ALGORITHMS - I

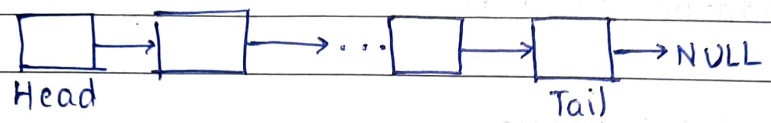
JASVIN JAMES
MT2021059

10/08/21 Data Structures in C - Preparatory Term

PAGE No.
DATE / /

Linked List

```
struct List{
    int data; ⇒[Can be any data structure or type, not just int]
    struct List *next;
};
```



Various functions with LL

i) ~~Bool~~ bool Search (struct List *node, int X){

```
    while (node){
        if (node->data==X){
            return true;
    }
    return false;
}
```

O(N) Time
O(1) Space

ii) Insert at begginning

a) struct List *addatbeg(struct List *node, int i){

```
    struct List *temp = (struct List*) malloc(sizeof(struct List));
    temp->data = i;
    temp->next = node;
    return temp;
}
```

O(N) Time, O(1) Space

head = addatbeg(~~node~~ head, i);

b) ~~struct~~ void addatbeg (struct List **node, int i){

```
    struct List* temp = (struct List*) malloc(sizeof(struct List));
    temp->data = i;
    temp->next = *node;
    *node = temp;
}
```

Called as,
addatbeg(&head, i);

PAGE No.
DATE / /

~~c) struct List* add~~

c) void addatbeg(struct List* node, int i) {

```
    ...
    node = temp;
}
addatbeg(head, i)
```

Why is this wrong?

Because we pass the reference of head pointer by value to the function. So updating value of "node" has no bearing on the value of "head". This is why that pointer to pointer logic was used in (b) and is actually preferred over (a) because it includes no return value and so no extra work to be done by client to use the function.

iii) Delete at beginning

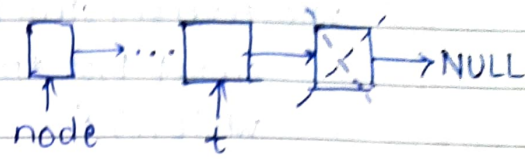
```
void deleteatbeg (struct List** node) {
    struct List* temp = *node;
    if (*node) {
        *node = (*node) -> next;
        free(temp);
    }
}
deleteatbeg(&head);
```

O(1) Time
O(1) Space

PAGE No.
DATE / /

iv) Delete at end

```
void deleteatend (struct List **node) {
    if (*node) {
        if (*node -> next) {
            struct List *t = *node;
            while (t -> next -> next)
                t = t -> next;
            free (t -> next);
            t -> next = NULL;
        }
        else
            *node = NULL;
    }
}
```



Time: O(N)
Space: O(1)

Reversing a Linked List → Recursive Approach

```
struct List *recreverse (struct List *head) {
    if (head && head -> next) {
        struct List *temp = head, *t;
        if (head -> next) {
            head = recreverse (head -> next);
            t = head;
            while (t -> next)
                t = t -> next;
            t -> next = temp;
            temp -> next = NULL
        }
    }
    return head;
}
```

Time: $O(N^2)$
N recursive calls
in which the no. of iterations
will grow from 1 to N

Space: O(N) Recursion
Call stack

PAGE No.
DATE / /

Reversing a Linked List → Iterative Approach

```
void reverse (struct List **head){
    struct List *p = NULL, *c = *head, *n;
    while (c) {
        n = c->next;
        c->next = p;
        p = c;
        c = n;
    }
    *head = p;
}
```

p = Previous
c = Current
n = Next

Time: O(N)
Space: O(1)

Detecting Cycle in Linked List → Floyd's Algorithm

```
bool cycle (struct List **head){
    struct List* t1 = *head, t2 = *head;
    while (t2) {
        t2 = t2 -> next;
        if (t2 == t1)
            return true;
        t1 = t1 -> next;
        if (t2)
            t2 = t2 -> next;
    }
    return false;
}
```

Time: O(N)
Space: O(1)

Why does this algorithm work?
- This is called the hare and tortoise approach where we have two ponters, t1 & t2.

PAGE No.
DATE / /

- In each iteration, t1 increases by 1 node and t2 increases by 2 nodes
- It should be atleast clear that if LL is not a cycle, then t2 would quickly become NULL and f^n returns false.
- If its a cycle, then why does it return true?
- Can visualize better with the help of an example.
- Running a race on 1km circular track, where you run at 5 m/s and your friend runs at 10 m/s.
- ~~Tabulating laps and times we get,~~
- When friend completes a lap, you will have finished half a lap.
- When friend completes 2 laps, you will have finished a lap and will meet your friend at the finish line.
- This ensures that in 2 laps (2 cycles), you (slow pointer) will always meet your friend (fast pointer).

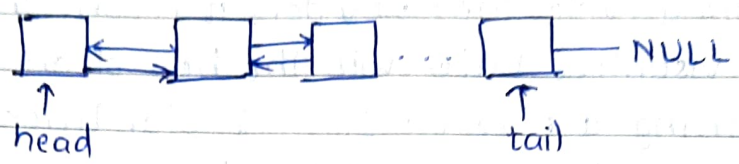
- This leads in to the next question, why is fast pointer twice as fast? Why not thrice or even more?
→ - Because twice guarantees that both pointers meet each other after two cycles exactly. This is also why its O(N) time.
- If it was thrice, then ~~slow~~ they will meet after 3 cycles.
- The underlying logic is that the number of cycles taken is the LCM of how fast they are. In this case, LCM(3, 1) = 3 cycles.

- The two pointers will meet each other eventually, even if they start from different nodes provided its a cycle.
- In racing terms, if both racers have different start points or different start times, then too they will meet.
- Visualize this example to verify this.

PAGE No.
DATE / /

Doubly Linked List

```
struct DLL{
    int data;
    struct List *next, *prev;
}
```



- The main advantage of this is that given a node p, we can delete it in O(1) time which is not the case with SLL where its O(N).
- p→prev→next = p→next;

14/8/21

Stack

Maintained using array S and index, top=0

```
push(i)
    s[top++] = i;

pop()
    if (top>0)
        top--; {or return s[top--]}
```

Queue

Q is an array and l=r=0;

```
enqueue(i)
    Q[r++] = i;

dequeue()
    if (l<r)
        l++; [and return Q[l-1];]
```

Stack as LL

List is LL.

Push: addatbeg OR addatend →

Pop: deleteatbeg deleteatend → O(n), so, not preferred
Must have tail pointer otherwise and even then delete is O(1)

PAGE No.
DATE / /

Queue using LL

LL with head and tail is List.
enqueue : addatend()
dequeue : deleteatbeg()

Always preferred to not use deleteatend() because its always O(n).

When to use LL over array implementation?

There's 1000 stacks.
10^6 nos. will go into any one particular stack.
In this case, ~~arrays wi~~ for array implementation,
We need 10^3 stacks of 10^6 size = 10^9 memory
for LL implementation.
We need 1 stack of 10^6 size + $10^3 - 1$ stacks of size 0 = 10^6 memory.
Clearly, LL is better in this usecase.
But arrays have advantage of random access.
So use tradeoff wisely as per usecase.

Queue Using Two Stacks → Must Know

Find the answer to that algo q^n, and why is it O(n).

Binary Tree

```
struct btree {
    int data;
    struct btree *lc, *rc, *p;
};
```

[~~Bin~~ Similar to LL, but two pointers instead of 'next']

[Parent Pointer makes this like a DLL]

Root: root->p = NULL
Leaf: leaf->lc = leaf->rc = NULL
Internal: All other nodes.

PAGE No.
DATE / /

Traversals : Preorder, Inorder, Postorder
V-L-R L-V-R L-R-V
[V = Visit node, L = Go to Left Child, R = Go to Right Child]

Height:
h(Leaf) = 0
h(Node) = 1 + max(h(lc), h(rc))
h(Root) = Height of Binary Tree

Level
l(Root) = 0
l(node) = 1 + l(parent node) [node ≠ root]
~~l(leaf) =~~
Maximum level is not always at ~~a~~ any leaf since leaves can be at any level, which may not be the deepest always.
Height ~~is~~ of binary tree = Max. level of binary tree.

→ When you go bottom-up, level always decreases by 1 each iteration
→ When you go top-down, ~~lev~~ height increase by 1 in each iteration.
Prove these two theorems to get result; height = max level

Level Order Traversal ⇒ Prints nodes of tree, level by level.
[Like BFS]

Iterative Pre Order Traversal

```
push(S, root);
while(S) {
    node = pop(S);
    visit(node);
    if (node → rc) push(S, node → rc);
    if (node → lc) push(S, node → lc);
}
```

Level Order Traversal

```
enqueue(Q, root);
while(Q) {
    node = dequeue(Q);
    visit(node);
    if (node → lc) enqueue(node → lc);
    if (node → rc) enqueue(node → rc);
}
```

[Note: rc pushed before lc because stack LIFO]

PAGE No.
DATE / /

Q. Write iterative inorder and postorder traversal.
Hint: Use ~~stacks~~ two stacks.

Q. Those two gigantic questions!

To calculate height of tree
You need postorder traversal because you need height of deepest nodes first and then move up.

height = 1 + max(height(node → lc), height(node → rc));

21/08/21 Iterative Inorder & Postorder
- Maintain two stacks, i) Contains tree nodes in DFS order.
  ii) Contains true or false.
- ~~Whe~~ Each node will be pushed in the stack twice.
- First time, we want children to be explored so push ~~false~~ true in stack 2.
- Second time we want to visit that node so push false in stack 2.
- So stack 2 tracks whether node is pushed into stack 1 for first time or not.

~~String Matchi~~ Some identities that were proved,

i) $L <= I + 1$ where, $n = L + I$
ii) $n + 1 <= 2^{h+1}$
iii) $L <= 2^h$

n → Total Nodes, L → Leaf Nodes, I → Internal Nodes

String Matching: Brute Force

```
for i = 0 to n
    for j = 0 to m
        if (T[i+j] != P[j])
            break;
    if (j == m)
        print "Match found";
```

T → Text
P → Pattern to find

PAGE No.
DATE / /

Rabin Karp Algorithm

This is for the case where string is binary input.

In case of regular strings we can extend the idea of precomputing hashes used here with the polynomial string hash f^n that you might remember :(

The hash for each m-size window is, (size of pattern)
Integer represent^n of binary string
1011 → 11, etc.

While extending sliding window, you can compute new hash as,
$t = t*2 + T[i] - T[i-m]*2^m$;

```
p=0; t=0;
for i=0 to m {
  p = p*2 + P[i];
  t = t*2 + T[i];
}
while (i<n){
   if (p==t) print "Match found";
   t = t*2 + T[i] - T[i-m]*2^m; i++;
}
if (p==t) print "Match found";
```

[You can precompute 2^m and avoid unnecessary ~~optim~~ operations]

[DECLARE $b = 2^m$ as long long int ~~otherwise can easily lead to overflow~~]

We can extend this sol^n as well,
We do, $p*2+P[i]$ because its binary string, if it was decimal we would've done $p*10+P[i]$

~~Strings are~~ Alphabet is just a-z where a=0, z=25 ⇒ 26 characters
So we can consider this to be base 26.

$p*26+P[i]$
$t = t*26 + T[i] - T[i-m]*b$; → $b = 26^m$.

The problem with ~~us~~ working with values of p and t directly is that their "hash values" can become too large (not even long long int).
This is why ~~the~~ hashing with a good hash f^n can help.
Here is one that implements hashing + collission handling.

PAGE No.
DATE / /

Rabin-Karp (Hashing + Collision Handling)

Hash f^n : (key % q)

```
p=0; t=0; b=1;
for i=0 to m {
  p= (p*2 + P[i]) % q;
  t= (t*2 + T[i]) % q;
  b= (b*2) % q;
}
while (i<m) {
  if (p==t)  //Can be match/collision
      Check if P[:m] == T[i-m:i] manually
  t= (t*2 + T[i] - T[i-m]*b) % q;
  if (t<0)
     t= t+q;  //To avoid negative modulo and converting into
                equivalent positive modulo
}
if (p==t)
  //Can be match/collision
```

$O(m \cdot n)$ Worst Case

$O(n)$ ~~Best Case~~ No Match Case

On average we need to consider probability of collision.

For each hash, its $\frac{1}{q}$. For n hashes, its $\frac{n}{q}$.

If we get a collision then we do m operations (manually checking pattern).

So in total worst case we can do $m * \frac{n}{q}$ operations.

But note that if we choose a really large prime value for $q >>> n$, then this probability is really low, and collisions are constant $O(1)$.

∴ Expected Running Time : $O(n)$

PAGE No.
DATE / /

Q. Given two strings and l, check if there is a string of length l which is substring of both the strings.

Q. Compute longest common substring between the two strings.
$O(n\log n)$ on average + $O(n)$ space.

27/08/21

→ Hash all strings of length l in string 1 and then do the same for string 2. If any hash computed for string 2 matches any in string 1, then, actually check both substrings whether they are equal to eliminate possibility of collision.

→ $O(n)$ on average because assuming collisions don't happen, ~~hashing~~ checking whether two substrings are equal is $O(1)$.
We hash all substrings of l in strings 1 & 2, but because of precomputation it will take $O(n)$ for string 1 & $O(m)$ for string 2.

→ Repeat substring matching algorithm with changing value of l.
To find correct value of l we use binary search.

find(0, n-1)
$l = (0 + (n-1))/2$
~~Check wh.~~ Create two hash tables for strings 1 & 2 for all substrings of length l.
Check if there's any common substring.
If yes, then there can be any substring with higher l, so store this substring's indices globally and
find($l+1$, n-1);
If no, then if substring exists its length has to be lower,
find(0, $l-1$);

Note: Clear both hash tables before making next binary search call.
Making binary search iterative will help with this.
$O(n\log n) + O(n)$ space.

CS 511 Algorithms

2/9/21

What is an Algorithm?
Method of solving a problem by a finite sequence of instructions.

Is there a problem for which there cannot be an Algorithm?
When can we conclude that an Algorithm doesn't work?
→ When it halts and gives wrong answer.
But what if algorithm doesn't halt for a long period of time?
This relates to the Halting Problem → Undecidable Problems

Note: NP-Hard and NP-Complete Problems are solvable, they just aren't efficient algorithms.

Measuring Performance of Algorithms
Running Time & Memory Required

Q. If an algorithm is $O(n^3)$ and $n = 10000$ then how long does it take?
→ It depends on the no. of instructions that can be executed per second.
Assuming processors are GHz → 10^9 operations per second.
Algorithm given is such that we do one operation per second.
$\therefore n^3 = (10^4)^3 = 10^{12}$ operations → $\frac{10^{12}}{10^9} = 10^3 = 1000$ seconds to execute.

Q. What is the largest integer array you can declare?
→ It is about 10^6 bytes. (5×10^6 to be precise). But it is still dependent on the system.

☆☆☆ So we will take 10^9 operations per second and 10^6 bytes of space to be allocated.

- Approximation algorithms give a somewhat guarantee on the approximate correctness of an algorithm.
- Heuristics give no such guarantee but they can be useful.
- Used for NP problems.

PAGE No.
DATE / /

Asymptotic Notations

Big-O "Bounded Above By" $T(n) = O(f(n))$
For some c and N, $T(n) \le c \cdot f(n)$ whenever $n > N$

Big-Omega "Bounded Below By" $T(n) = \Omega(f(n))$
For some c and N, $T(n) \ge c \cdot f(n)$ whenever $n > N$

Big-Theta "Bounded Above and Below": $T(n) = \Theta(f(n))$

$$c_1 \cdot f(n) < T(n) < c_2 \cdot f(n)$$

Q. Given a polynomial, $T(n) = a_k \cdot n^k + a_{k-1} n^{k-1} + \dots$

$T(n) = O(n^k)$

Show that $T(n) = \Theta(n^k)$, ie, show that $T(n) = \Omega(n^k)$

→ To-do

Note: int variables are typically of size 4 bytes giving a value range of $-2,147,483,648$ to $2,147,483,647 \Rightarrow (2^{31}-1)$
So roughly 2 billion-ish.

Q. The problem which computed $2*s+1$ for an int variable s.

→ It's ~~always~~ outputs -1 when $s = 2$ (starting value).
It outputs that because of the foll. computation,

$$2 \times 2 + 1 = 5 = 2^2 + (2^1 - 1)$$
$$2 \times 5 + 1 = 11 = 2^3 + (2^2 - 1)$$
$$2 \times 11 + 1 = 23 = 2^4 + (2^3 - 1)$$
$$\vdots$$
$$2^{30} + (2^{29} - 1) \Rightarrow \text{No overflow yet}$$
$$2^{31} + (2^{30} - 1) \Rightarrow \text{Yes Overflow}$$

Solving further, $\underbrace{(2^{31}-1)}_{\text{After Overflow}} + 1 + (2^{30} - 1)$

$\rightarrow -2^{31} + 2^{30} - 1 = (-2^{30} - 1)$

Next iteration: $2 * (-2^{30} - 1) + 1 = -2^{31} - 2 + 1 = -2^{31} - 1 = 2^{31} - 1$ (After Overflow)

Next iteration: $2 * (2^{31} - 1) + 1 = 2^{31} + 2^{31} - 2 + 1 = (2^{31} - 1) + (2^{31} - 1 + 1)$

$= -2^{31} + 2^{31} - 1 = -1$

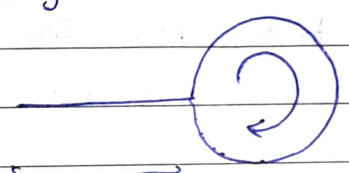
PAGE No.
DATE / /

I checked this for all s starting value from 2 to 10^6 and it always settled down to -1 which is ofcourse preceded by $2^{31}-1$.

Q. The code which did triple for loop with size of int array = 10^7 and n = 10^3 and computed, a[i*n*n+j*n+k] = 2*i*j*k+1;

→ Segmentation fault because max array size allowed is approx. 10^6 bytes
Reducing size to 10^6 also results in segmentation fault because
$i*n*n+j*n+k > 10^6$ when $n = 10^3$.
Setting size to 10^6 and $n = 10^2$, creates all elements of 'a' = 1.

