FA 3

Fundamental Algorithms : HW 3

Name : Advait Savant (N14108183)

1.a If $A[0, \_, (n-1)]$ is rotation sorted, we propose the following.
let $x_0, \_, x_{n-1}$ be the array elements in sorted order. let X be the corresponding sorted array.
let $a_0, \_, a_{n-1}$ be the indices for the array A.
let k be the index of the largest element.
Then,

- The index of the smallest element, s, $= (k+1) \bmod n$

- For an arbitrary non-boundary condition, where $a_k = x_{n-1}$,

$$\begin{bmatrix} a_0 & a_1 & - & a_{n/2} & a_{n/2+1} & \overset{x_{n-1}}{a_k} & \overset{x_0}{a_{k+1}} & a_{n-1} \end{bmatrix}$$

- The elements $a_0, a_1 \_, a_k$ are in sorted order and are the last $(k+1)$ elements of the array X.
- The elements $a_{k+1}, \_, a_{n-1}$ are in sorted order and are the first $(n-k-1)$ elements of the array X.

Proof :-

- At the boundary condition, we get the sorted array which is by definition rotation sorted with $c=0$, x_{n-1} is at position a_{n-1}

- For any other condition,
  let a_k be the position of x_{n-1}
- Hence, a_{k+1} is the position of x_0
- In order to get sorted array X from A using cyclic shifts, we must move necessarily element x_n at position a_k to position a_{n-1}.
  Hence, our c value must be $(n-1-k)$.
- For any k, we now show that the described configuration is the only one which holds for rotation sorting to be vali'
- Firstly, note that, for some k, we get a corresponding c, and, for each c, $0 \leq c < n$, the elements of A are cyclic shifted by that amount and the new array produced is unique. The mapping $R(A, c) \rightarrow A_{new}$ is
- one to one. R is the cyclic shift function.
- The inverse mapping is obtained by subtracting c under modulus n.
- If we show that our configuration when cyclic shifted produces X, it must be the only configuration of A for a given k which produces the sorted array X.
- For x_0 with index a_{k+1}:
  new index is $[k+1 + (n-k-1)] \bmod n$
  which is $n \bmod n$, which is therefore 0. x_0 will be correctly placed at a_0.

- Elements $a_{k+2}, \ldots a_{n-1}$ are increasing order from x_0.
- We can see that a_{k+i} is now in the correct position a_i, for these elements.
- Similarly, elements $a_0, \ldots, a_k$ are cyclic shifted $(n-k-1)$ forward and occupy the correct position in the ~~sorted~~ sorted array.
- Here, element $(k-j)$ is correctly in position $[k-j+(n-k-1)] \bmod n$ which is $(n-1-j)$.
- We thus obtain the sorted array X from A for our described configuration for a given k where k is the position of the largest element and gives us the corresponding c value.

- $k \in [0, n-1]$

  if $k \leq n/2 - 1$, elements $a_{k+1} - a_{n-1}$ will be formed by elements $x_0, x_1 -$ until the end of the array in increasing order and thus the second half of the array will be sorted.

  if $k > n/2 - 1$, elements $a_0 - a_{k-1}$ will be formed by the corresponding elements less than k in decreasing order from $k-1$ to 0. Thus the first half $A[0, -, n/2 - 1]$ will be sorted.

  if $k = n-1$, both halves will be sorted as essentially the whole array is sorted.

1 · D

```
Find_min (A, start, end)
{
    n = end - start + 1          # find no. of
                                    elements

    if (n == 2) :
        if (A[start] < A[end]) :
            return A[start]
        else :
            return A[end]           # base case

    if (n == 1) :
        return A[start]

    if A[n/2 - 1] > A[n/2] :        # case when
            return A[n/2]              xn is at a_{n/2 - 1}
                                       and x0 is at a_{n/2}

    if A[n/2] > A[n/2 - 1] :

# Condition 1      if A[n-1] < A[n/2 - 1] :

                        Find_min (A, start + n/2, end)

# Condition 2      if A[n-1] > A[n/2 - 1] :

                        Find_min (A, start, )
```

end $-\frac{n}{2}$)

}

- For each function call with the size of the array as n, the algorithm finds the correct subarray with size $n/2$ which contains the minimum element. It keeps on dividing the array until we reach the base case where it returns the correct answer.

