COM2001 - SUGGESTED SOLUTIONS

JAMES MARSHALL

1 EMPIRICAL PERFORMANCE ANALYSIS

1.2. **Graphing:**

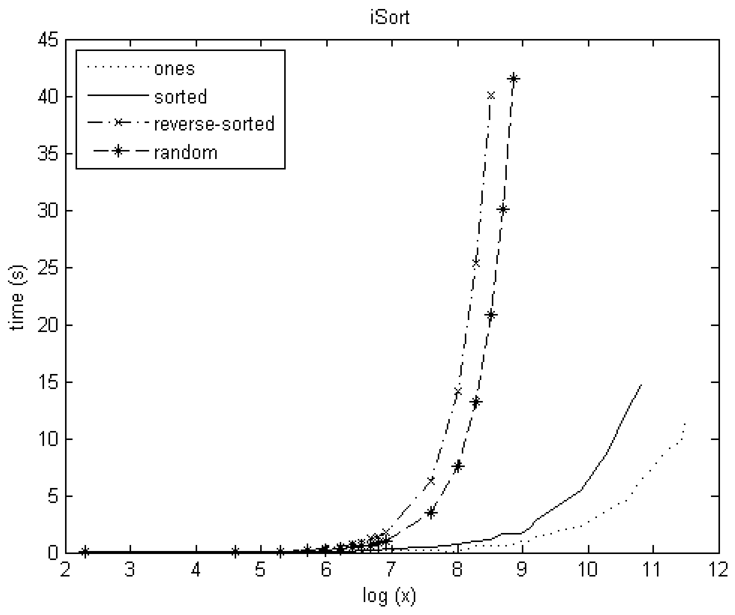

Figure 1: *iSort: this algorithm has an average time complexity of $O(n)$ and a worst case (for example, reverse-sorted and random lists) time complexity of $O(n^2)$.*

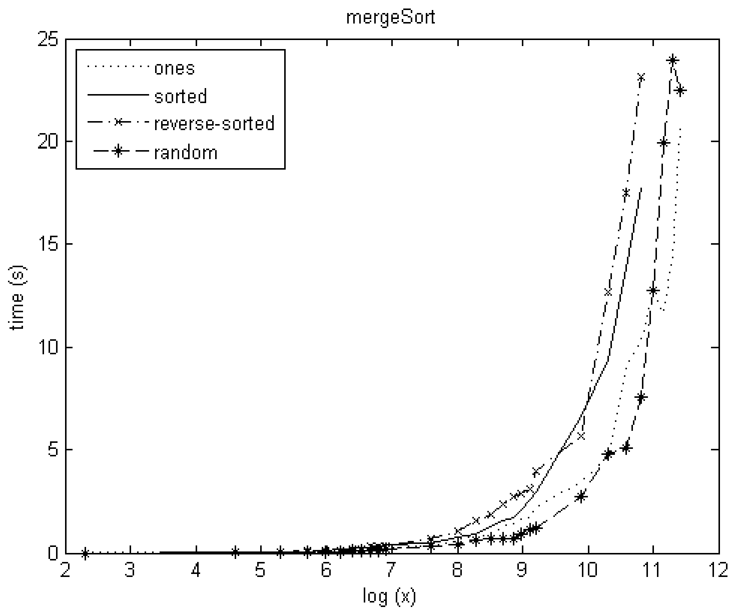

Figure 2: *mergeSort: this algorithm has a time complexity of $O(n\ log(n))$ for all types of lists.*

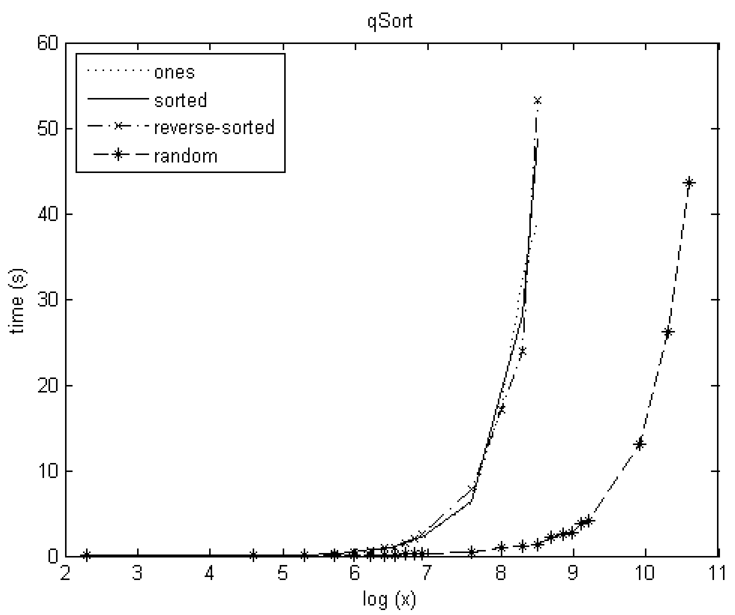

Figure 3: *qSort: this algorithm has an average time complexity of $O(n\ log(n))$ and a worst case time complexity of $O(n^2)$. The time complexity is $O(n^2)$ for sorted and reverse-sorted lists, but $O(n\ log(n))$ for random lists.*

