TCSS 343 - Assignment 3

Jake McKenzie

July 22, 2018

1 UNDERSTAND

In this problem use the Master Theorem to find and prove tight bounds for these recurrences (6 points each).

To solve this problem I used the master theorem taken from CLRS. I used a bit of a stronger statement than what was stated in the theorem but they are equivalent mathematically due to the nature of limits. I will include the theorem for the reader's benefit. To check my work I decided to include what I obtained using the Akra-Bazzi method.

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

1.
$$T(n) = \begin{cases} c & \text{if } n < 8 \\ 16T(\frac{n}{8}) + n\log n & \text{if } n \geq 8 \end{cases}$$

By Master Method I obtain case 1:

$$\begin{aligned} \lim_{n\to\infty} \frac{n\log n}{n^{\log_8 16+\varepsilon}} &= \lim_{n\to\infty} \frac{n\log n}{n^{\frac{4}{3}+\frac{2}{3}}} \\ &= \lim_{n\to\infty} (\frac{n}{n})(\frac{\log n}{n}) \to 0 \\ T(n) &\in \Theta(n^{\frac{4}{3}}) \end{aligned}$$

By Akra-Bazzi I obtain:

$$16(\frac{1}{8})^p = 1$$
$$p = \frac{4}{3}$$

$$\begin{aligned} T(n) &\in \Theta(n^p(1 + \int_1^n \frac{u \log u}{u^{p+1}} du)) \\ &\in \Theta(-9n + 10n^{\frac{4}{3}} - 3n \log n) \\ &\in \Theta(n^{\frac{4}{3}}) \end{aligned}$$

2.

$$T(n) = \begin{cases} c & \text{if } n < 4 \\ 8T(\frac{n}{4}) + n \log n & \text{if } n \geq 4 \end{cases}$$

By Master Method I obtain case 2:

$$\begin{aligned} \lim_{n \to \infty} \frac{n^{\frac{1}{3}}}{n^{\log_8 2 + \varepsilon}} &= \lim_{n \to \infty} \frac{n^{\frac{1}{3}}}{n^{\frac{1}{3}}} \\ &= \lim_{n \to \infty} 1 \to 1 \\ &\in \Theta(n^{\frac{1}{3}} \log n) \end{aligned}$$

By Akra-Bazzi I obtain:

$$2(\frac{1}{8})^p = 1$$
$$p = \frac{1}{3}$$

$$\begin{aligned} T(n) &\in \Theta(n^p(1 + \int_1^n \frac{u^{\frac{1}{3}}}{u^{p+1}} du)) \\ &\in \Theta(n^{\frac{1}{3}} + n^{\frac{1}{3}} \log n) \\ &\in \Theta(n^{\frac{1}{3}} \log n) \end{aligned}$$

3.

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 3T(\frac{n}{2}) + 9^n & \text{if } n \geq 2 \end{cases}$$

By Master Method I obtain case 3:

$$\begin{aligned}
\lim_{n\to\infty} \frac{9^n}{n^{\log_3 2+\varepsilon}} &= \lim_{n\to\infty} \frac{9^n}{n^{\frac{\log 2}{\log 3}-(\frac{\log 2}{\log 3}+1)}} \\
&= \lim_{n\to\infty} n9^n \to \infty \\
&\in \Theta(9^n)
\end{aligned}$$

This recurrence is not well suited for Akra-Bazzi.

4.

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 3T(\frac{3n}{5}) + n^2 & \text{if } n > 1 \end{cases}$$

By Master Method I obtain case 1:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^2}{n^{\log_{\frac{5}{3}} 3+\varepsilon}} &= \lim_{n\to\infty} \frac{n^2}{n^{\log_{\frac{5}{3}} 3+(3-\log_{\frac{5}{3}} 3)}} \\
&= \lim_{n\to\infty} \frac{n^2}{n^3} \to 0 \\
\log_{\frac{5}{3}} 3 &= 2.15066... \\
&\in \Theta(n^{2.15066...})
\end{aligned}$$

By Akra-Bazzi I obtain:

$$\begin{aligned}
3(\frac{3}{5})^p &= 1 \\
p &= \log_{\frac{5}{3}} 3
\end{aligned}$$

$$\begin{aligned}
T(n) &\in \Theta(n^p(1 + \int_1^n \frac{u^2}{u^{p+1}} du)) \\
&\in \Theta(7.63746n^{2.15066...} - 6.63746n^2) \\
&\in \Theta(n^{2.15066...})
\end{aligned}$$

5.