- Consider Condition 1 where we are showing that the element x_0 must be in $(n/2, n-1]$
    - let x_0 be at position $n/2 + k$
    - We will have an increasing sequence from positions $a_0, a_1 —, a_{n/2+k-1}$ where the last element will be x_n. let this be set 1. Here, $x_n \equiv A[n/2 + k - 1]$
    - Thus $A[n/2 - 1] < A(n/2]$ in condition 1; this is a valid statement.
    - Also, elements $a_{n/2+k}, —, a_{n-1}$ correspond to elements $x_0, —, x_{\frac{n}{2}-k-1}$ starting

with the smallest element x_0 and in increasing order. Let this be Set 2. Set 2 will be till $A(n-1]$. $A[0]$ will be the next element in X after Set 2 elements and Set 1 will start considering the sorted array X.

the sorted array x

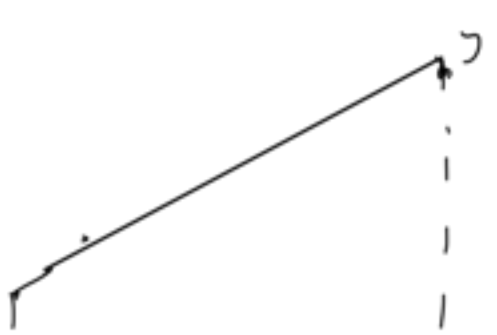
- Set 1 will be from $A[0]$ to $A[n/2+k-1]$
- So the second statement of condition 1 which states that $A[n/2-1]$ is greater than $A[n-1]$ will be valid as set 2 elements come before Set 1 elements in the sorted array.
- This demonstrates the correctness of our statements for condition 1.

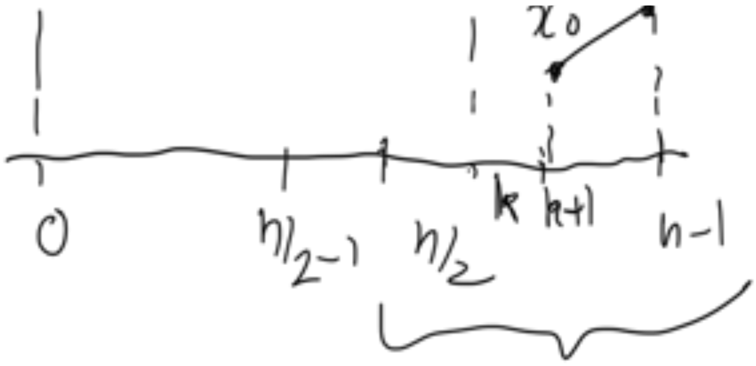
- For condition 2, the 2 statements can be shown to be correct similarly.
- At the boundary condition wherein $x_0 \equiv a_{n/2}$ and $x_n \equiv a_{n/2-1}$, we realise that this will be the only scenario where $A[n/2-1] > A[n/2]$ and we return the element x_0 at position $A[n/2]$ correspondingly.

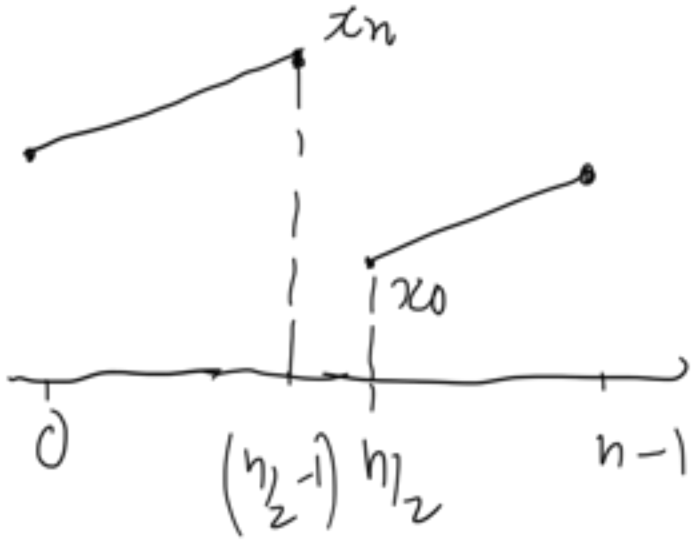
- Until we reach the base case, the size of the array is halved and since n is finite, the algorithm terminates.

