Quicksort

- Attributed to Tony Hoare (only one of his contributions to computer science; find out who he is if you don't already know)
- Pick an element p of the array; partition array so all values smaller than p are to p's left and all values greater than p are to the right; recursively quicksort sub-array on either side of p
- Average and best case time complexity is $O(n \cdot \lg n)$
- Worst case time complexity is $O(n^2)$
- Usually fastest when compared with other sequential, comparison sorting algorithms
- Does not require extra (explicit) space
- Ways to improve Quicksort
  - Remove recursion — it can consume unacceptable amounts of space for the implicit stack needed for recursion
  - Avoid small sub-arrays — switch to insertion sort when array size is small (e.g. , somewhere in the range 5 to 15)
    Introsoft switches switches from quick sort to heap sort when the recursion depth exceeds a certain level.
  - Try to avoid the worst case behavior by using a random pivot p, but, even then, a random sequence of pivots *could* lead to worst case behavior

Quicksort Algorithm

The quicksort algorithm is attributed to Tony Hoare (Hoare, 1961). Sedgewick's analysis of quicksort (Sedgewick, 1977) and (Sedgewick, 1978) provide an in-depth analysis and details of its implementation.

The basic quicksort idea is to place some element, the pivot `p`, of a list in its correct position. That is, smaller elements are placed before `p` and larger elements after `p`.

Listing 1: Functional Quicksort

1 ⟨*Functional Quicksort* 1⟩≡

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (p:xs) = qsort [x|x<-xs,x<p] ++ [p] ++ qsort [x|x<-xs,x>=p]
```

Analysis: Functional Quicksort

The time complexity to construct the lower `[x|x<-xs,x<p]` and upper `[x|x<-xs,x>=p]` lists for the recursive calls is $2(n-1) = O(n)$. An optimizing compiler may be able to scan the list `xs` only once to construct the lower and upper lists.

The cost of concatenation is a function of the length of the first list. Therefore, it is best to append the high list to `[p]`, then the low list to that result. This cost varies from 1 to n as the low list length k varies from empty to $n-1$.

Therefore, the time complexity is described by recursion

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

The worst case occurs when $k = 0$ for each recursive call. In this case, the recursion reduces to

$$T(n) = T(n-1) + (n-1) \quad \text{which implies } T(n) = O(n^2).$$

The best case occurs when $k = (n-1)/2$ for each recursive call. The recursion reduces to

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \frac{3(n-1)}{2} \quad \text{which implies } T(n) = O(n \lg n).$$

The standard assumption for the average case is that each value of k is equally likely for each recursive call. The analysis under this assumption is described below.

The Imperative Represenation

Here is a summary of Bentley's code for quick sorting (Bentley, 1984). The essence is partition an array about a `pivot` that gets placed in its correct position. Then quick sort the lower and upper arrays.

Listing 2: Imperative Quicksort

2 ⟨*Imperative Quicksort* 2⟩≡

```
void quickSort(int A[], int lo}, int hi)
{
  int pivot;
  if (hi > lo) {
    pivot = Partition(A, lo, hi);
    quickSort(A, lo, pivot-1);
    quickSort(A, pivot+1, hi);
  }
}
```

Example: Quicksort Partition Example

- *Scan from the right looking for a value greater than the pivot (index i)*
- *Scan from the left looking for a value less than the pivot (index j)*
- *If the right and left scans meet or cross, swaps pivot and right scan*
- *Otherwise, swap the right and left scan and resume scans*

position	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
array	*3*	*9*	*12*	*15*	*11*	*17*	*6*	*8*	*14*
pivot									*p*
indexes				*i*				*j*	
swap				*8*				*15*	
indexes						*i*	*j*		
swap						*6*	*17*		
indexes						*j*	*i*		
swap	*3*	*9*	*12*	*8*	*11*	*6*	*14*	*15*	*17*

Listing 3: Imperative partitioning about a pivot

3 ⟨*Quicksort partition* 3⟩≡