03/09/21 Proof for finding the first node in a Linked List cycle.



We have two pointers
t1 & t2 (twice as fast as t1)

a nodes outside cycle — b nodes in cycle ⇒ n = a+b ⇒ Total nodes in LL

Two cases : a<b & a>b

→ a = bq + r [Divide a by b]
When t_1 is at a+b−r
t_2 will have covered a+b−r twice.

All for loops : Exclusive right end
i.e, for $i = 0$ to n
$\Downarrow$
i in $[0, n-1]$

PAGE No.
DATE / /

Q. Given an array a of size n. Find max. difference b/w two elements of the array.

→ $O(n^2)$ algorithm

```
for i=0 to n
    for j=0 to n
        if (a_j - a_i > max)
            max = a_j - a_i
```

$O(n)$ algorithm
Find max(a)
Find min(a)
Then subtract them.

Clearly for $10^6 = n$, time taken will be like 10 minutes $\Rightarrow \frac{(10^6)^2}{10^9} = 10^3$ seconds

Q. Same question, but we have to find min. difference.

→ Sort array and find difference b/w every two adjacent elements.
$O(n \log n)$

But how to optimize even further?
No clue for now.
This just shows how big of a difference even slight tweaking of the problem makes.
Sir claims no algorithm faster than $O(n \log n)$ exists for this. [Excluding stuff like Countsort that's memory heavy]

☆☆ Why? Think on it some more.

Q. Another modification, find max. ~~absolute~~ difference $a_j - a_i$ such that, $j > i$

→ Connection to real-world (kind of)
Buying and Selling Stocks and maximizing profit.
j: Day of Selling, i: Day of Buying, A_i: price of stock on i-th day
$\therefore$ Maximizing $A_j - A_i$

PAGE No.
DATE / /

```
i=0
for j=1 to n
    if (a_j < a_i)
        i=j
    else if (a_j - a_i > M)
        M = a_j - a_i
```

You buy on day i and sell on ea ~~reach day~~ ay j> to get max $A_j - A_i$.
But if $A_j < A_i$, then you have exhausted all valid possibilities of buying on day A_i. And you should now buy on day j, that is why we set i=j

O(n) algorithm

More modification: $j \geq i+l$, where l is a given parameter.
Can ~~also~~ only sell stock atleast l days after buying stock.

```
i=0
for j=l to n
    if (a_{j-l} < a_i)
        i=j-l
    if (a_j - a_i > M)
        M = a_j - a_i
```

Only change here is that j has to be atleast l days away from i.
~~So when you reach~~ So you have a sliding window of sorts.
When you reach day j, day j-l enters into this window for the first time and it can be checked against A_i ~~which is~~ where i is definitely < j-l

O(n) algorithm

Final modification: $j \leq i+l$, where l is given.
Can only hold stock for max $l-1$ days before selling.
[Code is given by sir. Somehow interpret it.]
[Assume $l=d$ for this code]

```
l=0; r=0; D = Queue (Double-Ended)
for(i=0; i<n; ++i){
    while (l<r && A[D[r-1]] > A[i])
        r--;
    D[r++] = i;
    if (r-l > d)
        l++;
    if (A[D[r-1]]-A[D[l]]> max)
        max = A[D[r-1]] - A[D[l]];
}
```

(Pseudocode for this on next page)

PAGE No.
DATE / /

Consider each element in A
while Q is not empty and no. is smaller than the element at back of the Q, dequeue that element.
push A[i]
if the number of elements in Q is more than d then dequeue elements.
if last - first in Q > max, then update max.

Write intuitive approach for above!

09/09/21 Fibonacci Numbers

Q. $F(0) = 0$
$F(1) = 1$
$F(n) = [F(n-1) + F(n-2)] \% 10$
Write an efficient algorithm to compute this.
What is the largest value of n for which your laptop can compute this in 1 second?

→ Algorithm 1:

```
int rec(int n){
    if (n< 2)
        return n;
    else
        return (rec(n-1) + rec(n-2)) % 10;
}
```

What is value of n for which this run in 1s?

PAGE No.
DATE / /

What is time complexity?

Let $T(n)$ = No. of basic operations reqd. to find $F(n)$

$= T(n-1) + T(n-2) + 5$

1 Comparison
2 Subtractions
1 Addition
1 Modulo
[But you can ignore this anyways]

$< T(n-1) + T(n-1)$

Because, $T(n-1) = T(n-2) + T(n-3) + 5$
Definitely $> T(n-2) + 5$

$T(n) < 2T(n-1)$

Applying same logic, $T(n-1) < 2T(n-2)$

$\therefore T(n) < 2^2 T(n-2)$

$\vdots$

$T(n) < 2^k T(n-k)$

$\vdots$

$T(n) < 2^n T(n-n)$ $[T(0) = 1]$

i.e, $T(n) < 2^n$

$\therefore T(n) = O(2^n)$

So, $T(100) = O(2^{100}) \Rightarrow$ Very Bad ^_^

Find ~~upper~~ lower bound,

$T(n) = T(n-1) + T(n-2) + 5$

$> T(n-1) + T(n-2)$

$> 2T(n-2)$

$> 2^2 T(n-4)$

$\vdots$

$> 2^k T(n-2k)$

$\vdots$

$> 2^{n/2} T\left[n - 2\left(\frac{n}{2}\right)\right]$

$T(n) > 2^{n/2} \Rightarrow T(n) = \Omega(2^{n/2})$

Assume: $2^{10} \geq 10^3$ approx.

So $T(100) > 2^{50} > 10^{15}$

A computer with 10^9 ops/sec will require 10^6 seconds ~ 10 days

PAGE No.
DATE / /

Space Complexity,

$S(n)$ = Amount of space required to compute $F(n)$.

$S(n) = \max[S(n-1), S(n-2)] + 1$

Why? When we compute $F(n)$, we compute $F(n-1)$ & $F(n-2)$

Both of them require space.

When we compute $F(n-1)$, we store result in a variable and then we can reuse space used for it to compute $F(n-2)$.

So that's why $\max(S(n-1), S(n-2)) + 1$

storing result in variable

$\therefore S(n) = S(n-1) + 1$

$= n$

$\therefore S(n) = \Theta(n) \rightarrow$ [Both $O(n)$ & $\Omega(n)$]

<u>Linear Time Algorithm</u>

```
f[0] = 0
f[1] = 1
for (i=2; i<=n; i++)
    f[i] = (f[i-1] + f[i-2])%10
```

Clearly, $T(n) = O(n)$

$S(n) = O(n)$

So in 1s, you can get, $n = 10^9$. But!!!

Max space allowed for an array matters.

If we can only allocate 10^6 size int array, then, we can only get $f(n)$ upto 10^6.

But how can we get $f(n)$ for $n > 10^6$?

PAGE No.
DATE / /

Obvious Modification: Why to store array? We just need the last two Fibonacci Nos. to compute the next one and update accordingly.

```
a=0, b=1, c=n
for (i=2; i<=n; i++){
    c=(a+b)%10
    a=b
    b=c
}
```

$O(n)$ Time
$O(1)$ Space

So we can get $f(n)$ for up to 10^9 in 1 seconds.

Prefix Sum

$P_0 = 0$ where a is an array of length n.
$P_1 = a_0$
$P_2 = a_0 + a_1$
$\vdots$
$P_i = a_0 + a_1 + \dots + a_{i-1}$
$\vdots$
$P_n = a_0 + a_1 + \dots + a_{n-1}$

Consider the stocks example algorithms.
What if the elements of 'a' can have -ve elements and we have to find max. sum? $\longrightarrow$ Maximum Sum Subarray

Algorithm 1:

1. Compute all prefix sums, P.
2. $i = 0$, $max = -\infty$, $i_{max} = -1$, $j_{max} = -1$
3. for $j = 1$ to $n+1$
   if $P_j < P_i$
   $i = j$
   else if $P_j - P_i > max$ $\rightarrow$ Computes $\Sigma a[i \dots j-1]$
   $max = P_j - P_i$
   $i_{max} = i$, $j_{max} = j-1$

This is basically the same as the first stocks algo. the only difference being the prefix sum part.

$O(n)$ time
$O(n)$ space

PAGE No.
DATE / /

How to make this O(1) space?

- Store $P_j - P_i$ in a single variable s
  "if $P_j < P_i$" becomes "if $s<0$"
  "if $P_j - P_i > max$" becomes "if $s > max$"
- We have to keep updating s in each iteration now.

This is the final code,

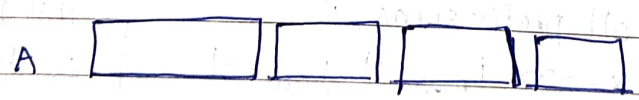
```
i=0, s=0, max=-∞
for j=1 to n+1
    s = s + A[j-1]   // s contains A[i...j-1]
    if (s<0)         // Start new subarrays from A[j-1]
        i=j
        s=0
    else if (s > max)
        max = s
        imax = i
        jmax = j-1
```

For all -ve nos. this will return 0. [Taking a subarray of length 0]

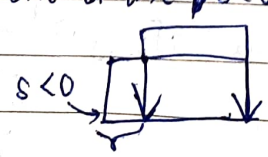
Can make modifications to this so that we return the largest no. in the array.

Proof:

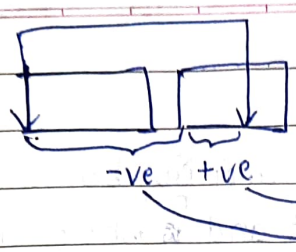
We can visualize this algorithm where array is cut at points where $s<0$



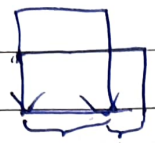
To prove this works, we have prove that each subarray can only start from one of the $s<0$ points.



Is this possible?
No!
Because, so it must start from the slot beginning.



Is this possible?
No!
Because, so we ~~have to~~ cannot have intersection b/w slots.



Possible?
No! So slots must start at $s<0$ point and also end at $s<0$.

∴ All cases are proved.

Q. Max sum subarray of size atleast l

→ This is like algorithm 2 of stocks (sell after atleast l days)

We have to keep track of two sums now, $A[j-l \ldots j]$ $s1$ $A[i \ldots j]$ s

```
s = 0, s1 = 0
j = 0 to l
  s = s + A[j]
  s1 = s
j = l to n
  s1 = s1 + A[j] - A[j-l]
  s = s + A[j]
  if (s < s1)
      s = s1;
  else if (s > max)
      max = s
```

[Detailed explanation in cpp file]

```
i = 0
j = l to n
  if (a_{j-l} < a_i)
      i = j - l
  else if (a_j - a_i > max)
      max = a_j - a_i
```

Line-by-Line Comparison and how it maps

Q. Max sum subarray of size atmost l → H.W

PAGE No.
DATE / /

11/9/21 Fibonacci Numbers (Contd.)

Fibonacci Number % 10 is Periodic

The moment two consecutive Fibonacci Numbers % 10 are 0 and 1, then, the sequence will repeat and $n-1$ where $f(n)=0$ will become period of this sequence.

So we want there to be $(f(i), f(i-1))$ such that it is equal to $(f(j), f(j-1))$. [these are consecutive Fibonacci nos.]

$f(n)$ can only take 10 values, so total distinct pairs $(f(n), f(n-1))$ possible are $10 \times 10 = 100$ pairs.

If you take 102 consecutive nos. in this series like,
$f(0), f(1), \ldots, f(102)$
Then they can be paired up as $(f(0), f(1)), (f(1), f(2)), \ldots$
$(f(101), f(102))$.
This will give us 101 ~~dis~~ pairs ~~which~~ and by the Pigeonhole Principle it is guaranteed that there exists some $(f(i), f(i-1))$ from these pairs such that it is equal to some $(f(j), f(j-1))$ from these pairs.

This same thing can be generalized for % m as,
Sequence must repeat in m^2+2 consecutive terms.
Like for % 10 it was, $10^2+2 = 102$

However stricter proof exists which states that,
$f(n) = [f(n-1)+f(n-2)] \% m$ is a periodic sequence with period less than $6m$.

So now all we have to do is for given m, find the period ~~and~~ length p and then $f(n) = f(n \% p)$
This is $O(6m + \log n) = \underline{O(m + \log n)}$

PAGE No.
DATE / /

But when m starts to become too large like 1000 digit large, then $O(m+\log n)$ starts to look bad.
So we need something even better, a $O(\log n)$ approach.

<u>Using Matrix Exponentiation to solve this</u>
We know that if A is 2x2 matrix then, A^N can be computed in $O(\log n)$ time.

We can write the Fibonacci Sequence in matrix form as,

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

$$= \cancel{A \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

$$\vdots$$

$$= \underbrace{\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}}_{A}^{n} \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = A^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

So we compute $F(n)$ using $F(1)$ & $F(0)$ only but you need A^n.

[<u>Note</u>: We are compute $F(n) \% m$, so while finding A^n, ensure that you do $\% m$ for all operations]

This will work for even 1000 digit but only 9 digit m. Because most programming languages only support modulo upto 9 digits after that you'll need to store m as an array and extra complexity of $\log^2 m$ will be applied for matrix multiplication calculations.
So total complexity $O(\log^2 m \cdot \log n)$

PAGE No.
DATE / /

Finding A^n ($A = 2\times2$) in logn time
[Also works for $A = 3\times3$]

Algorithm 1: Recursive sol^n

```
power(X, n){
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return power(X*X, n/2);
    else
        return ~~power~~ X * power(X*X, n/2);
}
```

Why does this work? What is Time complexity?

If n is even then $n = 2k \Rightarrow k = \frac{n}{2}$

$\therefore$ We can write $X^n = X^{2k} = (X^2)^k = (X^2)^{n/2} \Rightarrow$ power$(X*X, n/2)$

If n is odd then $n = 2k+1 \Rightarrow k = n/2$ (Integer Division!)

$\therefore X^n = X^{2k+1} = X * X^{2k} \Rightarrow X * (X^2)^{n/2} \Rightarrow X *$ power$(X*X, n/2)$

Time Complexity

$T(n)$ = No. of multiplications algorithm does to compute X^n.
(it also does divisions but we ignore them here)

$T(n) = 1 + T(n/2)$ if n is even
$= 2 + T(n/2)$ if n is odd

$\therefore T(n) \le 2 + T(n/2) \le 2 + 2 + T(n/4) = 4 + T(n/4)$
$\le 2k + T(n/2^k)$

This terminates when we reach $T(1)$, i.e, $T\left(\frac{2^k}{2^k}\right) + 2\cdot 2^k \Rightarrow n = 2^k$

$\boxed{\therefore k = \log_2 n}$ $\therefore T(n = 2^k) \le 2\log_2 n + T(1) \le 2\log_2 n$

Computing lower Bound,

$T(n) > 1 + T(n/2) > 2 + T(n/4) > \ldots > k + T(n/2^k)$

Again $T(n = 2^k) \Rightarrow k + T(1) = k = \log_2 n$

$\therefore \boxed{T(n) > \log_2 n}$

$\boxed{\therefore T(n) = \Theta(\log_2 n)} \longrightarrow \log n < T(n) < 2\log n$

Iterative version of this,

```
~~y=1;~~
~~while (n>0~~
power (X, n) {
    y = 1;
    while (n > 0) {
        if (n % 2 == 1)
            y = y * X;
        X = X * X;
        n = n / 2;
    }
    return y;
}
```

It's pretty clear how this algo. is $\log_2 n$.

Example run, Computing X^{28}

N	X	Y
28	X	1
14	X^2	1
7	X^4	X^4
3	X^8	X^{12}
1	X^{16}	X^{28}
0	X^{32}	$\boxed{X^{28}}$

Let, $n = \sum_i b_i 2^i$ be binary represent^n of n.

$\therefore X^n = X^{\sum b_i 2^i} = \prod_i X^{b_i 2^i}$ $[X^{a+b} = X^a \cdot X^b]$

In this product, if $b_i = 0$, then i-th term in series is 1.

if $b_i = 1$, then i-th term is X^{2^i}

PAGE No.
DATE / /

To generalize power (A, n) where $A = 2\times2$ Matrix
We start with $Y = I_{2\times2}$ and all multiplications are replaced by matrix multiplications.
Since all matrices are 2×2, we can assume that matrix multiplication takes constant time.
And this still leaves us with $\Theta(\log n)$ algorithm.

What if N is 1000 digits?
Normal operations don't work for it.
If N is given in binary represent^n then we can easily solve this as,
Checking for odd, $b[i] = 1$ ← Iteration no.
Binary represent^n array ($6 = 110 = [0, 1, 1]$)

Everything else remains the same.

If N is given in decimal represent^n, then this fails.
We can try to convert it to binary but that takes $O(\log n^2)$ which is not good. ($\log n$ iterations, one '/' by 2 per iteration)
There is an optimal approach involving 9 specific cases each with their different multiplications, instead of checking for odd or even.

<u>F(n) % 10 in O(1) Time</u>
F(n) % 10 is periodic with period 60.
So to compute F(n) % 10 all we need to do is,
F(n % 60)
We can store these first 60 values in an array F and just return F[n % 60] in O(1).
If $n < 2^{64} \sim 10^{18} = 19$ digits, then we can do this in O(1)

But this doesn't work for a 1000 digit no.
So we store such nos. as an array where,
$N = \sum_i A[i] \cdot 10^i$ so, $123 \Rightarrow A[0]=3,\ A[1]=2,\ A[2]=1$

And we will have to implement addition, multiplication, f^ns for such arrays.

