QuickSort

Chapter 7

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Introduction

We examine another sorting algorithm, namely **quicksort**. It exhibits an interesting behavior among all sorting algorithms. Its worst case performance is $\Theta(n^2)$. Despite this performance, it is one of the most practical choices of a sorting algorithm as its average case performance is $\Theta(n \lg n)$ with the constant factors hidden smaller than both mergesort and heap sort.

Quicksort is a very simple algorithm to describe. Its performance however it is difficult to describe and analyze.

Quicksort is a Divide-and-Conquer algorithm as is mergesort. The three steps of a divide and conquer approach to sort an array of size n is recursively shown for a subarray $A[l..r]$. In order to sort A we issue a call Quicksort($A[1..n]$).

Quicksort	$(A[l..r])$
Divide	Partition the elements of $A[l..r]$ into two nonempty subarrays/subsequences $A[l..m-1]$ and $A[m+1..r]$ such that all elements of the first subarray are smaller than or equal to $A[m]$ and all elements of the second subarray are at least $A[m]$. Index m **is not** the middle element as in mergesort but is computed as part of the partitioning algorithm. Note that during this step, elements of A move around but with in the bounds of the subarray $A[l..r]$.
Conquer	Each of the two subarrays is sorted recursively by calling quicksort.
Combine	Everything is sorted as a result of conquer phase (if one sorts $A[l..m-1]$ and the sorts $A[m+1..r]$, because of the partitioning in step Divide, $A[1..n]$ becomes sorted). Do nothing and return.

Below we describe in detail the phases of quicksort.

Because of its simplicity the major task in quicksort is to split $A[l..r]$ as evenly as possible so that $A[l..m-1]$ and $A[m+1..r]$ have approximately the same number of elements ie $m-l \approx r-m$. This means that the splitter element $A[m]$ (as all element of $A[l..m]$ are at most $A[m]$ and all elements of $A[l..m]$ are at least $A[m]$) must be well chosen (i.e. be close to the median of the input sequence).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Pseudocode

The `QuickSort` function is given below.

```
QuickSort(A,1,n);  // Sort array A[1..n]
```

The implementation of `QuickSort(A,l,r)` follows.

```
 QuickSort(A,l,r);
1. if (l<r)
2.  then m=PARTITION(A,l,r);  // A[l,...m-1,m,m+1,...r]
3.       QuickSort(A,l,m-1);
4.       QuickSort(A,m+1,r);
```

The implementation of `PARTITION(A,l,r)` follows.

```
 PARTITION(A,l,r); // PARTITION A[l..r]
 // Reorder so that A[l.. i i+1 i+2 ... r]
 //                    <=    =     >      splitter
 // During the course of the PARTITION
 //             A[l ... i ... j ... r-1 r]
 //               -------                 : A[l]  ... A[i]   Confirmed keys <= splitter   (Line 4)
 //                      ------           : A[i+1]... A[j-1] Confirmed keys >  splitter
 //                            ------     : A[j]  ... A[r-1] Unresolved : Work in progress(Loop 3)
 //                                     - :           A[r]   Splitter
1. splitter= A[r];                     //Splitter is the rightmost key. Record splitter.
2. i=l-1;                              //Start from left of leftmost position (See step 6)
3. for j=l to r-1 do {                 //Go through all the keys; ignore A[r], the splitter
4.   if A[j] <=splitter {              //Is examined key less than or equal to splitter?
5.     then  i=i+1;                    //Yes it is! Make room for it and store it
6.           exchange A[j] and A[i]    //Future Optimization: Perform swap if i != j ?
     }     //endif
   }      //endfor
7. exchange A[i+1] and A[r]            //Move splitter A[r] into A[i+1]
8. return i+1                          //Return position into which splitter moved, i.e. i+1.
```

(c) Copyright A. Gerbessiotis.

A **description** of the algorithm follows.