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 3T(\frac{3n}{5}) + n^{2.5} & \text{if } n > 1 \end{cases}$$

By Master Method I obtain case 3:

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^{2.5}}{n^{\log_{\frac{5}{3}} 3+\varepsilon}} &= \lim_{n\to\infty} \frac{n^{2.5}}{n^{\log_{\frac{5}{3}} 3+0.1}} \\
&= \lim_{n\to\infty} \frac{n^2.5}{n^{2.25}} \to \infty \\
&= \lim_{n\to\infty} n^{0.25...} \to \infty \\
&\in \Theta(n^{2.5})
\end{aligned}$$

2 EXPLORE

For the following problems stated as pseudo-code, let $A[\ell \ldots r]$ denote the sublist of the integer list A from the ℓ-th to the r-th element inclusive, let Cubic($A[1 \ldots n]$) denote an algorithm that runs in time $\Theta(n^3)$, and let Swift($A[1 \ldots n]$) denote an algorithm that runs in time $\Theta(n \log(\log n))$.

```
Three(A[1...n])
      If n ≤ 1 Then Return // nothing to do
      Cubic(A[1...n])
      Three(A[1...⌊n/2⌋])
      Three(A[⌊n/4⌋+1...⌊3n/4⌋])
      Three(A[⌊n/2⌋+1...n])
End Three.
```

(6 points) 1. State a recurrence that gives the complexity $T(n)$ for algorithm `Three`.

For this problem I decided to analyze the cost of running each line individually.

```
Three(A[1...n]) O(1)
      If n ≤ 1 Then Return // nothing to do O(1)
      Cubic(A[1...n]) O(n^3)
      Three(A[1...⌊n/2⌋]) T(n/2)
      Three(A[⌊n/4⌋+1...⌊3n/4⌋]) T(n/2)
      Three(A[⌊n/2⌋+1...n]) T(n/2)
End Three.  O(1)
```

Which gives me the following recurrece relation:

$$T(n) = \begin{cases} c & \text{if } n \leq 1 \\ 3T(\frac{n}{2}) + n^3 & \text{if } n > 1 \end{cases}$$

(6 points) 2. Find the tight complexity of algorithm `Three`.

By Master Theorem I obtain case 3:

$$\begin{aligned} \lim_{n\to\infty} \frac{n^3}{n^{\log 3+\varepsilon}} &= \lim_{n\to\infty} \frac{n^3}{n^{\log 3+(2-\log 3)}} \\ &= \lim_{n\to\infty} \frac{n^3}{n^2} \\ &= \lim_{n\to\infty} n \to \infty \\ T(n) &\in \Theta(n^3) \end{aligned}$$

By Akra-Bazzi I obtain:

$$\begin{aligned} 3(\frac{1}{2})^p &= 1 \\ p &= \log_2 3 \end{aligned}$$

$$
\begin{aligned}
T(n) &\in \Theta(n^p(1+\int_1^n \frac{u^3}{u^{p+1}}du)) \\
&\in \Theta(0.293305n^{1.58496} + 0.706695n^3) \\
&\in \Theta(n^3)
\end{aligned}
$$

```
One(A[1...n])
      If n ≤ 1 Then Return // nothing to do
      Swift(A[1...n])
      One(A[1...⌊n/2⌋])
End One.
```

(6 points) 3. State a recurrence that gives the complexity $T(n)$ for algorithm `One`.

```
One(A[1...n])O(1)
      If n ≤ 1 Then Return // nothing to do O(1)
      Swift(A[1...n])O(n log log n)
      One(A[1...⌊n/2⌋]) T(n/2)
End One.
```

Which gives me the following recurrece relation:

$$
T(n) = \begin{cases} c & \text{if } n \leq 1 \\ T(\frac{n}{2}) + n \log \log n & \text{if } n > 1 \end{cases}
$$

(12 points) 4. Find the tight complexity of algorithm `One`.

