Robert Barnes

20407 DSA 2013b

Maman 15

1) Assume a set of unique integers S, in the range $[0 \dots n^k - 1]$ such that $k \geq 1, z \in \mathbb{N}$

A) Find if there exist $a, b \in S : a \neq b \wedge a + b = z$ in $\Theta(n \cdot min(k, \lg n))$ time.

First we sort the elements. If $k \geq \lg n$ we use heapsort, which is $\Theta(n \lg n)$. Otherwise we use a base n Radix sort. Thus we get that the number of digits needed to represent a number is $\log_n n^k$ and we do $\Theta(n \log_n n^k + n \log_n n^k) = \Theta(2nk)$. Next we call the following `findPair` function on the entire array which takes $O(n)$ time:

```
// A is the array, z the number we're looking for, s the first index, e the
// last index.  It return's the indexes of values.  If s >= e no values where
// found.
findPair(A, z, s, e)
while ( s < e )
    while ( A[e] > z - A[s] ) e = e - 1
    if ( A[s] + A[e] = z ) return (s,e)
    while ( A[s] < z - A[e] ) s = s + 1
    if ( A[s] + A[e] = z ) return (s,e)
return (s, e) // failure
```

B) Find 3 values that equal z in $\Theta(n^2)$ time.

First we sort as in part A. We then iterate over the array making n calls to findPair on the whole array for a total running time of $\Theta(n^2)$.

```
findTriple(A, z, s, e)
while ( s <= e )
    (s1, e1) = findPair(A, z - A[s], 1, e)
    if ( s != s1 && s != e1 && s1 != e1 ) return (s, s1, e1)
    s = s + 1
return (s, s1, e1) // failure s = e + 1
```

C) Find 4 values that equal z in $\Theta(n^2 \cdot min(k, \lg n))$ time.

First generate an array of all the sums of two distinct elements, including a parallel helper array which contains the values each sum was created from.

This is equivalent to choosing 2 from n and we have that the time to do this is $C(n,2) = \frac{n(n-1)}{2} = \Theta(n^2)$. The largest possible value in this set will be $2(n^k - 1)$. If $k \geq \lg n$ we use heapsort which takes $\Theta(n^2 \lg n^2) = \Theta(n^2 \lg n)$ time. Otherwise we use a base n Radix sort. This takes $\Theta(n^2 \cdot \log_n 2n^k + n \log_n 2n^k) = \Theta(n^2 \cdot k)$. We then have a function `findQuad` which calls `findPair` on the sums array. If a pair is found, we look in the helper array to make sure the original four values are unique. If they are we return them. If not, and the `s<e`, we call `findPair` again on the remaining subarray. This all takes takes $O(n^2)$ time. Thus, the algorithm is dominated by the cost of sorting the sums array, and thus meets the run time requirements.

D) Find 5 values that equal z in $\Theta(n^3)$ time.

We combine the appoaches of part B and part C. We sort the original array as in part A. We then create and sort the sums array and it's helper array in $\Theta(n^2 \cdot min(k, \lg n))$ time. We then iterate over the original array subtracting the current element from z and calling `findQuad` from part C on each value. If we find a valid quad set, we check the values for uniqueness against the current element. If the uniqueness condition isn't met and `s<e`, we continue looking. This makes up to n calls to `findQuad` at a cost of $O(n^2)$ each for a total running time which is $O(n^3)$. The lower bound is the time to sort the sums array.

2)

A) Given a file containing lines of text, sort according to alphabetical order. Assume rows can be swapped in constant time. Do this in $O(n)$ time where n is the number of characters in the file.

For simplicity we assume an alphabet of 26 letter a-z representing values 1-26. The algorithm is equivalent for any size alphabet with a total ordering over the symbols in that alphabet. We treat the lines of text as a set of k strings with a total of n base 26 digists. Ordering of the strings is left to right such that $aa < z$. We have that $1 \leq k \leq n$.

```
// SL is a list of strings, indexable by string position and char position
// within the string.  firstChar is the index of the first char in the
// substrings we're sorting.  first and last are the subrange we're sorting
sortStrings( stringList SL, firstChar, first, last )
    // countingSort sorts the strings in the subrange based on the ith character.
    // If there is no ith character it's treated as the equivalent of the NIL
    // character.
    countingSort(SL, firstChar, first, last)
    // call sortStrings recursively on each range of strings that start with the
    // same letter
    start = end = first
```

```
while ( start < last )
    while ( SL[start][firstChar] = SL[end][firstChar] ) end = end + 1
    if ( start < end - 1 && SL[start][firstChar] != NIL )
        sortStrings(SL, firstChar+1, start, end - 1)
    start = end
```