The **splitter** key is fixed to be the rightmost element of subarray $A[l, .., r]$, i.e. `splitter`$= A[r]$. We then **reorder** the subarray $A[l..r]$ around the splitter into two subsequences called `LessEqual` and `Greater` consisting of the keys **less than or equal to the splitter** and the keys **greater than the splitter** respectively. The keys that have not been determined whether they belong to one or ther other area are in subsequence `Unknown`. This splitting is done by way of the loop of lines 3-6. The boundaries of the three areas `LessEqual`, `Greater`, and `Unknown` are determined by the indices i, j.

The `LessEqual` area starts with $A[l]$ and goes up to including $A[i]$. Note that if $i = l - 1$ (Line 2), this area is empty (initially). The `Greater` area starts with $A[i + 1]$ and goes up to $A[j - 1]$. Initially, before line 3 completes the $j = l$ loop, this area is also empty. The `Unknown` area starts with $A[j]$ and goes up to $A[r - 1]$. Initially, before line 3 is executed, all the keys belong to this area. The splitter is `A[r]`. The splitter is left untouched until the very end (line 7).

```
                 A[l]          ...                 A[r-1]   A[r]
Initially:    i   j


                 A[l]...A[i] ...   A[j]   ...    A[r-1]   A[r]
During   :               i          j
                 ----------- -----   ----------------
                    <=        >      Working on
                LessEqual    Greater   Unknown


Line 7   :       A[l]   ...   A[i]A[i+1]A[i+2]..A[r-1]   A[r]
                 ----------------   -------------------  ----
                  LessEqual         Greater              Splitter


Line 8   :       A[l]   ...   A[i]A[i+1]A[i+2]..A[r-1]   A[r]
                   l    ...    m-1  m     m+1      ...     r
                   --------------  ----    -----------------
                  LessEqual       Split.       Greater
```

(c) Copyright A. Gerbessiotis.

An example

```
i  lj                    r
   1  3  7  5  6  2  8  4  (a)
   li j                  r
   1  3  7  5  6  2  8  4  (b)
   l  i  j               r
   1  3  7  5  6  2  8  4  (c)

   l  i     j            r
   1  3  7  5  6  2  8  4  (d)
   l  i        j         r
   1  3  7  5  6  2  8  4  (e)
   l  i           j      r
   1  3  7  5  6  2  8  4  (f)

   l     i              j  r
   1  3  2  5  6  7  8  4  (g)
   l     i              j  r
   1  3  2  4  6  7  8  5  (h)
```

Assessing the performance of quicksort It is noted that the performance of function **PARTITION** is proportional to the number of elements to be partitioned ie $(n = r - l + 1)$. Given that merge-sort is an $O(n \lg n)$ algorithm we expect quicksort to have the same performance as long as **PARTITION** splits the array into two partitions of equal size. An **average** case argument lends credibility to such an argument. Half of the elements in the average case will be smaller than key and half larger.

(c) Copyright A. Gerbessiotis.

50% - 50% split	In each recursive step $n/2$ and $n/2$ are the sizes of the two subsequences (this is a very optimistic **best-case**).
1 **vs** $n-1$ **split**	We shall show that this as well as the symmetric one lead to worst-case performance.
cn **vs** $n-cn$ **split**	Assuming that $c < 1$ we shall show that this is still as good as the best-case.
average case	To be examined.

Partitioning a sequence of size n take dn time for some constant d, and $T(1) = \Theta(1)$, where $T(n)$ the worst-case running time of quicksort.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-k) + \Theta(n-k+1) + \ldots + \Theta(n) \\ &= T(1) + \Theta(2) + \ldots + \Theta(n) \\ &= T(1) + \Theta(2 + \ldots + n) = T(1) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Therefore the **best-case** of insertion sort is **worst-case** for quick sort.

Under the best-case the recurrence for running time is shown below.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$

The cn split provides the recurrence relation shown below

$$T(n) = T(cn) + T((1-c)n) + \Theta(n) = \Theta(n \lg n)$$

Therefore, as long as c is a constant, a constant fraction split leads to an algorithm that is $\Theta(n \lg n)$. The closer the constant c is to $1/2$ the smaller the constant in the Θ notation is.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Worst-case running time

In general the performance of quicksort is given by the recurrence, where c is some predefined positive constant determined by PARTITION.

