1997) by P Kirschenhofer, H Prodinger, C Martínez supports your contention (that 'median-of-three' is three random items).
- There's an article described at portal.acm.org that is about 'The Worst Case Permutation for Median-of-Three Quicksort' by Hannu Erkiö, published in The Computer Journal, Vol 27, No 3, 1984. [Update 2012-02-26: Got the text for the article. Section 2 'The Algorithm' begins: '*By using the median of the first, middle and last elements of A[L:R], efficient partitions into parts of fairly equal sizes can be achieved in most practical situations.*' Thus, it is discussing the first-middle-last Mo3 approach.]
- Another short article that is interesting is by M. D. McIlroy, "A Killer Adversary for Quicksort", published in Software-Practice and Experience, Vol. 29(0), 1–4 (0 1999). It explains how to make almost any Quicksort behave quadratically.
- AT&T Bell Labs Tech Journal, Oct 1984 "Theory and Practice in the Construction of a Working Sort Routine" states "Hoare suggested partitioning around the median of several randomly selected lines. Sedgewick [...] recommended choosing the median of the first [...] last [...] and middle". This indicates that both techniques for 'median-of-three' are known in the literature. (Update 2014-11-23: The article appears to be available at IEEE Xplore or from Wiley — if you have membership or are prepared to pay a fee.)
- 'Engineering a Sort Function' by J L Bentley and M D McIlroy, published in Software Practice and Experience, Vol 23(11), November 1993, goes into an extensive discussion of the issues, and they chose an adaptive partitioning algorithm based in part on the size of the data set. There is a lot of discussion of trade-offs for various approaches.
- A Google search for 'median-of-three' works pretty well for further tracking.

Thanks for the information; I had only encountered the deterministic 'median-of-three' before.

edited Nov 24 '14 at 5:13 answered Oct 2 '08 at 19:42
Jonathan Leffler
462k 65 530 852

2 Median of 3 is NOT first last middle. Choose three random indexes, and take the middle value of this. The whole point is to make sure that your choice of pivots is not deterministic - if it is, worst case data can be quite easily generated. – mindvirus Feb 3 '09 at 18:30

I was reading abt introsort which combines good features of both quicksort and heapsort. The approach to select pivot using median of three might not always be favourable. – Sumit Kumar Saha Feb 10 '13 at 20:31

3 The problem with choosing random indices is that random number generators are pretty expensive. While it does not increase the big-O cost of sorting, it will probably make things slower than if you had just picked the first, last, and middle elements. (In the real world, I bet nobody is making contrived situations to slow down your quick sort.) – Kevin Chen Nov 24 '14 at 5:01

Heh, I just taught this class.

There are several options.
Simple: Pick the first or last element of the range. (bad on partially sorted input) Better: Pick the item in the middle of the range. (better on partially sorted input)

However, picking any arbitrary element runs the risk of poorly partitioning the array of size n into two arrays of size 1 and n-1. If you do that often enough, your quicksort runs the risk of becoming O(n^2).

One improvement I've seen is pick median(first, last, mid); In the worst case, it can still go to O(n^2), but probabilistically, this is a rare case.

For most data, picking the first or last is sufficient. But, if you find that you're running into worst case scenarios often (partially sorted input), the first option would be to pick the central value(Which is a statistically good pivot for partially sorted data).

If you're still running into problems, then go the median route.

answered Oct 2 '08 at 19:46
Chris Cudmore
18.4k 10 44 87

1 We did an experiment in our class, getting the k smallest elements from an array in sorted order. We generated random arrays then used either a min-heap, or randomized select and fixed pivot quicksort and counted the number of comparisons. On this "random" data, the second solution performed worse on average than the first. Switching to a randomized pivot solve the performance problem. So even for supposedly random data, fixed pivot performs significantly worse than randomized pivot. – Robert S. Barnes May 9 '13 at 5:59

Never ever choose a fixed pivot - this can be attacked to exploit your algorithm's worst case O(n^2) runtime, which is just asking for trouble. Quicksort's worst case runtime occurs when partitioning results in one array of 1 element, and one array of n-1 elements. Suppose you choose the first element as your partition. If someone feeds an array to your algorithm that is in decreasing order, your first pivot will be the biggest, so everything else in the array will move to the left of it. Then when you recurse, the first element will be the biggest again, so once more you put everything to the left of it, and so on.

A better technique is the median-of-3 method, where you pick three elements at random, and choose the middle. You know that the element that you choose won't be the the first or the last, but also, by the central limit theorem, the distribution of the middle element will be normal, which means that you will tend towards the middle (and hence, n lg n time).

If you absolutely want to guarantee O(nlgn) runtime for the algorithm, the columns-of-5 method for finding the median of an array runs in O(n) time, which means that the recurrence equation for quicksort in the worst case will be T(n) = O(n) (find the median) + O(n) (partition) + 2T(n/2) (recurse left and right.) By the Master Theorem, this is O(n lg n). However, the constant factor will be huge, and if worst case performance is your primary concern, use a merge sort instead, which is only a little bit slower than quicksort on average, and guarantees O(nlgn) time (and will be much faster than this lame median quicksort).