2 ASYMPTOTIC COMPLEXITIES

(a). $O(1) \subset O(log(n)) \subset O(n) \subset O(n\ log(n)) \subset O(n^2) \subset O(2^n)$

(b). $\Omega(2^n) \subset \Omega(n^2) \subset \Omega(n\ log(n)) \subset \Omega(n) \subset \Omega(log(n)) \subset \Omega(1)$

(c). No. $\Theta(g(n))$ means having upper and lower bounds which are constant multiples of $g(n)$). $\Theta(n^2) \subset \Theta(2^n)$ means that every algorithm having upper and lower running time bounds of order n^2 also has upper and lower running time bounds of order 2^n; clearly 2^n can upper-bound n^2, but a constant multiple of it clearly can't also provide a lower bound on n^2.

(d). The summation has a closed-form solution of $(4^{n+1} - 1)/3$. An appropriate asymptotic time complexity is thus $\Theta(4^n)$ (everything else in the closed-form solution being a constant that does not depend on n). The running time will be growing exponentially as the input size increases, as such this algorithm will have incredibly large running times with larger inputs, and will begin to struggle with smaller inputs very quickly.

(e). The summation has a closed-form solution of $7n + 3n(n + 1)/2$. An appropriate asymptotic time complexity is thus $\Theta(n^2)$ (since the n^2 term dominates as n becomes larger, and everything else in the solution is a constant that does not depend on n). This is a polynomial time algorithm, in this case the running time will be growing quadratically as the input size increases. This algorithm will perform well with larger inputs, with the growth curve growing significantly slower than an exponential time algorithm.

3 RECURRENCES AND THE MASTER THEOREM

(a).$a = 1, b = 3/2$ hence case 2 applies as $f(n) \in \Theta(1) = \Theta(n^0) = \Theta(n^{\log_{3/2} 1})$. Therefore $T(n) \in \Theta(\log_2 n)$.

(b). $a = 9, b = 3$ hence case 1 applies with $\epsilon = 1$ as $f(n) \in \Theta(n) = \Theta(n^{\log_3 9 - 1})$. Therefore

$T(n) \in \Theta(n^2)$.

(c). $a = 3, b = 4$ hence case 3 applies with $\epsilon \approx 0.2$ as $f(n) \in \Omega(n^1) = \Omega(n^{log_4 3+\epsilon})$, and $3n/4 \log_2(n/4) \leq cn \log_2 n$ for $c = 3/4$ and for large enough n. Therefore $T(n) \in \Theta(n \log_2 n)$.

(d). $a = 4, b = 2$ hence case 2 applies as $f(n) \in \Theta(n^2) = \Theta(n^{\log_2 4})$. Therefore $T(n) \in \Theta(n^2 \log_2 n)$.

(e). $a = 2$ but b is non-constant, hence the Master Theorem does not apply

(f). $a = 4, b = 2$ hence case 1 applies with $\epsilon = 1$ as $f(n) \in \Theta(n) = \Theta(n^{\log_2 4-1})$. Therefore $T(n) \in \Theta(n^2)$.

(g). $a = 2, b = 2$ however the Master Theorem does not apply. Case 2 clearly does not apply as $f(n) \notin \Theta(n)$. Case 1 does not apply as if $f(n) \in O(n^{1-\epsilon})$ we would have $n \log_2 n \leq cn/n^\epsilon$ which can be rearranged to give $c \geq n^\epsilon \log_2 n$, hence c cannot be a constant. Similar reasoning shows case 3 cannot apply either as if $f(n) \in \Omega(n^{1+\epsilon})$ we would have $c \leq (\log_2 n)/n^\epsilon$. The problem is we cannot establish a tight (case 2), polynomially larger (case 1) or polynomially smaller (case 3) bound on $f(n)$.

(h). $a = 4, b = 2$ hence case 3 applies with $\epsilon = 1$ as $f(n) \in \Theta(n^3) = \Theta(n^{\log_2 4+1})$, and $4(n/2)^3 \leq cn^3 \implies 1/2 \leq c$ which satisfies the condition $c < 1$. Therefore $T(n) \in \Theta(n^3)$.