- For brevity of representation, consider the cases pictorially.

<u>Condition 1</u>

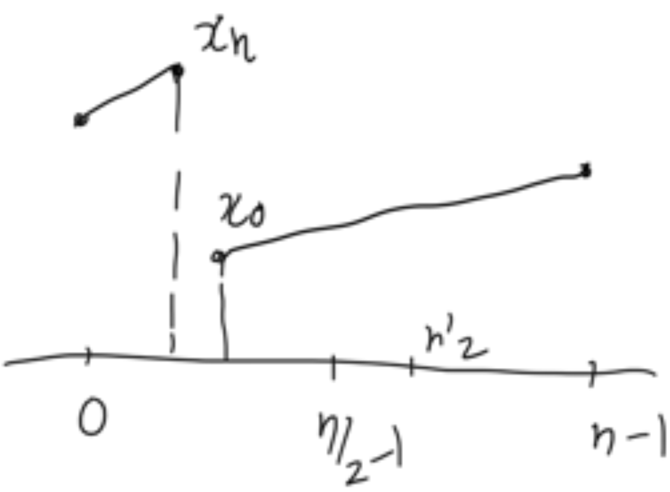


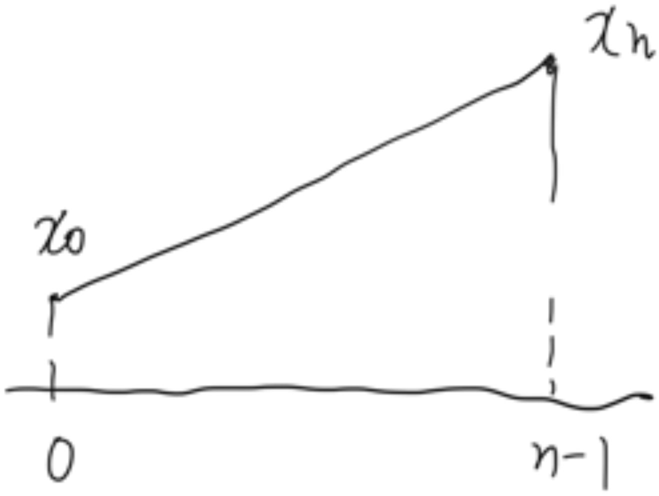




Boundary case

Condition 2





Condition 2

Statements
are valid when
array is sorted
fully

Consider the time complexity,

$$T(n) = T(n/2) + O(1)$$

$k = \log_2 n$, $T(n) = S(k)$

$$S(k) = S(k-1) + O(1)$$

Telescoping series

$$S(k) = O(k)$$

$$T(n) = O(\log n)$$

Can be verified with master theorem,

$$T(n) = a\,T(n/b) + f(n)$$
$$T(n) = 1\,T(n/2) + C$$
$$f(n) = \text{constant}$$
$$n^{\log_b a} = \text{constant}$$

$$T(n) = \Theta(n^{\log_b a} \log n)$$

$$\therefore T(n) = O(\log n)$$

Thus, using divide and conquer, we get a better running time than the linear time naive algorithm.

2.a

$$X = X_0 + X_1 2^{1\cdot\frac{n}{m}} \cdots X_{m-2} 2^{(m-2)\frac{n}{m}} + X_{m-1}^{(m-1)\frac{n}{m}}$$

$$Y = Y_0 + Y_1 2^{1\frac{n}{m}} \cdots Y_{m-2} 2^{(m-2)\frac{n}{m}} + Y_{m-1}^{(m-1)\frac{n}{m}}$$

X, Y are n bit integers.

$$Z = XY$$

$$Z = Z_0 + Z_1 2^{1\cdot\frac{n}{m}} \cdots Z_{2m-3}^{(2m-3)\frac{n}{m}} + Z_{2m-2}^{(2m-2)\frac{n}{m}}$$

$$Z_0 = X_0 Y_0$$
$$Z_1 = X_0 Y_1 + Y_0 X_1$$
$$Z_2 = X_0 Y_2 + X_1 Y_1 + X_2 Y_0$$
$$Z_3 = X_0 Y_3 + X_1 Y_2 + X_2 Y_1 + X_3 Y_0$$
$$\vdots$$
$$Z_m = X_0 Y_m \cdots X_m Y_0$$
$$Z_{m+1} = X_1 Y_m \text{———} X_m Y_1$$
$$\vdots$$
$$Z_{2m-2} = X_{m-1} Y_{m-1}$$

consider the number of bits which each operation $X_i Y_j$ will occupy individually.

