Soham Chatterjee
Email: soham.chatterjee@tifr.res.in
Course: Algorithms

Assignment - 2
Dept: STCS
Date: September 28, 2024

Problem 1 P2 (10 marks)

Show that $n-1$ comparisons are necessary and sufficient to find the minimum element in an unsorted array of n elements.

Solution: Suppose there is an algorithm which can find minimum element with less than $n-1$ comparisons. Let $A=(a_1,\dots,a_n)$ be the unsorted array of n elements. Create a graph of vertices $[n]$ and put an edge between i and j if the elements a_i and a_j are compared. Since there are $, n-1$ comparisons in the graph we constructed there are at most $n-2$ edges. Hence the graph has at least 2 connected components. Let C_1 and C_2 be two connected components. No two elements one from each component is compared. Let the concluded minimal element from C_1 is x and the concluded minimal element from C_2 is y. Hence we can make any of them bigger than the other depending on the algorithm output to make the algorithm wrong. Hence at least $n-1$ comparisons are necessary to find the minimum element.

Now we can also find the minimum element of an unsorted array of n elements with $n-1$ comparisons by the following algorithm:

Algorithm 1: Find-Minimum

```
  Input: Array $A=(a_1,\dots,a_n)$
  Output: Minimum element of $A$
1 begin
2     $MinVal \longleftarrow A[1]$
3     for $i=2,\dots,n$ do
4         if $MinVal > A[i]$ then
5             $MinVal \longleftarrow A[i]$
6     return $MinVal$
```

This algorithm uses only $n-1$ comparisons to find the minimum element of the array. Therefore $n-1$ comparisons are sufficient to find the minimum element in an unsorted array of n elements. ■

Problem 2 P3 (15 marks)

Show a comparison based algorithm for finding the minimum and maximum in an unsorted array of n elements using $\left\lfloor\frac{3n}{2}\right\rfloor-2$ comparisons. Also show that $\left\lfloor\frac{3n}{2}\right\rfloor-2$ comparisons are necessary to find the minimum and maximum.

Solution: First assume that all the elements are distinct. We will show the algorithm with which you only need to do $\left\lceil\frac{3n}{2}\right\rceil-2$ comparisons.

So first we partition the elements into pairs. If n is odd then the last element will not in any groups. Hence there are total $\left\lfloor\frac{n}{2}\right\rfloor$ pairs. Now for each pair we compare then. Hence we have done now $\left\lfloor\frac{n}{2}\right\rfloor$ many comparisons. Now in each pair we have an element which was lesser than the other. So from each pair we take the lesser element and make a new set S. So S has $\left\lfloor\frac{n}{2}\right\rfloor$ elements. And from each pair we take the greater element and make another new set T. If n is odd then the last element which didn't take part in the pairs is added to both the sets S and T. So S and T both have now $\left\lceil\frac{n}{2}\right\rceil$ elements.

Now since S has the lesser elements the minimum element of the array is in S. And since T has the greater elements, the maximum element of the array is in T. So by the Problem 1: P2 we have an algorithm to find out the minimum element of S using $\left\lceil\frac{n}{2}\right\rceil-1$ comparisons. And since finding maximum element of array is very similar as finding minimum element using the same algorithm but using less than instead of greater than we get the maximum element of T using $\left\lceil\frac{n}{2}\right\rceil-1$ comparisons. Hence now we have the minimum and the maximum element of the array.

```
Algorithm 2: FIND-MINIMUM-MAXIMUM($A$)
  Input: Array $A = (a_1, \dots, a_n)$
  Output: Minimum and Maximum element of $A$
  begin
      $n \longleftarrow A.length$
      if $n == 1$ then
          return $(A[1], A[1])$
      $k \longleftarrow \left\lfloor \frac{n}{2} \right\rfloor$, $S \longleftarrow \emptyset$, $T \longleftarrow \emptyset$
      for $i = 1, \dots, k$ do
          if $A[2i-1] < A[2i]$ then
              $S \longleftarrow S \cup \{A[2i-1]\}$, $T \longleftarrow T \cup \{A[2i]\}$
          else
              $S \longleftarrow S \cup \{A[2i]\}$, $T \longleftarrow T \cup \{A[2i-1]\}$
      if $n$ mod $2 == 1$ then
          $S \longleftarrow S \cup \{A[n]\}$, $T \longleftarrow T \cup \{A[n]\}$
      $Minval \longleftarrow \infty$, $Maxval \longleftarrow -\infty$
      for $i \in S$ do
          if $i = 1$ then
              $Minval \longleftarrow S[1]$
          else if $Minval > S[i]$ then
              $Minval \longleftarrow S[i]$
      for $j \in T$ do
          if $j = 1$ then
              $Maxval \longleftarrow T[1]$
          else if $Maxval < T[i]$ then
              $Maxval \longleftarrow T[i]$
      return $(Minval, Maxval)$
```

