An algorithm is any well defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

An algorithm is a sequence of computational steps that transform the input into the output. It is a tool for solving a well-specified computational problem.

Study of algorithms with the problem of sorting a sequence of numbers into non-descending order is our primary aim.

Defining a sorting problem.

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Input sequence is called an instance of the sorting problem. An instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Which algorithm is best for a given application if multiple algorithms are given?

- The number of items to be sorted.
- Extent to which items are already sorted.
- storage device available i.e main memory, disk or tape.
- Time utilised.

An algorithm is said to be correct, if for every input instance, it halts with the correct output.

INSERTION SORT

It is an efficient algorithm for sorting a small number of elements. It works in the same way as cards are sorted. we suppose that we have an empty left hand and the cards are lying on the table. we then remove one card at a time from the table and insert it into the correct position in the left hand. To find out the correct position for a card we compare it with each of the cards, already in the hand from right to left.

Pseudocode for insertion sort

It takes as a parameter an array A[1...n] where n represents the length of the array.

```
INSERTION-SORT (A)
1.  for j ← 2 to length [A]
2.     do key ← A[j]                    ↙ comment.
3.        ▷ Insert A[j] into the sorted sequence A[1...j-1]
4.        i ← j-1
5.        while i > 0 and A[i] > key
6.           do A[i+1] ← A[i]
7.              i ← i-1
8.        A[i+1] ← key.
```

The index j indicates the "current card" being inserted into the hand. Array elements A[1...j-1] constitute the currently sorted hand, and elements A[j+1...n] correspond to the pile of cards still on the table. The index j moves from left to right to sort the data.

Input Instance : 5, 2, 4, 6, 1, 3.

27/7/2013

```
                                  i
                                  j   key, j
                                  ↓   ↓
j = 2 to n                        5, 2, 4, 6, 1, 3
j = 2
Key ← A[2]  i.e  key ← 2
i ← j-1
i ← 1
while  i > 0 and A[1] > key (2)
                          5 > 2  true
A[i+1] ← A[i]                     i
i.e A[2] ← A[1]                   ↓
                          5, 5, 4, 6, 1, 3
                            ↺
i ← i-1
i ← 0
```

while i > 0 and A[i] > key

0 > 0 false

A[i+1] ← key

A[0+1] ← 2

A[1] ← 2 2, 5, 4, 6, 1, 3

i key j

j = 3 2, 5, 4, 6, 1, 3

key ← A[3] ie key ← 4

i ← j-1 ie i ← 2

while i > 0 and A[2] > key

5 > 4 true

A[i+1] ← A[i]

A[3] ← A[2] 2, 5, 5, 6, 1, 3

i ← i-1 ie i ← 1

while i > 0 and A[1] > key

2 > 4 false

A[1+1] ← key 2, 4, 5, 6, 1, 3

key, j

j = 4 2, 4, 5, 6, 1, 3

key ← A[4] ie key ← 6

i ← j-1 ie i ← 3

while i > 0 and A[i] > key

5 > 6 false

A[i+1] ← key

A[4] ← 6

i key, j

j = 5 2, 4, 5, 6, 1, 3

key ← A[j] ←[5] ie key ← 1

i ← j-1 ie i ← 4

while i > 0 and A[i] > key

6 > 1 true

A[i+1] ← A[i] ie A[5] ← A[4]

A[5] ← 6

2, 4, 5, 6, 6, 3

i ← i-1 ie i ← 3

while i > 0 and A[i] > key

5 > 1 true

A[i+1] ← A[i]

ie A[4] ← A[3] 2, 4, 5, 5, 6, 3

i ← i-1 ie i ← 2

while i > 0 and A[2] > key

4 > 1 true

A[i+1] ← A[i]

ie A[3] ← A[2] 2, 4, 4, 5, 6, 3

i ← i-1 ie i ← 1

while i > 0 and A[i] > key

2 > 1 true

A[i+1] ← A[i]

ie A[2] ← A[1] 2, 2, 4, 5, 6, 3

i ← i-1 ie i ← 0

while i > 0 and A[i] > key

false

A[i+1] ← key 1,2,4,5,6,3

ie A[1] ← 1

i key,j

j ← 6 1,2,4,5,6,3

key ← A[6] ie key ← 3

i ← j-1 ie i ← 5

while i > 0 and A[i] > key

6 > 3 true

A[i+1] ← A[i]