$$
\begin{aligned}
\lim_{n\to\infty} \frac{n \log \log n}{n^{\log 1+\varepsilon}} &= \lim_{n\to\infty} \frac{n \log \log n}{n^{0+\gamma}} \\
\gamma &= 0.577216 \\
&= \lim_{n\to\infty} n^{0.422784} \log \log n \to \infty \\
T(n) &\in \Theta(n \log \log n)
\end{aligned}
$$

By Akra-Bazzi I obtain:

$$
\begin{aligned}
(\frac{1}{2})^p &= 1 \\
p &= 0
\end{aligned}
$$

As for computing this integral, it is non-elementary. It invoIves computing the $Li(u)$ which I've looked up and it defintely grows slower than any of the functions we normally use.

It's worth noting that we don't need to work about that, Akra-Bazzi tells us that if $p < k$ then we get case 3 of the master theorem.

$$\begin{aligned}
T(n) &\in \Theta(n^p(1 + \int_1^n \frac{u \log\log u}{u^{p+1}} du)) \\
&\in \Theta(1 + \gamma - i\pi + n \log\log n - Li(n)) \\
&\in \Theta(n \log\log n)
\end{aligned}$$

3 PROGRAMMING

Quicksort runs in it's worst case when it chooses the greatest or smallest element for each successive pivot. That recurrence relation can be written in the form of

$$T(n) = \begin{cases} c & \text{if } n < 1 \\ T(n-1) + O(n) & \text{if } n \geq 1 \end{cases}$$

We've solved this recurrence relation plenty times before and the solution will just be stated as $\Theta(n^2)$. Now in practice was this what I obtained? By in large, on the 50 or so times I ran this program, this is what I obtained as a upper bound.

I was able to completely get rid of stackoverflows by implementing XORSwap, which is my favourite way to reduce the size of the stack. This is not the best swaping without a temporary variable routine in Java, arithmetic swap performs better due to compiler optimizations(arithmetic operators are used more often than bitwise ones so the compiler writers have written better optimizations for airthmetic) but it avoids overflows and leads to more desirable behaivor generally for this reason.

As for the space complexity I'm going to be honest and I couldn't make heads or tails of the space complexity of this algorithm from what I wrote. I think consulting the literature is important when this happens so here we go:

"For Quicksort, the combination of end- recursion removal and a policy of processing the smaller of the two subfiles first turns out to ensure that the stack need only contain room for about, lg N entries, since each entry on the stack after the top one must represent a subfile less than half the size of the previous entry." Robert Sedgwick's Algorithms

There two possible correctors to this question of space complexity. $\log n$ and n so which is it? Sedgwick seems to tell me it's $\log n$ which makes sense but if this true, why is that by implementing XORSwap I removed stackoverflows? Well let me consult James Aspnes lecture notes from a Computational Complexity Theory at Yale:

"There is no overhead in space complexity (except possibly dealing with the issue of a non-writable input tape with multiple heads), but the time complexity of a computation can easily go from T to $O(T^2)$.

(chapter 3 page 14 http://www.cs.yale.edu/homes/aspnes/classes/468/notes.pdf) What this telling me is that space complexity is less fickle and due to the nature of how information is stored it really does matter what the constant is. By reducing the number of swaps by a 1/3 ten times in my program I considerably dropped the space quicksort was using on the stack. But in both cases it was $\Theta(\log n)$ but inorder to run it I needed to worry about that constant.

The excution times and the code will be included below. It's worth noting here for the reader that I did implement XORShift to generate better random numbers. I think this is a best practice. Just from eyeballing it, before included it I was getting numbers that seemed a little too good. The runningtimes are more in line with what I expect with a more normally distributed set of numbers.