Hence total comparisons needed to find the minimum and maximum element is

$$\left\lfloor \frac{n}{2} \right\rfloor + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

Hence this algorithm can find out the minimum and maximum using $\left\lceil \frac{3n}{2} \right\rceil - 2$ comparisons. Now we need to prove that at least this many comparisons are needed.

Now we work as an adversary to the algorithm and will force the algorithm to do at least $\left\lceil \frac{3n}{2} \right\rceil - 2$ many comparisons. So for each element in the array we keep the tags M and m. The M tag represents that the element can be the maximum element of the array and the m tag represents the element can be the minimum element of the array. Now on each comparison of two elements the algorithm does we remove tags for example if two elements both having a M tag and a m tag is compared then the smaller one can not be maximum so the M tag from the smaller one is removed and the larger one can not be the minimum so the m tag from the larger one is removed.

Initially before the algorithm starts all elements have one M tag and one m tag. After the algorithm successfully find the maximum element and the minimum element there is only one M tag on some element and only one m tag on some element of the array. So initially there are $2n$ tags and at the end there are 2 tags. Hence $2n - 2$ tags have to be removed in order to find the maximum and minimum of the array.

Now at any point during the execution of the algorithm if there is an element with both the M and m tags then this element has not been compared to other element because otherwise if its compared then either it becomes the larger and then the m tag is removed or it becomes the smaller and then the M tag is removed. Now consider the following cases:

- Two (M,m) tagged elements are compared: Then the adversary makes one of the larger and other one smaller randomly.
- If (M,m) tagged element, a is compared with M tagged element, b: a has not been compared to any one. So the adversary simply makes b larger. So only the M tag from a is removed.
- If (M,m) tagged element, a is compared with m tagged element, b: a has not been compared to any one. So the adversary simply makes b smaller. So only the m tag from a is removed.
- If M tagged element, a is compared with M tagged element, b: Then the whatever the adversary makes larger only one M tag will be removed.
- If m tagged element, a is compared with m tagged element, b: Then similarly like the above case whatever the adversary makes smaller only one m tag will be removed.
- If M tagged element, a is compared with an element b with no tag: Then if a becomes smaller then M tag of a is removed and in the other case no tag is removed.
- If m tagged element, a is compared with an element b with no tag: Then if a becomes larger then m tag of a is removed and in the other case no tag is removed.

Apart from the first case in all the above case at most 1 tag is getting removed. For first case 2 tags are removed. But at most $\left\lfloor \frac{n}{2} \right\rfloor$ many such comparisons possible. So the only case remaining is when a M tagged element, a is compared with a m tagged element b. We want to claim if the adversary makes the a larger than b then no contradictions will arise.

We want to argue for this using the graphs. We have n vertices which are basically the elements of the array. So for each comparison if a_i and a_j are compared we put a directed edge $a_i \longrightarrow a_j \iff a_i > a_j$. This way we construct the graph $G = (V,E)$. Let $G^{(t)} = (V, E^{(t)})$ be the graph constructed by the comparisons which happened before the t^{th} comparison in the algorithm. So in $G^{(0)}$, $E^{(0)} = \emptyset$. Now we know for any t, $G^{(t)}$ is a directed acyclic graph. So suppose at t^{th} comparison an element a with tag M is compared with an element b with tag m. Consider the weakly connected components of $G^{(t)}$. A weakly connected component is a subset of vertices such that for any two vertices u,v in that set either there is a path from $u \rightsquigarrow v$ or there is a path from $v \rightsquigarrow u$.