ie A[6] ← A[5] 1,2,4,5,6,6

i ← i-1 ie i ← 4

while i > 0 and A[4] > key

5 > 3 true

A[i+1] ← A[i]

ie A[5] ← A[4] 1,2,4,5,5,6

i ← i-1 ie i ← 3

while i > 0 and A[i] > key

ie A[3] > key

ie 4 > 3 true

A[i+1] ← A[i]

ie A[4] ← A[3] 1,2,4,4,5,6

i ← i-1 ie i ← 2

while i > 0 and A[i] > key

2 > 3 false

A[i+1] ← key

ie A[3] ← 3 1,2,3,4,5,6

for loop terminates - array sorted

Input instance : 31,41,59,26,41,58

j = 2

key,j

key ← A[j] ie key ← 41 31,41,59,26,41,58

i ← j-1 ie i ← 1

while i > 0 and A[i] > key

31 > 41 false

A[i+1] ← key

ie A[2] ← 41 31,41,59,26,41,58

i key,j

j = 3 31,41,59,26,41,58

key ← A[3] ie key ← 59

i ← j-1 ie i ← 2

while i > 0 and A[i] > key

41 > 59 false

A[i+1] ← key

ie A[3] ← 59 31,41,59,29,41,58

j = 4

key ← A[4] ie key ← 26

i ← j-1 ie i ← 3

while i > 0 and A[i] > key

ie A[3] > 26

ie 59 > 26 true

A[i+1] ← A[i]

A[4] ← A[3] 31,41,59,59,41,58

i ← i-1 ie i ← 2

while i > 0 and A[2] > key

41 > 26 true

A[i+1] ← A[i]

A[3] ← A[2] 31, 41, 41, 59, 41, 58

i ← i-1 ie i ← 1

while i > 0 and A[i] > key

31 > 26 true

A[i+1] ← A[i]

ie A[2] ← A[1] 31, 31, 41, 59, 41, 58

i ← i-1 ie i ← 0

while i > 0 and A[i] > key

0 > 0 false

A[i+1] ← key

A[1] ← key 26, 31, 41, 59, 41, 58

j = 5

key ← A[5] ie key ← ~~A[4+1]~~ = 41

j ← i-1

Analysing Algorithms

Analysing an algorithm means predicting the resources that the algorithm requires such as memory, communication bandwidth or H/w etc.

The time taken by the insertion sort procedure depends on the I/P (sorting more no's takes more time).

Insertion sort can take different amount of time to sort two input sequences of the same size depending on how much it is sorted. Therefore, running time of a program is defined as a function of the size of its input as well as the arrangement of no's.

The running time of an algorithm on a particular I/P is the no. of operations or the steps performed. A constant amount of time is required to execute each line of pseudocode.

One line may take different amount of time than the other line. We assume that each execution of the ith line takes c_i th time i.e a constant time. The analysis procedure includes the analysis with time i.e cost of each statement & the no. of times each statement is executed.

For each $j = 2, .., n$ where n is the length of the array we let t_j be the no. of times the while loop is executed for that value of j. When a for or while loop exists, the test is executed one time more than the loop body.

	Insertion_Sort (A)	Cost	Time
1.	for $j = 2$ to length [A]	c_1	n
2.	do key $\leftarrow$ A[j]	c_2	$n-1$
3.	$i \leftarrow j-1$	c_3	$n-1$
4.	while ($i>0$ & A[i] > key)	c_4	$\sum_{j=2}^{n} t_j$
5.	do A[i+1] $\leftarrow$ A[i]	c_5	$\sum_{j=2}^{n} t_j - 1$
6.	$i \leftarrow i-1$	c_6	"
7.	A[i+1] $\leftarrow$ key	c_7	$n-1$

Total running time is the sum of running times for each statement and is represent as $T(n)$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} t_j - 1 + c_6 \sum_{j=2}^{n} t_j - 1 + c_7(n-1)$$

1/8/2013

Best Case Analysis

The best case occurs when the array is already sorted. In this case of insertion sort for each $j = 2, 3, \ldots, n$, the $A[i] \leq key$ is found in line 4.

i has the initial value of $j-1$ in line 3. $\therefore$ $t_j = 1$ for $j = 2,3,4,\ldots,n$ and best case running time is calculated as

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4 n - c_4 + c_7 n - c_7$$

$$T(n) = n(c_1 + c_2 + c_3 + c_4 + c_7) - (c_2 + c_3 + c_4 + c_7)$$

$T(n) = an - b$ where $a = c_1 + c_2 + c_3 + c_4 + c_7$,

$b = c_2 + c_3 + c_4 + c_7$

In the above equation, a and b are constants and it's a linear function of n.

$$T(n) = an - b$$

$$T(n) = \theta(n)$$

Worst case Analysis.