X_i : $\frac{n}{m}$ bits : max : $2^{n/m} - 1$

Y_j : $\frac{n}{m}$ bits : max : $2^{n/m} - 1$

$X_i Y_j$: max : $2^{2n/m} - 2^{n/m+1} + 1$

To represent 2^n we need $(n+1)$ bits
$2^n - k$ can be represented using n bits.

$X_i Y_j$ will be allocated $2n/m$ bits.

Z_0 - $\frac{2n}{m}$ bits

Z_1 - 1 addition of two numbers with $2\frac{n}{m}$ bits.

Z_k - k additions of numbers with $2n/m$ bits each

$1 \leq k \leq m$

$\vdots$

Z_{m+1} - $(m-1)$ such additions, after $k=m$, the number of additions in each pass decreases as seen from the equations.

Z_{2m-2} - $2n$ bits

$$2^{n} \quad \frac{}{m}$$

Consider this as we find bounds on the length of bits needed,

$$Z_k \equiv k \text{ additions of } \frac{2n}{m} \text{ bit terms each max } \{2^{n/m}-1\}$$

$$\equiv \underbrace{\left(2^{2n/m}-1\right) + \left(2^{2n/m}-1\right) \text{ --- } \left(2^{2n/m}-1\right)}_{(k+1) \text{ times}}$$

$$\equiv (k+1)\, 2^{2n/m} - (k+1)$$

$$\equiv 2^{\log_2(k+1) + 2n/m} - 2^{\log_2(k+1)}$$

This can be represented in $\frac{2n}{m} + \log_2(k+1)$ bits.

for $k \in \{1, m\}$
This is bounded by $\frac{2n}{m} + \log_2(m+1)$

for $k \in \{m+1, 2m-2\}$
The bit representation bound $\frac{2n}{m} + \log_2(m+1)$ will be valid based on our previous discussions; the summation is symmetric.

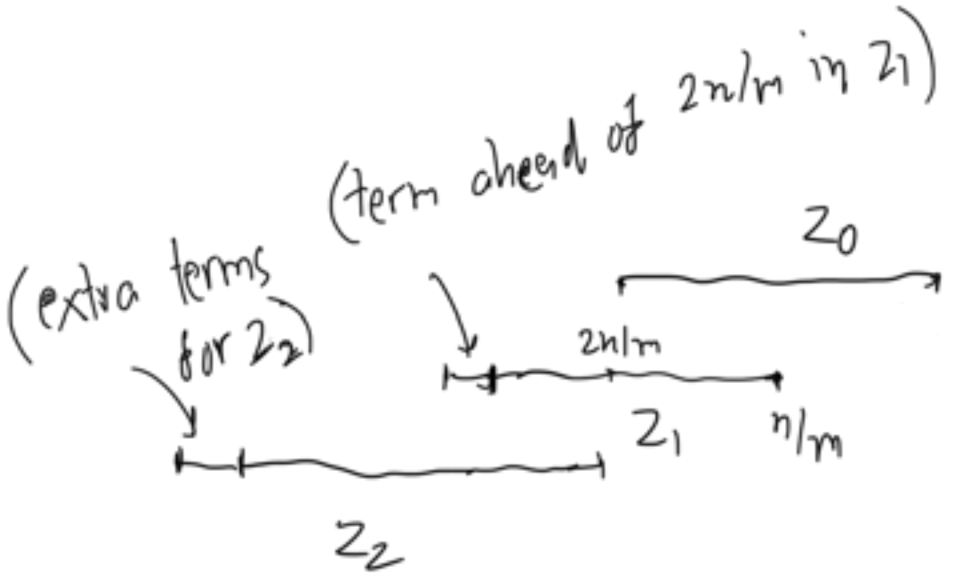
We place 2 in positions starting from 0

to the end of its bit representation.
We place z_1 from position $\frac{n}{m}$ to the end of its bit representation.