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

Then, the worst case performance of quicksort is

$$T(n) = \max_{q=0}^{n-1} (T(q) + T(n-q-1)) + n$$

We will show that this implies that $T(n) = \Theta(n^2)$ by using the substitution method.

(a) Show that $T(n) = O(n^2)$, i.e. $T(n) \leq cn^2$ for some $c > 0$ and for all $n \geq n_0$. We use the fact that $i^2 + (n-1-i)^2$ is maximized for $i = 0$ or $i = n-1$ (i.e. for one of the boundary values). Then

$$\begin{aligned}
T(n) &= \max_{q=0}^{n-1} (T(q) + T(n-q-1)) + n \\
&\leq \max_{q=0}^{n-1} \left(cq^2 + c(n-q-1)^2 \right) + n \\
&= c \cdot \max_{q=0}^{n-1} \left(q^2 + (n-q-1)^2 \right) + n \\
&\leq c \cdot (n-1)^2 + n \\
&= c(n^2 - 2n + 1) + n \\
&= cn^2 + (1-2c)n + c \\
&\leq cn^2 + (1-2c)n + cn \\
&= cn^2 + (1-c)n \\
&\leq cn^2
\end{aligned}$$

The last inequality is derived provided that $c \geq 1$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

(b) Show that $T(n) = \Omega(n^2)$, i.e. $T(n) \geq fn^2$ for some $f > 0$ and for all $n \geq n_1$. Then we prove similarly that

$$\begin{aligned} T(n) &= \max_{q=0}^{n-1} \left(T(q) + T(n-q-1)\right) + n \\ &\geq \max_{q=0}^{n-1} \left(fq^2 + f(n-q-1)^2\right) + n \\ &\geq f\left((n-1)^2\right) + n \\ &= fn^2 + (1-2f)n + f \\ &\geq fn^2 + (1-2f)n \\ &\geq fn^2 \end{aligned}$$

The last inequality is derived provided that $1 \geq f/2$, i.e. $f \leq 2$, and $f > 0$.

Question. What happens with the best-case performance of quicksort? How can we show that $T(n) = \Omega(n \lg n)$ or a tight Θ bound? We can work similarly to the worst case. The only change will be finding not the maximum of $f(i) = i^2 + (n-1-i)^2$ but the MINIMUM of function $f(i) = i \lg i + (n-1-i) \lg (n-1-i)$. The latter has a minimum at $i = (n-1)/2$, which shows that the best case performance is attained for a $50 - 50$ split.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Average case

To analyze the average case of quicksort we assume that all permutations of the input are equally likely to occur. In the average case, we would have a mix of **bad** and **good** splits. For example if a bad split 1 and $n-1$ is followed by good split of the second sequence we get for the second a split $(n-1)/2$ and $(n-1)/2$. This gives a total $(n-1)/2+1$ and $(n-1)/2$.

A randomized version of quicksort, before the execution of the algorithm, randomizes the position of the keys so that the average case be observed. Note that in this version of quicksort the worst case or base case are rarely observed. Their appearance depends on the sequence being so unlucky that the random numbers generate a sorted sequence out of the input sequence

A **randomized algorithm** is one whose behavior is also determined (besides the input) by the values of a random number generator. This way bad inputs don't necessarily lead to worst-case running time. The disadvantage is that in some case good inputs may lead to worst-case running time. However the chances of this occurring are negligible. And even if they occur (we can count number of operations to detect such a case) we always have a quick fix. Stop the current execution, reobtain the original sequence, rerandomize it and hope that the second time around we would be luckier.

Another way to proceed is to change the partition method: The splitter becomes a key in a random position instead of the key in position 1.

```
Randomized-PARTITION (A,l,r)
1. i<- Random(l,r)
2. exchange A[i] and A[r]
3. return PARTITION(A,l,r)
4.
```

```
Rand-QuickSort(A,l,r)
1. If (l<r)
2.  m= Randomized-PARTITION(A,l,r)
3.  Rand-Quicksort(A,l,m-1)
4.  Rand-Quicksort(A,m+1,r)
```

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.