<u>Computing Modulo p for such an array.</u>

$p = 60$
$r = A[k-1]$ [where k = no. of digits in N]
$i = k-2$
while $(i > -1)$
$\quad r = (r * 10 + A[i--]) \% p$

$O(k) \rightarrow O(\log n)$
$\therefore$ Find $F[n \% 60]$
$\uparrow$
$O(\log n)$

[Note: We go from MSB to LSB in mod f^n so that we don't have to compute powers of 10.
Try going from LSB to MSB and you'll see]

<u>Summary</u>: Recursive Algo. $O(2^n)$ time, $O(n)$ space
Iterative Algo. $O(n)$ time, $O(1)$ space
Sequence is Periodic $\Rightarrow$ Store Period $\Rightarrow$ Period of $\% 10 = 60$, then compute $n \% 60 \Rightarrow O(1)$ time, $O(1)$ space
But for large n ~~mod~~ $> 2^{64}$, $\% 60$ is $\log n$ operation and if m instead of $\% 10$ then $O(m + \log n$ time), $\underbrace{O(\log n)}$ space to store array

To avoid dependence on m use Matrix Exponentiation,

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = A^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}, \text{ where } A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

A^n can be found $O(\log n)$ time using iterative power f^n assuming 2×2 case only.
[In case digits are large then store in array and stuff :(]

$\cancel{(1+2+0+6)\%10 = 9}$

$\cancel{(9+2+5+6)\%10 = 2}$

PAGE No.
DATE / /

Homework:

Find $f(n) = [f(n-1) - 2f(n-2) + 5f(n-3) + 6]\%m$

where $m = 10, 3468, 1436$,

$f(0) = 0,\ f(1) = 1,\ f(2) = 1$

Use A^n method to find this,

→ For,

$f(n) = (f(n-1) + f(n-2))\%m$

$$\begin{bmatrix} f(n+1) \\ f(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix}$$

For $f(n) = (f(n-1) - 2f(n-2) + 5f(n-3) + 6)\%m$

$$\begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 5 & 6 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ 1 \end{bmatrix} = A \cdot F = A \cdot A \cdot \begin{bmatrix} f(n-1) \\ f(n-2) \\ f(n-3) \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ 1 \end{bmatrix} = A^{n-1} \begin{bmatrix} f(3-1) \\ f(3-2) \\ f(3-3) \\ 1 \end{bmatrix} =$$

↓

Multiply by A

$$A \begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ 1 \end{bmatrix} = A \begin{bmatrix} f(n+2) \\ f(n+1) \\ f(n) \\ 1 \end{bmatrix}$$

$$A \cdot \left(A^{n-1} \begin{bmatrix} f(2) \\ f(1) \\ f(0) \\ 1 \end{bmatrix} \right) = A^n \begin{bmatrix} f(2) \\ f(1) \\ f(0) \\ 1 \end{bmatrix} = \begin{bmatrix} f(n+2) \\ f(n+1) \\ f(n) \\ 1 \end{bmatrix}$$

Our Answer!!

16/9/21

Sorting Problem

Given a sequence of n integers $a_0, a_1, \ldots, a_{n-1}$
Arrange them in increasing order ; $a_0 \le a_1 \le \ldots \le a_{n-1}$

1. Bubble Sort

Look at ~~any~~ two consecutive numbers a_j and a_{j+1}
if $(a_j > a_{j+1})$ swap them.

```
for (i=0; i<n-1; i++)
    for (j=0; j<n-1-i; j++)
        if (a[j] > a[j+1])
            swap(a[j], a[j+1]);
```

After ~~one~~ first iteration of ~~j th~~ j-for loop, the largest number moves to A[n-1]. A[n-1] is then ignored and algorithm continues.
So after doing this 'n-1' times A[1] will have second smallest no. leaving A[0] as the smallest no.

Time Complexity

$T(n) = \underbrace{n \cdot 1}_{\text{(if comparisons/No. of swaps)}} + \underbrace{T(n-1)}_{\text{(Next iteration of i-for loop)}}$

$= n-1 + n-2 + n-3 + \ldots 1$

$= \frac{n(n-1)}{2}$

$T(n) = \Theta(n^2) \rightarrow$ [$\Theta = O$ and Ω both are same]

Worst Case: Array in Decreasing Order → $n(n-1)/2$ swaps
Best Case: Array in Ascending Order → 0 swaps

2. Selection Sort

Aim is to minimize no. of swaps.

In each iteration, i, find the ~~$(n-1)^{th}$ lar~~ i^{th} largest integer and swap it at $(n-1-i)^{th}$ position in A.

~~max = 0 for (i = 0; i < n-1; i++)~~
~~for (j = i+1;~~

```
for (i = 0; i<n-1; i++) {
    max = 0;
    for (j = 1; j<n-i; j++) {
        if (A[j] > A[max])
            max = j;
    }
    if (max < n-1-i)
        swap(A[max], A[n-1-i]);
}
```

Time Complexity

$T(n)$ = No. of comparisons

$= n-1 + T(n-1)$

$= n-1 + n-2 + \dots + 1$

$= n(n-1)/2 = \Theta(n^2)$

No. of swaps : In worst case, $T_s(n) = 1 + T_s(n-1)$

$= n-1$

$= \Theta(n)$.

In best case, $T_s(n) = 0$

PAGE No.
DATE / /

Insertion Sort

~~We a~~ In i^{th} iteration we assume that first $(i-1)$ elements are already sorted, then we try to place $(i)^{th}$ element into its correct position. This approach is useful in case your input is a stream of integers, so you want array to be sorted at any point in time.

```
for (i = 1; i<n; i++) {
    j = i-1 ; t = A[i];
    while (j>=0 && A[j] > t)
        A[j+1] = A[j--];
    A[j+1] = t;
}
```

Time Complexity

$T(n)$ = No of Comparisons

Worst Case ⇒ Reverse-Sorted Array

$T(n) = n-1 + T(n-1)$

$T(n) = \Theta(n^2)$

Best Case ⇒ Sorted Array

$T(n) = 1 + T(n-1)$

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

Searching :

1. Linear Search

Worst Case ⇒ Element not present in array ⇒ $\Theta(n)$

What is the average no. of comparisons if element is found?

$$T(n) = \sum_{j=1}^{i} \frac{1}{n} i$$ → where element is found at i-th index

$\frac{1}{n}$ = Probability that i-th index has element

$$= \frac{1}{\cancel{n}} \frac{\cancel{n}(n+1)}{2}$$

So we are computing Expectation here, $\sum x \cdot P(x)$, x = index of array

$$T(n) = \frac{n+1}{2}$$

2. Binary Search

Sorted Array required.

PAGE No.
DATE / /

```
while (l<=r) {
    mid = (l+r)/2;
    if (A[mid] == x)
        return true;
    if (A[mid] < x)
        l = mid+1;
    else
        r = mid-1;
}
return ~~tr~~ false;
```

<u>Time Complexity</u>

$T(n)$ = No. of comparisons

In ~~the~~ worst case ⇒ Element not found

$T(n) \leq 2 + T(n/2)$

$\leq 2\log_2 n$

$T(n) = \Theta(\log_2 n)$

While searching for a word in a dictionary, BS can be used. But there <u>biased binary search</u> is more useful than regular BS. Example, You want a ~~sta~~ word that starts with 'a' then why should we check the middle page of the dictionary? We can bias our search ~~set~~ so that it only looks at the starting section. All of this is possible because we know extra info. that dictionary will always have 'a' words in beginning and 'z' words at the end.

PAGE No.
DATE / /

Efficient Sorting

1. Merge Sort

Divide the array into 2 equal parts.
Merge sort the 2 parts.
Merge the 2 sorted arrays into a single sorted array.

```
mergesort (int A[], l, r){
    if (l<r) {
        mid = (l+r)/2;
        mergesort (A, l, m);
        mergesort (A, m+1, r);
        merge (A, l, r);
    }
}

merge (int A[], l, r) {
    k=0; mid=(l+r)/2;
    i= l; j= mid+1;
    int B[r-l+1];
    while (i<=mid && j<=r) {
        if (A[i]<=A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }
    while (i<=mid)
        B[k++]= A[i++];
    while (j<=r)
        B[k++] = A[j++];
    i=l; k=0;
    while (i<=r)
        A[i++] = B[k++];
}
```

Time Complexity

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

- One n/2 to sort left half
- Second n/2 to sort right half

- One 'n' to get B
- One 'n' to copy B into A[l...r]

$\therefore T(n) = 2^2 T(n/2^2) + 4n$

$= 2^k T(n/2^k) + 2k \cdot n$

When $n = 2^k$, $T(n/2^k) = 1$

$k = \log_2 n$

$= 2^{\log_2 n}(1) + 2\log_2 n \cdot n$

$= n^1 + 2(\log_2 n)n$

$T(n) = O(n \log_2 n)$

Space Complexity: $O(n)$

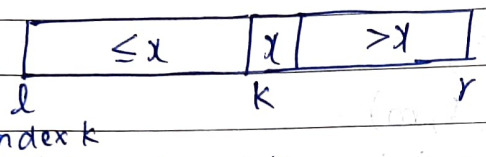
So if $n > 10^6$ we cannot use Merge Sort.

17/9/21

2. Quick Sort (Randomized)

O(1) Space! O(nlogn) Time

- Partition the array into 2 parts around a pivot.
- Sort the parts
- Concatenate the two parts.



x at index k

Our pivot, should ~~pivot~~ partition array such that all, $a[i] \le x$ if $i<k$ and $a[i] > x$ if $i>k$

```
rqsort(int A[], int l, int r){
  if (l<r){
    k= rand() % (r-l+1) +l;
    swap (A[l], A[k]);
    p = partition(A, l, r);
    rqsort(A, l, p-1);
    rqsort(A, p+1, r);
  }
}
```

```
partition (int A[], l, r) {
  pivot = A[l]; i= l+1; j= r;
  while (i<=j) {
    while (i<=j && A[i]<=pivot)
      i++;
    while (i<=j && A[j]>pivot)
      j--;
    if (i<j) {
      swap(A[i], A[j]);
      i++; j--;
    }
  }
```

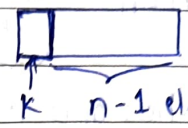
Alternate Version

```
partition (int A[], l, r){
  j=l, x= A[l];
  for(i= l+1; i<=r; i++){
    if (A[i]<=x){
      j++;
      swap(A[i], A[j]);
    }
  }
  swap(A[l], A[j]);
  return j;
}
```

```
    i--;   //
    A[l] = A[i];
    A[i] = pivot;
    return i;
}
```

<u>Time Complexity</u>

Worst Case ⇒ Pivot's final position is at l ⇒



$\therefore T(n) = T(n-1) + n$ (To get partition)

(n-1) elements remaining in second half.

$T(n) = \Theta(n^2)$

If we can pick a pivot such that its final position is in the middle of the array, then we get two partitions of n/2 elements and then we call quicksort on both of those.

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

$= \Theta(n \log n)$

How to get such a partition? ⇒ Random Pivot Selection

Average Runtime of Quicksort $= \Theta(n \log n)$ → How?

<u>Quicksort Analysis</u>

Let a_n be avg. time taken to sort an array of size n using RQS.

∴ After finding pivot at index k we will have two halves with avg. times,

a_{k-1} & a_{n-k}

∴ Recurrence Rel^n is, $a_n = a_{k-1} + a_{n-k} + (n+1)$

For partioning

For picking pivot

PAGE No.
DATE / /

Now we pick the pivot at random so probability is involved.
$P(\text{Pivot is at pos } k) = \frac{1}{n}$ for all k.

Now we can get a_n (Expected value) as follow,

$$a_n = \sum_{k=1}^{n} \frac{1}{n} \cdot (a_{k-1} + a_{n-k} + (n+1))$$

$$\therefore n a_n = \underbrace{2\sum_{k=0}^{n-1} a_k}_{} + (n+1)\sum_{k=1}^{n} 1$$

When goes k goes from 1 to n, a_{k-1} terms gives you $a_0, a_1, \dots a_{n-1}$ and a_{n-k} terms gives you, $a_{n-1}, a_{n-2}, \dots a_0$, so together over the Σ they both give

$$2\sum_{k=0}^{n-1} a_k$$

$$n a_n = 2\sum_{k=0}^{n-1} a_k + n(n+1)$$

$$n a_n = 2(a_0 + a_1 + \dots + a_{n-1}) + (n+1)n \longrightarrow (i)$$

$\therefore$ For a_{n-1},

$$(n-1) a_{n-1} = 2(a_0 + a_1 + \dots + a_{n-2}) + n(n-1) \longrightarrow (ii)$$

Subtracting (ii) from (i),

$$n a_n - (n-1) a_{n-1} = 2(a_0 + a_1 \dots + a_{n-1}) + n(n+1) - 2(a_0 + a_1 \dots + a_{n-2}) - n(n-1)$$

$$n a_n - (n-1) a_{n-1} = 2a_{n-1} + 2n$$

$$n a_n = (2 + n - 1) a_{n-1} + 2n$$

$$\therefore n a_n = (n+1) a_{n-1} + 2n$$

Divide by $n(n+1)$,

$$\frac{a_n}{n+1} = \frac{a_{n-1}}{n} + \frac{2}{n+1}$$

For $n-1$, $\frac{a_{n-1}}{n} = \frac{a_{n-2}}{n-1} + \frac{2}{n}$ ...

For 2, $\frac{a_2}{3} = \frac{a_1}{2} + \frac{2}{3}$; For 1, $\frac{a_1}{2} = \frac{a_0}{1} + \frac{2}{2}$

Now add all of the equations all a_n terms cancel out except for a_n & a_0

$$\frac{a_n}{n+1} = \cancel{\frac{a_{n-1}}{n}} + \frac{2}{n+1}$$

$$+ \quad \cancel{\frac{a_{n-1}}{n}} = \cancel{\frac{a_{n-2}}{n-1}} + \frac{2}{n}$$

$$\vdots$$

$$\frac{a_n}{n+1} = \frac{a_0}{1} + 2\left[\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{3} + \frac{1}{2}\right]$$

$$\therefore a_n = a_0(n+1) + 2(n+1)H(n+1)$$

$a_0(n+1)$ ↓ Can be written as $O(n)$

$H(n+1)$ ↓ $H(n) = \sum \frac{1}{k}$ is the Harmonic Series

$$a_n = O(n) + 2(n+1)H(n+1)$$

We can write $H(n) = \sum \frac{1}{k} \leq \int \frac{1}{x}\,dx = (\log x)! \Rightarrow H(n+1) = O(\log(n+1))$
$= O(\log n)$

$$\therefore a_n = O(n) + 2(n+1)O(\log n)$$

$$\boxed{a_n = O(n\log n)}$$

Finding Rank ⇒ Median of an Array

- The Randomised Quicksort Algo. is non-deterministic.
- We can make it deterministic by choosing a good pivot without randomization, i.e, partition() function implementation.
- If we can find median of an array in $O(n)$ time deterministically, then this can be the new implementation of partition() and QS is now deterministic.

What is Rank?

In an array, rank of ~~the~~ any ~~number~~ element x (x may or may not $\in A$) is the no of elements that are greater than it + 1 ⇒ rank(max(A)) = 0 + 1 = 1, ..., rank(min(A)) = n

~~So the one~~

PAGE No.
DATE / /

We can therefore say that element $\in A$ with rank $n/2$ is the median of the array.

Example, {12, 38, 28, 33, 22, 23}
Rank of 32 in array is $2+1=3$

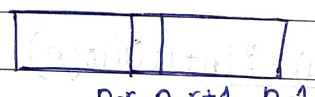
<u>Rank(X) can be found in linear time.</u>

```
rank = 1;
for (i=0; i<n; i++)
    if (A[i] > x)
        rank++;
```

~~Or~~
~~We want to find rank(r) = i~~

Q. Given an r $(1 \le r \le n)$ we want to find an i $(0 \le i < n-1)$ such that $rank(A[i]) = r$

→ Approach 1:
Sort A.
Return $n-r$ → Because



$$rank(A[n-r]) = 1 + \underbrace{(n-1)-(n-r+1)}_{\text{No. of elements} > A[n-r]} + 1$$

$$\therefore rank(A[n-r]) = r$$

$O(n \log n)$

Approach 2: <u>Later!</u>

Q. Find an element whose rank is $< n/2+1$, i.e, a no. bigger than the median.

→ Pick a random p between 0 and $n-1$, what is the probability that $A[p]$ is smaller than the median?
It is $1/2$.

PAGE No.
DATE / /

Pick k nos ~~of~~ from array, what is probability that all of them are smaller than the median?

$\frac{1}{2^k}$ (assuming independent events)

$\therefore 1 - \frac{1}{2^k}$ is the probability that atleast one of ~~them~~ the k numbers will be greater than the median.

So approach is,
Pick k nos. from given sequence and report max. of them.
Then with probability '$1 - \frac{1}{2^k}$', the reported no. will be greater than median.

Let $k = \log n$, then probability '$1 - \frac{1}{n}$' is what we're working with.

So larger the value of n, the more likely we are to find an element greater than median.

Th^m: A deterministic algo. for this will require atleast n/4 comparisons.

So this is O(n) algo.
But the randomized algo. gives us O(logn) time with probability of correctness as $1 - \frac{1}{n}$

Randomized Algorithms

1. Las Vegas
- Always gives correct answer.
- Running time depends on ~~the~~ input and choices the algo. makes.
Eg, Randomized Quick Sort, Randomized FindRank, Hashing.

2. Monte Carlo
- Efficient but answers can be wrong with very low probability.
Eg, Algo. to find elements bigger than median, Primality Testing
$O(k^6) \rightarrow O(k^3)$

PAGE No.
DATE / /

Q. Given a binary string A, return an index k such that no. of zeros in $A[:k]$ = no. of ones in $A[k^{th}:]$

→ For A = 000...., k = -1
For A = 11...., k = n

Does a k always exist?
Given that string contains atleast 1 0 & 1, then does a k always exist?
Yes. But proof?

Proof Hints:
By induction.
On No. of elements in string.
For n = 1, n = 2, ...

0 1 00 11 01 10

For any string of length n → Assume last bit 1
Induction Hypothesis: k exists for string of length n-1

(can be 0 or 1)
if 1 then k+1 → include 1 in Left Partition
last 1 compensates

if 0 then also k+1
→ include 0 in Left Partition
count of 0s in LHS increases by 1
but count of 1s in RHS decreases by 1
because of last 1 bit.

