VOL.2 NO.2September 2013 - ISSN 2233 – 1859

Operations Research Society in Bosnia and Herzegovina

Southeast Europe Journal of Soft Computing

Available online:www.scjournal.com.ba

Randomized Algorithms in Bioinformatics: RANDOMIZEDCAMSORT

Mehmet Can, Hüseyin Lutin

International University of Sarajevo, Faculty of Engineering and Natural Sciences, Hrasnicka Cesta 15, Ilidža71210 Sarajevo, Bosnia and Herzegovina

Article Info

Article history:
Recivied 17 Sep.2013
Recivied in revised form 17 Oct 2013

Keywords:
Randomized algorithms, sorting problems, linear time algorithm

Abstract

Randomized algorithms make random decisions throughout their operation. At first glance, making random decisions does not seem particularly helpful. Basing an algorithm on random decisions sounds like a recipe for disaster, but the fact that a randomized algorithm undertakes a nondeterministic sequence of operations often means that, unlike deterministic algorithms; no input can reliably produce worst-case results. (Karp 1991). Randomized algorithms are often used in hard problems where an exact, polynomial-time algorithm is not known. In this paper we will see how randomized algorithms solve the Sorting problems.

1. INTRODUCTION

QUICKSORT discovered by Tony Hoare in 1962 (Hoare, 1962;Hoos, and Stützle 1998) is a fast and simple sorting technique. It selects an element *m*(typically, the first) from an array *c* and simply partitions the array into two subarrays: *csmall*, containing all elements from c that are smaller than *m*; and *clarge* containing all elements larger than *m*.

This partitioning can be done in linear time, and by following a divide-and conquer strategy, QUICKSORT recursively sorts each subarray in the same way. The sorted list is easily created by simply concatenating the sorted *csmall*, element *m*, and the sorted *clarge*. A pseudocode is as follows (Jones, and Pevzner 2004).

QUICKSORT(c)

1 if c consists of a single element
2 return c
3 m ← c1
4 Determine the set of elements csmall smaller than m
5 Determine the set of elements clarge larger than m
6 QUICKSORT(csmall)
7 QUICKSORT(clarge)
8 Combine csmall , m, and clarge into a single sorted array csorted
9 return csorted

It turns out that the running time of QUICKSORT depends on how lucky we are with our selection of the element *m*. If we happen to choose m in such a way that *c* is split into even halves (i.e., |csmall| = |clarge|), then

$$T(n) = 2T(n/2) + an,$$

where $T(n)$ represents the time taken by QUICKSORT to sort an array of n numbers, and an represents the time required to split the array of size n into two parts; a is a positive constant. It leads to $O(n \log n)$ running time. However, if we choose m in such a way that it splits c unevenly (e.g., an extreme case occurs when *csmall* is empty and*clarge* has $n-1$ elements), then the recurrence looks like

$$T(n) = T(n-1) + an$$

It leads to $O(n^2)$ running time, something we want to avoid. Indeed, QUICKSORT takes quadratic time to sort the array $(n, n-1, \ldots, 2, 1)$.

Worse yet, it requires $O(n^2)$ time to process $(1, 2, \ldots, n-1, n)$, which seems unnecessary since the array is already sorted.

