Chapter 5

Quicksort

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

Quicksort is famous because of its ability to sort in-place. I.e., it directly modifies the contents of the list and uses $O(1)$ temporary storage. This is not only useful for space requirements (recall that our merge sort used $O(n \log(n))$ space), but also for practical efficiency: the allocations performed by merge sort prevent compiler optimizations and can also result in poor cache performance\footnote{See Chapter 3 of “Code Optimization in C++11” (Serang 2018)}. Quicksort is often a favored sorting algorithm because it can be quite fast in practice.

Quicksort is a sort of counterpoint to merge sort: Merge sort sorts the two halves of the array and then merges these sorted halves. In contrast, quicksort first “pivots” by partitioning the array so that all elements less than some “pivot element” are moved to the left part of the array and all elements greater than the pivot element are moved to the right part of the array. Then, the right and left parts of the array is recursively sorted using quicksort and the pivot element is placed between them (Listing 5.1).

Runtime analysis of quicksort can be more tricky than it looks. For example, we can see that if we choose the pivot elements in ascending order, then during pivoting, the left part of the array will always be empty while the right part of the array will contain all remaining $n - 1$ elements (i.e., it excludes the pivot element). The cost of pivoting at each layer in the call tree will therefore be $\Theta(n), \Theta(n - 1), \Theta(n - 2), \ldots, \Theta(1)$, and the total cost
will be $\in \Theta(n + n - 1 + n - 2 + \cdots + \Theta(1) = \Theta(n^2)$. This poor result is because the recursions are not balanced; our divide and conquer does not divide effectively, and therefore, it hardly conquers.

### 5.1 Worst-case runtime using median pivot element

One approach to improving the quicksort runtime would be choosing the median as the pivot element. The median of a list of length $n$ is guaranteed to have $\frac{n-1}{2}$ values $\leq$ to it and to have $\frac{n-1}{2}$ values $\geq$ to it. Thus the runtime of using the median pivot element would be given by the recurrence

$$r(n) = 2r\left(\frac{n-1}{2}\right) + \Theta(n),$$

$$< 2r\left(\frac{n}{2}\right) + \Theta(n),$$

where the $\Theta(n)$ cost comes from the pivoting step. This matches the runtime recurrence we derived for merge sort, and we can see that using quicksort with the median as the pivot will cost $\Theta(n \log(n))$.

There is a large problem with this strategy: we have assumed that the cost of computing the median is trivial; however, if asked to compute a median, most novices would do so by sorting and then choosing the element in the middle index (or one of the two middle indices if $n$ should happen to be even). It isn’t a good sign if we’re using the median to help us sort and then we use sorting to help compute the median. There is an $O(n)$ divide-and-conquer algorithm for computing the median, but that algorithm is far more complex then quicksort. Regardless, even an available linear-time median algorithm would add significant practical overhead and deprive quicksort of its magic.

### 5.2 Expected runtime using random pivot element

We know that quicksort behaves poorly for some particular pivoting scheme and we also know that quicksort performs well in practice. Together, these suggest that a randomized algorithm could be a good strategy. Now, obviously, the worst-case performance when choosing a random pivot element
5.2. **EXPECTED RUNTIME USING RANDOM PIVOT ELEMENT**

would still be $\Theta(n^2)$, because our random pivot elements may correspond to visiting pivots in ascending order (or descending order, which would also yield poor results).

Let us consider the expected runtime\footnote{I.e., the average runtime if our quicksort implementation were called on any array filled with unique values.}. We denote the operation where elements $i$ and $j$ are compared using the random variable $C_{i,j}$:

$$C_{i,j} = \begin{cases} 
1 & \text{i and j are compared} \\
0 & \text{else}
\end{cases}.$$

The total number of comparisons will therefore be the sum of comparisons on all unique pairs:

$$\sum_{\{i,j\}:i \neq j} C_{i,j}.$$

Excluding the negligible cost of the recursion overhead, the cost of quicksort will be the cost of all comparisons plus the cost of all swaps, and since each comparison produces at most one swap, use the accounting method to simply say that the runtime will be bounded above by this total number of comparisons.

The expected value of the number of comparisons will be the sum of the expected values of the individual comparisons:

$$\mathbb{E} \left[ \sum_{\{i,j\}:i \neq j} C_{i,j} \right] = \sum_{\{i,j\}:i \neq j} \mathbb{E}[C_{i,j}].$$

The expected value of each comparison will be governed by $p_{i,j}$, the probability of whether elements $i$ and $j$ are ever compared:

$$\mathbb{E}[C_{i,j}] = p_{i,j} \cdot 1 + (1 - p_{i,j}) \cdot 0 = p_{i,j}.$$
and \( j \), then the probability they will be compared will be \( \frac{2}{|r_j - r_i + 1|} \), where \( r_i \) gives the index of element \( i \) in the sorted list.