[This same proof can be repeated assuming last bit 0]

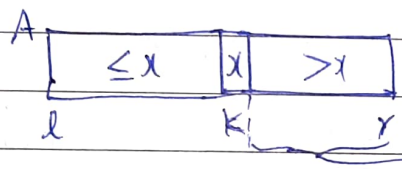
Algo Hint:
l = 0, r = n-1
Skip all 1s for l and Skip all 0s for r till first 0 & 1 encounter
Then alternatively increment & decrement l & r.
When l > r, return some value.

PAGE No.
DATE / /

Q. Given string S of A, B & Cs, return a and b such that
A's in $S[:a]$ = B's in $S[a+1:b]$ = C's in $S[b+1:]$
Use $O(n)$ time and $O(1)$ space
$O(n)$ space also works.

This doesn't always have an answer.
Find counter example for it ⇒ BBBBAAC

FindRank in $O(n)$ time → Approach 2
Randomised FindRank
- Pick a pivot element randomly
- Partition the array (as in quick sort)
- What is the rank of pivot?



From this it can be inferred that, Rank(A[k]) = $r-(k+1)+1+1$
$= r-k+1$

∴ If $r-k+1$ == Rank that we want (r_{res}) then,
return k
else if $r-k+1 > r_{res}$, then,
pivot must lie in right half because we need a bigger no.
else
pivot must lie in left half but r_{res} will be subtracted by $(r-k+1)$ i.e. $r_{res} = r_{res} - (r-k+1)$

[Code on P.T.O] This is because if we go to left half we change $r=k-1$
[Explanation P.T.O] so, we have to modify rank in array $A[l...r]$ to rank in array $A[l...k-1]$.
~~If rank is $(r-k+1)$ in $A[l...r]$, then in $A[l...k-1]$ it will be~~

PAGE No.
DATE / /

```
RFindRank(A[], l, r, rank){
  if (l<r){
    p = rand() % (r-l+1) + l;
    swap(A[l], A[p]);
    k = partition(A, l, r);
    if (rank == r-k+1)
      return k;
    else if (rank > r-k+1)
      return RFindRank(A, l, k-1, rank-(r-k+1));
    else
      return RFindRank(A, k+1, r, rank);
  }
  return r; // Case where l==r ⇒ rank=1
}
```

// This can be anything, -1 as well.
// This just tells us that given rank does n't exist in A[l...r]

- When we go to the right side, we are ignoring the smaller nos. which doesn't affect the ranks.
- When we go to the left side, we are ignoring larger nos which contribute to the rank.
- No. of nos. b/w k and r is r-k+1 and we are ignoring these because we go to (l, k-1), so we need to find rank-(r-k+1)

<u>Time Complexity</u>

T(n) = No. of comparisons done

$T(n) \le n + \max_{0 \le k \le n-1} \{T(k), T(n-k-1)\}$

Partition()

k splits A into [0... k-1] & [k+1...n-1]
k n-k-1

We choose one half.
For worst case assume we take half which results in doing more comparisons.

PAGE No.
DATE / /

In Worst Case (Same as Quick Sort)

$$T(n) = \underbrace{n + T(n-1)}_{} = \Theta(n^2)$$

Partition returns $k=0$ and we go to right half with $n-1$ nos.

Assume partition gave two equal groups, then we go into either of the two halves with $n/2$ elements each.

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

$$= n + \frac{n}{2} + T\left(\frac{n}{4}\right) = n + \frac{n}{2} + \frac{n}{4} + \dots +$$

$$< 2n$$

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

- This is the best possible pivot but we don't know how to find one each time. So we relax the condition to define a good pivot.
- A pivot is good if its rank is b/w $n/4$ and $3n/4$.
- If we pick a random pivot, the probability that it will be good is $1/2$.
- On average, in every ~~it~~ alternate iteration we shall pick a good pivot.

- If we pick a good pivot we shall discard atleast $n/4$ nos., so atmost $3n/4$ nos. will be left for the next iteration.

$\therefore$ Considering two consecutive iterations. [one of them guaranteed good pivot]

$$T(n) \leq 2n + T(3n/4)$$

$$\leq 2n + 2 \cdot \frac{3n}{4} + T\left(\frac{9n}{16}\right)$$

$$\frac{1}{1-\frac{3}{4}} = 4$$

$$\leq 2n\left[1 + \frac{3}{4} + \frac{9}{16} + \dots\right]$$

$$\leq 8n \quad \left[\text{G.P with } a=1,\ r=\frac{3}{4} \Rightarrow S_\infty = \frac{a}{1-r} = 4\right]$$

$\therefore T(n) = \Theta(n)$ on average.

PAGE No.
DATE / /

Average Complexity using the other recurrence rel^n

$$T(n) \le n + \max\{T(k), T(n-k-1)\} \; ; \; 0 \le k \le n-1$$

$$\le \sum_{k=0}^{n-1} \frac{1}{n} * \left[n + \max\{T(k), T(n-k-1)\}\right]$$

prob.

$$\le n + \frac{1}{n}\sum_{k=0}^{n-1} \max\{T(k), T(n-k-1)\}$$

$T(0)$	$T(n-1)$
$T(1)$	$T(n-2)$
$\vdots$	
$T(n/2-1)$	$T(n/2)$
$T(n/2)$	$T(n/2-1)$
$\vdots$	
$T(n-1)$	$T(0)$

$$\sum \text{max values} = 2\sum_{k=n/2}^{n-1} T(k)$$

$$\therefore T(n) \le n + \frac{2}{n}\sum_{k=n/2}^{n-1} T(k)$$

We want to prove that $T(n) < c \cdot n$ for some c & $n > n_0$

Proof by Induction.

Induction Hypothesis: $T(k) < c \cdot k$

$$\therefore T(n) \le n + \frac{2}{n}\sum_{k=n/2}^{n-1} c \cdot k \le n + \frac{2c}{n}\sum_{k=n/2}^{n-1} k$$

$$\cancel{\le n + \frac{2c}{n}\left[\sum_{k=1}^{n} k\right] \le n + \frac{2c}{n}\frac{n(n+1)}{2} \le n + nc + c}$$

$$\le n + \frac{2c}{n}\left[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right]$$

$$\le n + \frac{2c}{n}\left[\frac{n(n-1)}{2} - \frac{(n/2-1)\,n/2}{2}\right]$$

$n-1 \approx n$ & $n/2 - 1 \approx n/2$

$\leq n + c[n - n/4] \leq n + \frac{3cn}{4}$

We want $T(n) \leq n + \frac{3nc}{4} \leq c \cdot n$

$\therefore cn \geq n + \frac{3nc}{4}$

$\frac{c}{4} \geq 1 \Rightarrow c \geq 4$

$\therefore T(n) \leq c \cdot n$ for any $c \geq 4 \Rightarrow T(n) = O(n)$

<u>Deterministic FindRank</u>

Algorithm is the same as Randomized version. only difference is in choosing random pivot. Here we select pivot by calling a p = goodpivot(l, r); function.

How to implement this?

Divide array into groups of 5. Sort each group. Find median of each group and find median of these medians.

We can prove that this median of medians will be a good pivot, i.e, rank b/w n/4 and 3n/4.

Eg,

121	38	90		
87	35	88		
42	21	55	78	65
23	19	12		
7	11	2		

Descending Order ↓

→ Medians

Atleast n/4 elements $\geq 55 \Rightarrow$ rank $= n/4$ ⇓ Good pivot

Median of Medians

Atleast n/4 elements $\leq 55 \Rightarrow$ rank $= 3n/4 \Rightarrow$ Good pivot

PAGE No.
DATE / /

To summarize:

- Divide nos. into groups of 5; i.e, A[:4], A[5:9], ...
  [Simplifies no. of operations]
  If size < 5, then not O(n)
  size > 5, constants involved change but still O(n)
- Sort the groups
- Find median of the medians.

```
GoodPivot(int A[], l, r) {
    m = (r - l + 1)/5 ; j = l; i = l;   //m = No. of groups of 5
    while (i+4 <= r) {
        BubbleSort(A, i, i+4);
    swap(A[j], A[j+2]);   // Move median of group, i.e, A[i+2]
    j++; i+=5;                to the beginning of the array
    }                         i.e, group no., i.e, median of group 2
                              is put at A[l+2-1] = A[j]
    return FindRank(A, l, l+m-1, m/2);
}
                //Return median of medians
```

[Excluding 'r' index]

Time Complexity

- Divide nos. into groups of 5 → free
- Sort the groups → $\frac{n}{5} * \frac{5 \times 4}{2} = 2n$

  No. of groups * No. of comparsions by Bubble sort, $\frac{n(n-1)}{2}$
- Find median of medians,

  $\frac{n}{5} + T\left(\frac{n}{5}\right)$

$\therefore T(n) = \underbrace{2n + n/5 + T(n/5)}_{\text{Finding good pivot}} + \underbrace{n + T(3n/4)}_{\text{Finding rank}}$

Partition

New partition guarantee to discard n/4 nos.

$< 4n + T(n/5) + T(3n/4) \quad (n/5 \approx n)$

We can prove by induction that $T(n) = O(n)$.

[If we divided into groups of 3, then, $T(n) < 3n + T(n/3) + T(3n/4)$

$\left[< \frac{cn}{3} + \frac{c3n}{4} \Rightarrow > cn \text{ Bad!}\right]$

PAGE No.
DATE / /

Proof:

$T(n) = T(n/5) + T(3n/4) + 4n < cn$

I.H: $T(k) < c \cdot k$ where, $k < n$

$\therefore T(n) \leq c \cdot \frac{n}{5} + c \cdot \frac{3n}{4} + 4n$

$\leq n\left[\frac{c}{5} + \frac{3c}{4} + 4\right]$

$T(n) \leq n\left[\frac{19}{20}c + 4\right]$ & $T(n) \leq cn$

$\therefore \frac{19}{20}c + 4 \leq c$

$\frac{1}{20}c \geq 4$

$\therefore c \geq 80$

$\therefore T(n) \leq c \cdot n$ for all $c \geq 80$ and $n \geq n_0$

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

So for deterministic FindRank $c \geq 80$ whereas for randomized $c \geq 4$
This is why in practice the randomized ~~all~~ algo. will perform better than deterministic algo. despite same asymptotic notation.

Deterministic FindRank can be used to make Quick Sort efficiently deterministic as well but it too will have shortcomings as compared to randomized alternative.

23/9/21

Find K Largest Numbers

- Given an array find the K largest nos.
- FindRank() with rank = k will return array such that arr[k:] will be the k largest nos $\Rightarrow O(n)$ time and $O(n)$ space.
- But this requires us to ~~us~~ store all nos. which is capped by 10^6.
- So we have to get around this. If data is a stream of integers ($n > 10^6$)

PAGE No.
DATE / /

Algorithm:- Store first 2k nos. in an array, arr.
- Call FindRank(k) which returns ~~pk~~
- arr[k:] ~~arr[k]~~ is the k-largest nos.
- Overwrite first k nos. in ~~stre~~ array with next k nos. in i/p stream.
- Repeat this till will reach end of stream.

Can optimize this, by only storing those stream nos. which are bigger than median, i.e, arr[k].
Because only those nos. can ~~be~~ have rank < k.

$$\underbrace{O(k)}_{\text{Median finding}} \times \frac{n}{k}$$

$\frac{n}{k}$ ← Max No. of times to call FindRank()

⇓

$\Theta(n)$ time & $O(k)$ space

One more approach:
Start with array of size $m = 10^6$.
Assume it is filled with integers from stream.
Apply FindRank() ⇒ returns p.
If $p < k$ ⇒ All elements in A[:p] are definitely not part of kth largest nos., so they can be overwritten by stream integers.
If $p > k$ ⇒ Partition in A[:p] to find new p and repeat this till $p < k$ case happens.

$O(n)$ & $O(m)$ space

PAGE No.
DATE / /

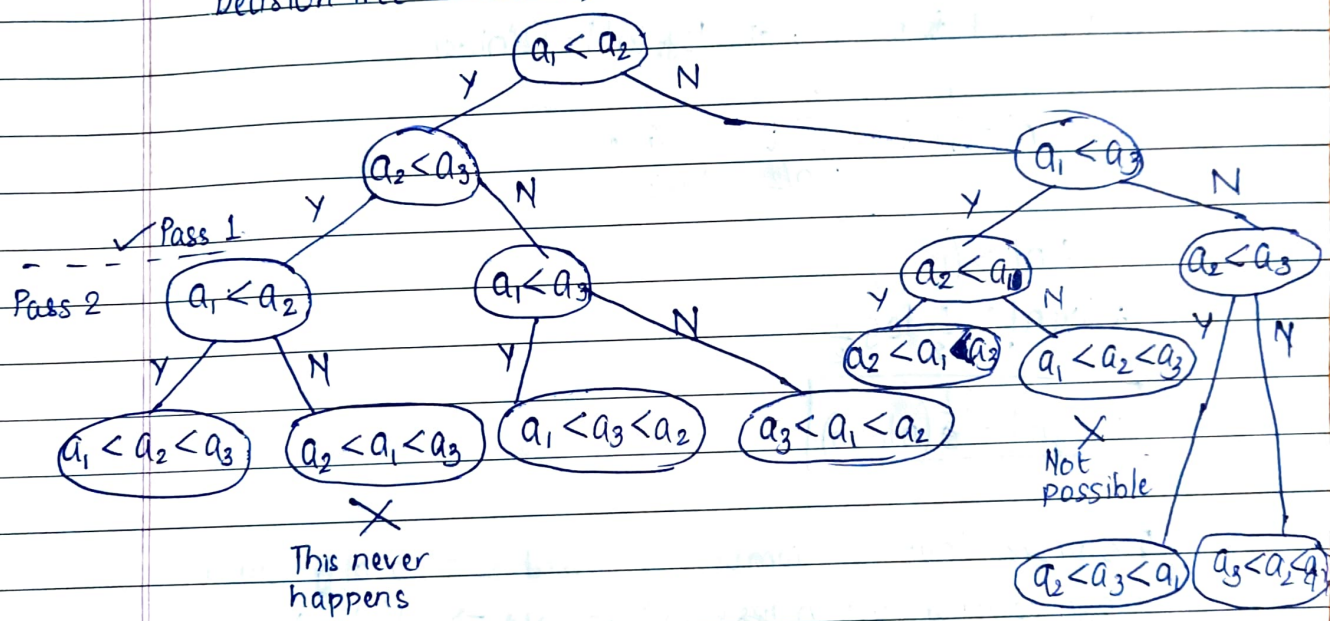
Lower Bound for Comparison-based Sorting

Decision Tree for Sorting Algo.

- Binary Tree where at each node ~~whe~~ we ask a Yes or No qn.
- If ans. is yes then we take the left child, else we take the right child.
- We repeat till we reach the leaf where algo. terminates.

Consider Bubble Sort on array with 3 elements a_1, a_2, a_3

Decision Tree for this is,



Full Binary Tree with level $= 3 \Rightarrow 2^3$ leaf nodes $- 2$ (not reachable leafs)
$= 6$ actual leaf nodes

When we reach a leaf node then we get sorted array.
So in worst case, no. of comparisons done = Height of tree (h)
$\therefore T(n)$ for any comparison-based sorting algo. ~~with~~ can only be O(min height of tree)

Let l be no. of leaf nodes then for binary tree, $l \leq 2^h$
$\therefore h \geq \log_2 l$
What should be value of l?
Atleast All valid arrangements of n integers $\Rightarrow n!$
$\therefore l \geq n!$, we say atleast because as in Bubble sort tree we can have invalid leaf nodes as well.

$\therefore$ No of comparisons $= h$

$= \log_2 l$

$\geq \log_2(n!)$

Now $n! = \prod_{i=1}^{n} i$

$\leq n^n$

$\therefore \log n! \leq n \log n$

$\therefore \log n! = O(n \log n) \rightarrow$ Not that useful for us

We want to prove that $\log_2(n!) = \Omega(n \log n)$

$$n! = \prod_{i=1}^{n} i > \prod_{i=n/2}^{n} i > \prod_{i=n/2}^{n} (n/2)$$

$$\therefore n! > (n/2)^{n/2}$$

$$\therefore \log n! > \frac{n}{2} \log \frac{n}{2}$$

$$\boxed{\therefore \log n! = \Omega(n \log n)}$$

This proves that any comparison-based sorting algo will do $n \log n$ comparisons in ~~bes~~ even best-case $\Rightarrow \Omega(n \log n)$

<u>Count Sort</u> $\Rightarrow O(n)$ time

Sorting on array of 0s & 1s

Count no. of 0s & 1s $\Rightarrow$ c0 & c1

Fill arr[:c0] = 0, arr[c1:] = 1

Assume ~~ar~~ A can contain m values (0, m-1), then

Store counts of each value i in C[i], C = Count array.

Fill A according to count.

But this algo is not stable sort, equal elements may or may not appear in original order.

The stable count sort uses a different approach.

$C[i] =$ ~~Cumulative~~ No. of elements $\leq i$

One pass to fill this array.

Then, start traversing A from the back,

if $A[i] == m$, then

$B[C[m]] = m$

$C[m]$-- // So that next ~~one~~ m occurs, it will be placed at $C[m]$.

Return B

B preserves order of equal elements ⇒ Stable Sort.

Pseudocode: ~~m = No. of unique values in~~ $(0, m-1)$ values in A

```
i=0 to m
    C[i]=0;
i=0 to n
    C[A[i]]++;
i=1 to m
    C[i] = C[i] + C[i-1];
i=n-1 to -1
    ~~B[C[A[i]]]~~ C[A[i]]--;  //Decrement count first because 0-based indexing
    B[C[A[i]]] = A[i];
i=0 to n
    A[i] = B[i]
```

$O(n+m)$ ~~O(n)~~ space & $O(n)$ time

Nos. have to be b/w 0 & $m-1$ ⇒ Integers & $m < 10^6$

Example for Stable Sort.

Given students sorted by roll number, sort them according to gender.

If sort is stable, then after sorting all boys ~~will~~ will be sorted according to roll no.