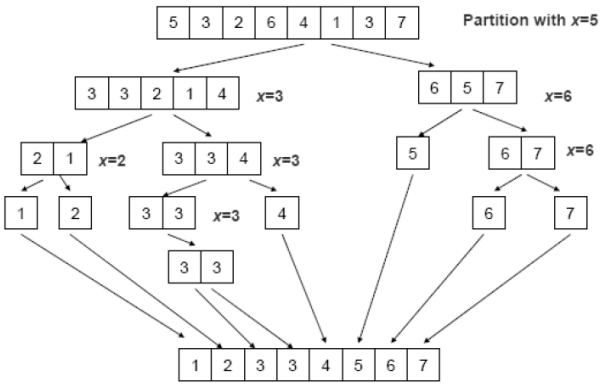

Figure 1. Implementation of QUICKSORT

EXAMPLE 1. A MATHEMATICA Code for

QUICKSORT

```
c= {18,14,15,5,20,11,4,1,6,7,10,2,19,16,13,9}
n=Length[c];
```

Divide

```
c1={}; c2={};
m=First[c]
Do[{If[c[[i]]<m,c1=Append[c1,c[[i]]],c2=Append[c2,c[[i]]]]},{i,1,n}]
c1 c2
8 {6,5,1,2,7,3,4} {8,13,16,11,20,12,19,17,15,9}
***
c=c1;
n=Length[c];
c11={}; c12={};
m=First[c]
Do[{If[c[[i]]<m,c11=Append[c11,c[[i]]],c12=Append[c12,c[[i]]]]},{i,1,n}]
c11 c12
6 {5,1,2,3,4} {6,7}
***
c=c11;
n=Length[c];
c111={}; c112={};
m=First[c]
Do[{If[c[[i]]<m,c111=Append[c111,c[[i]]],c112=Append[c112,c[[i]]]]},{i,1,n}]
c111 c112
5 {1,2,3,4} {5}
***
c=c2;
n=Length[c];
c21={}; c22={};
m=Take[c,{3}][[1]];
Do[{If[c[[i]]<m,c21=Append[c21,c[[i]]],c22=Append[c22,c[[i]]]]},{i,1,n}]
c21 c22
16 {8,13,11,12,15,9} {16,20,19,17}
***
c=c21;
n=Length[c];
c211={}; c212={};
m=Take[c,{3}][[1]]
Do[{If[c[[i]]<m,c211=Append[c211,c[[i]]],c212=Append[c212,c[[i]]]]},{i,1,n}]
c211 c212
11 {8,9} {13,11,12,15}
***
c=c212;
n=Length[c];
c2121={}; c2122={};
m=First[c]
Do[{If[c[[i]]<m,c2121=Append[c2121,c[[i]]],c2122=Append[c2122,c[[i]]]]},{i,1,n}]
c2121 c2122
13 {11,12} {13,15}
***
c=c22;
n=Length[c];
c221={}; c222={};
m=Take[c,{3}][[1]]
Do[{If[c[[i]]<m,c221=Append[c221,c[[i]]],c222=Append[c222,c[[i]]]]},{i,1,n}]
c221 c222
19 {16,17} {20,19}
```

Concur

```
sc11=Join[c111,c112]
{1,2,3,4,5}
```

sc1=Join[sc11,c12]

{1,2,3,4,5,6,7}

c222=Reverse[c222]

{19,20}

sc22=Join[c221,c222]

{16,17,19,20}

sc212=Join[c2121,c2122]

{11,12,13,15}

sc21=Join[c211,c212]

{8,9,13,11,12,15}

sc2=Join[sc21,sc22]

{8,9,13,11,12,15,16,17,19,20}

sc=Join[sc1,sc2]

{1,2,3,4,5,6,7,8,9,13,11,12,15,16,17,19,20}

2. RANDOMIZATION OF THE SORTING ALGORITHM

In QUICKSORT algorithm, if we can choose a good "splitter" m that breaks an array into two equal parts, we might improve the running time. To achieve $O(n \log n)$ running time, it is not actually necessary to find a perfectly equal (50/50) split. For example, a split into approximately equal parts of size, say, 51/49 will also work. In fact, one can prove that the algorithm will achieve $O(n \log n)$ running time as long as the sets *csmall* and *clarge* are both larger in size than *n/4* , which implies that, of *n* possible choices for *m*, at least

3/4n – 1/4n = ½n

of them make good splitters. In other words, if we choose m uniformly at random (i.e., every element of *c* has the same probability to be chosen), there is at least a 50% chance that it will be a good splitter. This observation motivates the following randomized algorithm (Motwani, and Raghavan 1996):

RANDOMIZEDQUICKSORT(c)
1 if c consists of a single element
2 return c
3 Choose element m uniformly at random from c
4 Determine the set of elements csmall smaller than m
5 Determine the set of elements clarge larger than m
6 RANDOMIZEDQUICKSORT(csmall)
7 RANDOMIZEDQUICKSORT(clarge)
8 Combine csmall , m, and clarge into a single sorted array csorted
9 return csorted

RANDOMIZEDQUICKSORT is a very fast algorithm in practice but its worst case running time remains O(n2) since there is still a possibility that it selects bad splitters. Although the behavior of a randomized algorithm varies on the same input from one execution to the next, one can prove that its expected running time is O(n log n).

The running time of a randomized algorithm is a random variable, and computer scientists are often interested in the mean value of this random variable. This is referred to as the expected running time (Floyd,and Rivest1975).

The key advantage of randomized algorithms is performance: for many practical problems randomized algorithms are faster in the sense of expected running time than the best known deterministic algorithms. Another attractive feature of randomized algorithms, as illustrated by RANDOMIZED-QUICKSORT is their simplicity.