Explanation of the Median of Medians Algorithm

edited Apr 21 '13 at 3:26
alan
123 2 2 9

answered Oct 25 '08 at 21:50
mindvirus
2,222 2 17 36

Don't try and get too clever and combine pivoting strategies. If you combined median of 3 with random pivot by picking the median of the first, last and a random index in the middle, then you'll still be vulnerable to many of the distributions which send median of 3 quadratic (so its actually worse than plain random pivot)

E.g a pipe organ distribution (1,2,3...N/2..3,2,1) first and last will both be 1 and the random index will be some number greater than 1, taking the median gives 1 (either first or last) and you get an extermely unbalanced partitioning.

answered Oct 26 '08 at 3:54
paperhorse
1,998 2 16 11

It is entirely dependent on how your data is sorted to begin with. If you think it will be pseudo-random then your best bet is to either pick a random selection or choose the middle.

answered Oct 2 '08 at 19:46
Joe Phillips
23.2k 21 69 120

If you are sorting a random-accessible collection (like an array), it's general best to pick the physical middle item. With this, if the array is all ready sorted (or nearly sorted), the two partitions will be close to even, and you'll get the best speed.

If you are sorting something with only linear access (like a linked-list), then it's best to choose the first item, because it's the fastest item to access. Here, however,if the list is already sorted, you're screwed -- one partition will always be null, and the other have everything, producing the worst time.

However, for a linked-list, picking anything besides the first, will just make matters worse. It pick the middle item in a listed-list, you'd have to step through it on each partition step -- adding a O(N/2) operation which is done logN times making total time O(1.5 N *log N) and that's if we know how long the list is before we start -- usually we don't so we'd have to step all the way through to count them, then step half-way through to find the middle, then step through a third time to do the actual partition: O(2.5N * log N)

edited Oct 2 '08 at 19:57

answered Oct 2 '08 at 19:42

James Curran
73.9k 23 139 222

It is easier to break the quicksort into three sections doing this

1. Exchange or swap data element function
2. The partition function
3. Processing the partitions

It is only slightly more inefficent than one long function but is alot easier to understand.

Code follows:

```
/* This selects what the data type in the array to be sorted is */

#define DATATYPE long

/* This is the swap function .. your job is to swap data in x & y .. how depends on
data type .. the example works for normal numerical data types .. like long I chose
above */

void swap (DATATYPE *x, DATATYPE *y){
  DATATYPE Temp;

  Temp = *x;        // Hold current x value
  *x = *y;          // Transfer y to x
  *y = Temp;        // Set y to the held old x value
};


/* This is the partition code */

int partition (DATATYPE list[], int l, int h){

  int i;
  int p;          // pivot element index
  int firsthigh;  // divider position for pivot element

  // Random pivot example shown for median   p = (l+h)/2 would be used
  p = l + (short)(rand() % (int)(h - l + 1)); // Random partition point

  swap(&list[p], &list[h]);                   // Swap the values
  firsthigh = l;                                  // Hold first high value
  for (i = l; i < h; i++)
    if(list[i] < list[h]) {                 // Value at i is less than h
      swap(&list[i], &list[firsthigh]);   // So swap the value
      firsthigh++;                        // Incement first high
    }
  swap(&list[h], &list[firsthigh]);           // Swap h and first high values
  return(firsthigh);                          // Return first high
};



/* Finally the body sort */

void quicksort(DATATYPE list[], int l, int h){

  int p;                                      // index of partition
  if ((h - l) > 0) {
    p = partition(list, l, h);              // Partition list
    quicksort(list, l, p - 1);        // Sort lower partion
    quicksort(list, p + 1, h);              // Sort upper partition
    };
  };
```

edited Mar 10 '11 at 15:59
James
4,347 12 28

answered Mar 10 '11 at 2:19
Uglybb
11 2

Ideally the pivot should be the middle value in the entire array. This will reduce the chances of getting worst case performance.

answered Apr 17 '13 at 14:57
Faizan
589 1 16 36

Quick sort's complexity varies greatly with the selection of pivot value. for example if you always choose first element as an pivot, algorithm's complexity becomes as worst as O(n^2). here is an smart method to choose pivot element- 1. choose the first, mid, last element of the array. 2. compare these three numbers and find the number which is greater than one and smaller than other i.e. median. 3. make this element as pivot element.

choosing the pivot by this method splits the array in nearly two half and hence the complexity reduces to O(nlog(n)).

answered Dec 5 '13 at 5:05
vivek
29 8

On the average, Median of 3 is good for small n. Median of 5 is a bit better for larger n. The ninther, which is the "median of three medians of three" is even better for very large n.

The higher you go with sampling the better you get as n increases, but the improvement dramatically slows down as you increase the samples. And you incur the overhead of sampling and sorting samples.

answered Oct 19 '16 at 10:04
S0lo
26 2

In a truly optimized implementation, the method for choosing pivot should depend on the array size - for a large array, it pays off to spend more time choosing a good pivot. Without doing a full analysis, I would guess "middle of O(log(n)) elements" is a good start, and this has the added bonus of not requiring any extra memory: Using tail-call on the larger partition and in-place partitioning, we use the same O(log(n)) extra memory at almost every stage of the algorithm.

answered Oct 8 '13 at 19:50
Morten Kloster
1

Finding the middle of 3 elements can be done in constant time . Any more, and we essentially have to sort the sub array. As n becomes large, we run right back into the sorting problem again. – Chris Cudmore Jun 10 '16 at 15:40