& same for all girls.

PAGE No.
DATE / /

Radix Sort

A = [742, 648, 764, 262, 183, 563]

Sort using stable Count Sort & LSB digit of each no.

∴ A = [742, 262, 183, 563, 764, 648]

Repeat but sort using middle digit

A = [742, 648, 262, 563, 764, 183]

[Note: Maintain sequence of equal nos.]

Repeat using MSB

A = [183, 262, 563, 648, 742, 764]

A is sorted!!!

But how?

If the MSBs are different digits then Count Sort guarantees proper sorting without ever looking at other bits.

But for 742, 764 ⇒ MSB = 7 ⇒ Use 10s place

So array must be sorted according to 10s place

But if digit at 10s place also equal then we need to sort according to 1s place.

∴ We start sorting from 1s place to ensure that all equality conflicts are already resolved when we reach MSB.

Pseudocode:

[k = Max No. of digits in ~~A~~ any no. in A, m = 10]

j = 1

while (--k > 0) {

  i = 0 to m

    c[i] = 0

  i = 0 to n

    c[(A[i]/j)%10]++;

  i = 1 to m

    c[i] += c[i-1];

  i = n-1 to -1

    B[--c[(A[i]/j)%10]] = A[i];

  i = 0 to n → A[i] = B[i]; j = j * 10;

}

$O(k) \times O(n)$

$\boxed{T(n) = O(nk)}$

PAGE No.
DATE / /

Bucket Sort

A = [742, 648, 764, 262, 183, 563]

~~Star~~ Sort using Count Sort with MSB

∴ A = [183, 262, 563, 648, 742, 764]

Array already sorted!

- Thing to notice is that if we sort from MSB, then if any two consecutive nos. have different MSBs then they are sorted.
- If they have same MSB then we have to sort just those nos. with same MSBs using next place (10s in this case).
- So we are essentially placing elements of A into 10 buckets depending on their MSBs and this process continues.

- Bucketing can have different approaches but they must satisfy certain properties. Any element that goes in bucket[i] ≤ bucket[i+1]

- Buckets will have constant no. of nos., so we can sort individual buckets and concatenate them to get final sorted array.

- Distributing A into buckets → O(n)
  Sorting a bucket with constant no. of nos. → O(1)
  ∴ T(n) = O(n)

[This seems a lot similar to Hashing but we have applied condition on the hash function]

25/9/21

Binary Heaps → Priority Queue

Operations: Add

Delete Min/Max

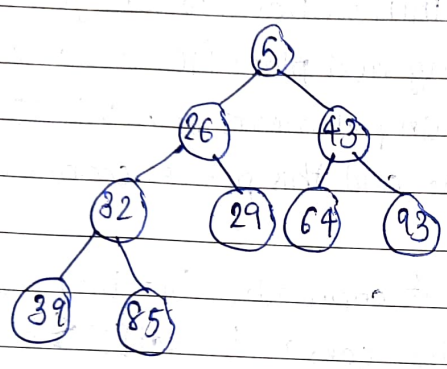
(Implementing min-heap but same logic follow for max)

Binary Heap (Min.)

It is a complete binary tree in which each node is associated a key. At each node of the tree, value of the key is smaller than or equal to the value of both of its children.

→ This is called the Heap Property

Eg,



Complete Binary Tree

Full Binary Tree till Second-last level + Last level filled from Left to right.

This can be represented as an array of size n.
[n = no. of nodes in tree]

0	1	2	3	4	5	6	7	8

H = [5 26 43 32 29 64 93 39 85]

So no need for tree, just use the array.

For node i,

parent = $(i-1)/2$ [Integer division]

l_child = $(2*i+1)$

r_child = $(2*i+2)$

Height of Binary Heap = $O(\log n)$

Proof: At l^{th} level (non-leaf) there will be 2^l nodes.

$\cancel{\therefore n > 2^l}$

$\therefore n > \sum_{i=1}^{h-1} 2^i$ (h = Height of tree)

$\therefore n > 2^h - 1$

$\therefore 2^h < n+1$

$\therefore h < \log_2(n+1) \longrightarrow (i)$

~~$h < \log_2 n$~~

~~$\therefore h = O(\log n)$~~

Finding upper bound for n, $n \leq \sum_{i=0}^{h} 2^i$

$n \leq 2^{h+1} - 1$

$2^{h+1} \geq n+1$

$\therefore h+1 \geq \log_2(n+1) \longrightarrow (ii)$

From (i) & (ii), $\log(n+1) - 1 \leq h < \log(n+1)$

Ignoring constants we can say $h = \Theta(\log n)$

Add Operation

Must maintain tree structure → Insert as last node in tree ⇒ Put at end of array

Must maintain heap property → Check ~~last~~ new node with parent. If parent not smaller, then, swap with parent and continue to sift this node up

SiftUp → BottomUpHeapify

[n = No. of nodes]

```
void Add(int H[], int X, int *n) {
    H[*n] = X;          // Insert new element X as last node in array
    *n = *n + 1;        // No. of nodes increased by 1
    BottomUpHeapify(H, *n - 1);   // Sift the new node up
}
```

(P.T.O)

PAGE No.
DATE / /

```
void BottomUpHeapify (int H[], int i){
    int p = (i-1)/2 ~~temp~~;
    while (p>-1){
        if(H[i]<H[p]){
            swap(H[i], H[p]);
            i=p;
            p=(i-1)/2;
        }
        else
            p=-1;
    }
}
```

Worst case we can go up to the ~~parent~~ root node.
∴ while loop runs for $h = \log n$ iterations
∴ Add() → $O(\log n)$.

<u>Delete Min Operation</u> → Remove root node
Swap Root & last Node
Delete last node (Decrement n by 1)
Sift new root down to maintain heap property.

SiftDown → TopDownHeapify
Replace ~~parent~~ node with smallest child node.
Terminate when node is smaller than both children.

```
void DeleteMin(int H[], int *n){
    int t; *n=*n-1;
    swap(H[0], H[n]);
    TopDownHeapify(H, 0, *n);
}
```

PAGE No.
DATE / /

```
void TopDownHeapify (int H[], int i, int n) {
    while (i<n){
        c=i;
        if (2*i+1<n && H[2*i+1] < H[c])
            c=2*i+1;
        if (2*i+2<n && H[2*i+2] < H[c])
            c=2*i+2;
        if (c!=i){
            swap(H[i], H[c]);
            i=c;
        }
        else
            i=n;
    }
}
```

In every iteration, level of node increases by 1.
∴ Worst case while loop runs for $\log n$ iterations
∴ DeleteMin → $O(\log n)$

DecreaseKey operation

```
void DecreaseKey (H[], i, X){
    H[i]=X;
    BottomUpHeapify (H, i);
}
```

IncreaseKey operation

```
void IncreaseKey (H[], i, X){
    H[i]=X;
    TopDownHeapify (H, i);
```

PAGE No.
DATE / /

BuildHeap → From given array

Algo 1: for i = 0 to n
Add(i) or BottomUpHeapify(H, i)

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

Algo 2: Start from last node that ~~heap~~ has child and Sift it down

```
i = ((n-1)-1)/2   // Parent of (n-1)th node
while (i >= 1) {
    TopDownHeapify(H, i);
    i--;
}
```

For a node at height i, no. of operations done is i at max.
No. of nodes at height i $\leq \frac{n}{2^i}$

$\therefore T(n) \leq \sum_{i=0}^{h} i * \frac{n}{2^i}$ ← [No. of nodes at height i]

[No. of operations for node at height i]

$\leq n \sum_{i=0}^{h} \frac{i}{2^i}$

$\leq 2n \Longrightarrow$ We have to prove, $\sum_{i=0}^{h} \frac{i}{2^i} \leq 2$

$T(n) = O(n)$

Consider, $\sum_{i=0}^{\infty} \frac{1}{2^i}$. We know that, $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

Diff. on b.s, $\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{-1\ (-1)}{(1-x)^2}$

Multiply b.s by x,

$$\therefore \sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$$

Put $x = 1/2 \Rightarrow \sum_{i=0}^{\infty} i \cdot \left(\frac{1}{2}\right)^i = \frac{1/2}{(1-1/2)^2}$

$$\therefore \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

$$\therefore \sum_{i=0}^{n} \frac{i}{2^i} < 2$$

$$\therefore T(n) \leq n\left(\sum_{i=0}^{n} \frac{i}{2^i}\right) \leq 2n$$

$$\therefore \underline{T(n) = O(n)}$$

Heap Sort

```
BuildHeap(H, n)
m = n; // Stores no. of elements in heap
for(i=0; i<n; i++)
    DeleteMin(H, &m); // Delete Min element in heap and place it at
                      ~~end of array~~ A[m] after which m-- happens
```

H will now contain numbers in decreasing order.

If use ~~min~~ max heap, it will be increasing order.

If you report nos. as they are deleted then it will be in ~~decreasing~~ increasing order (decreasing order if max heap).

$$T(n) = \underset{\text{BuildHeap}}{O(n)} + \underset{n \text{ calls to DeleteMin()}}{\Theta(n\log n)} = \Theta(n\log n)$$

DATE / /

Delete Operation

Deleting any arbitrary node i

- DecreaseKey (H, i, H[0] - 1); [DecreaseKey to one less than min. so its new root]
- DeleteMin (H);

∴ T(n) = Θ(logn)

[Concatenate both arrays and run build heap]

Two Binary Heaps ~~cannot~~ can only be merged in O(n) time

But this can be ~~done~~ improved by Binomial Heaps.

Q. Design a Data Structure which can perform the foll. operations efficiently,

Add()

DeleteMedian()

Median = Element with rank = n/2 + 1

[Same even if n is odd or even]

Approach 1: Maintain array and use FindRank() for delete

Add ⇒ O(1), Delete ⇒ O(n)

So if there are n operations, then Delete ⇒ $O(n^2)$

Approach 2: Maintain 2 heaps.

★★★ - One will have all the elements smaller than median & median itself ⇒ HS ⇒ Max Heap

★★★ - Other will have all elements greater than median ⇒ HB ⇒ Min Heap

HS → ns no. of elements, HB → nb no. of elements

★★★ Note: ns = nb or ns = nb + 1

(n = even) (n = odd)

Median = Root of HS

When median deleted then, if nb > ns ⇒ This condition violated, so root of HB is new median.

PAGE No.
DATE / /

```
DeleteMedian() {
    Median = HS[0];
    DeleteMax(HS, &ns);
    if (nb > ns) {        // New Median is root of HB
        DeleteMin(HB, &nb);
        Add(HS, HB[nb], &ns);
    }
}
```

Worst Case : DeleteMax + DeleteMin + Add

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

```
Add(x) {
    if (x > HS[0]) {      // Add to HB → Check if ns & nb condition violate
        Add(HB, x, &nb);
        if (nb > ns) {
            DeleteMin(HB, &nb);
            Add(HS, HB[nb], &ns);
        }
    }
    else {                // Add to HS → Check if ns & nb condition violate
        Add(HS, x, &ns);
        if (ns > nb + 1) {
            DeleteMax(HS, &ns);
            Add(HB, HS[ns], &nb);
        }
    }
}
```

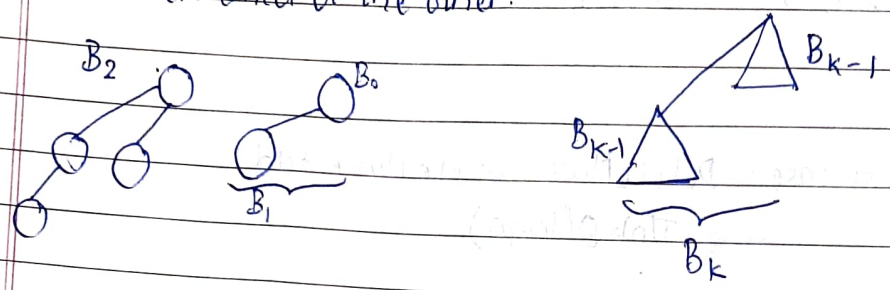
$\therefore T(n) = \Theta(\log n)$

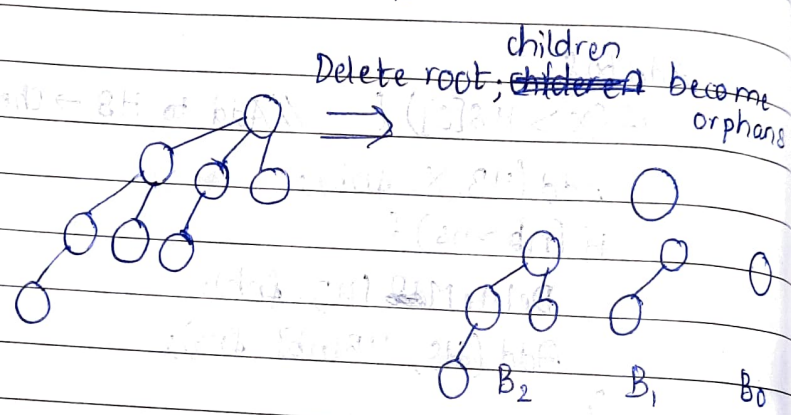
30/9/21

Binomial Heaps

What are Binomial Trees?

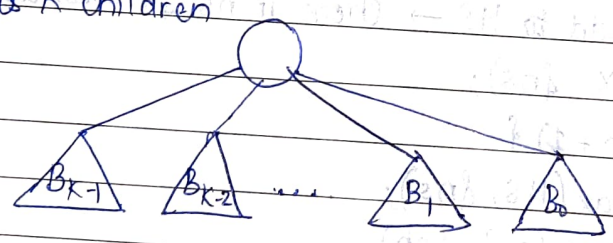
- A Binomial Tree of order k is defined inductively and denoted by
- B_0 is just one node.
- B_k is constructed by taking two copies of B_{k-1} and making one as the left child of the other.





Children of B_k root node are $B_0, B_1, \ldots, B_{k-1}$

$\therefore B_k$ has k children



Let $N(B_k)$ = No. of nodes in a ~~Bino~~ B_k

$\therefore N(B_k) = 2^k$

$\because N(B_k) = N(B_{k-1}) + N(B_{k-1}) = 2^{k-1} + 2^{k-1} = 2^k$

Let $H(B_k)$ = Height of B_k

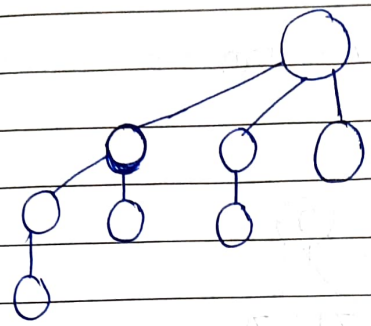
$\therefore H(B_k) = k$, $\because H(B_k) = 1 + H(B_{k-1}) = 1 + k - 1 = k$

PAGE No.
DATE / /

Let $L(k, i)$ be the no. of nodes at level i in B_k

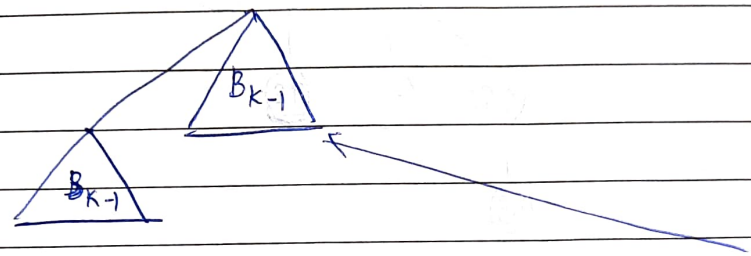
$\therefore L(k, i) = {}^kC_i$

Eg, B_3



L	N
0	1 $= {}^3C_0$
1	3 $= {}^3C_1$
2	3 $= {}^3C_2$
3	1 $= {}^3C_3$

Proof:



- ~~All nodes at lev~~ The levels of all nodes in the first B_{k-1} subtree remains same in B_k but all nodes in second B_{k-1} subtree will have their levels increased by 1 in B_k.
- So total no. of nodes at level i in B_k = No. of nodes at level i in B_{k-1} + No. of nodes at level $i-1$ in B_{k-1}

$$\therefore L(k, i) = L(k-1, i) + L(k-1, i-1)$$
$$= {}^{k-1}C_i + {}^{k-1}C_{i-1}$$
$$= \frac{(k-1)!}{(k-1-i)! * i!} + \frac{(k-1)!}{(k-1-(i-1))! * (i-1)!}$$
$$= (k-1)! \left[\frac{1}{(k-1-i)! * i!} + \frac{1}{(k-i)! * (i-1)!} \right]$$
$$= (k-1)! \left[\frac{k-i}{(k-i)! * i!} + \frac{i}{(k-i)! * i!} \right]$$
$$= \frac{(k-1)!}{(k-i)! * i!} [k-i+i]$$
$$= \frac{k!}{(k-i)! * i!}$$

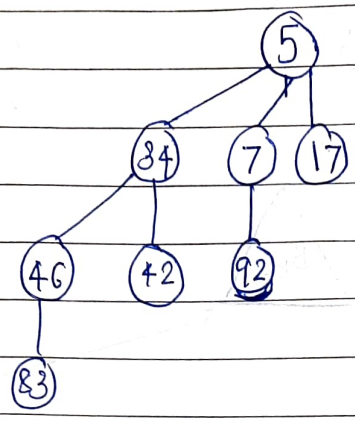
$\therefore L(k, i) = {}^kC_i \longrightarrow$ Binomial Coefficients $\rightarrow$ Why tree is called Binomial.

PAGE No.
DATE / /

Binomial Heaps

- A collection of Binomial Trees such that for each order there is atmost one B_k.
- At each node, heap property is satisfied, i.e, value of node is <= value of its children.

Eg,



☆☆ Given n and the Binomial Heap with n nodes. B_k will be present in the collection iff the kth bit in binary representation of n is 1.