Worst case running time occurs when the array is in reverse order or decreasing order.

In worst case each element A[j] is compared with each element in the entire sorted subarray $A[1 \ldots j-1]$ and $t_j = j$ for $j = 2,3,4, \ldots n$

$\therefore \sum_{j=2}^{n} t_j$ in line can be written as sum of n terms $\sum_{j=2}^{n} j = 2+3+4+\ldots$

$$= 2+3+4+\ldots \; n-1$$

$$= (1+2+3+4+\ldots) - 1$$

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

But for the first element there is no need for comparison & thus one is deducted from the sum of terms.

$\sum_{j=2}^{n} t_j - 1$ in line 5 can be written as

$$\sum_{j=2}^{n} j - 1 = 1 + 2 + 3 + \ldots + n - 1$$

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

$\therefore$ worst case running time for the entire insertion sort algorithm can be written as

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left\{\frac{(n)(n+1)}{2} - 1\right\} + c_5\left\{\frac{n(n-1)}{2}\right\} + c_6\left\{\frac{n(n-1)}{2}\right\} + c_7(n-1)$$

$$T(n) = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4\left\{\frac{n^2+n}{2} - 1\right\} + c_5\left\{\frac{n^2-n}{2}\right\} + c_6\left\{\frac{n^2-n}{2}\right\} + c_7 n - c_7$$

$$T(n) = c_1 n + c_2 n - c_2 + c_3 n - c_3 + c_4\frac{n^2}{2} + c_4\frac{n}{2} - c_4 + c_5\frac{n^2}{2} - \frac{c_5 n}{2} + c_6\frac{n^2}{2} - c_6\frac{n}{2} + c_7 n - c_7$$

$$T(n) = (c_4 + c_5 + c_6)\frac{n^2}{2} + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

$\therefore$ worst can running time can be expressed in a quadratic function of n as

$T(n) = an^2 + bn + c$

Total running time for the worst case can be expressed as $\theta(n^2)$ $[an^2+bn+c \approx \theta(n^2)]$.

Average case Analysis

Average case is as bad as worst case for the insertion sort. In this we choose n no.s & try to apply the insertion sort. we have to determine where in the subarray $A[1 \ldots j-1]$ to insert element $A[j]$.

In average case, half the elements in $A[1..j-1]$ are less then $A[j]$ and half are greater.
$\therefore$ half of the array is checked & thus $t_j = j/2$
If average case running time is computed it comes out to be same as worst case running time.

$$\sum_{j=2}^{n} \frac{j}{2} = \frac{1}{2}(2+3+4+\cdots+n)$$

$$= \frac{1}{2}(2+3+4+\cdots+n+1-1)$$

$$= \frac{1}{2}\left[\frac{n(n+1)}{2} - 1\right]$$

$$= \frac{1}{2}\left[\frac{n^2+n-2}{2}\right]$$

$$= \frac{n^2+n-2}{4}$$

Divide and Conquer Approach

The recursive algorithms follow divide and conquer approach. In divide and conquer approach, a big problem is divided into several sub problems & then solve the sub-problems recursively & at the end combine these solutions to create a solution to the original problem.

Merge sort algorithms follow the divide & conquer approach & involves three steps. First step is divide: divide n element sequence to be sorted into sub-sequences of n/2 elements each. Second step is conquer: sort two sub-sequences recursively using merge sort. Third step is combine: merge two sorted sub sequences to produce the sorted answer for the given problem.

The major task in merge sort is the merging of two sorted sub-sequences. The procedure that performs this task is

MERGE (A, p, q, r)

where A is the given array & p, q, r are index such that $p \le q < r$.

The two sorted subarrays are $A[p \ldots q]$ and $A[q+1 \ldots r]$.

