Compsci Assignment 2

Rahul Issar

Question 1

The conventional one (k =2), the sorting time is $T_2(n) = cnlog_2n$

T(n) = 2T(n/2) + n Substituting T(n/2) = 2T(n/4) + n/2

T(n) = 2T(n/2) + n = 2[2T(n/4) + n/2] + n = 4T(n/4) + n + n

T(n) = 4T(n/4) + 2n, Substituting T(n/4) = 2T(n/8) + n/2

T(n) = 4T(n/4) + 2n = 4[2T(n/8) + n/4] + 2n = 8T(n/8) + n + 2n

T(n) = 8T(n/8) + 3n, Substituting T(n/8) = 2T(n/16) + n/2

T(n) = 8T(n/8) + 3n = 8[2T(n/16) + n/8] + 3n = 16T(n/16) + n + 3n

T(n) = 16T(n/16) + 4n

Rewrite the above equations and look for a pattern.

1st step of recursion T(n) = 2T(n/2) + n = 21T(n/21) + 1n

2nd step of recursion T(n) = 4T(n/4) + 2n = 22T(n/22) + 2n

3rd step of recursion T(n) = 8T(n/8) + 3n = 23T(n/23) + 3n

4th step of recursion T(n) = 16T(n/16) + 4n = 24T(n/24) + 4n

Closed-form formula

T(n) = $2^kT\left(\frac{n}{2^k}\right) + kn$

We want T(1), so we let n = 2^k. Solving for k, we get k = logn. If we plug this back in:

T(n) = $2^{logn}T\left(\frac{2^k}{2^k}\right) + (logn)n = n * T(1) + (logn)n = n + nlogn\ T(n) = n + nlogn.$

Modified mergesort

If we substitute any k > 2 into the equations above, the closed-form formula will be:

When k = 3 $3^k T\left(\frac{n}{3^k}\right) + kn$

When k = 4 $4^k T\left(\frac{n}{4^k}\right) + kn$

When k = 5 $5^k T\left(\frac{n}{5^k}\right) + kn$

When k = 6 $6^k T\left(\frac{n}{6^k}\right) + kn$

Thus, the sorting time will be $T_k(n) = cn\log_k n$, hence $T_k(n) = cn\log_k n$.

This means that the greater the value of k, the faster the sorting time. Therefore, the modified mergesort could be faster for some k > 2 than the conventional one (k =2).

Question 2

Tqselect (n) = cn
Tqsort(n) = $cnlog_2(n)$
cn < $cnlog_2(n)$
1 < $log_2(n)$
2 < n Therefore option (a) is faster for any n greater than 2.

Since k = 15 and n = 100,000,000 ≈ $2^{36.6}$, while c remains constant for both.
Tqselect(n) = cn = 100,000,000c
Tqsort(n) = $cnlog_2(n)$ 2657542475.90989c

10000000c < 2657542475.90989c

Therefore option(a) is faster

Question 3

Process(a) – in linearithmic time O(nlogn), heapsort repeats n times the deletion of the maximum key and restoration of the heap property, where each restoration is logarithmic in all best, worst, and average case).

Process(b) – Building a heap in linear time ø recursive structure. This means that the comparing process starts from the base case. The lower bound is ø since every input element must be inspected.

Because process(a) does not have the recursive structure, and that the heapsort in process(a) repeats n times the deletion of the maximum key and restoration of the heap property, process (b) yields such acceleration.

Question 4

Maximum

1,2,3,4,5,6,7
7,6,5,4,3,2,1
1,7,6,5,4,3,2
7,1,2,3,4,5,6,

Minimum

4,2,1,3,6,5,7
4,6,5,7,2,1,3
4,2,3,1,6,5,7
4,6,7,5,2,1,3

Question 5

UID: 588382864

Combined array of 9's compliment and UID = {1,2,3,4,5,6,7,8}

Minimum

4,2,1,3,6,5,7,8
4,6,7,8,5,2,1,3

Question 6a

Knuth-Morris-Pratt

i	1	2	3
Pattern[i]	a	b	c
Prefix[i]	0	0	0

First is always 0
Pre: a, suffix is b
Pre: a, ab, suffix: c, bc

Pattern: abc
String: aabcbcbabcabcabc

1) a matches
2) b doesn't match shift down 1
3) a matches
4) b matches
5) c matches

Boyer-Moore algorithm

aabcbcbabcabcabc
abc

1) c doesn't match
2) So, we shift one to the right so we get both b's matching.

aabcbcbabcabcabc
 abc

3) c matches
4) b matches
5) a matches

Question 6b

Knuth-Morris-Pratt
Pattern = abc
String = abababababababab

1) a matches
2) b matches
3) c doesn't match
4) a matches
5) b matches
6) c doesn't match
7) a matches
8) b matches
9) c doesn't match
10) a matches
11) b matches
12) c doesn't match
13) a matches
14) b matches
15) c doesn't match
16) a matches

17) b matches
18) c doesn’t match
19) a matches
20) b matches
21) c doesn’t match

Boyer-Moore algorithm

Pattern = abc
String = abababababababab

```
abababababababab
abc
```

1) c doesn’t match
2) Shift so a’s match

```
abababababababab
  abc
```

3) c doesn’t match
4) Shift so a’s match

```
abababababababab
    abc
```

5) c doesn’t match
6) Shift so a’s match

```
abababababababab
      abc
```

7) c doesn’t match
8) Shift so a’s match

```
abababababababab
        abc
```

9) c doesn’t match
10) Shift so a’s match

```
abababababababab
          abc
```

11) c doesn't match
12) Shift so a's match

```
abababababababab
      abc
```

13) c doesn't match
14) Shift so a's match

```
abababababababab
        abc
```

15) c doesn't match
16) Shift so a's match

Question 7

The algorithm searches a string for matches to a key, when it matches a character of the string to the first character of the key, it begins comparing the next character of the string. If that matches then it compares the next until it fails.

So, to maximise comparisons is to have the algorithm fail just before finding a match, but also to hide partial matches within such a string so it has to also consider that partial match after it fails.

e.g. Given the key "ABABCDB" we could use the following:

```
ABABABAB
```

Here it will compare the first four characters ABAB, then it will go back and start at the second A and compare ABAB again, then the third, and so on. It will compare fail on each turn after

```
ABAB
ABAB
ABAB
AB
```

So, in this case, we have 14 comparisons.

Question 8

The Boyer-Moore-algorithm uses bad character rule and suffix rule. Since we know it must be scanned from the right hand side we need to know the most possible comparisons.

The pattern can be P = bbababab. In the first scan it will be compared 8 times, since the first index of the pattern is found to be the mismatch, the pattern will shift by 1. If we look at the 7^{th} index of the patter we can see it will be a shift by 7 even though there is a match. Every second shift will conclude to be a maximum comparison of 8 times. This will create the largest possible amount of comparisons, using the Boyer-Moore algorithm