[$\because n = \sum_i 2^i$ because B_k contains 2^k nodes $\Rightarrow 2^i = i^{th}$ bit in binary repr'n's decimal value]

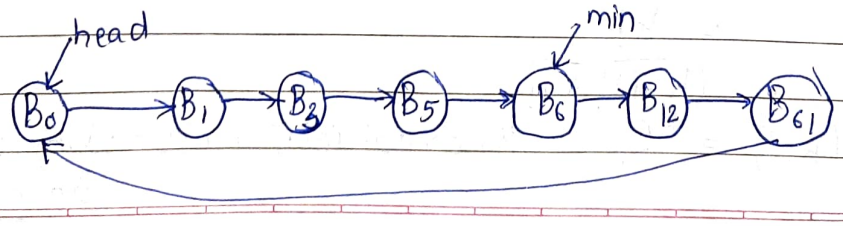
Eg, $n = 107 = 1101011$

$\therefore$ Binomial Heap will contain, B_0, B_1, B_3, B_5, B_6

Structure of a Binomial Heap

- The roots of the trees are connected via a Linked List → a root list (this will be circular list as well).
- The root list is such that the order of the trees increases as we go from left to right.
- Because of the heap property, the root of any tree is also its minimum, so the min. of the entire heap is the min. value in the root list, for which we'll maintain a min. pointer.

Eg,



PAGE No.
DATE / /

Structure of a Node

- Value
- Pointer to next node (for nodes in root list), else NULL)
- Parent Pointer (NULL for node in root list)
- Child List
- No. of children of the node.

Lemma: The no. of nodes in the root list is exactly equal to the no. of 1s in the binary representation of n which is atmost $O(\log n)$.
$\therefore$ Traversing the root list in one direction will take $O(\log n)$.

- The height of a Binomial Heap is defined as the maximum height of any node in the Binomial Heap.
- This max. height node has to be in the root list.
- Let it be root of B_k which has height $= k$

We know, no. of nodes in $B_k \leq$ no. of nodes in heap

$\therefore 2^k \leq n$

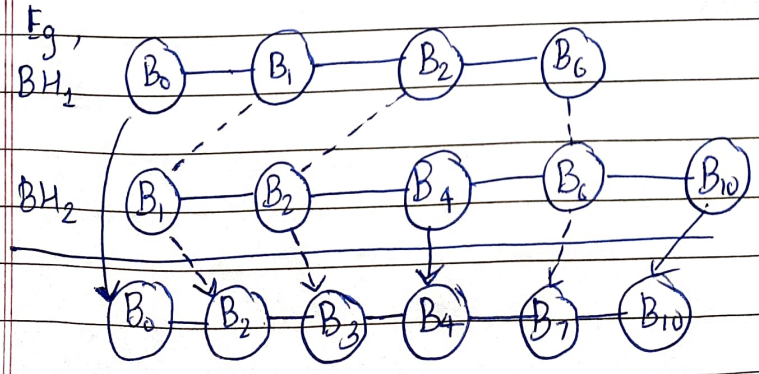
$\therefore k \leq \log n$

$\therefore$ Max. height of any tree in Binomial Heap is $O(\log n)$.
$\therefore$ Max. height of Binomial Heap is $O(\log n)$.

Binary Heap Operations

Union/Merge of Binomial Heap

Eg,



→ Start traversing both root lists from L to R.
→ If $(B_k)_1 \neq (B_k)_2$ put smaller B_k in list
If they are same then, Merge to get B_{k+1}

This is very similar to adding two binary nos.
In the previous example,

$BH_1 = 00001000111$
$BH_2 = 10001010110$
$\rightarrow 10010011101$

$BH_3 \Rightarrow B_{10} \rightarrow B_7 \rightarrow B_4 \rightarrow B_3 \rightarrow B_2 \rightarrow B_0$ (reverse order of)

~~This addition takes~~
No. of elements in root list = $\Theta(\log n)$
$\therefore$ Union also takes $\Theta(\log n)$ which is better than Binary Heap's $\Theta(n)$ time

Add Operation

- Create a new node $\rightarrow B_0$
- Merge this with the heap
- $\Theta(\log n)$ time

DeleteMin Operation

- Delete the node (root) pointed to by "min".
- Concatenate all its children,
  Eg, Say B_k was min. node $\rightarrow$ Delete it
  Its children $B_0, B_1, \ldots B_{k-1}$
  Concatenate them as, $B_0 \rightarrow B_1 \rightarrow \ldots \rightarrow B_{k-1} \Rightarrow \Theta(\log n)$
  This is also a Binomial Heap!
- Merge the two heaps.
  $\Theta(\log n)$ time

PAGE No.
DATE / /

Decrease Key

- Decrease node's value and call BottomUpHeapify (same as binary heap).
- $O(\log n)$ time.

Delete

Decrease value to 1 less than min. and call Delete Min

∴ Decrease Key + Delete Min

∴ $O(\log n)$ time

Increase Key

- Increase value and call TopDown Heapify
- But! This is not efficient because a node can have ~~any no of~~ many children, adding extra overhead for each level that we have to go lower for. (whereas in binary heap each node had just 2 children) $O(\log n * \log n)$
- Instead it will be better if we just delete that node and add a node with the new value.
- $O(\log n)$ time.

Build Heap

Algo 1: Add n nos. one by one → $O(n \log n)$

So you think!

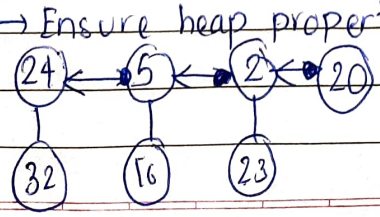
However it can be proved that this is $O(n)$

[Proof later!]

Algo 2: Eg, 24 → 32 → 5 → 16 → 23 → 2 → 20 → Array/Root List

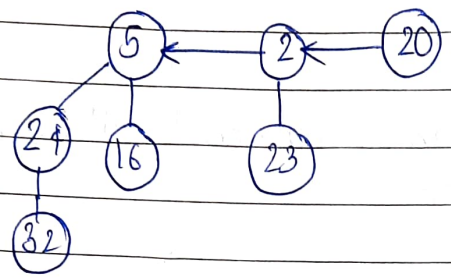
Array becomes initial root list

All nodes are B_0 → Not allowed → Start merging two nodes to make B_1's → Ensure heap property

∴ 

~~Traverse~~

Traverse new root list from the back and see if any two consecutive trees have same order. If yes, merge them while maintaining heap property. If not, then go to next tree.



Traverse new root list from the back and check if same order happens.
None here ⇒ Binomial Heap made ⇒ BuildHeap stops

Time Complexity

- In iteration 1, atmost $n/2$ merges happened to form atmost $n/2$ B_1's from n B_0's.
- In iteration 2, atmost $n/4$ merges can happen to form B_2's
- So in each iteration, the no. of merges reduces by half and we're only merging two trees, so merging itself is constant-time.

$$\therefore T(n) \leq \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots$$

$$T(n) \leq n$$

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

PAGE No.
DATE / /

Amortized Analysis

Stack with operations:

Push

Multipop(k) → Pops k elements from stack.

- We are gonna do n operations and Multipop operation is O(n).

∴ Total complexity of algo. is $O(n^2)$

- This is technically true but is the cost really $O(n^2)$?
- This situation happens when we have one costly operation that will not be called all the time but Asymptotic notation will consider it all the time.
- This is why Amortized Analysis required.

Amortized Cost ⇒ Assume it is like credit received for doing an operation.

Operation	Actual Cost	Amortized Cost
Push	1	2
Multipop(k)	k	0

Something that we assign

- For each push, we'll get 2 rs ⇒ 1 rs to pay for the actual push. Remaining 1rs will be stored for future pop.
- Since no. of pops can never exceed no. of pushes, we will never go into "debt" and for each pop we'll pay using the Rs. that was left over.

∴ In worst case, Total no. of pops = Total no. of pushes $= n + n = 2n$

∴ For n operations, $T(n) = O(2n)$

Amortized Analysis: Potential Function, Φ

$\Phi : D \rightarrow R^+$

Data Structure

Amortized cost of ith Operation, $\overline{C_i} = C_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Potential difference after } i^{th} \text{ operation}}$

Actual cost of ith operation

Data Structure after i^{th} operation

PAGE No.
DATE / /

$\therefore \bar{C_1} = C_1 + \phi(D_1) - \phi(D_0)$

$\bar{C_2} = C_2 + \phi(D_2) - \phi(D_1)$

$\vdots$

$\bar{C_n} = C_n + \phi(D_n) - \phi(D_{n-1})$

$$\sum_{i=1}^{n} \bar{C_i} = \sum_{i=1}^{n} C_i + [-\phi(D_0) + \phi(D_n)]$$

$\therefore$ If, $\phi(D_n) \geq \phi(D_0) \Rightarrow \sum_{i=1}^{n} \bar{C_i} \geq \sum_{i=1}^{n} C_i$

If, Potential difference between first & last operations is non-negative $\Rightarrow$ Amortized Cost $\geq$ Actual Cost

For this to remain true define ϕ such that, $\phi(D_0) = 0$

For the Stack Problem, ϕ = No. of elements in stack

$\therefore \phi(D_0) = 0$

Push adds one element $\Rightarrow$ Increases potential by 1

$\Rightarrow$ Potential Difference = 1

$\therefore$ Amortized Cost for Push $= \underset{\text{(Actual Cost)}}{1} + \underset{(p.d)}{1} = 2$

Multipop $\Rightarrow$ Deletes k elements $\Rightarrow p.d = -k$

$\therefore$ Amortized Cost for Multipop $= k + (-k) = 0$

$\therefore \bar{C_i} = 0$ or $2 \Rightarrow \bar{C_i} \leq 2$

$\therefore \sum_{i=1}^{n} \bar{C_i} \leq 2n$

Because, $\phi(D_n) \geq \phi(D_0)$ $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \bar{C_i}$

$$\therefore \sum_{i=1}^{n} C_i \leq 2n$$

$\therefore$ Actual Cost for n operations $\leq 2n = O(n)$

So Amortized Analysis can be used to make calculation easier.

This requires us to define a potential function that increases slightly when we do an easy/fast/cheap operation and should decrease considerably when we do an expensive operation. This is to compensate for the expensive operation that we are doing.

Amortized Analysis for Binary Increment

Binary Increment → 0000 → Algo.

0001

0010

⋮

0111

1000

⋮

1111

10000

```
Increment(A){
  i = 0
  while (A[i] == 1)
    A[i++] = 0;
  A[i] = 1;
}
```

- Flip all bits from LSB to 0 till 1st 0-bit encountered.
- Change that 0-bit to 1.
- Done.

In worst case, we will flip all bits like in 1111 → 10000

There are logn bits in total → One increment = O(logn)

∴ n increments = O(nlogn)

- This is correct but there exists tighter upper bound.
- Because worst case doesn't happen that often.

In this algo. we do flips, 1 → 0 ⇒ k flips (we don't know k)

0 → 1 ⇒ 1 flip (last one)

To determine k, for n operations,

In every alternate iteration → LSB flips (1→0) → n/2

In every 4th iteration → LSB+1-bit flips (1→0) → n/4

⋮ → n/8

Total no. of (1→0) flips = $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots \leq n$

∴ In n operations, (0→1) n times & (1→0) ≤ n times

PAGE No.
DATE / /

We can think of this more intuitively.
In each iteration we flip only 1 0-bit to 1
For n operations, we flip n 0-bits to 1s.
∴ Through n operations we can only flip atmost n 1-bits to 0s
[Similar to push-multipop situation, we can only pop as man
we have pushed. Likewise, we can only (1→0) as many as w
(0→1) which is n]

$\therefore T(n) \leq n + n = 2n$

<u>Potential Function</u>

Let $\phi : D \rightarrow$ No. of 1s in D

$\therefore \phi(D_0) = 0$

Amortized Costs: $\bar{c}_i(0 \rightarrow 1) = c_i(0 \rightarrow 1) + \phi(D_i) - \phi(D_{i-1})$

$= 1 + 1$

$\bar{c}_i(0 \rightarrow 1) = 2$

$\bar{c}_i(1 \rightarrow 0) = k \text{ flips} = c_i(1 \rightarrow 0) + \phi(D_i) - \phi($

$= k + (-k)$

$\therefore \bar{c}_i(1 \rightarrow 0) = 0$

$\therefore \bar{c}_i \leq 2$

$\therefore \sum_{i=1}^{n} \bar{c}_i \leq 2n$ and $\because \phi(D_0) = 0 \leq \phi(D_n)$

Actual Cost $= \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \bar{c}_i \leq 2n$

$\therefore$ Actual Cost $= O(n)$

PAGE No.
DATE / /

- The Binary Increment function is analogous to the Add() operation of Binomial Heap.
- A Binomial Heap with n nodes can be written in binary represent^n of n which will tell us which B_k's it contains.
- Inserting one node into this heap is ~~like~~ incrementing n by 1, ie, adding 1 to LSB of binary represent^n of n → Binary Increment.
- ~~So this amortized~~ Flips correspond to Merge operations in this case.
- So the amortized analysis can be reused to conclude that n insertions into Binomial Heap will take ~~at~~ 2n iterations = O(n).

<u>Dynamic Table Amortized Analysis</u>

- Essentially a vector data structure.
- ~~Something~~ Table that grows in size dynamically as per requirement.

- <u>Logic:</u> Create array in sizes of power 2.
  - Start with $2^0 = 1$ size
  - Upon trying to insert~~ing~~ second element, array is full, so create new array of size $(2^1 = 2)$, copy the old array into this and add new element at the end.
  - Upon trying to insert~~ing~~ third element, new array of size $2^2 = 4$ is created
  - And this process continues.

- Every time we have to allocate a new array and copy old array into it, we are doing O(n) for insertion.
- Otherwise insertion takes O(1).

$$\therefore C_i = \begin{cases} 1 \\ 1 + 2^k & \text{when } i = 1 + 2^k \end{cases}$$

So with n operations in worst case we can do $\frac{n}{2} + 1$ operations.

$\therefore T(n) = O(n^2)$

But ofc, thats not the case ^^

PAGE No.
DATE / /

$C_i = 1 + 2^k, \quad i = 1 + 2^k$

$\therefore \sum_{i=1}^{n} C_i = n + \sum_{k=0}^{\log n} 2^k$

Because at worst array size will be incremented by power of 2, logn times.
Also, $i = 1 + 2^k \Rightarrow$ ~~$k = (i-1)$~~
when $i = n$, $k = \log_2(n-1) \approx \log_2(n)$

$\leq n + 2^{\log n} \leq n + 2^{\log n + 1}$

$\therefore \sum_{i=1}^{n} C_i \leq n + 2 \cdot 2^{\log n}$

$\therefore \sum_{i=1}^{n} C_i \leq 3n \Rightarrow$ So we proved actual cost is $O(n)$

Amortized Analysis for this,
Actual Costs, Simple Insertion $\rightarrow 1\ (C_i)$
Resize & Insertion $\rightarrow C_i = k + 1$

[Assume k elements were array]

Potential Function $\rightarrow$ Gotta think on it

ϕ_1 : No. of filled slots − No. of empty slots

Simple Insertion $\rightarrow \overline{C_i} = C_i + D_i - D_{i-1} = 1 + 2 = 3$ (Filled − Empty)

~~$= 1 + [(k+1 - (k$~~

Resize & Insertion $\rightarrow \overline{C_i} = C_i + D_i - D_{i-1}$

$= k + 1 + \{[k+1-(k-1)] - [k-0]\}$

$= \cancel{k} + 1 + \cancel{k} + 1 - \cancel{k} + 1 - \cancel{k}$

$\overline{C_i} = 3$

This ϕ decreases to 2 from k when we do costly insert and increases by 2 when we do cheap insert.

$\therefore$ It satisfies our criteria for a good ϕ

Also in this case, $\overline{C_i} = 3$ any case

$\therefore \sum_{i=1}^{n} \overline{C_i} = 3n$ & $\because \phi(D_0) = 0$ & $\phi(D_n) \geq \phi(D_0)$

Actual Costs $= \sum_{i=1}^{n} C_i = 3n \Rightarrow O(n)$

$k+1 + k+1 - 2k+1 - k$

PAGE No.
DATE / /

Φ_2 : 2 * (No. of filled slots) − Size of array

∵ Array is going to be atleast half-filled all of the time,

$\Phi_2(D_i) \geq 0$ & $\Phi_2(D_0) = 0$

[Array of size 0 ⇒ Filled = 0 & size = 0 ⇒ 2 * 0 − 0 = 0]

Simple Insert → $\overline{C_i} = 1 + [2] = 3$

Resize & Insert → $\overline{C_i} = k + 1 + \{[2(k+1) - 2k] - [2(k) - k]\}$

$= \cancel{k} + 1 + \cancel{2k} + 2 - \cancel{2k} - \cancel{2k} + \cancel{k}$

$\overline{C_i} = 3$

Same analysis as with Φ_1.

So there can be multiple valid Potential Functions.

<u>Summary</u>: If we are able to define a Potential Function such that it increases slightly for cheap operation and decreases considerably for an expensive operation, then, there is merit to using Amortized Analysis.

- We cannot guarantee anything for a single operation, but for a sequence of operations, Amortized Analysis will be able to give us a proper bound by accounting how often do the cheap and expensive operations occur.

8/9/21

Fibonacci Heaps

Summary of Heap studied till now + Fibonacci Heap

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap
Add	O(1)	O(logn)	O(logn)	$\boxed{O(1)}$
Delete Min	O(n)	O(logn)	O(logn)	* O(logn)
Min	O(1)	O(1)	O(1)	O(1)
Decrease Key	O(1)	O(logn)	O(logn)	* $\boxed{O(1)}$
Delete	O(n)	O(logn)	O(logn)	* O(logn)
Build Heap	O(n)	O(n)	O(n)	O(n)
Union	O(1)	O(n)	O(logn)	$\boxed{O(1)}$

Doubly Circular Linked List ~~with min pointer~~

[Deletes are still O(n) because we have to find new min. after delete]

Note: * = Amortized Costs!