```
int partition(int A[], int lo, int hi)
{
  int p, i, j;
  p = A[hi]; i = lo - 1; j = hi;
  for (;;) {
    while (i < hi && A[++i] < p) ;
    while (j > lo && A[--j] > p) ;
    if (i >= j) break;
    swap (A[i], A[j]);
  }
  swap (A[i], A[hi]);
  return i;
}
```

Let's analyze the worst, best, and average case of `Quicksort`.

Quicksort: Worst Case

- Given an array of length n, `quicksort` makes two calls to itself, once with an array of length k and once with an array of length $n-k-1$
- Here k is the size of the array from `low` to `pivot-1`
- The cost of the call to `partition` is $n+1 = O(n)$
- In the worst case $k = 0$ and

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

with initial condition $T(1) = 1$

- By mathematical induction, or unrolling the recurrence,

$$\begin{aligned}
T(n) &= (n+1)+(n)+(n-1)+\cdots+3+T(1) \\
&= (n+1)+(n)+(n-1)+\cdots+3+1 \\
&= (n+1)+(n)+(n-1)+\cdots+3+2+1-2 \\
&= \frac{(n+1)(n+2)}{2}-2 \\
&= O(n^2)
\end{aligned}$$

Quicksort: Best Case

- In the best case $k = n/2$ and the recursion reduces to

$$T(n) = (n+1)+2T(n/2)$$

  with initial condition $T(1) = 1$
- Note we are cheating a little here since $n-k-1 = n/2-1 \neq n/2$, but this fudge will not alter the timing analysis
- *Unrolling* the formula

$$\begin{aligned}
T(n) &= (n+1)+2T(n/2) \\
&= (n+1)+2((n/2+1)+2T(n/4)) \\
&= (n+1)+(n+2)+4T(n/4)) \\
&= (n+1)+(n+2)+(n+4)+\cdots+2^qT(n/2^q) \\
&= (n+1)+(n+2)+(n+4)+\cdots+nT(1) \\
&= (n+1)+(n+2)+(n+4)+\cdots+(n+\frac{n}{2})+nT(1) \\
&= (n+n+n+\cdots+n)+(1+2+4+\cdots 2^{q-1}) \\
&= nq+(2^q-1) \\
&= n\lg n+(n-1) \\
&= O(n\lg n)
\end{aligned}$$

  where $n = 2^q$ is a power of 2 and $q = \lg n$.

Quicksort: Average Case

Assume it costs nothing to sort nothing: Initial condition: $T(0) = 0$.

- In the average case, compute the average of the n cases as k, the length of the low array, goes from 0 to

$n-1$

$$\begin{aligned}
T(n) &= (n+1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k-1)) \\
&= (n+1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + \sum_{k=0}^{n-1} T(n-k-1)) \\
&= (n+1) + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + \sum_{k=0}^{n-1} T(k)) \\
&= (n+1) + \frac{2}{n}\sum_{k=0}^{n-1} T(k)
\end{aligned}$$

- Massaging $T(n)$ into shape:
  First rewrite

$$T(n) = (n+1) + \frac{2}{n}\sum_{k=0}^{n-1} T(k)$$

  by removing the fraction, multiplying by n

$$nT(n) = n(n+1) + 2\sum_{k=0}^{n-1} T(k)$$

  Relabel n using $n+1$ to write

$$(n+1)T(n+1) = (n+1)(n+2) + 2\sum_{k=0}^{n} T(k)$$

  Subtract the two equations:

$$(n+1)T(n+1) - nT(n) = 2(n+1) + 2T(n)$$

- Now suppose we knew a function $G(z)$ such that

$$\begin{aligned}
G(z) &= T_0 + T_1 z + T_2 z^2 + \cdots \\
&= \sum_{n=0}^{\infty} T_n z^n
\end{aligned}$$

  Recall,

$$\sum_{n=0}^{\infty} z^n = \frac{1}{1-z}$$

  which gives

$$\sum_{n=0}^{\infty} (n+1) z^n = \frac{1}{(1-z)^2}$$

  where we write T_n for $T(n)$
- Notice that

$$\begin{aligned}
G'(z) - zG'(z) &= \textstyle\sum_{n=0}^{\infty} nT_n z^{n-1} - z\sum_{n=0}^{\infty} nT_n z^{n-1} \\
(1-z)G'(z) &= \textstyle\sum_{n=0}^{\infty}(n+1)T_{n+1} z^n - \sum_{n=0}^{\infty} nT_n z^n \\
&= \textstyle\sum_{n=0}^{\infty} [(n+1)T_{n+1} - nT_n]\, z^n \\
&= \textstyle\sum_{n=0}^{\infty} [2(n+1) + 2T(n)]\, z^n \\
&= \frac{2}{(1-z)^2} + 2G(z)
\end{aligned}$$

- Thus
$$(1-z)G'(z) = \frac{2}{(1-z)^2} + 2G(z)$$
- Multiplying by $(1-z)$ and rearranging terms gives a perfect differential
$$(1-z)^2G'(z) - 2(1-z)G(z) = \frac{2}{1-z}$$
that is
$$\frac{d(1-z)^2G(z)}{dz} = (1-z)^2G'(z) - 2(1-z)G(z)$$
- Integrating both sides
$$(1-z)^2G(z) = -2\ln(1-z) + C$$
where $C = 0$ since $G(0) = T_0 = 0$
- It follows that
$$\begin{aligned} G(z) &= \frac{-2}{(1-z)^2}\ln(1-z) \\ &= 2\sum_{i=1}^{\infty}(i+1)z^i \sum_{j=1}^{\infty}\frac{z^j}{j} \\ &= 2\sum_{n=1}^{\infty}\left[\sum_{k=1}^{n}\frac{n+1-k}{k}\right]z^n \\ &= \sum_{n=1}^{\infty}\sum_{k=1}^{n}\left[\frac{2(n+1)}{k} - 2\right]z^n \\ &= \sum_{n=1}^{\infty}[2(n+1)H_n - 2n]z^n \end{aligned}$$
- Therefore
$$T(n) = 2(n+1)H_n - 2n = O(n\lg n)$$

The Cauchy product of two power series is
$$\left(\sum_{i=0}^{\infty}a_ix^i\right)\cdot\left(\sum_{j=0}^{\infty}b_jx^j\right) = \sum_{n=0}^{\infty}c_nx^n$$
where
$$c_n = \sum_{k=0}^{n}a_{n-k}b_k$$

Recall the Harmonic numbers
$$\begin{aligned} H_n &= \sum_{k=1}^{n}\frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= H_{n-1} + \frac{1}{n} \quad n \geq 1 \\ H_0 &= 0 \end{aligned}$$

The harmonic number H_n can be asymptotically approximated by
$$H_n \sim \ln n + \gamma + O(1/2n)$$
where $\gamma \approx 0.5772156649\cdots$ is Euler's constant.

References

Bentley, J. (1984). Programming Pearls: How to sort. *Commun. ACM*, 27(4):287–ff. [page 2]

Hoare, C. A. R. (1961). Quicksort. *Computer Journal*, 5(1):10–15. [page 1]

Sedgewick, R. (1977). The analysis of quicksort programs. *Acta Informatica*, 7:327–355. [page 1]

Sedgewick, R. (1978). Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857. [page 1]