ASSIGNMENT: HW9 ①

NAME: PRASANTH BHAGAVATULA FAEL2015

CWID: A20355611

① If we use $\omega_n = e^{2\pi q i/n}$, instead of $\omega_n = e^{2\pi i/n}$, we are simply raising the ~~power of~~ n^{th} roots of unity to the power of q. and so we are considering $(n/q)^{th}$ roots of unity. Although we are considering 'n' values to compute the point-values, we are actually computing the same value 'q' times and so, there will be only 'n/q' distinct values.

So, if there are 'n' distinct values with the original FFT algorithm there will be 'n/q' distinct values with the new algorithm.

ASSIGNMENT: HW9

NAME: PRASANTH BHAGAVATULA

CWID: A20355611

②

FALL2014

② The iterative-FFT computes the twiddle factor as many number of times as the innermost loop (lines-9-13) runs for each stage. The outer loop runs for n/m times and inner loop runs for $m/2$ times for each of outer loop run. Hence for each stage, the total number of times the twiddle factor computed are

$$n/m * m/2 = n/2$$

If we look at the example of iterative-fft with $n=8$, in the first stage we use $w_m = w_2$ and the inner loop runs only 1 time. So, the unique values of w are $1, w_2$. Similarly, if we look at second stage, we use $w_m = w_4$ and the inner loop runs for 2 times and so, the values of w are $1, w_4, w_4^2$. Hence, we need to compute only w_4 and w_4^2 for second stage. Similarly, for the 3rd stage $w_m = w_8$ and inner loop runs 4 times and the unique values of w are $1, w_8, w_8^2, w_8^3$. Hence computing these values before hand, will reduce the number of computations and so, at each stage s we require only 2^{s-1} twiddle factor and so, only those number of computations.

ASSIGNMENT: HW9

NAME: PRAVANTH BHAGAVATULA

CWID: A20355611

(3)

FALL2015

(3) (a) The twiddle factor $\omega_n^{n/2}$ is ~~will be the~~ computed by most number of multiplications. When the 'log n' stage is running, the ω_m = will be ω_n = , the nth roots of unity, and the inner loop runs for 'n/2', times each time multiplying ω by ω_n ~~and~~, and starting 'w' value is '1'.

Hence, $\omega_n^{n/2}$ is computed ~~the~~ by $n/2$ multiplications.

(b)

ASSIGNMENT: HW9
NAME: PRANEETH BHAGAWATULA
CWID: A20355611

(4) Given a function $rev_k(a)$ which runs in $\Theta(k)$ time consider, the following algorithm to compute the bit-reversal permutation on an array of input size $n = 2^k$.

```
BIT-REVERSAL-PERMUTATION (A)
n = A.length
for i = 0 to n-1
      B[i] = 0
for i = 0 to n-1
      if B[i] == 0
            reverse = rev_k (i)
            temp = A[i]
            A[i] = A[reverse]
            A[reverse] = temp
            B[i] = 1
            B[reverse] = 1
      end-if
end-for
```

We consider a new array B, which will ~~had~~ have '0' if a swap is not performed for that index and 1 if a swap is performed. The elements $A[i]$ needs to be swapped with $A[rev_k(i)]$ only once and so using this array we ensure that we swap only once.

~~Since the loop runs for 'n' times~~

Although the loop at line 4 runs for 'n' times, ~~a~~ swapping ~~of~~ ~~atmost~~ will happen ~~n/2~~ ~~element~~ for $n/2$ times and so the $rev_k(a)$ function is called $n/2$ times. Hence the running time is $O(nk)$

Although a slight improvement can be ~~fo~~ found by running the loop in line 4 ~~from~~ from 1 to $n-2$, since $rev_k(0) = 0$ and $rev_k(n-1) = n-1$ but that does not have much impact ~~on~~ ~~the~~ ~~entire~~ ~~average~~ ~~running~~ if n is pretty large.

(b) The bit-reversed increment is pretty much similar to the bit-increment algorithm in the textbook (page - 454). and the difference is that in BIT-INCREMENT, we start from lower bit, here we start from higher order bit.

```
BIT-REVERSED-INCREMENT (A)
~~i=0~~ n = A.length
~~while~~ ~~i=0~~ ~~and~~ i = n-1
While  i >= 0  and A[i] == 1
          A[i] = 0
          i = i - 1
end-while
if  i >= 0
          A[i] = 1
end-if.
```

~~Goingo by this approa~~

As we know that ~~by~~ the amortitized cost of BIT-INCREMENT is constant, the amortized cost of this algorithm is also constant and so for an array of 'n' element, we can find the reverse of bit-~~reversed~~ number of an index ~~is~~ each in constant time and so the ~~cos~~ overall cost is $O(n)$

© The BIT-REVERSED-INCREMENT ~~do~~ does not use any shifting of words and so, ~~the algorithm still~~ we can still calculate the n-element bit reversal permutation in $O(n)$ time.