Command Prompt

N	piv	sorted?	safe?	time?
1000	init	unsorted	unsafe	0.001
1000	init	sorted	unsafe	0.000
1000	mid	unsorted	unsafe	0.001
1000	mid	sorted	unsafe	0.001
1000	init	unsorted	safe	0.001
1000	init	sorted	safe	0.001
1000	mid	unsorted	safe	0.000
1000	mid	sorted	safe	0.000
10000	init	unsorted	unsafe	0.009
10000	init	sorted	unsafe	0.002
10000	mid	unsorted	unsafe	0.005
10000	mid	sorted	unsafe	0.003
10000	init	unsorted	safe	0.002
10000	init	sorted	safe	0.003
10000	mid	unsorted	safe	0.005
10000	mid	sorted	safe	0.002
100000	init	unsorted	unsafe	0.031
100000	init	sorted	unsafe	0.030
100000	mid	unsorted	unsafe	0.037
100000	mid	sorted	unsafe	0.033
100000	init	unsorted	safe	0.034
100000	init	sorted	safe	0.029
100000	mid	unsorted	safe	0.034
100000	mid	sorted	safe	0.023
1000000	init	unsorted	unsafe	0.277
1000000	init	sorted	unsafe	0.225
1000000	mid	unsorted	unsafe	0.430
1000000	mid	sorted	unsafe	0.385
1000000	init	unsorted	safe	0.283
1000000	init	sorted	safe	0.204
1000000	mid	unsorted	safe	1.094
1000000	mid	sorted	safe	0.299
10000000	init	unsorted	unsafe	11.957
10000000	init	sorted	unsafe	4.651
10000000	mid	unsorted	unsafe	6.424
10000000	mid	sorted	unsafe	4.164
10000000	init	unsorted	safe	5.128
10000000	init	sorted	safe	3.742
10000000	mid	unsorted	safe	4.429
10000000	mid	sorted	safe	3.839

```
C:\Users\Epimetheus\Documents\GitHub\qsort>
```

```java
/**
 * @author Jake McKenzie
 * Implementation of the qsort algorithm using mid and 1st pivot with an accompaning analysis of it.
 */

public class qsort {
    private static String first = "init";
    private static String mid = "mid";
    private static String unsorted = "unsorted";
    private static String sorted = "sorted";
    private static String unsafe = "unsafe";
    private static String safe = "safe";
    private static Integer[] init_A;
    private static Integer[] mid_A;
    private static long startTime;
    private static int[] size = {1000,10000,100000,1000000,10000000};
    private static StringBuilder sb;
    private static String brdSize;
    private static String fill;
    public static void main(String[] args) {
        writeTop();
        for (int s : size) {

            init_A = new Integer[s];
            mid_A = new Integer[s];

            shakeUrn(init_A);
            shakeUrn(mid_A);

            writeRow(s, first, unsorted, unsafe, test_qsort_init_unsorted());
            writeRow(s, first, sorted, unsafe, test_qsort_init_sorted());
            writeRow(s, mid, unsorted, unsafe, test_qsort_mid_unsorted());
            writeRow(s, mid, sorted, unsafe, test_qsort_mid_sorted());

            shakeUrn(init_A);
            shakeUrn(mid_A);

            writeRow(s, first, unsorted, safe, test_sf_qsort_unsorted());
            writeRow(s, first, sorted, safe, testsf_qsortSorted());
            writeRow(s, mid, unsorted, safe, test_sf_qsort_mid_unsorted());
            writeRow(s, mid, sorted, safe, test_sf_qsort_mid_sorted());
        }

        writeBottom();

    }
```

```java
public static void qsort_init(Integer[] a) {
    qsort_init(a, 0, a.length - 1);
}

public static void qsort_init(Integer[] a, int l, int r) {
    if (r > l) {
        int p = piv_init(a, l, r);
        qsort_init(a, l, p - 1);
        qsort_init(a, p + 1, r);
    }
}

public static int piv_init(Integer[] a, int l, int r) {
    Integer p = a[l];
    int i = l + 1;
    int j = r;
    while (i < j) {
        while (p.compareTo(a[i]) > 0 && i < r) {
            i++;
            if (i > r) break;
        }
        while (p.compareTo(a[j]) < 0) j--;
        if (i >= j) break;
        if (i <= j) {
            swap(a, i, j);
            i++;
            j--;
        }
    }
    swap(a, l, j);
    return j;
}

private static void sf_qsort_init(Integer[] a) {
    sf_qsort_init(a, 0, a.length - 1);
}

private static void sf_qsort_init(Integer[] a, int l, int r) {
    while (r > l) {
        int p = piv_init(a, l, r);
        if (p - l <= r - p) {
            sf_qsort_init(a, l, p - 1);
            l = p + 1;
        } else {
            sf_qsort_init(a, p + 1, r);
            r = p - 1;
        }
    }
```

```
    }

    public static void qsort_mid(Integer[] a) {
        qsort_mid(a, 0, a.length - 1);
    }

    public static void qsort_mid(Integer[] a, int l, int r) {
        if (r > l) {
            int p = piv_mid(a, l, r);