3/8/2013

The original array is A[p.... r]

MERGE (A, p, q, r)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. ■ (create arrays L[1.... n_1+1] and R[1.... n_2+1]
4. for i ← 1 to n_1
5. do L[i] ← A[p + i − 1]
6. for j ← 1 to n_2
7. do R[j] ← A[q + j]
8. $L[n_1+1] \leftarrow \infty$
9. $R[n_2+1] \leftarrow \infty$
10. i ← 1
11. j ← 1
12. for k ← p to r
13. do if L[i] ≤ R[j]
14. then A[k] ← L[i]
15. i ← i + 1
16. else A[k] ← R[j]
17. j ← j + 1

The line 1 computes the length of the sub array A[p.... q] and line 2 computes the length of sub array A[q+1 ... r]. In line 3, left and right arrays are created of length $n1+1$ and n_2+1 In line 4 and 5, the array elements are copied from A[p ... q] into L[1...n1]. Similary in line 6 and 7, A[q+1... r] is copied to R[1...n2].

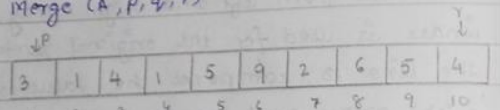
Line 8 and 9 are used to put the sentinal values at the end of each left and right arrays. In line 10 and 11 the indices are initialized to 1. The line 12 represents for loop used to add the values from left and right array. The k index is used for the original sorted array. The line 13, compares the value of left & right array and whichever is smaller is copied to kth index in line 14 to 17.

Initialization - before 1st iteration of loop, in line 12 the k = p such that k − p = 0 and i = j = 1, which means that L[i] and R[j] contains smallest element and not yet copied to the array A. The lines 1 to 3 and 8 to 11 takes constant time and line 4 to 7 takes $\theta(n_1 + n_2)$ time. In the line 12 to 17, the for loop executes for n times and all lines inside the for loop takes constant time.

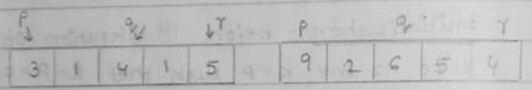
The merge sort MERGE_SORT(A, p, r) procedure sorts the elements in the sub array A[p..r]. If there are more than 1 element in the Array, then we q use the index q to divide the array into two parts to A[p..q] and A[q+1..r]. If p = r the array has only one element and is sorted.

MERGE_SORT (A, P, r)

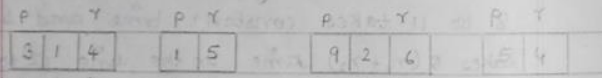
1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. Merge_Sort (A, P, q)
4. Merge_Sort (A, q+1, r)
5. Merge (A, P, q, r)



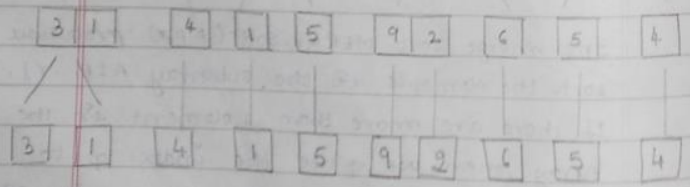
$$q = \left\lfloor \frac{p+r}{2} \right\rfloor = \left\lfloor \frac{1+10}{2} \right\rfloor = 5$$



$q = \left\lfloor \frac{1+5}{2} \right\rfloor = 3$ $\quad$ $q = \left\lfloor \frac{1+5}{2} \right\rfloor = 3$



$q = \left\lfloor \frac{1+3}{2} \right\rfloor = 2$ $\quad$ $q = \left\lfloor \frac{1+2}{2} \right\rfloor = 1$ $\quad$ $q = \left\lfloor \frac{1+3}{2} \right\rfloor = 2$ $\quad$ $q = \lfloor \frac{1+2}{2} \rfloor = 1$



Analysis

when an algorithm contains a recursive call to itself, its running time can be expressed by a recurrence relation. A recurrence of divide and conquer algorithm is based on three steps. - Divide, conquer and combine.

Let $T(n)$ be the running time of the problem of size n. if the problem size is small i.e $n \leq c$ where c is some constant, the total running time can be written as $\Theta(1)$ i.e constant time.

suppose division of the problem yields 'a' subproblems of size '1/b' each. in Merge sort 'a' and 'b' both are usually ~~two~~ 2.