If a and b are in same weakly connected component then there is a path from $a \rightsquigarrow b$ or a path from $b \rightsquigarrow a$. If there is a path from $a \rightsquigarrow b$. Then naturally we conclude that $a > b$. So assume there is a path from $b \rightsquigarrow a$. In that case let the last edge of the path is $b' \longrightarrow a$. Since the edge $b' \longrightarrow a \in E^{(t)}$ they were compared before the t^{th} comparison and in that comparison a was smaller than b'. Then the M tag of a should have been removed but a still has the tag. Contradiction. If a and b are in same weakly connected component then there is a path from $a \rightsquigarrow b$ and therefore $a > b$.

Now assume a and b are in different weakly connected component. Then there is no path from $a \rightsquigarrow b$ or $b \rightsquigarrow a$. Hence the adversary can make a larger than b and there will not be any contradictions. Cause making $a > b$ only affects the vertices which are reachable from a and b. But a is not reachable from the vertices reachable from b and similarly b is not reachable from the vertices reachable from a. Therefore the adversary can make a larger than b and then no tags will be removed.

So now we shown that in at most $\left\lfloor \frac{n}{2} \right\rfloor$ many comparisons $2\left\lfloor \frac{n}{2} \right\rfloor$ tags will be removed. So the for the remaining at least $2n - 2\left\lfloor \frac{n}{2} \right\rfloor$ tags for rest of the comparisons in each comparison at most 1 tag is removed. And at the end 2 tags are remaining. So the number of comparisons needed is:

$$\left[(2n-2) - 2\left\lfloor \frac{n}{2} \right\rfloor\right] + \left\lfloor \frac{n}{2} \right\rfloor = 2n - \left\lfloor \frac{n}{2} \right\rfloor - 2$$

We will show that $2n - \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{3n}{2} \right\rceil$. If $n = 2k$ for some $k \in \mathbb{N}$ then $2n - \left\lfloor \frac{n}{2} \right\rfloor = 4k - k = 3k = \left\lceil \frac{3n}{2} \right\rceil$. If $n = 2k+1$ for some $k \in \mathbb{N}$ then $2n - \left\lfloor \frac{n}{2} \right\rfloor = 4k + 2 - k = 3k + 2$ and on the other hand $\left\lceil \frac{3n}{2} \right\rceil = \left\lceil \frac{6k+3}{2} \right\rceil = \left\lceil 3k + \frac{3}{2} \right\rceil = 3k+2$. So we have $2n - \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{3n}{2} \right\rceil$. Hence at least $\left\lceil \frac{3n}{2} \right\rceil - 2$ many comparisons are needed for any algorithm to find out the minimum and maximum element of an unsorted array of n elements. ■

[I discussed the solution with Soumyadeep]

Problem 3 P4 (10 marks)

Let $G = (V, E)$ be a directed acyclic graph $G = (V, E)$. Additionally, you are given a nonnegative, integral weight w_e on each edge $e \in E$, and two special vertices $s, t \in V$. Give an algorithm to find a max-weight path from s to t.

Solution: We can solve this problem using dynamic programing. Let $dist(v)$ denotes the weight of maximum weight path from $s \rightsquigarrow v$ for any $v \in V$. $dist(s) = 0$. Now for any $v \in V$ we have the following relation:

$$dist(v) = \max_{u:(u,v)\in E} \{dist(u) + w(u,v)\}$$

Hence we have the following algorithm:

Algorithm 3: FIND-MAX-PATH(G, s, t, W)

```
Input: A directec acyclic graph $G = (V, E)$ with 2 vertices $s, t$ and weights on edges $W$.
Output: Maximum weight path from $s \rightsquigarrow t$
begin
    $dist(s) \longleftarrow 0$, $s.parent \longleftarrow$ NULL
    for $v \in V - \{s\}$ do
        $dist(v) \longleftarrow -\infty$
        $v.parent \longleftarrow$ NULL
    $U \longleftarrow \{s\}$
    while $U \neq \emptyset$ do
        $u \longleftarrow$ Extract first element of $U$
        for $(u, v) \in E$ do
            if $\textsc{MaxPath}[v] \leq \textsc{MaxPath}[u] + w(u,v)$ then
                $\textsc{MaxPath}[v] \longleftarrow \textsc{MaxPath}[u] + w(u,v)$
                $v.parent \longleftarrow u$
            $U \longleftarrow U \cup \{v\}$
    $P \longleftarrow \emptyset$, $p \longleftarrow t$
    while TRUE do
        $P \longleftarrow P \cup \{p\}$
        if $p.parent ==$ NULL then
            BREAK
        $p \longleftarrow p.parent$
    $P \longleftarrow$ REVERSE$(P)$
    return $P$
```