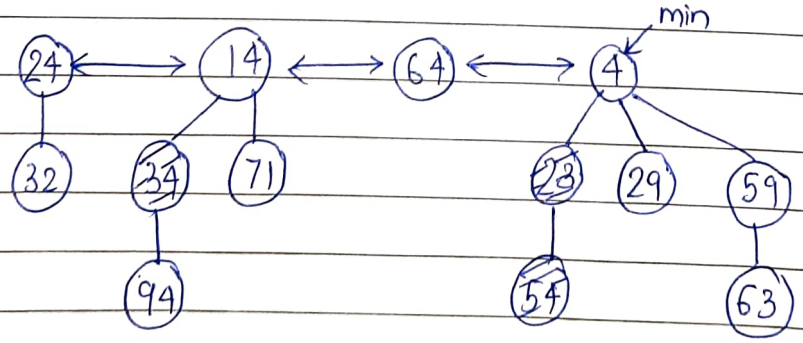
Fibonacci Heap Intro

- It is a collection of Trees.
- At each node, Heap property is satisfied → Parent > Child nodes
- No strict condition on the collection of trees as was with Binomial Heaps.
- The principle followed here is,

Be lazy when you are allowed to be lazy.

When you are forced to do something, do more than what is required

Eg,



[⊗ = Marked Node]

- Roots are connected by Doubly Circular LL.
- Trees have no restriction other than Heap property.

PAGE No.
DATE / /

- So a Fibonacci Heap just requires a pointer to this min. node root, i.e, min pointer.
- Using min. pointer you can access everything → Root List + Any tree. Plus every node stores info. like no. of children.

More terms involved,
Rank(x) : No. of children of node x.
Rank(H): Max. rank of any node in Fibonacci Heap.
Trees(H) : No. of trees of H
Marks(H): No. of marked nodes of H [In eg, we had 3 marked nodes]
n: Total no. of nodes in H
[So every node stores a boolean flag to indicate whether its marked or not]

Potential Function for H
$\Phi(H) = Trees(H) + 2 * Marks(H)$
Prev. eg. had Trees(H) = 4 & Marks(H) = 3
$\therefore \Phi(H) = 4 + 2 \times 3 = 10$

Insert Operation
- Create a new node.
- Add to the root list.
- Change the min. pointer if required.

[This shows our lazy work principle]
O(1) time
$\Phi(H_{i+1}) = \Phi(H_i) + 1$ → [∵ Trees(H) increases by 1]

PAGE No.
DATE / /

Union Operation

- Just join the two root lists (doubly LLs).
- Min. of min_1 & min_2 is the new min pointer

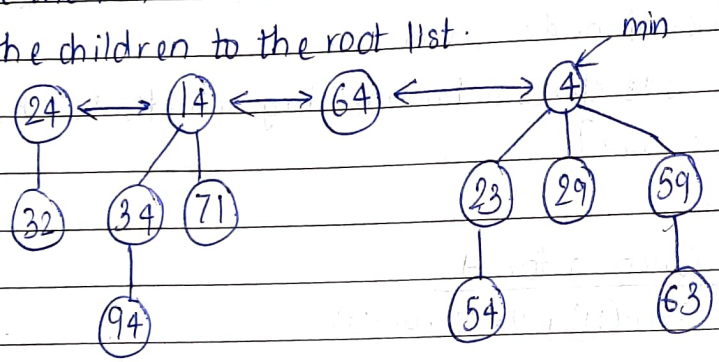
Build Heap

- Just call Add() n times → Root List with n nodes/trees.
- O(n) time.

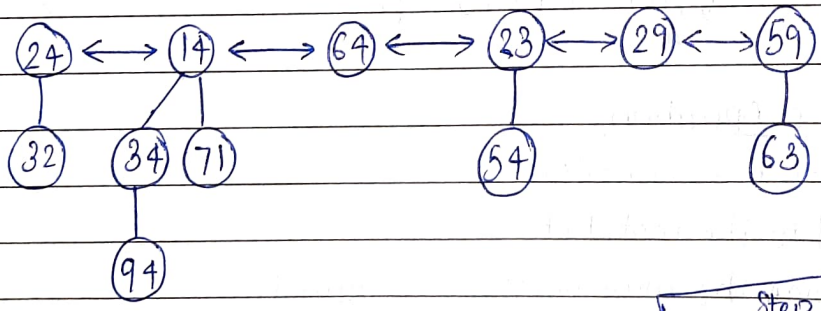
★★★ Delete Min

- Delete the Min
- Add the children to the root list.

Eg,



⇓



Now how do we update the minimum? ⇒ $\boxed{\text{Step Consolidation}}$

- We have to traverse the root list to find minimum.
- Our principle kicks in, forced to not be lazy ⇒ do extra work.
- While traversing the root list if we see two trees with the same rank, we merge them.
- But this requires us to keep track of rank of trees, eg, if we see tree with rank 2, then how will we know that there's another tree with rank 2 in the heap? Because root list is not sorted by rank (unlike Binomial Heaps)

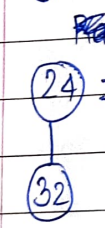
To keep track of ranks we maintain a table,

Rank	1	2	...	i	...	Rank(H)
Pointer						

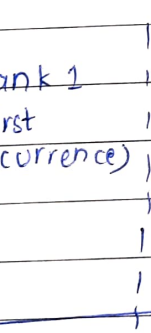
Pointer (Rank(i)) = Pointer to a tree with rank i

- So ~~we scan~~ initially all pointers are NULL and as we traverse the root list and find trees with given rank we update pointers in table at the correct index.
- If tree with given rank already exists, then merge them (smaller root node tree becomes parent of larger root node tree ⇒ O(1), rank increases by 1.
- Check if tree with new rank already exists, then repeat this process till there's no such tree left and pointer(rank(i)) = NULL

Eg. Trying this on the heap that we got,

i) 

⇒ Rank 1 (first occurrence)

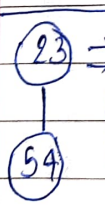
ii) 

Rank 2 (1st occurrence)

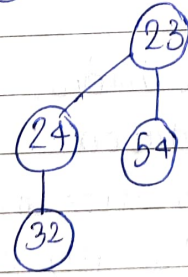
iii)

Rank 0 (1st Occurrence)

Rank	Pointer
0	~~N~~ 64
1	~~N~~ ~~24~~
2	~~N~~ ~~14~~
3	14

iv) 

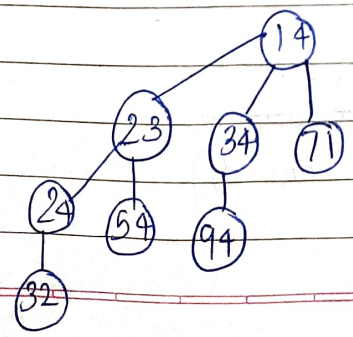
⇒ Rank 1 (2nd Occurrence) ⇒ Merge 23 & 24 trees



⇒ Rank 2 (2nd occurrence)

⇓

Merge 23 & 14 trees

v) 

⇒ Rank 3 (1st Occurrence)

PAGE No.
DATE / /

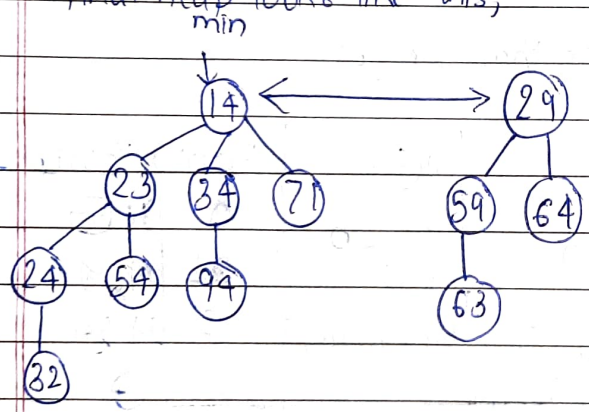
New Table:

Rank	0	1	2	3
Pointer	~~64~~	~~N~~ ~~29~~	~~N~~ 29	14

vi) 29 ⇒ Rank 0 (already exists) ⇒ Merge ⇒ 29 (64) ⇒ Rank 1 (doesn't exist)

vii) 59 (63) ⇒ Rank 1 (already exists) ⇒ Merge ⇒ 29 (59 (63), 64) ⇒ Rank 2 (doesn't exist)

∴ Final heap looks like this,



Summary:

- DeleteMin ⇒ Cut the children and attach them to root list
  Consolidation
- Will prove that both steps are O(Rank(H)) } Amortized Costs
- Will prove that ~~to~~ Rank(H) is O(logn)

Note that Actual Cost is O(n) for ~~Decrease Min~~ DeleteMin
But for sequence of operations, it is O(Rank(H))

PAGE No.
DATE / /

Analysis → Amortized

~~Act~~ Step 1: Cut the children and attach them to root list

~~Actual Cost = O(n) in the case where we do a single DeleteMin~~

Say we delete node X

Actual Cost → We take all its children, Rank(X), and attach them to root list

$\therefore$ Rank(X) * O(1)

$\therefore$ Actual Cost = O(Rank(X))

~~Change~~ $\phi(H) = \text{Trees}(H) + 2 * (\text{Marks}(H))$

[Increased by Rank(X) after this step]

[Remains the same after this step Unless we delete a marked node But it will never increase in any case, which is what matters]

$\therefore \phi(H_{i+1}) - \phi(H_i) = \text{Rank}(X) - 1$ → [For removed tree of min node]

$\therefore$ Amortized Cost = Actual Cost + Change in Potential

$= \text{Rank}(X) + \text{Rank}(X)$

$= O(\text{Rank}(H))$ → [$\because$ Rank(X) $\leq$ Rank(H)]

Step 2: Consolidation

Consider a merge operation,

Amortized Cost = $1 + (-1) = 0$

[Attach one ~~child~~ node as parent of other]

[No. of trees reduces by 1 & No. of marked nodes remains same]

So we can consider any merge as a free operation.

What matters is no. of merges

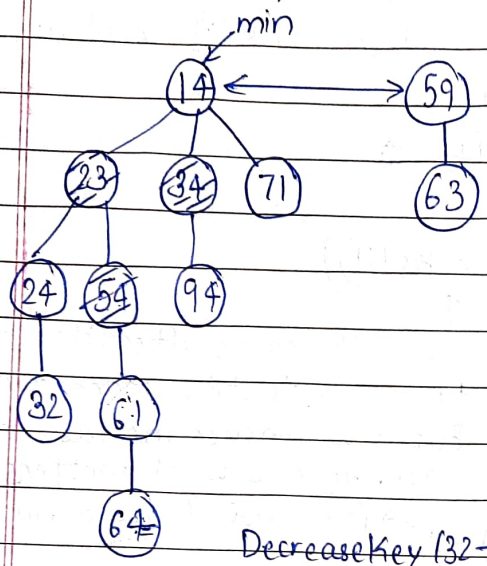
And this will be indicated by; No. of trees left after consolidation.

$\therefore$ Amortized Cost = No. of trees left after consolidation

$\leq$ Rank(H) + 1 [Because consolidation only allows one tree of given rank to be left in list & so at max we can have two trees of Rank(H) which will merge to give Rank(H) + 1]

Decrease Key

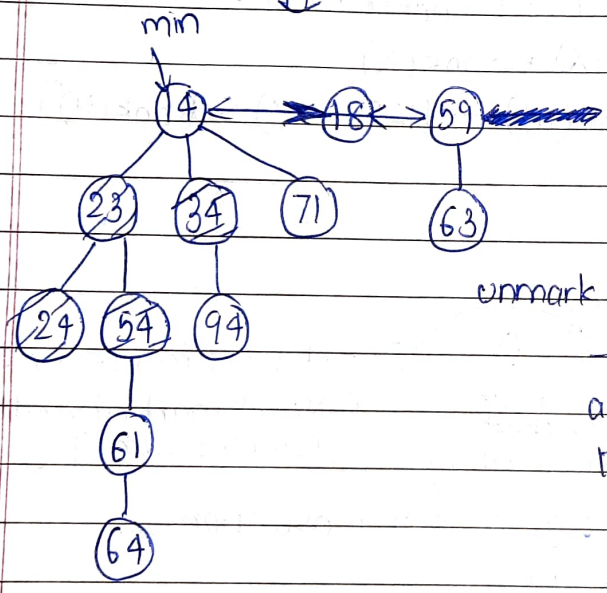
We cannot do BottomUpHeapify because the height of a tree in Fibonacci Heap can be O(n).



- Decrease key of 32 to 26
- No violation of Heap property
- Can just do it ✓

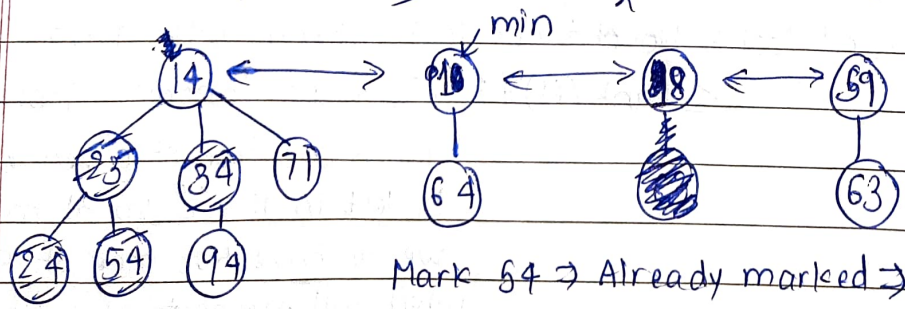
- Decrease key of 32 to 18.
- Heap property violated.
- Cut the node (and all its subtrees) and add it to root list.
- Mark the parent node (24) if its not

DecreaseKey (32→18)

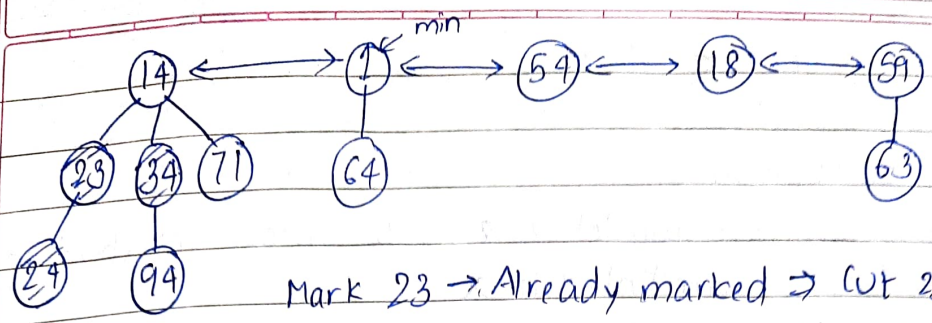


- But if parent node is already marked, then we cut it and add it to root list as well and unmark it & check do the same for its parent.
- This continues till we either reach an unmarked node or a node in the root list.

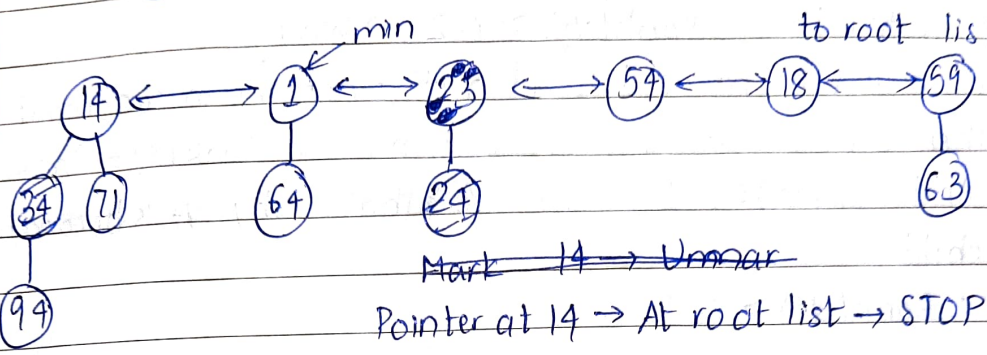
DecreaseKey (61 → 1) → Cut 61 (make it 1) and add to root list



Mark 54 → Already marked → Cut 54 and add to root list



Mark 23 → Already marked ⇒ Cut 23 and add to root list



~~Mark 14 → Unmar~~

Pointer at 14 → At root list → STOP!

<u>Analysis</u>

- If the parent node is marked, cut the node and add it to the root list and unmark it.
- When this happens, no. of trees ↑ by 1, no. of marked nodes ↓ by 1.

∴ Amortized Cost $= 1 + [1 + 2(-1)] = 0$

So it doesn't matter how many times this operation happens as we're incurring zero amortized cost, so its constant time.

- If the parent node is unmarked then we cut the child node and add it to the root list and mark the parent.

∴ Trees(H) ↑ by 1 & Marks(H) ↑ by 1

∴ $\phi(H)$ ↑ by $1 + 2*1 = 3$ ⇒ Amortized Cost $= 1+3 = 4$

∴ Using this 4 we can pay for 3 (4-1) cases of the first type, where parent is cut.

∴ Therefore for n operations, we will always be able to pay for the first case using the second case balance.

∴ DecreaseKey is $O(1)$.

~~Case~~ Note: DecreaseKey can add k new trees out of which ~~1 node was~~ all were marked and now will become unmarked because root and iteration stops on one unmarked node or root, so, no. of marked nodes changes by $-(k-1)$, no. of trees by k ⇒ $\Delta\phi = 2(-k+1) + k$
$= -k + 4$

PAGE No.
DATE / /

Rank of a Fibonacci Heap

Th^m:

Fix a point in time.

Let x be a node and let $y_1, \ldots, y_k$ denote its children in the order they were linked to x.

Then, for $i > 1$, $Rank(y_i) \geq i-2$ $\Rightarrow$ Why?

When y_i was made child of x (this happens in consolidation step) then that means at that point in time, $Rank(x) = Rank(y_i)$

And $Rank(x) \geq i-1$ because that's why y_i is becoming i^{th} child.