z_0 : $\frac{2n}{m}$ terms

z_1 : $\frac{2n}{m} + \log_2(1+1)$ terms

z_2 : $\frac{2n}{m} + \lceil \log_2(2+1) \rceil$ terms



If we are adding two binary numbers of length n_1, n_2, let $n_1 \geq n_2$, as a maximum, we get $2^{n_1}-1 + 2^{n_1}-1 = 2^{n_1+1}-2$
This can be represented in (n_1+1) bits.

As we iterate through the additive process, taking 2 consecutive z_i s and the corresponding carry overs, the number of operations in each addition will be proportional to the number of bits in concern,

$$A_i \sim \frac{2n}{m} + \log_2(i+1) + 1 \qquad 1 \le i \le m$$

term symmetric for higher i

$\sum_{i=0}^{2m-2} A_i$ is bounded by,

$$S \equiv \sum_{i=0}^{2m-2} \left(\frac{2n}{m} + \log_2(m+1) + 1 \right)$$

$$S \equiv \left(\frac{(2m-2)n}{m} + m\log_2(m+1) + m \right)$$

Using Asymptotic notation,

$$T(n,m) = \Theta(n + m\log m)$$

$$f(n,m) = \Theta(n + m\log m)$$

2.b

Since each z_i can be computed through

$O(m \log m)$ multiplications and $O(m \log m)$ additions over $k = n/m$ bit integers

$$T(n) = a_m \, T_m(n/b) + f(n)$$

In the naive method,

$\sum Z_i =$ We get $1\,T(n/m) + 2\,T(n/m) - m\,T(n/m)$

$$\propto m^2 \, T(n/m)$$

We can obtain a tigther bound given the current information.

We are given,

$$\sum_i T(z_i) = O(m \log m) \, T(n/m)$$

$$T(n) = \sum_i T(z_i) + d(n)$$

Where we know,

$$d(n) = f(n,m) = \theta(n + m\log m)$$

let,

$$T(n) = (2m-1) \log m \; T(n/m) + \theta(n + m\log m)$$

$$T(n) = 4m\,T(n/m) + O(n + m\log m)$$

for $n=2$,
We get

$$T(n) = 3T(n/2) + \theta(n)$$

which is for the Karatsuba multiplication algorithm

Also, $(2m-1)\log m \equiv O(m\log m)$

$$T(n) = (2m-1)\log m\, T(n/m) + \theta(n + m\log m)$$

Since there are $O(m\log m)$ additions and $O(m\log m)$ multiplications, we find something in the order of $2m\log m$ for our solution for A_m. Our big Oh notation will be valid.

2.c

$$T(n) = (2m-1)\log m\, T(n/m) + \theta(n + m\log m)$$

Based on master theorem,

$$f(n) = n + m\log m$$

$$n^{\log_b a} = n^{\log_m (2m-1)\log m}$$

$$\lim_{n \to \infty} \frac{n + m\log m}{n^{\log_m (2m-1)\log m}}$$

$$= \frac{1}{(\log_m (2m-1)\log m)\, n^{(\log_m (2m-1)\log m \; - 1)}}$$

If

$$\log_m (2m-1)\log m - 1 \geq 0$$

then $n^{\log_b a}$ term will dominate.

$$\log_m (2m-1)\log m \geq \log_m m$$

$$(2m-1)\log m \geq m$$

$$\log m \geq \frac{m}{2m-1}$$, this is solvable.