$D(n)$ be the time to divide the problem into subproblems. and $C(n)$ be the time to combine the solutions of the subproblems.

∴ Recurrence can be written as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ a\left(T\left(\frac{n}{b}\right)\right) + D(n) + C(n) & \end{cases}$$

↓ no. of subproblems ↓ size of each subproblem

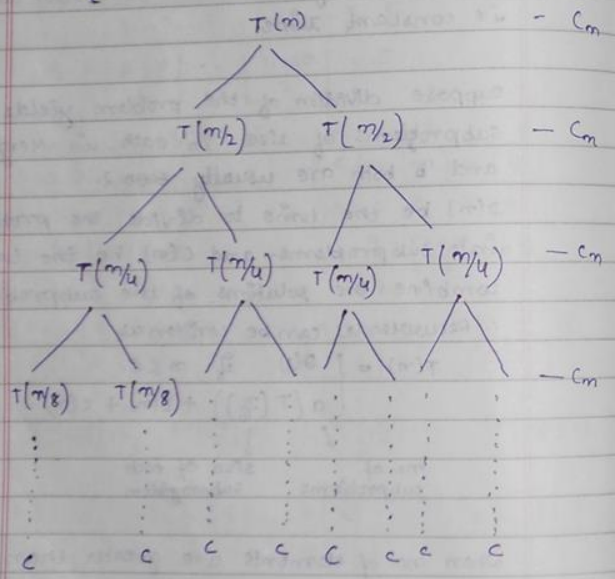
When no of elements are greater than 1, the running time is divided into 3 steps -

Divide - it helps to compute middle of the subarray & takes constant time i.e $D(n) = \Theta(1)$

Conquer - In this step, the two subproblems are solved each of size $n/2$ which contributes $2T(n/2)$ to the running time.

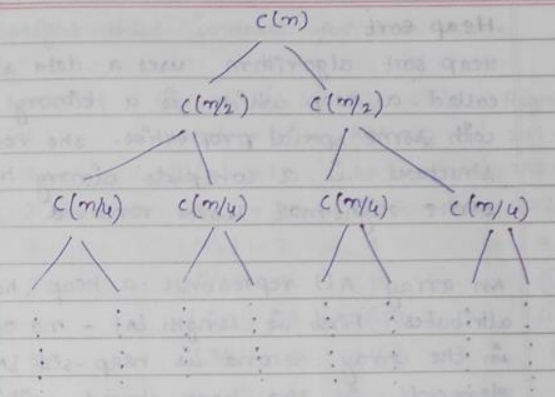
Combine - The solns of subproblems are combined i.e in total n elements are combined. so ~~C~~ $C(n) = \Theta(n)$

i.



No. of levels = $\log n$

The total cost is computed by adding the cost at all levels i.e $1Cn + 2Cn + 3Cn + \cdots + Cn\log n$
$= \Theta(n \log n)$



The level i has 2^i nodes, each contributing a cost of $c \cdot \frac{n}{2^i}$, so the total cost at each level is $2^i \cdot \frac{c \cdot n}{2^i}$ which is equal to cn. Let the total no of levels be m $\therefore$ total cost of each node at level m is $\frac{c \cdot n}{2^m}$. Let $\frac{cn}{2^m} = C$

Total cost of m^{th} level $= 2^m \cdot \frac{cn}{2^m}$

$\frac{cn}{2^m} = c \Rightarrow cn = 2^m \cdot c$
$\Rightarrow n = 2^m$

$\log n = m \log 2$
$\log n = m.$

No. of levels $= \log n + 1$
$\therefore$ total cost can be expressed as
$= cn[\log n + 1] = cn \log n + cn.$
$= C(n \log n + n) = \Theta(n \log n)$

Heap sort

Heap sort algorithm uses a data structure called a heap which is a binary tree with some special properties. The heap data structure is a complete binary tree with some rightmost leaves removed.

An array A[] represents a heap, has two attributes. First is length[A] - no of elements in the array. second is heap-size[A] - no of elements in the heap stored within the array.

Heap-size[A] ≤ length[A].