Time Complexity: The lines 2-3 takes $O(n)$ time and line 5 takes constant time. The while loop at line 7 picks a vertex and then goes through all its neighbors and adds them to U. So the while loop visits at most every vertex and and every edges. In each iteration it takes constant time. Therefore the while loop takes $O(|V| + |E|) = O(n^2)$ time. Now for the while loop at line 14 it can be at most n many iterations and in each iteration it takes constant time. The REVERSE function also take linear time. Therefore the algorithm runs in $O(n^2)$ time or $O(|V| + |E|)$ time. ■

Problem 4 P5 (15 marks)

Given a matroid $(S,\mathcal{I})$, show that $(S,\mathcal{I}')$ is also a matroid, where $A \in \mathcal{I}'$ if $S \setminus A$ contains a maximal independent in $\mathcal{I}$.

Solution:

① **Downward Closure:** Let $A \in \mathcal{I}'$ and $B \subseteq A \implies S \setminus A \subseteq S \setminus B$. Since $A \in \mathcal{I}'$, $\exists\ X \in \mathcal{I}$ such that X is a maximal independent set such that $X \subseteq S \setminus A$. Therefore we have $X \subseteq S \setminus A \subseteq S \setminus B$. Therefore $B \in \mathcal{I}'$. Hence $\mathcal{I}'$ follows the downward closure property.

② **Exchange Property:** Let $A, B \in \mathcal{I}'$ and $|B| < |A|$. Let $X, Y \in \mathcal{I}$ be maximal independent sets such that $X \subseteq S \setminus A$ and $Y \subseteq S \setminus B$. We have $|X| = |Y|$. Now consider the set $X \setminus B \in \mathcal{I}$ as $X \in \mathcal{I}$. We can extend $X \setminus B$ to a maximal independent set Z by adding elements from Y not in $X \setminus Y$. Then let $Z = (X \setminus B) \sqcup T$ where $T \subseteq Y \setminus (X \setminus B) = Y \setminus X$.

First of all we have $(X \setminus B) \cap (A - B) = \emptyset$ since $X \cap A = \emptyset$. Also we have

$$X \cap B \subseteq (S \setminus A) \cap B = B \setminus A$$

Now we want to claim $A \setminus B \not\subseteq T$ because if it is true then $\exists\ e \in A \setminus B$ such that $e \notin T$. Then $Z \subseteq S \setminus (B \cup \{e\})$. So suppose $A \setminus B \subseteq T$. Then $|T| \geq |A \setminus B|$. Now

$$|B \setminus A| = |B| - |A \cap B| < |A| - |A \cap B| = |A \setminus B|$$

Hence

$$|T| \geq |A \setminus B| > |B - A|$$

Now we have

$$|Z| = |X \setminus B| + |T| > |X \setminus B| + |B \setminus A| \geq |X \setminus B| + |X \cap B| = |X|$$

Therefore we have $|Z| > |X|$ but both are maximal independent sets of $(S,\mathcal{I})$ and their size should be equal. Hence contradiction. Therefore $A \setminus B \not\subseteq T$.

Therefore $\exists\ e \in A \setminus B$ such that $e \notin T$. Then $e \notin Z \implies Z \subseteq S \setminus B \cup \{x\}$. Therefore $B \cup \{x\} \in \mathcal{I}'$. Hence $(S,\mathcal{I}')$ follows the exchange property.

Therefore $(S,\mathcal{I}')$ is a matroid. ■

Problem 5 P6 (15 marks)

In class, we showed that if $(S,\mathcal{I})$ is a matroid, then for any nonnegative weights w no the elements of S, the greedy algorithm obtains a maximum weight independent set. Show that this is only true if $(S,\mathcal{I})$ a matroid. That is, for a fixed downward-closed set system $(S,\mathcal{I})$, if the greedy algorithm obtains a maximum weight element of $\mathcal{I}$ for every assignment of nonnegative weights to elements of S, then $(S,\mathcal{I})$ is a matroid.