Using master theorem,

$$T(n) = n^{\log_m (2m-1)\log m}$$

for $n=2$,

$$T(n) = O(n^{\log_2 3})$$

Based on our question,

$$T(n) = n^{1+\epsilon}$$

$$1+\epsilon = \log_m((2m-1)\log m)$$

$$\epsilon = \log_m(2m-1) + \log_m \log m - \log_m m$$

$$\epsilon = \log_m\left(\frac{2m-1}{m}\right) + \log_m(\log m)$$

$$= L_1 + L_2$$

$$\lim_{m\to\infty} L_1 = \log_m\left(2-\frac{1}{m}\right) = 0$$

$$L_1 = \log_m(\log m)$$

$\log m = t \quad , m = e^t$

$$L_1 = \log_{e^t} t$$

$$= \frac{\log t}{\log e^t}$$

$$L_1 = \frac{\log t}{t}$$

$$\lim_{t \to \infty} L_1 = \frac{\frac{1}{t}}{1} = 0$$

$$\therefore \lim_{m \to \infty} \epsilon(m) = L_1 + L_2$$

$$= 0$$

2.d

$$X = X_0 + X_1 2^{n/m} \text{——} X_{m-1} 2^{(m-1)\frac{n}{m}}$$

$$Y = Y_0$$

$$Z = XY$$

$$= X_0 Y_0 + X_1 Y_0 2^{n/m} \text{ —— } X_{m-1} Y_0 2^{(m-1)\frac{n}{m}}$$

We apply Karatsuba individually to each of the m multiplications of k bit numbers

$\frac{n}{m} = k$, $m = \frac{n}{k}$

$$T_d(n) = c_0 \frac{n}{k} \cdot (k)^{\log_2 3}$$

$$= c_0\, n\, k^{\log_2 3 - 1}$$

for the divide step subproblems

The conquer step will remain the same as in the previous questions.

$$d(n) = c_1 m \log 3 + c_2 n$$

$$= c_1 \frac{n}{k} \log \frac{n}{k} + c_2 n$$

$$T(n) = c_0 n k^{\log_2 3 - 1} + c_1 \frac{n}{k} \log \frac{n}{k} + c_2 n$$

- If k is reasonable relative to n, this algorithm is decent.
- For naive Karatsuba, we have to do padding and we eliminate redundancies with such a formulation.

3.a

$A[1, \_\_\_, n]$
$A[i] \in \{0,1,2\}$

To prove: If (start, end) is a minimum span then $A[\text{start}, \_, \text{end}]$ has the form $a\, b^{\text{end-start}-1}\, c$ where (a,b,c) is a permutation of $(0,1,2)$

let a, c be the first and last elements of the minimum span. This is without any loss of generality.

- Consider the cases apart from our given configuration.

1. $a\, c^{x_1}\, b^{\text{end-start}+1}\, c$

2. $a\, b^{\text{end-start}+1}\, a^{x_2}\, c$

- In the first case, $a c^{x_1} b$ will be a span and is of the form $a\, b^{\text{end-start}+1}\, c$.
The original array (1) will not be a minimum span and the minimum span will be $a\, c^{x_1}\, b$

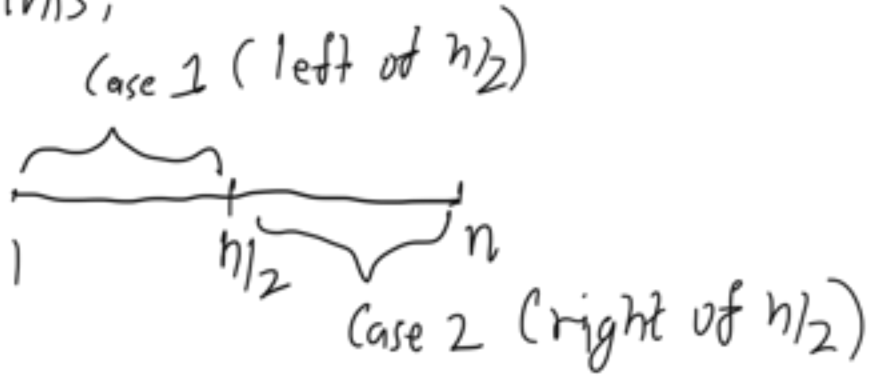
- In the second case, $b\,a^{x_2}c$ will be the minimum span and will be of the same structure as in the question.

- Thus, if there is a minimum span which starts with a and ends with c, it is of the form $a\,b^{\text{end}-\text{start}+1}\,c$.

3.b

To prove : If [start, end] is a minimum span, then either end $\leq \frac{n}{2}$ or start $> n/2$ or start $\leq n/2 <$ end

Consider this,



Case 3 - through $n/2$

The following options are exhaustive and cover all possibilities for the minimum span subarray.