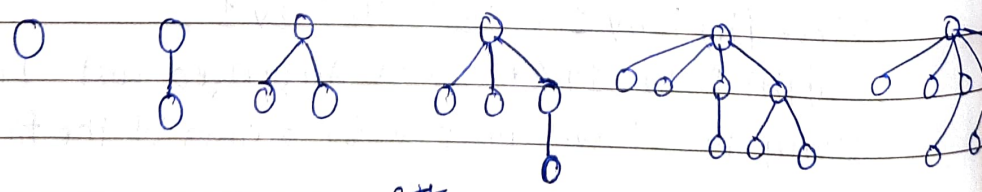
$\therefore Rank(y_i) \geq i-1$

Now x can have more children added and more operations happen. But y_i can only have one of its children deleted (via Decreasekey, because after this delete, y_i will become marked and deleting one more child of y_i (Decreasekey) will also cut y_i and add it to root list.

$\therefore Rank(y_i) \geq i-2 \rightarrow$ We'll use this below.

Let $F(r)$ = Min. no. of nodes in a Fibonacci Heap of rank r.

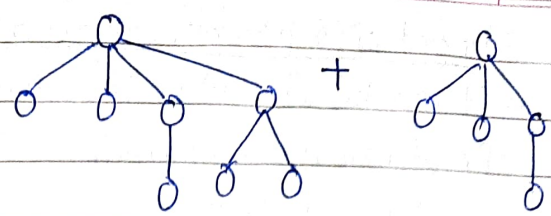
r	0	1	2	3	4	5
$F(r)$	1	2	3	5	8	13



[3rd child must have atleast $(3-2) = 1$ child]

$\therefore F(r)$ = Fibonacci Series!

Cut F(5) at last child



$\therefore F(5) \Rightarrow F(4) + F(3)$

$\therefore$ We can write, $F(r) = F(r-1) + F(r-2)$

[Ignore last child of F(r)] [last child of F(r)]

This is why Fibonacci Heaps are called so.

Consider a Fibonacci Heap with rank r and no. of nodes n.

$\therefore n \geq \underbrace{F(r)}_{\text{Min. no. of nodes in H with rank } r}$

$$\begin{aligned}
\therefore n &\geq F(r-1) + F(r-2) \\
&\geq F(r-2) + F(r-2) \quad [\because F(r-1) \geq F(r-2)] \\
&\geq 2F(r-2) \\
&\geq 2 \cdot 2 \cdot F(r-4) \\
n &\geq 2^k F(r-2k)
\end{aligned}$$

This terminates when $r = 2k \Rightarrow k = r/2$

$n \geq 2^{r/2} F(0) \geq 2^{r/2}$

$\therefore \frac{r}{2} \leq \log n$

$\therefore r \leq 2\log n$

$\therefore \text{Rank}(H) = O(\log n)$

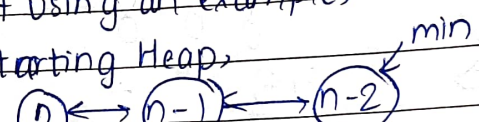
$\therefore \text{DeleteMin} = O(\text{Rank}(H)) = O(\log n)$

PAGE No.
DATE / /

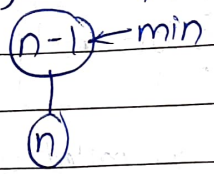
Height of a Fibonacci Heap

- It can be O(n) unlike the other heaps which were O(logn)
- Proof using an example,

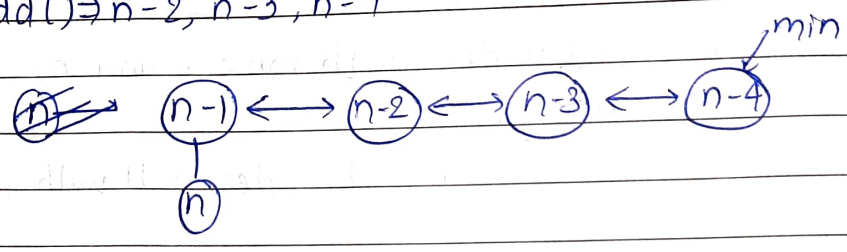
Starting Heap,



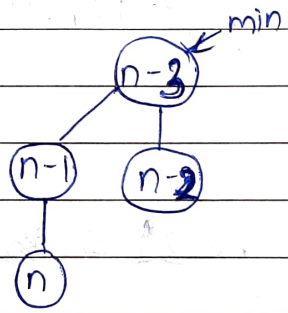
DeleteMin() → n-2 X ⇒ Consolidation ⇒ n & n-1 merge



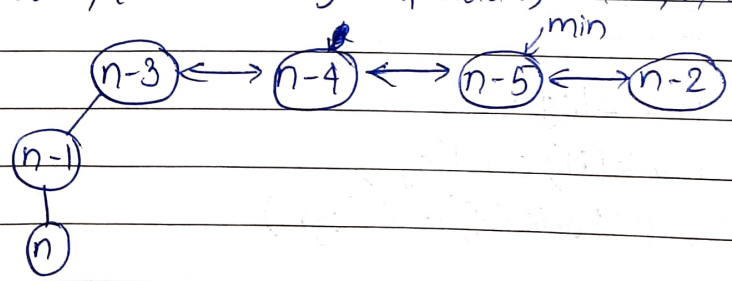
Add() ⇒ n-2, n-3, n-4



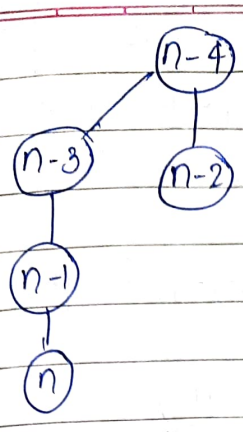
DeleteMin() ⇒ n-4 X ⇒ Consolidation ⇒ n-2 & n-3 merge
n-3 & n-1 merge



DecreaseKey (n-2 → n-4) & Add() n-2, n-5



DeleteMin() → n-5 X ⇒ Consolidation ⇒ n-4 & n-2 merge
n-3 & n-4 merge



Height increased from 2 to 3 (by 1)

- So in each iteration we DecreaseKey(), Add 2 two nos. then call DeleteMin ⇒ So ~~only one ele~~ n increases by 1 overall.
- In every iteration height increases by 1.
- ∴ For n such iterations, Height = $O(n)$.

★★★★ No. of marked nodes in a Fibonacci Heap can be linear

How?

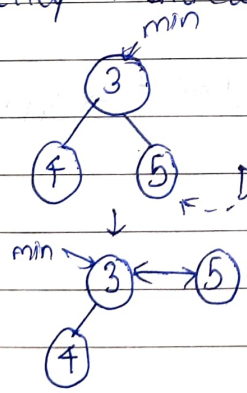
Find an example for this ⇒ Can come in mid-term

A ⇒ Heap with 1 Tree → n children → Each child has 1 child → DecreaseKey on each grandchild, gchild added to root list, child marked, n iterations, n new trees + n marked node

★★★★ DecreaseKey can increase or decrease the potential.

Find an example ⇒ Can come in mid-term.

→ DecreaseKey → Increasing Potential

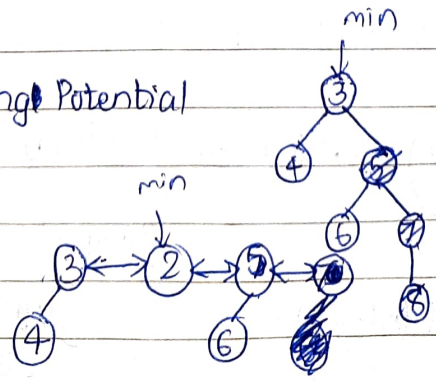


$\Phi = 1 + 2 * 0 = 1$

DecreaseKey (5 → 2)

$\Phi = 2 + 2 * 0 = 2$

DecreaseKey → Decreasing Potential



$\Phi = 1 + 2 * 2 = 5$

DecreaseKey (8 → 2)

$\Phi = 4 + 2 * 0 = 4$

PAGE No.
DATE / /

Master Theorem

$T(n) = a \cdot T(n/b) + f(n)$ $\quad b > 1,\ a \geq 1$

~~Case 1: $T(n) = \Theta(n^{\log_b a})$ if $f(n) < n^{\log_b a}$~~

~~Case 2: $T(n) = \Theta(f(n) * \log n)$ if $f(n) = n^{\log_b a}$~~ [Not formal thm but still usable]

~~Case 3: $T(n) = \Theta(f(n))$ if $n^{\log_b a} < f(n)$~~

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

Case 2: If $f(n) = \Theta(n^{\log_b a})$, ~~for some~~ then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$
and $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some $c < 1$ for all sufficiently large

then $T(n) = \Theta(f(n))$

[Note: $f(n)$ is work done at each level & $\log_b a$ is no. of levels
$\Downarrow$
$n^{\log_b a}$ is ~~no. of nodes~~ at leaf of recursion tree]

Practice

a) $T(n) = 3T(n/3) + 1$

$a = 3,\ b = 3,\ n^{\log_b a} = n^{\log_3 3} = n^1$

$f(n) = n^0 = n^{1-1} = n^{\log_b a - 1} \Rightarrow$ Case 1

$\therefore T(n) = \Theta(n^{\log_b a}) = \Theta(n^1)$

b) $T(n) = 2T(n/3) + n\log n$

$a = 2,\ b = 3,\ n^{\log_b a} = n^{\log_3 2} = n^0 = 1$

$f(n) = n^{\log n} > n^{\log_b a}$

$2f\left(\frac{n}{3}\right) \leq c \cdot f(n) \Rightarrow 2 \cdot \frac{n}{3} \log \frac{n}{3} \leq c \cdot n \log n$

$\frac{2}{3}(\log n - \log 3) \leq c \log n$

$\frac{2}{3} \log n \leq c \log n$

$c \geq \frac{2}{3} \Rightarrow c = \frac{2}{3} < 1$

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

$\frac{2207}{38600}$

$\frac{n}{3} + \frac{n}{4} = \frac{7n}{12} + \frac{n}{5} = \frac{47}{60}n$

PAGE No.
DATE / /

c) $T(n) = n + \frac{1}{n}\sum_{i=0}^{n-1} T(i) = \cancel{n + \frac{1}{n}\frac{n}{2}[2T(0) + (n-1)]} =$

$= n + \frac{1}{n}[T(0) + T(1) + \ldots + T(n-1)]$

$\leq n + \frac{1}{n}[n \cdot T(n-1)]$

$T(n) \leq n + T(n-1)$

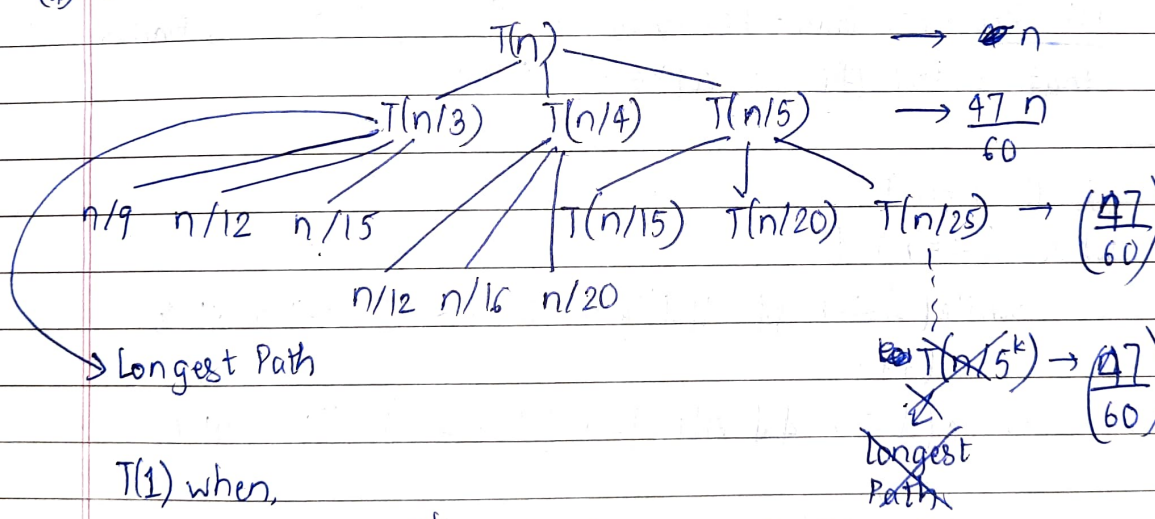
$\leq n + n-1 + n-2 + \ldots + T(1)$

$\leq \frac{n(n+1)}{2}$

$T(n) \leq n^2$

$\therefore T(n) = O(n^2) \Rightarrow$ Not tightest upper bound $\Rightarrow$ Turn few pages

(d) $T(n) = n + T(n/3) + T(n/4) + T(n/5)$



$T(1)$ when,

$\frac{n}{3^k} = 1 \Rightarrow n = 3^k \Rightarrow k = \log_3 n \rightarrow$ No. of levels

At each level $\left(\frac{47}{60}\right)^i n$ work is done.

$\therefore$ Total work done $= \cancel{k \cdot \left(\frac{47}{60}\right)^k n}$ $\cancel{\sum_{i=0}^{k}\left(\frac{47}{60}\right)^k n}$

$= \cancel{n\left[\frac{1\left[\left(\frac{47}{60}\right)^{k+1} - 1\right]}{1-}\right]}$ $= \cancel{n\left[\frac{1\left[1 - \left(\frac{47}{60}\right)^{k+1}\right]}{1 - \frac{47}{60}}\right]}$

No. of nodes in last level

$= 3^{\log_3 n} = n$ nodes

$\therefore$ Work done in last level $= n \times \Theta(1) = \Theta(n)$

$\therefore$ Total cost $= \sum_{i=0}^{k-1}\left(\frac{47}{60}\right)^i n + \Theta(n)$

$= \frac{60n}{13}$

(P.T.O)

$$= n\left[\frac{1\left[1-(47/60)^k\right]}{1-47/60}\right] + \Theta(n)$$

$$= \frac{60n}{13}\left[1-\left(\frac{47}{60}\right)^{\log_3 n}\right] + \Theta(n)$$

$$= \frac{60n}{13} - \frac{60n}{13}\left(\frac{47}{60}\right)^{\log_3 n} + \Theta(n)$$

$T(n) = \Theta(n) \rightarrow \left[\because \frac{47}{60} < 1 \Rightarrow \left(\frac{47}{60}\right)^{\log_3 n} << 1 \text{ (for large } n) \Rightarrow \text{can be ignored}\right]$

No. of BSTs & BTs

BSTs = Catalan No. = $\cancel{(2n+1)!}$ $\frac{(2n)!}{(n+1)!*n!}$

BTs = BST with no order = Arrange BST anyway = $\frac{(2n)!}{(n+1)!\,n!} * n!$

HW Problem: Given binary array A[0...n] of 0s & 1s. Find k such that A[:k] (#0s) = A[k+1:] (#1s)

```
l = 0, r = n-1
while (l <= r) {
    while( l <= r && A[l] != 0 && l < n) //Skip all 1s
        l++;
    while( l <= r && A[r] != 1 && r > -1) //Skip all 0s
        r--;
    if (l > r) //If crossing happened then 'r' is the answer
        return r;
    l++; r--; //Count 1 '0' on LHS and 1 '1' on RHS
}
```

PAGE No.
DATE / /

HW Problem: Given an array of As, Bs & Cs find i & j such that, $A[:i]$ (# As) = $A[i+1:j]$ (# Bs) = $A[j+1:n-1]$ (# Cs)

1. Count no. of Bs ⇒ countB , count = 0 (count of As & Cs)
2. l = 0, r = n-1;

```
while (l<=r) {
    while (l<=r && A[l] != 'A' && A[l] < n){ //Skip all non As
        if (A[l] == 'B')
            countB--;
        l++;
    }
    while (l<=r && A[r] != 'C' && A[r] > -1) { //Skip all non Cs
        if (A[r] = 'B')
            countB--;
        r--;
    }
    count++;
    if (count==countB)
        return {l, r-1};
}
return {-1, -1} // No solution
```

No. of BTs with traversal given:

No. of trees with given preorder
= No. of trees with given postorder
= No. of trees with given inorder
= No. of BSTs
= No. of unlabelled binary trees
= Catalan Number = $\frac{2n!}{(n+1)!\,n!}$

PAGE No.
DATE / /

(b) $T(n) = n + \frac{1}{n}\sum_{i=0}^{n-1} T(i) = n + \frac{1}{n}\{T(0) + \dots + T(n-1)\}$

$T(n-1) = (n-1) + \frac{1}{n-1}\{\underbrace{T(0) + \dots + T(n-2)}_{\text{Let this be } X}\}$

$\therefore \cancel{\frac{1}{(n-1)} T(n-1)} = X = (n-1)T(n-1) - (n-1)^2$

$T(n) = n + \frac{1}{n}\{X + T(n-1)\} = n + \frac{1}{n}\{(n-1)T(n-1) - (n-1)^2 + T(n-1)\}$

$= n + \frac{1}{n}\{nT(n-1) - (n-1)^2\}$

$= n + T(n-1) - \frac{(n-1)^2}{n}$

$= T(n-1) + \frac{\cancel{n^2} - \cancel{n^2} + 2n - 1}{n}$

$T(n) = T(n-1) + \frac{2n-1}{n} = T(n-1) + 2 - \frac{1}{n}$

$T(n-1) = T(n-2) + 2 - \frac{1}{n-1}$

$T(n-2) = T(n-3) + 2 - \frac{1}{n-2}$

$\vdots$

$T(n-k+1) = T(n-k) + 2 - \frac{1}{n-k+1}$ where $k = n \Rightarrow T(n-k) = 0$

$\therefore \cancel{T(n) = T(n-k)}$

$T(n) = 2 - \frac{1}{n} + 2 - \frac{1}{n-1} + 2 - \frac{1}{n-2} + 2 - \frac{1}{n-3} + \dots + 2 - \frac{1}{n-k+1}$ $\rightarrow n-k+1=1$

$= \underset{k=n}{2k} - \left(\frac{1}{n} + \frac{1}{n-1} + \dots 1\right) \rightarrow$ Harmonic Series $= O(\log n)$

$= 2n - O(\log n)$

$\boxed{T(n) = O(n)}$