Solution: Here we only have to prove that $(S,\mathcal{I})$ follows the exchange property. So let $A, B \in \mathcal{I}$ and $|B| < |A|$. So we create a weight function W. Let $|S| = n$. Then $\forall\ b \in B$ we have $W(b) = \frac{1}{n}$. And $\forall\ a \in A - B$, we have $W(a) = \frac{1}{n+1}$. Now

$$W(B) \leq \frac{|B|}{n}$$

Now suppose $|A| = k$, $|B| = l$ and $|A - B| = m$ then we have $n \geq k > l$ and $m > 0$ and $|A \cap B| = k - m \leq l$. Hence

$$W(A) = \frac{k-m}{n} + \frac{m}{n+1} = \frac{(k-m)(n+1) + nm}{n(n+1)} = \frac{(n+1)k - m}{n(n+1)}$$

Now the algorithm sorts the elements by their weight and after sorting firstly the algorithm will encounter the elements of B and then the elements of $A - B$. Since $B \in \mathcal{I}$ the algorithm will pick all the elements of B and construct the set B when it starts to encounter the elements of $A - B$. Now the algorithm picks any element from $A - B$ if there is already a set $S \in \mathcal{I}$ such that $W(S) > W(B)$. Now we claim $W(A) > W(B)$.

$$W(A) = \frac{(n+1)k - m}{n(n+1)} > \frac{l}{n} = W(B) \iff \frac{(n+1)k - m}{n+1} > l \iff (n+1)k - m > l(n+1) \iff (n+1)(k-l) > m$$

The last inequality is true since $k > l$ and $m \leq n$. Hence we have a set which has weight greater than the weight of B. So the algorithm will pick an element x from $A - B$ such that $B \cup \{x\} \in \mathcal{I}$.

Hence $(S, \mathcal{I})$ follows the base exchange property. Therefore $(S, \mathcal{I})$ is a matroid. ■

Problem 6 P7: Exercise 10.4-6 (on tree representations with pointers) from CLRS. (10 marks)

The *left-child*, *right-sibling* representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

Solution: Fir each node in the tree the we will keep a boolean value called *isRightSibling*. If value of *isRightSibling* is 1 then there is a right sibling of the current node and there is a *right-sibling* pointer of the node and the node has not *parent* pointer pointing towards its parent. If value of *isRightSibling* is 0 then there is not *right-sibling* pointer of the current node. Instead the node has a *parent* pointer pointing towards its parent.

So the following are two algorithms for going to a parent of a given node and going to a given specific child of a given node.

Algorithm 4: PARENT($node$)

```
1 begin
2     while TRUE do
3         if $node.isRightSibling == 1$ then
4             $node \longleftarrow node.right - sibling$
5         else
6             return $node.parent$
```

Algorithm 5: CHILD($node$)

```
1 begin
2     CHILDREN $\longleftarrow \emptyset$
3     CHILDREN $\longleftarrow$ CHILDREN $\cup \{node.left - child\}$
4     while $node.isRightSibling == 1$ do
5         CHILDREN $\longleftarrow$ CHILDREN $\cup \{node.right - sibling\}$
6         $node \longleftarrow node.right - sibling$
7     return CHILDREN
```

The algorithm for finding the parent goes through all the children of the parent of the given node to go to the right most sibling of the given node then returns the parent to the node. So it takes linear time in the number of children. The algorithm for finding all the children it goes to the left child of the given node and then goes through all the right siblings of the children one by one and then returns the array of all the children. Hence this algorithm also reaches all the children of the given node in linear time in the number of children.

■

Problem 7 P8 (10 marks)

Given a directed graph $G = (V, E)$ with weights on the edges, and which has a negative-weight directed cycle that is reachable from the source s, Give an efficient algorithm to list the vertices of such a cycle.

Solution: If the graph if there is a negative-weight cycle reachable from s then there is a vertex j reachable from s such that the shortest path from $j \rightsquigarrow j$ has negative weight. So we will use the Floyd Warshal Algorithm. But we will also keep track of the paths. We are assuming that G is given as an adjacency matrix.