The root of the tree is A[1]. The indices of ~~of~~ parent i.e Parent(i), left child i.e left(i) right child i.e Right(i) can be computed as

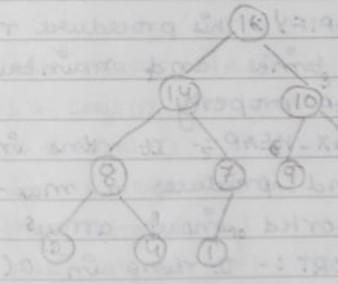
Parent(i)	Left(i)	Right(i)
return $\lfloor i/2 \rfloor$	return $2i$	return $2i+1$.

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

7/8/2013 since A[1] is the root node and can be drawn first. To check the parent of index 1, we calculate using parent function which return $\lfloor i/2 \rfloor$ where i is the index. The following table shows parent, left & right child indexes for each node

Index, i	Parent $\lfloor i/2 \rfloor$	Left, $2i$	Right, $2i+1$
1	$\lfloor 1/2 \rfloor = 0$	$2\times1 = 2$	$2\times1+1 = 3$
2	$\lfloor 2/2 \rfloor = 1$	$2\times2 = 4$	$2\times2+1 = 5$
3	$\lfloor 3/2 \rfloor = 1$	$3\times2 = 6$	$3\times2+1 = 7$
4	$\lfloor 4/2 \rfloor = 2$	8	9
5	$\lfloor 5/2 \rfloor = 2$	10	(11)
6	$\lfloor 6/2 \rfloor = 3$	(12)	(13)
7	$\lfloor 7/2 \rfloor = 3$	(14)	(15)

Values do not exist.



There are two types of binary heaps

1) MAX-HEAPS
2) MIN-HEAPS.

The values in the node satisfy the heap property. The MAX-HEAP property is that for every node i, other than the root, A[Parent(i)] ≥ A[i] (root of every node is having a greater value than its child node).

A MIN-HEAP property is that for every node i other than the root $A[Parent(i)] \leq A[i]$.

For heap sort, we use max-heaps. In a tree representation of a heap, the height of node in a heap is defined as the no. of edges on the longest downward path from node to a leaf.

Height of a heap is defined by height of its root. The height of n element heap is order of log n i.e $\Theta(\log n)$. There are ~~five~~ four procedures that are used

1. MAX-HEAPIFY :- This procedure runs in $O(\log n)$ times and maintains the MAX-HEAP property.
2. BUILD-MAX-HEAP :- It runs in linear time and produces a max-heap from an unsorted input array.
3. HEAP-SORT :- It runs in $O(n \log n)$ times and sorts an array in place
4. MAX-HEAP-INSERT
   HEAP-EXTRACT-MAX
   HEAP-INCREASE-KEY
   HEAP-MAXIMUM.
   These procedures run in $O(\log n)$ times and allows the heap data structure to be used as a priority queue.

Maintaining the Heap Property.

1. MAX-HEAPIFY is a procedure that manipulates MAX-HEAPS. Inputs are array A and index i. When MAX-HEAPIFY is called, it is assumed that, left of i and right of i are already MAX-HEAPS but the root may be smaller than its children & thus voilates the MAX-HEAP property.

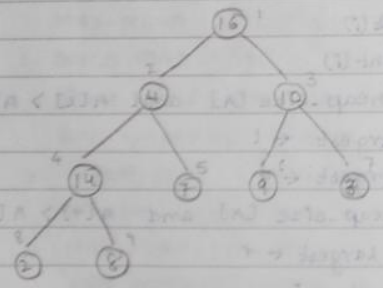
This procedure takes A[i] to its proper position in downward direction or in a subtree.

```
MAX-HEAPIFY (A, i)
1.  l ← Left(i)
2.  r ← Right(i)
3.  if l ≤ heap-size[A] and A[l] > A[i]
4.  then largest ← l
5.  else largest ← i
6.  if r ≤ heap-size[A] and A[r] > A[largest]
7.    then largest ← r
8.  if largest ≠ i
9.  then exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY (A, largest)
```

At each step, the largest of elements A[i], A[left(i)], A[right(i)] is determined

and its index is stored in the largest variable. If A[i] is largest, then subtree at node i is a max heap and the procedure terminates. Else one of the children has largest element and A[i] is swapped with A[largest]. Then MAX-HEAPIFY is recursively called at the largest index.