            qsort_mid(a, l, p - 1);
            qsort_mid(a, p + 1, r);
        }
    }

    public static int piv_mid(Integer[] a, int l, int r) {
        Integer p = a[l + (r - l) >> 1];
        int m = l + (r - l) >> 1;
        int i = l + 1;
        int j = r;
        swap(a, m, l);
        while (i <= j) {
            while (p.compareTo(a[i]) > 0) {
                i++;
                if (i > r) break;
            }
            while (p.compareTo(a[j]) < 0) --j;
            if (i <= j) {
                swap(a, i, j);
                i++;
                j--;
            }
        }
        if (j < i) {
            swap(a, l, j);
            return j;
        } else if (i == j) {
            swap(a, l, i - 1);
            return i--;
        }
        return j;
    }

    private static void sf_qsort_mid(Integer[] a) {
        sf_qsort_mid(a, 0, a.length - 1);
    }
```

```java
    private static void sf_qsort_mid(Integer[] a) {
        sf_qsort_mid(a, 0, a.length - 1);
    }
    /**
     * Sometms this will run in ~150 seconds when there are 10^7 elements but most of the tm
     * it runes in 3 to 5 seconds. I do not think this is necisarily a bug but a nature of the safe
     * qsort mid requirements.
     */
    private static void sf_qsort_mid(Integer[] a, int l, int r) {
        while (r > l) {
            int p = piv_mid(a, l, r);
            if (p - l <= r - p) {
                sf_qsort_mid(a, l, p - 1);
                l = p + 1;
            } else {
                sf_qsort_mid(a, p + 1, r);
                r = p - 1;
            }
        }
    }
    /**
     * Never leave home without xorswap. This is my favourite way to deal with memory issues.
     * Credit goes to george marsaglia for discovery the algorithm, from FSU (my beloved uni
     * back in stomping grounds back in florida, go seminoles!), who happens to be one of my
     * favourite computer scientists.
     *
     * Essentially it is a way of swapping two variables without a temporary variable.
     *
     * This avoids stack overflow because the stack is reduced significantly for each recursive call.
     * This method alone is called 10 tms in this program, the stack contribution here is reduced by
     * a 1/3. That is very significant, thus no stack overflows. I've increased the size of N and
     * the stack still does not overflow so it can withstand 10^8 size arrays.
     */
    private static void swap(Integer[] a, int i, int j) {
        a[i] = a[i] ^ a[j];
        a[j] = a[j] ^ a[i];
        a[i] = a[i] ^ a[j];
    }

    private static void writeTop() {
        sb = new StringBuilder();
        fill = "| %-9d | %-5s | %-9s | %-7s | %-7.3f |%n";
        sb.append(String.format("|===========|=======|===========|=========|=========|%n"));
        sb.append(String.format("|     N     |  piv  |  sorted?  |  safe?  |  time?  |%n"));
        System.out.print(sb.toString());
    }
```

```java
    private static void sf_qsort_mid(Integer[] a) {
        sf_qsort_mid(a, 0, a.length - 1);
    }
    /**
     * Sometimes this will run in ~150 seconds when there are 10^7 elements but most of the time
     * it runes in 3 to 5 seconds. I do not think this is necisarily a bug but a nature of the safe
     * qsort mid requirements.
     */
    private static void sf_qsort_mid(Integer[] a, int l, int r) {
        while (r > l) {
            int p = piv_mid(a, l, r);
            if (p - l <= r - p) {
                sf_qsort_mid(a, l, p - 1);
                l = p + 1;
            } else {
                sf_qsort_mid(a, p + 1, r);
                r = p - 1;
            }
        }
    }
    /**
     * Never leave home without xorswap. This is my favourite way to deal with memory issues.
     * Credit goes to george marsaglia for discovery the algorithm, from FSU (my beloved uni
     * back in stomping grounds back in florida, go seminoles!), who happens to be one of my
     * favourite computer scientists.
     *
     * Essentially it is a way of swapping two variables without a temporary variable.
     *
     * This avoids stack overflow because the stack is reduced significantly for each recursive call.
     * This method alone is called 10 times in this program, the stack contribution here is reduced by
     * a 1/3. That is very significant, thus no stack overflows. I've increased the size of N and