Here we introduce a $[n+1]\times[n]\times[n]$ $D = \left(d_{i,j}^{(k)}\right)$ where $k \in [n+1]$. So $d_{i,j}^{(k)}$ is the weight of the shortest path from i to j using the vertices $[k]$. Naturally we have the relation:

$$d_{i,j}^{(k+1)} = \min\left\{d_{i,k+1}^{(k)} + d_{k+1,j}^{(k)},\ d_{i,j}^{(k)}\right\}$$

Here we also need to keep the path. So when the minimum path uses the $k+1$ vertex then we just concatenate the paths from $i \rightsquigarrow k+1 \rightsquigarrow j$ and otherwise the path is same as it was not using the vertex $k+1$. So here is the algorithm:

Algorithm 6: FIND-NEGATIVE-CYCLE(A, s, W)

```
Input: A directec graph $G = (V = [n], E)$ with source vertex $s$ and weights on edges $W$ with promise that
       $G$ has a negative-weighted cycle reachable from $s$
Output: Find a negative-weighted cycle reachable from $s$
1  begin
2      Create a $[n+1]\times[n]\times[n]$ array $D = \left(d_{i,j}^{(k)}\right)$ for all $k \in [n+1]$, $i,j \in [n]$ with all entries $(\infty, \text{NULL})$
3      for $i,j \in [n]$ do
4          $d_{i,j}^{(0)} = \left(A[i,j], [i,j]\right)$
5      for $k = 1,\dots,n$ do
6          for $i = 1,\dots,n$ do
7              for $j = 1,\dots,n$ do
8                  if $d_{i,j}^{(k-1)}[1] > d_{i,k}^{(k-1)}[1] + d_{k,j}^{(k-1)}[1]$ then
9                      $d_{i,j}^{(k)}[2] \longleftarrow d_{i,k}^{(k-1)}[2]\ ++\ d_{k,j}^{(k-1)}[2]$
10                 else
11                     $d_{i,j}^{(k)}[2] \longleftarrow d_{i,j}^{(k-1)}[2]$
12     $Visited \longleftarrow \{s\}$, $U \longleftarrow \{s\}$
13     Create a $n$ length array $VIS$ with all entries 0
14     $VIS[s] = 1$
15     while $U \neq \emptyset$ do
16         $u \longleftarrow$ Extract first element of $U$
17         for $(u,v) \in E$ do
18             if $VIS[v] == 0$ then
19                 $U \longleftarrow U \cup \{v\}$
20                 $Visited \longleftarrow Visited \cup \{v\}$
21                 $VIS[v] \longleftarrow 1$
22     for $v \in Visited$ do
23         if $d_{v,v}^{(n)}[1] < 0$ then
24             return $d_{i,i}^{(n)}[2]$
```

Time Complexity: The line 2 takes $O(n^3)$ time to create the array. In line 3 the for loop has n^2 iterations where each iteration takes constant time. In line 5-7 it takes total n^3 iterations and each iteration takes constant time. So the for loops in line 5-7 in total takes $O(n^3)$ time (We are assuming concatenating two lists take constant time. So in line 9 it takes constant time). In line 12-14 it takes linear time. In line 15 the while loop and the for loop at line 17 goes through each vertex and then also goes through each neighbor of it. Hence there are total $O(|V|+|E|) = O(n^2)$ iterations each of which takes constant time. In line 22 the for loop has $O(n)$ many iterations at most and each iteration it takes constant time. Hence the algorithm runs in $O(n^3)$ time. ■

Problem 8 P9 (15 marks)

Let us modify the "cut rule" (in the implementation of decrease-key operation for a Fibonacci heap) to cut a node x from its parent as soon as it loses its 3rd child. Recall that the rule that we studied in class was when a node loses its 2nd child. Can we still upper bound the maximum degree of a node of an n–node Fibonacci heap with $O(\log n)$.

Solution: To have the maximum degree bound first we prove two lemmas:

Lemma 1. *Let x be a node in the Fibonacci heap. Let $y_1, \dots, y_k$ be children of x in the order which they were added. Then $y_1.degree \geq 0$, $y_2.degree \geq 1$ and $y_i.degree \geq i-3$ for $i = 3, \dots, k$*

Proof: Degree of any node is at least 0. So we already get $y_1.degree \geq 0$. Now the children are indexed in the order which they were added. So when y_1 was added to x *degree* of x was 0. So after adding y_1 *degree* of x became 1. Now when y_2 was added $x.degree \geq 1$ and $y_2.degree = x.degree$. So $y.degree \geq 1$.