This would sort all strings based on their first letter, then all strings which have the same first letter would be sorted as a group on their second letter and so on. If the length of a string is j, then it is in it's final position after at most $j + 1$ sorts and will no longer be included in further recursive calls. Looking at it from the point of view of columns, the time it takes to sort the $i + 1$th column is proportional to the number of non-NIL characters in the ith column. Thus the run time is the sum of the run times of sorting each column, which is proportional to the sum of the number of non-NIL characters in each column, thus the algorithm is $O(n)$.

B) Place a number at the beginning of each line of text. The total number of digits in the numbers is n. Sort in $O(n)$ time.

We assume that the numbers are unique, because other wise we don't have enough information to solve the problem. Thus we sort only on the numbers, ignoring the text. The numbers have the property that longer numbers are larger than shorter numbers. We can count the number of digits in each number in $O(n)$ time and then sort the rows by length using counting sort in $O(n)$ time since the maximum length is n and the number of numbers is less than equal to n. We then radix sort each set of numbers that are the same length. Again, if we look at this from the point of view of columns, it's equivlent to calling counting sort once on each column and the run time is proportional to the sum of the number of characters in each column, which is of course n, meaning the algorithm runs in $O(n)$ time. For instance if the initial sort creates k groups and there are k_j digits in ith column then counting sort is called once on each of those groups and the total cost of sorting the column is the sum of the costs of each call which is proportional to the number of digits in the column.

3) Given an array `A`, we implement two stacks `LS` and `RS` and a dequeue `DQ` by placing the base of `LS` at the beginning of `A` and growing up, placing the base of `RS` at the end of `A` and growing down, and placing the base of `DQ` in the middle of `A` and growing in both directions.

For `LS` and `RS` we store `top` values which index the next free position. Similarly, we store two top values for `DQ`, a left one and a right one. When inserting values we check if it has passed one of it's neighbor tops to check for overflow. We then write the value to `top` and then increment / decrement it. For example,

```
DQ.leftEnqueue(x)
if ( LS.top > DQ.leftTop )
    error( "stack overflow" )
else
    A[DQ.leftTop] = x
    DQ.leftTop = DQ.leftTop + 1
```

All pushing / enqueueing would be similar in spirit. In order to pop / dequeue elements we simply do something in the form:

```
RS.pop()
if ( RS.top = A.length )
    error( " stack underflow")
else
    RS.top = RS.top + 1
    return A[RS.top]

DQ.leftDequeue()
if ( DQ.leftTop = A.length / 2 )
    error("stack underflow")
else
    DQ.leftTop = DQ.leftTop + 1
    return A[DQ.leftTop]
```

To check overflow status we simply check if neighboring tops have passed each other, such as `if ( RS.top < DQ.rightTop ) return true`.

4) Accept a string a character at a time and check if it's a palindrome.

The algorithm can be implemented using two dynamic stacks implemented using linked lists. Initially we read each character and push it on the first stack, `S1`. Once we've read all the characters, we pop $\lfloor n/2 \rfloor$ characters one at a time and push them onto the second stack, `S2`. If n is odd we pop one more value off `S1` and disgard it. We then pop one value at a time off of the two stacks and compare them. If all the values are identical the string is a palindrome, otherwise it's not.

```
IsPalindrome()
n = 0
while ( ( c = readChar() ) != EOF )
    push(S1, c)
    n = n + 1
count = n
while ( count != floor( n / 2 ) )
```

```
    push(S2, pop(S1))
if ( isOdd(n) )
    pop(S1)
while ( notEmpty(S1) )
    if ( pop(S1) != pop(S2) )
        return false
return true
```

5) Give a $O(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

Remove elements one at a time from the head of the linked list, and insert them at the head of a second linked list. Once the first list is empty assign it's head to point to the head of the second list.

```
Reverse(L1)
while (L1.head != NIL)
    temp = L1.head
    L1.head = L1.head.next
    temp.next = L2.head
    L2.head = temp
L1.head = L2.head
return
```