We emphasize that RANDOMIZEDQUICKSORT, despite makingrandom decisions, always returns the correct solution of the sorting problem. The only variable from one run to another is its running time, not the result. In contrast, other randomized algorithms we consider in this chapter usually produce incorrect (or, more gently, approximate) solutions. Randomized algorithms that always return correct answers are called Las Vegas algorithms (Lubi, 1993; Hoos, and. Stützle, 1998; Luby, et. Al. 1993.), while algorithms that do not are called Monte Carlo algorithms. Of course, computer scientists prefer Las Vegas algorithms to Monte Carlo algorithms but the former are often difficult to come by (Liu 2001). Although for some applicationsMonte Carlo algorithms are not appropriate (when approximate solutions are of no value), they have been popular in different applications for over a hundred years and often provide good approximations to optimal solutions (Nielsen, 2009).

EXAMPLE2. A MATHEMATICA Code for RANDOMIZEDQUICKSORT

c={13,21,19,24,14,7,15,22,23,12,3,1,10}
n=Length[c];

Divide

c1={};c2={};
m=RandomSample[c,1][[1]]
Do[If[c[[i]]<m,c1=Append[c1,c[[i]]],c2=Append[c2,c[[i]]]],{i,1,n}]
c1c2
14{13,7,12,3,1,10}{21,19,24,14,15,22,23}
***
c=c1;
n=Length[c];
c11={};c12={};
m=RandomSample[c,1][[1]]
Do[{If[c[[i]]<m,c11=Append[c11,c[[i]]],c12=Append[c12,c[[i]]]]},{i,1,n}]

```
c11c12
12{7,3,1,10}{13,12}
***
c=c11;
n=Length[c];
c111={};c112={};
m=RandomSample[c,1][[1]]
Do[{If[c[[i]]<m,c111=Append[c111,c[[i]]],c112=Append[
c112,c[[i]]]]},{i,1,n}]
c111c112
7{3,1}{7,10}
***
c=c2;
n=Length[c];
c21={};c22={};
m=RandomSample[c,1][[1]];
Do[{If[c[[i]]<m,c21=Append[c21,c[[i]]],c22=Append[c22,
c[[i]]]]},{i,1,n}]
c21c22
{21,19,14,15}{24,22,23}
***
c=c21;
n=Length[c];
c211={};c212={};
m=RandomSample[c,1][[1]];
Do[{If[c[[i]]<m,c211=Append[c211,c[[i]]],c212=Append[
c212,c[[i]]]]},{i,1,n}]
c211c212
{14,15}{21,19}
***
c=c22;
n=Length[c];
c221={};c222={};
m=RandomSample[c,1][[1]];
Do[{If[c[[i]]<m,c221=Append[c221,c[[i]]],c222=Append[
c222,c[[i]]]]},{i,1,n}]
c221c222
{22,23}{24}
```

Concur

```
sc111=Reverse[c111]
{1,3}
sc11=Join[sc111,c112]
{1,3,7,10}
sc12=Reverse[c12]
{12,13}
sc1=Join[sc11,sc12]
{1,3,7,10,12,13}

sc212=Reverse[c212]
{19,21}
sc21=Join[c211,sc212]
{14,15,19,21}

sc22=Join[c221,c222]
{22,23,24}

sc2=Join[sc21,sc22]
{19,21,28,30,41,42,43,44,45,47,49}

sc2=Join[sc21,sc22]
{14,15,19,21,22,23,24}

sc=Join[sc1,sc2]
{1,3,7,10,12,13,14,15,19,21,22,23,24}
```

3. RANDOMIZED CUT AND MERGE SHORTENS EXPECTED RUNNING TIME

RANDOMIZEDQUICKSORT algorithm is a divide and concur algorithm. We realized that it is possible to develop an algorithm that after fist partitioning does not continue the partitioning of smaller strings, instead merges the sets *csmall* and *clarge.* Iteration continues till to obtain the correct sorting. The correct result is guaranteed hence this algorithm is classified as a Las Vegas algorithm.

Here also if we can choose a good "splitter" m that breaks an array into two equal parts, we might improve the running time. In fact, one can prove that the algorithm will achieve $O(n \log n)$ running time as long as the sets *csmall* and *clarge*are both larger in size than n/4.This observation motivates the following randomized algorithm:

RANDOMIZEDCAMQUICKSORT(c)

1 if c consists of a single element
2 return c
3 Choose element m uniformly at random from c
4 Determine the set of elements csmall smaller than m
5 Determine the set of elements clarge larger than or equal to m
8 Combine csmall and clarge into a single sorted array csorted
9. If c(ı)>c(i+1), for some 1<i<n
RANDOMIZEDCAMQUICKSORT(c)
10 return csorted