For $i \geq 3$ when y_i was linked to x then $y_1, \dots, y_{i-1}$ were children of x. So at that time $x.degree \geq i-1$. Since y_i was linked to x we must have $y_i.degree = x.degree$. So at that time we must have $y_i.degree \geq i-1$. Since then because of the Cascading-Cut operation y_i can loose at most 2 children and still remain as the child of x. So $y_i.degree \geq i-3$. Therefore for all $i = 3, \dots, k$ we have $y_i.degree \geq i-3$ and $y_1.degree \geq 0$, $y_2.degree \geq 1$. □

Lemma 2. *Consider the sequence of numbers $\{a_n\}_{n\geq 0}$ where*

$$a_0 = 0,\ a_1 = 1,\ a_2 = 1,\ a_n = 1 + \sum_{i=3}^{n} a_{i-3}$$

Then we have the following:

① $a_n = a_{n-1} + a_{n-3}$ *for* $n \geq 3$

② *The equation* $x^3 - x^2 - 1 = 0$ *has a real root* λ *which is greater than* 1

③ $a_{n+3} \geq \lambda^n$ *for all* $n = \mathbb{N} \cup \{0\}$

Proof:

① We have

$$a_n = 1 + \sum_{i=3}^{n} a_{i-3} = \left[1 + \sum_{i=3}^{n-1} a_{i-3}\right] + a_{n-3} = a_{n-1} + a_{n-3}$$

Therefore we obtain the given recurrence relation for all $n \geq 3$.

② Value of the polynomial $p(x) = x^3 - x^2 - 1$ at $x = 1$ is $p(1) = 1 - 1 - 1 < 0$. And $p(2) = 8 - 4 - 1 > 0$. Hence $p(x)$ has a real root between 1 and 2. Therefore $p(x)$ has a real root λ which is greater than 1.

③ We will prove this by induction. For $n = 0$ we have $\lambda^0 = 1$ and $a_3 = 1 + a_0 = 1 \geq \lambda^0$. Hence the base case is followed. Suppose for $n = 0, \dots, i-1$ we have $a_{n+3} \geq \lambda^n$. For $n = i$

$$a_{i+3} = a_{i+2} + a_i \geq \lambda^{i-1} + \lambda^{i-3} = \lambda^{i-3}(\lambda^2 + 1) = \lambda^{i-3}\lambda^3 = \lambda^i$$

Hence by mathematical induction we have $a_{n+3} \geq \lambda^n$ for all $n \geq 0$

□

Now using these two lemmas we will prove a upper bound the size of a node with degree k which we can use to bound the maximum degree since the number of nodes is n and degree can not exceed the total number of nodes.

Lemma 3. *Suppose x be a node in a Fibonacci heap with $k = x.degree$. Then $x.size \geq a_{k+3} \geq \lambda^k$*

Proof: Let s_k denote the minimum possible size of any node with degree k. Now we have $s_0 = 1$, $s_1 = 2$, $s_3 = 3$. So we have $x.size \geq s_k$. Suppose $y_1, \dots, y_k$ be the children of x in the order which they were linked to x. Now to bound s_k we count one for the root and one each for the first two children of the root and then we have the sizes for the other children. So we have

$$x.size \geq s_k \geq 3 + \sum_{i=3}^{k} s_{y_i.degree} \geq 3 + \sum_{i=3}^{k} s_{i-3}$$

Now we will inductively show that $s_k \geq a_{k+3}$. For base case using Lemma 1 we have $s_0 = 1 = a_3$, $s_1 = 2 = a_3 + a_1 = a_4$, $s_2 = 3 = a_4 + a_2 = a_5$. Therefore for $k = 0, 1, 2$ we have $s_k = a_{k+3}$. So the base case is followed. Now suppose this is true for $k = 0, \dots, i-1$. For $k = i$ we have

$$s_i \geq 3 + \sum_{j=3}^{i} s_{j-3} \geq 3 + \sum_{j=3}^{i} a_j = 3 + \sum_{j=0}^{i} a_j - \sum_{j=0}^{2} a_j = 3 + \sum_{j=0}^{i} a_j - 2 = 1 + \sum_{j=0}^{i} a_j = a_{i+3}$$