(i). $a = 1$ but b is non-constant, hence the Master Theorem does not apply.

4 INDUCTIVE PROOFS

```
(a). expsum z n == (1 - z ^ (n + 1)) / (1 - z)
```

Base case:

```
expsum z 0 == (1 - z^(0 + 1)) / (1 - z)

expsum z 0 == (1 - z^1) / (1 - z)

expsum z 0 == (1 - z) / (1 - z)

expsum z 0 == 1
```

From `(expsum)`

```
1 == 1
```

Inductive step:

```
expsum z n == (1 - z^(n + 1)) / (1 - z) ==>
 expsum z (n + 1) == (1 - z^(n + 2)) / (1 - z)

expsum z (n + 1) == (1 - z^(n + 2)) / (1 - z)
```

From `(expsum2)`

```
z^(n + 1) + expsum z n == (1 - z^(n + 2)) / (1 - z)
```

From inductive hypothesis

```
z^(n + 1) + (1 - z^(n + 1)) / (1 - z) == (1 - z^(n + 2)) / (1 - z)

(1 - z^(n + 1)) / (1 - z) ==
 (1 - z^(n + 2) - (1 - z) * z ^ (n + 1)) / (1 - z)

(1 - z^(n + 1)) / (1 - z) ==
 (1 - z^(n + 2) - z^(n + 1) + z^(n + 2)) / (1 - z)

(1 - z^(n + 1)) / (1 - z) == (1 - z^(n + 1)) / (1 - z)
```

Q.E.D.

```
(b). reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Base case A:

```
reverse ([] ++ ys) == reverse ys ++ reverse []
```

From `(++1)`

```
reverse ys == reverse ys ++ reverse []
```

We must now prove the above statement by proving the following.
Inductive hypothesis B:

```
xs ++ [] == xs
```

Base Case B:

```
[] ++ [] == []
```

From `(++1)`

```
[] == []
```

Inductive step B:

```
xs ++ [] == xs ==> (x:xs) ++ [] == (x:xs)

(x:xs) ++ [] = (x:xs)
```

From `(++2)`

```
x:(xs ++ []) == (x:xs)
```

From inductive hypothesis B

```
x:(xs) == (x:xs)
```

Inductive step A:

```
reverse (xs ++ ys) == reverse ys ++ reverse xs ==>
 reverse ((x:xs) ++ ys) == reverse ys ++ reverse (x:xs)

reverse ((x:xs) ++ ys) == reverse ys ++ reverse (x:xs)
```

From `(++2)`

```
reverse (x:(xs ++ ys)) == reverse ys ++ reverse (x:xs)
```

From `(reverse2)`

```
reverse (xs ++ ys) ++ [x] == reverse ys ++ (reverse xs ++ [x])
```

From inductive hypothesis

```
(reverse ys ++ reverse xs) ++ [x] ==
  reverse ys ++ (reverse xs ++ [x])
```

which is true by associativity of the `++` operator, Q.E.D.

```
(c). foldr (*) 1 (map (2^) xs) == (2^) (foldr (+) 0 xs)
```

Base case:

```
foldr (*) 1 (map (2^) []) == (2^) (foldr (+) 0 [])
```

From `(map1)`

```
foldr (*) 1 [] == (2^) (foldr (+) 0 [])
```

From `(foldr1)`

```
1 == (2^) 0

1 == 1
```

Inductive step:

```
foldr (*) 1 (map (2^) xs) == (2^) (foldr (+) 0 xs) ==>
 foldr (*) 1 (map (2^) (x:xs)) == (2^) (foldr (+) 0 (x:xs))

foldr (*) 1 (map (2^) (x:xs)) == (2^) (foldr (+) 0 (x:xs))
```

From `(map2)`

```
foldr (*) 1 ((2^) x: map (2^) xs) == (2^) (foldr (+) 0 (x:xs))
```

From `(foldr2)`

```
(*) ((2^) x) (foldr (*) 1 (map (2^) xs)) ==
 (2^) ((+) x (foldr (+) 0 xs))
```

From inductive hypothesis

```
(*) ((2^) x) ((2^) (foldr (+) 0 xs)) ==
 (2^) ((+) x (foldr (+) 0 xs))
```

Let `k = (foldr (+) 0 xs)`

```
(*) ((2^) x) ((2^) k) == (2^) ((+) x k)

(*) (2^x) (2^k) == (2^) (x + k)

(2^x) * (2^k) == 2^(x + k)
```

which can be confirmed by algebra, Q.E.D.