10	2	9	6	5	4	7	13	3	8	1	14	12	15
1	2	4	3	6	5	7	8	9	10	13	12	14	15
1	2	4	3	5	6	7	8	9	10	12	13	14	15
1	2	3	4	5	6	7	8	9	10	12	13	14	15
1	2	3	4	5	6	7	8	9	10	12	13	14	15

Figure 2. Implementation of RANDOMIZEDCAMQUICK-SORT

The RANDOMIZEDCAMQUICKSORT(c) is faster than the RANDOMIZEDQUICKSORT algorithm since it does not need the necessary steps for a divide-and conquer strategy. Its worst case running time is O(n) since there is a possibility that it selects bad splitters. Although the behavior of a randomized algorithm varies on the same input from one execution to the next, one can prove that its expected running time is O(n).

The key advantage of randomized algorithms is performance: for many practical problems randomized algorithms are faster (in the sense of expected running time) than the best known deterministic algorithms. Another attractive feature of rand-omized algorithms, as illustrated by RANDOMIZEDQUICK-SORT, and RANDOMIZEDCAMQUICKSORT algorithms, is their simplicity.

EXAMPLE 3. A MATHEMATICA Code for RANDOMIZEDCAMQUICKSORT(c)

```
c={17,3,1,39,16,2,25,5,33,19,4,38,11,23,29,13,31,6,32,18,
35,20,8,12,37,40,9,36,28,27,7,24,30}
n=Length[c];
maxiter=80;
Do[{c1={},c2={},m=RandomSample[c,1][[1]],
Do[{If[c[[i]]<m,c1=Append[c1,c[[i]]],c2=Append[c2,c[[i]
]]]},{i,1,n}],c=Join[c1,c2],s = 0, Do[{If[c[[i + 1]] > c[[i]],
s = s + 1]}, {i, 1, n - 1}],
 If[s == n - 1, {iter = k, ct = iter/N[n Log[2, n]],
Print["running time", " ", ct, " ", "sorted c", "  ", c],
Break[]}]}, {k, 1, maxiter}]},{k,1,maxiter}]
{1,2,3,4,5,6,7,8,9,11,12,13,16,17,18,19,20,23,24,25,28,27,
29,30,31,32,33,35,36,37,38,39,40}
```

The running time of this randomized algorithm is also a random variable. The mean value of this random variableis referred as the expected running time.

For RANDOMIZEDCAMQUICKSORT(c) expected running time is around$O(n\ log\ n)$.

Table 1. Expected running time of RANDOMIZEDCAM-QUICKSORT is around$O(n\ log\ n)$.

n	ct = iter/N[n Log[2, n]]
100	0.675593
1000	0.669025
5000	0.589887

4. CONCLUSIONS

For many practical problems randomized algorithms are faster than the best known deterministic algorithms. Another attractive feature of randomized algorithms, as illustrated by RANDOMIZEDQUICKSORT, and RANDOMIZEDCAM-QUICKSORT algorithms, is their simplicity. For RANDOM-IZEDCAMQUICKSORT(c) expected running time is around$O(n\ log\ n)$.

REFERENCES

R. W. Floyd, and R.L. Rivest (1975) Expectedtime bounds for selection. Commun. ACM 18, pp. 165–172.

C. A. R. Hoare (1962) Quicksort, Computer Journal, 5:10–15.

H. H. Hoos, T. Stützle, (1998) Evaluating Las Vegas algorithms: pitfalls and remedies, N Proceedings Of The Fourteenth Conference On Uncertainty In Artificial Intelligence (Uai-98).

N. C. Jones and P. A. Pevzner. (2004) An Introduction to Bioinformatics Algorithms. The MIT Press.

R.M. Karp (1991)An introduction to randomizedalgorithms, Discrete Applied Mathematics 34, pp. 165-201.

J.S. Liu (2001) Monte Carlo strategies in scientific computing, Harvard Univ.

M. Luby (1993) Optimal speedup of Las Vegas algorithms, Information Processing Letters, 47, 4, pp 173–180.

M. Luby, A. Sinclair, A., and D. Zuckerman (1993). Optimal Speedup of Las Vegas Algorithms. Information Processing Letters, 47:173-180.

P. Motwani, P. Raghavan (1996) Randomized Algorithms, ACM Computing Surveys, Vol. 28, No. 1, pp. 33-37.

M. Nielsen, C. Lundegaard, P.Worningl, C. Sylvester Hvid, K.Lamberth, S.Buus, S.Brunak, and O. Lund (2004) , Improved prediction of MHC class I andclass II epitopes using a novel Gibbssampling approach, BIOINFORMATICS Vol. 20 no. 9, pages 1388–1397.