Therefore we have $s_i \geq a_{i+3}$. Hence by mathematical induction we have $s_k \geq a_{k+3}$ and then using Lemma 2 we have $s_k \geq \lambda^k$ for all $k \in \mathbb{N} \cup \{0\}$. □

Now we will bound the maximum degree. Let x be a node in the Fibonacci heap. Let $k = x.degree$. Then by Lemma 3 we have $k \geq \lambda^k$. Let there are total n nodes in the Fibonacci heap. Therefore we have

$$n \geq k \geq \lambda^k \implies \lg_\lambda n \geq \log_\lambda \lambda^k \implies k \leq \log_\lambda n = \log \lambda \cdot \log n = O(\log n)$$

Hence the maximum degree is bounded by $O(\log n)$. ■

Problem 9 P10 (15 marks)

The following are Fibonacci-heap operations: *extract-min*$(\cdot)$, *decrease-key*$(\cdot,\cdot)$, and also *create-node*(x,k) which creates a node x in th root list with key value k. Show a sequence of these operations that results in a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

Solution: We construct a Fibonacci heap consisting of just one tree that is a linear chain of n nodes iteratively. For $n = 1$ we just do a *create-node*(x,k) operation and then we have a Fibonacci heap with just one node. Now we will discuss for $n > 1$ case. At i^{th} iteration we create a linear chain with i nodes for any $i \in [n-1]$. Let $x^{(i+1)}$ denote the root of the linear chain with $i+1$ nodes after i^{th} iteration and $k^{(i+1)}$ denote the value of the key of the root $x^{(i+1)}$. With this set up we start the process of creating linear chain with n nodes:

- **Initialization:** At $i = 1$ we have the empty heap. So we will create a new heap $\mathcal{H}$ by inserting 3 nodes with *create-node*(a, k_a), *create-node*(b, k_b), *create-node*(c, k_c) with the property that $k_a > k_b > k_c$. So we have the min pointer on k_c. Now we do the operation *extract-min*$(\mathcal{H})$. The *extract-min* operation will then remove the node c. And since now there are 2 nodes with degree 0, the CONSOLIDATE operation add the two nodes a, b together with b being the root $x^{(2)}$ with the value $k^{(2)} = k_b$ since $k_b < k_a$ and the degree of b becomes 1. Hence now we have a Fibonacci heap with 2 nodes which is a linear chain. So the operations used in this step are:
  - 3 times *create-node*
  - *extract-min*
- **Iteration:** So now assume we have a Fibonacci heap which is a linear chain of $i+1$ nodes. The root of the chain is $x^{(i+1)}$ and the value of the key of $x^{(i+1)}$ is $k^{(i+1)}$. We will now show the process of creating a linear chain of $i+2$ nodes.

  First we create three nodes with at least two of them have the key value less than $k^{(i+1)}$, $i \in [n-1]$. So we do *create-node*(α, k_α), *create-node*(β, k_β), *create-node*(γ, k_γ) with $k_\alpha < k_\beta < k_\gamma$ and suppose $k_\alpha, k_\beta < k^{(i+1)}$.

So now the min pointer is on the node α. Now we run the operation *extract-min*$(\mathcal{H})$. This operation then removes the node k_α. Now β and γ are two nodes with degree 0. The Consolidate operation then add the two nodes β and γ together with γ being the child of β and then degree of β becomes 1. Now we have two roots, $x^{(i+1)}$ and β both having degree 1. So the Consolidate operation links $x^{(i+1)}$ and β with $x^{(i+1)}$ being a child of β since initially we took $k_\beta < k^{(i+1)}$. So now degree of β is 2 and the two child of β are $x^{(i+1)}$ and γ. So now we run the operation *decrease-key*$(\gamma, k_\beta - 1)$. So now key of γ becomes less than key of β. So γ is now added to the root list and degree of β becomes 1. Since γ has the minimum key the min pointer now points to gm. So now we do an *extract-min*$(\mathcal{H})$ operation. This will remove the γ from the root list and now we have a new linear chain with $i + 2$ node where the root $x^{(i+2)} = \beta$ and the key is $k^{(i+2)} = k_\beta$. So the operations used in one iterations are:

- 3 times *create-node*
- *extract-min*
- *decrease-key*
- *extract-min*

So we do the iteration process $n - 1$ many times to have a linear chain with n nodes. ■