1. The minimum span lies to the left of $n/2$ where start < end $\leq n/2$
2. The minimum span lies to the right of $n/2$

where $n/2 < start < end$.

3. The minium span passes through $n/2$ where $start \le n/2 < end$

If we pick any (start, end) combination from the array as candidate for our minimum span, we see that it must fall within one of these cases.

3.c

```
Find_Crossing_span (A[1, _ , n])
{   i = 1
    j = 1

    while (A[n/2 - i] == A[n/2])
    {
        i = i - 1
    }

    while (A[n/2 + j] == A[n/2])
    {
        j = j + 1
    }
```

```
if ( A[n/2 - i] != A[n/2 + j] )
    {
        return ( n/2 - i , n/2 + j )
    }

if ( A[n/2 - i] == A[n/2 + j] )
    {
        while ( A[n/2 + j] == A[n/2 - i]
                OR   A[n/2 + j] == A[n/2] )
            {
                j = j + 1
            }
        return ( n/2 , n/2 + j )
    }
}
```

- We start with $A[n/2]$.
- let $A[n/2]$ be b.
- Until we find some c different from b towards the right, we iterate forward.

- Until we find some a different from b towards the left, we iterate backward.
- If our c is different from a, we have our minimum span through $n/2$ of the form $a\, b^{x} c$ as discussed earlier.
- If c is the same as a, the elements for a, b are covered in the sub array starting with $A[n/2] = b$ and forward. The element for c is to be discovered to complete the minimum span and we iterate forward accordingly. We need not include the elements behind $A[n/2]$ in our miniumum span. As we are finding a crossing minimum span wherein we have a crossing condition start $\leq n/2 <$ end. We return the sub array from $n/2$ to the element index by which all three elements are covered as discussed.
- Our first condition satisfies the structure as proven, any element less from the left and we won't have a minimum span as all $\{0, 1, 2\}$ are not covered. Any extra element on the left, at the cost of reducing an element from the left, even if the sub array is a span, will add to the redundancy, since our configuration is a minimum span of the same cardinality.
- Similar arguments can be made for elements at the right hand side of the sub array.

• $T(n) = O(n)$

We start with $A(n/2)$, iterate backwards towards $A[1]$, iterate forwards towards $A[n]$, until respective conditions are met.

We iterate over the remaining array in the forward direction as pertaining to case 2.

∴ We get $O(n)$ time complexity as we make a pass over the array at most once.

3.d

```
Find_min_span (A[1,___,n])
{   if (length(A) <= 2)
                  { return (∞,∞) }

        (l1, r1) = Find_min_span (A[1,___,n/2])

        (l2, r2) = Find_min_span (A[n/2,___,n])

        (l3, r3) = Find_crossing_span (A[1,___,n])

    if (r1 - l1 + 1) < (r2 - l2 + 1)
        {
            if ( r1 - l1 + 1) < (r3 - l3 + 1)
              {
                return (l1, r1)
```

```
        }
    }
    if (r_2 - l_2 + 1) < (r_1 - l_1 + 1)
    {
        if (r_2 - l_2 + 1) < (r_3 - l_3 + 1)
        {
            return (l_2, r_2)
        }
    }

    return (l_3, r_3)

}
```

Since we have used strictly less than sign, this algorithm returns (∞, ∞) when no minimum span exists for $A[1, \ldots, n]$.

We must predefine $\infty - \infty + 1 \approx \infty$ as for our comparisions to work.

- We propose find_min_span $(A[1, \ldots, n])$ finds the minimum span in an array of size n.
- Consider the base case, when the array consists of 2 or less elements, we return (∞, ∞) as there is no minimum span, we have 3 unique elements $\{0, 1, 2\}$

- In the inductive hypothesis, we assume that we get correct solutions for $A[1, \_, \frac{n}{2}]$ and $A(n/2, \_, n]$.

- Based on problem 3.b, the minimum span for $A[1, \sim, n]$ will either be the minimum span of $A[1, -, \frac{n}{2}]$ or $A(n/2, \_, n]$

or the minimum crossing span.

- The smallest of these three spans will be our answer.
- We return the smallest of these three spans as our answer for $A[1, \_, n]$ and hence our proof by induction is valid.