If our list contains unique elements\(^3\) then we can also sum over the comparisons performed by summing over the ranks rather than the elements (in both cases, we visit each pair exactly once):

\[
\text{total runtime} \propto \sum_{\{i,j\} \neq \{i,j\}} \mathbb{E}[C_{i,j}]
= \sum_{\{i,j\} \neq \{i,j\}} p_{i,j} \\
= \sum_{\{r_i,r_j\} \mid r_i \neq r_j} \frac{2}{|r_j - r_i + 1|} \\
= \sum_{r_i \choose r_j > r_i} \frac{2}{r_j - r_i + 1}.
\]

We transform into a more accessible form by letting \( k = r_j - r_i + 1 \):

\[
= \sum_{r_i \choose k=r_j-r_i+1} \sum_{r_j > r_i \land r_j \leq n} \frac{2}{k} \\
\leq \sum_{r_i \choose k=2} ^n \sum_{k=2} ^n \frac{2}{k} \\
= \sum_{k=2} ^n \sum_{r_i=1} ^n \frac{2}{k} \\
= 2n \sum_{k=2} ^n \frac{1}{k}.
\]

Here we use the fact that \( \sum_{k=1} ^n \frac{1}{k} \) is a “harmonic sum”, which can be seen as an approximation of

\[
\int_1 ^n \frac{1}{x} \, dx.
\]

Specifically, if we plot the continuous function \( \frac{1}{x} \), \( x \geq 1 \) and compare it to the discretized \( \frac{1}{[x]} \), we see that \( \frac{1}{[x]} \) will never underestimate \( \frac{1}{x} \) because \( x \geq [x] \)

\(^3\)Quick sort works just as well when we have duplicate elements, but this assumption simplifies the proof.
5.2. EXPECTED RUNTIME USING RANDOM PIVOT ELEMENT

and thus \( \frac{1}{x} \leq \frac{1}{\lfloor x \rfloor + 1} \) (Figure 5.1). If we shift the discretized function left by 1 to yield \( \frac{1}{\lfloor x \rfloor + 1} \), we see that it never overestimates \( \frac{1}{x} \) (because \( x < \lfloor x \rfloor + 1 \) and thus \( x > \frac{1}{\lfloor x \rfloor + 1} \)). It follows that the area

\[
\sum_{k=1}^{n} \frac{1}{k+1} = \int_{1}^{n} \frac{1}{\lfloor x \rfloor + 1} \, dx \\
< \int_{1}^{n} \frac{1}{x} \, dx \\
= \log_e(n).
\]

We can rewrite

\[
\sum_{k=2}^{n+1} \frac{1}{k} = \sum_{k=1}^{n} \frac{1}{k+1} \\
< \log_e(n).
\]

Our harmonic sum, \( \sum_{k=2}^{n} \frac{1}{k} \), is bounded above by a sum with an additional nonnegative term, \( \sum_{k=2}^{n+1} \frac{1}{k} < \log_e(n) \). And hence, our expected quicksort runtime is bounded above by \( 2n \log_e(n) \in O(n \log(n)) \).

Listing 5.1: Quicksort.

```python
import numpy as np

def swap_indices(arr, i, j):
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp

def quicksort(arr, start_ind, end_ind):
    n = end_ind - start_ind + 1
    # any list of length 1 is already sorted:
    if n <= 1:
        return

    # choose a random pivot element in {start_ind, ..., end_ind}
    pivot_index = np.random.randint(start_ind, end_ind+1)
    pivot = arr[pivot_index]

    # count values < pivot:
```

Figure 5.1: Illustration of a harmonic sum. $\frac{1}{\lfloor x \rfloor}$ is an upper bound of $\frac{1}{x}$ while $\frac{1}{\lfloor x \rfloor + 1}$ is a lower bound of $\frac{1}{x}$. 
5.2. EXPECTED RUNTIME USING RANDOM PIVOT ELEMENT

vals_lt_pivot=0
for i in xrange(start_ind, end_ind+1):
    if arr[i] < pivot:
        vals_lt_pivot += 1

# place pivot at arr[vals_lt_pivot], since vals_lt_pivot will need to
come before it
swap_indices(arr, pivot_index, start_ind+vals_lt_pivot)
# the pivot index has been moved to index vals_lt_pivot
pivot_index = start_ind+vals_lt_pivot

# move all values < pivot to indices < pivot_index:
vals_lt_pivot=0
for i in xrange(start_ind, end_ind+1):
    if arr[i] < pivot:
        swap_indices(arr, i, start_ind+vals_lt_pivot)
        vals_lt_pivot += 1

# pivoting is complete. recurse:
quicksort(arr, start_ind, start_ind+vals_lt_pivot)
quicksort(arr, start_ind+vals_lt_pivot+1, end_ind)

arr = [10,1,9,5,7,8,2,4]
quicksort(arr, 0, len(arr)-1)
print arr