     * the stack still does not overflow so it can withstand 10^8 size arrays.
     */
    private static void swap(Integer[] a, int i, int j) {
        a[i] = a[i] ^ a[j];
        a[j] = a[j] ^ a[i];
        a[i] = a[i] ^ a[j];
    }

    private static void writeTop() {
        sb = new StringBuilder();
        fill = "| %-9d | %-5s | %-9s | %-7s | %-7.3f |%n";
        sb.append(String.format("|===========|=======|===========|=========|=========|%n"));
        sb.append(String.format("|     N     |  piv  |  sorted?  |  safe?  |  time?  |%n"));
        System.out.print(sb.toString());
    }
```

```java
    private static void writeTop() {
        sb = new StringBuilder();
        fill = "| %-9d | %-5s | %-9s | %-7s | %-7.3f |%n";
        sb.append(String.format("|===========|=======|===========|=========|=========|%n"));
        sb.append(String.format("|     N     |  piv  |  sorted?  |  safe?  |  time?  |%n"));
        System.out.print(sb.toString());
    }

    private static void writeRow(int size, String piv, String sort, String sf, double tm) {
        fill = "| %-9d | %-5s | %-9s | %-7s | %-7.3f |%n";
        String s = String.format(fill, size, piv, sort, sf, tm);
        writeBottom();
        System.out.print(s.toString());
    }

    private static void writeBottom() {
        String s = String.format("|===========|=======|===========|=========|=========|%n", brdSize);
        System.out.print(s.toString());
    }

    private static void shakeUrn(Integer[] a) {
        for (int i = 0; i < a.length; i++) a[i] = (int)XORShift128plus();
    }

    private static double test_qsort_init_unsorted() {
        startTime = System.currentTimeMillis();
        qsort_init(init_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double test_qsort_mid_unsorted() {

        startTime = System.currentTimeMillis();
        qsort_mid(mid_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double test_qsort_init_sorted() {

        startTime = System.currentTimeMillis();
        qsort_init(init_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }
```

```
    private static double test_qsort_mid_sorted() {

        startTime = System.currentTimeMillis();
        qsort_mid(mid_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double test_sf_qsort_unsorted() {

        startTime = System.currentTimeMillis();
        sf_qsort_init(init_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double test_sf_qsort_mid_unsorted() {

        startTime = System.currentTimeMillis();
        sf_qsort_mid(mid_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double testsf_qsortSorted() {

        startTime = System.currentTimeMillis();
        sf_qsort_init(init_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }

    private static double test_sf_qsort_mid_sorted() {

        startTime = System.currentTimeMillis();
        sf_qsort_mid(mid_A);
        return (System.currentTimeMillis() - startTime) / 1000.0;
    }
```

```java
    /**
     * I implemented these in my data structures course to achieve better randomness. In my genetic algorithm assignment it improved the runtime of
     * that program significantly by injecting better randomness into the distribution of choices the algorithm was making when choosing traits and
     * children. Here all it does is generate a better seed for the random number generator in java.
     *
     * "The goal of practical random number generation should be to compute it in as little space,
     * and as little time, while still producing good results that satisfy statistical tests" ~  Melissa O'Neill
     * Harvey Mudd College
     *
     * XORShift satifies all of Mellisa's requirements. She talks about XORshift at timestamp 30:22.
     * https://youtu.be/45Oet5qjlms?t=30m22s
     * https://en.wikipedia.org/wiki/Xorshift
     * https://www.javamex.com/tutorials/random_numbers/xorshift.shtml#.Wtmgwi7wZEY
     */
    private static long XORShift() {
        long x = System.currentTimeMillis();
        x ^= (x << 21);
        x ^= (x >>> 35);
        x ^= (x << 4);
        return x;
    }
    private static long XORShift128plus() {
        long x = System.currentTimeMillis();
        long y = XORShift();
        x ^= (x << 23);
        long z = x ^ y ^ (x >> 26);
        return z + x;
    }

}
```