3.e

Continuing from algorithm Find_min_span,

$$T(n) = 2T(n/2) + cn$$

$$S(k) = 2S(k-1) + c2^k$$

$$\# \; n = 2^k, \; T(n) = S(k)$$

$$\frac{S(k)}{} = \frac{S(k-1)}{} + c$$

2^k $\qquad\qquad$ 2^{k-1}

$$f(k) = f(k-1) + c$$

Telescoping series,

$$f(1) = \text{constant}$$

$$f(k) \equiv ck$$

$$\frac{S(k)}{2^k} \equiv ck$$

$$S(k) \equiv ck2^k$$

$$T(n) \equiv cn\log n$$

$$T(n) = \Theta(n\log n)$$

3.f

We make the sub problems return the following,

- The minimum spanning sub array: (a, b)

- The position from the start of the array until we get to a new element. (s_1)
- The position from the end of the array until we get to a new element. (e_1)
- returning $((a,b), s_1, e_1)$
- For example,

$$(1, 1, 2, 0) \quad \text{will return} \quad ((1,4), 2, 1)$$
$$\;\;1\;\;2\;\;3\;\;4$$

- We are here using the structure of the minimum spanning sub array.

- for $A[1, \_\_, n]$, the minimum spanning array will either lie in $A[1, \_\_, n/2]$ or $A[n/2, \_\_, n]$ or will have start $< n/2 <$ end

- Notice that we are using strictly greater than conditions here.

- What we do for the crossing problem is that we check for $A[1, \_, n/2]$, the number of steps to be taken until we are able to get a new element. $A[n/2 - x]$ will have a new element different from $A[n/2]$. x is returned by the sub problem. x is e_1 for $A[1, -, n/2]$

- Similarly, we check for y such that $A[n/2 + y]$ is different from $A[n/2]$. y is returned by the other subproblem. y is s_1 for $A[n/2, \_, n]$.

- if $A[n/2 - x]$ is not equal to $A[n/2 + y]$, the crossing subarray will be $(n/2 - x, n/2 + y)$ with cardinality $x + y - 1$.
- This operation can be done in constant time given the return values of our subproblem.

- If $A[n/2 - x]$ is equal to $A[n/2 + y]$, the crossing sub array will not be the minimum sub array of our original array. Any crossing sub array which is a spanning array will have extra redundancies with respect to a minimum spanning array in the two sub arrays.

- The answer to our question regarding the minimum spanning array will therefore be among the answers for our two sub arrays.
- We return (∞, ∞) in this case for the crossing problem as it is not determining our final answer.

```
Find_min_span (A[1,___,n])
{
      flag = 0
      s = ∞, e = ∞
      if (length (A) <= 2)
       {
         (l_min, r_min) = (∞, ∞)
```

```
flag = 1

if (A[0] != A[1])
    {   s = 1
        e = 1
    }
  return ((lmin, rmin), s, e))
}

((l1, r1), s1, e1) = Find_min_span
                        (A[1, — n/2])

((l2, r2), s2, e2) = Find_min_span
                        (A[n/2, —, n])

if (A[n/2 - e1] != A[n/2 + s2])
   {
      (l3, r3) = (n/2 - e1, n/2 + s2)

   }
else
   {  (l3, r3) = (∞, ∞)
   }

if (s1 == ∞)
   {   if (A[1] == A(n/2 + 1)) {
              s = length(A[1, —, n/2])
                     + s2
       }
```

}

else { s = s_1 }

if (e_2 == ∞) {
    if (A[n/2] == A[n]) {
        e = e_1 + length (A[1, -, n/2])
    }
}
else { e = e_2 }

if (s_1 == ∞ and s_2 == ∞) {
    if (A [1] != A[n]) {
        s = length (A [1, -, n/2]) + 1
        e = 1 + length (A [n/2, - n]
}

(l_min, r_min) = min ((l_1, r_1) , (l_2, r_2) , (l_3, r_3))

return ((l_min, r_min) , s, e)

}

We see that,

$$T(n) = 2T(n/2) + O(1)$$

$\because$ Using master theorem,

$$T(n) = O(n)$$

We hence get a divide and conquer algorithm with linear time complexity for the minimum spanning array problem.

EOF

- as14634
- Advait Savant