MAX-HEAPIFY on a subtree of size n at node i takes running time as $\Theta(1)$ + the time to run max-heapify on subtree. The size of child subtree is $2n/3$. ∴ running time of MAX-HEAPIFY is defined by recurrence equation $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$



i = 1

L = 2, r = 3

L ≤ heap-size [A] i.e 2 ≤ 10 and A[L] ≥ A[r]
true i.e 4 > 10
false

largest ← 3

if r ≤ heap-size [A] and A[r] > A[largest]
3 ≤ 10 and 10 > 10
true false

if largest ≠ i i.e 3 ≠ 1
A[1] ↔ A[3]

It cannot be called from the root node because the root node is not violating the MAX-HEAP property.

i ← 2

L ← 4, r ← 5

if 4 ≤ 10 and A[4] > A[5]
true 14 > 7 true.

largest ← 4

if r ≤ heap-size[A] and A[r] > A[largest]
5 ≤ 10 and 7 > 14
true false

if largest ≠ i
4 ≠ 2 then A[i] ↔ A[largest]
i.e A[2] ↔ A[4]

i ← 4

L ← 8, r ← 9

if 8 ≤ 10 and 2 > 4
true false

largest ← 4

if 9 ≤ 10 and 8 > 4
true true

largest ← 9

if 9 ≠ 4 true

then A[4] ↔ A[9]

∴ the running time of MAX_HEAPIFY on a node of height h can be written as ~~O(h)~~ O(h) where height of the tree is ~~log(n+1)~~ $\lfloor \log n \rfloor + 1$ ∴ running time is ~~O(log(n+1))~~

$O(\lfloor \log n \rfloor + 1)$

$\theta(\lfloor \log n \rfloor + 1) \Rightarrow \theta(\log n)$

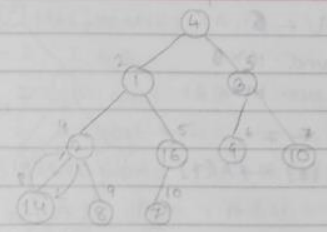
8/8/2013 **Building a Heap**

Max-Heapify in a button up manner is used to convert an array into a max-heap. Because the elements in the subarray $A[(\lfloor \frac{n}{2} \rfloor + 1) \ldots n]$ are all leaves of the tree and thus ∴ no need to process them. Build-max-heap algo goes through the remaining nodes & runs heapify on each one.

BUILD_MAX_HEAP

1. heap_size[A] ← length(A)
2. for i ← ⌊length[A]/2⌋ down to 1
3. do MAX_HEAPIFY(A,i)

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



n = 10

heapsize[A] = 10

for i = 5 to 1

i = 5 , MAX-HEAPIFY (A,5)

l ← 10 , r ← does not exist

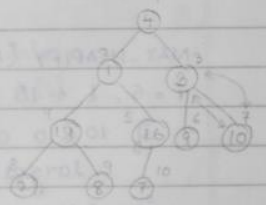
if 10 ≤ 10 and 7 > 16

true false

largest ← 5

if 5 ≠ i i.e 5 ≠ 5

i = 4 , MAX_HEAPIFY (A,4)



l ← 8 , r ← 9

if 8 ≤ 10 and 14 ≥ 2

true true

largest ← 8

if 9 ≤ 10 and 9 > 14

true false

if 8 ≠ 4 true A[4] ↔ A[8]

i = 3, MAX-HEAPIFY(A,3)

l ← 6 , r ← 7

if 6 ≤ 10 and 9 > 3

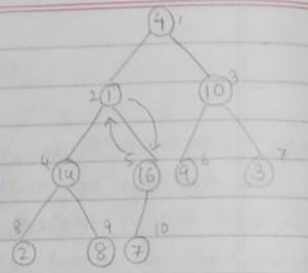
true ~~false~~ true

largest ← 3

if 7 ≤ 10 and 10 > 9
true true

largest ← 7

if 7 ≠ i A[3] ↔ A[7]



i = 2, MAX-HEAPIFY (A,2)

l ← 4, r ← 5

if 4 ≤ 10 and 14 > 1
true true

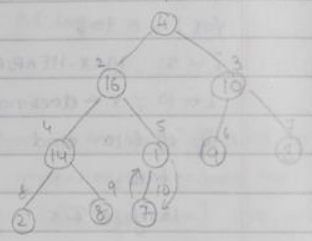
largest ← 4

if 5 ≤ 10 and 16 > 14
true true

largest ← 5

if 5 ≠ 2

A[5] ↔ A[2]



MAX-HEAPIFY (A,5)

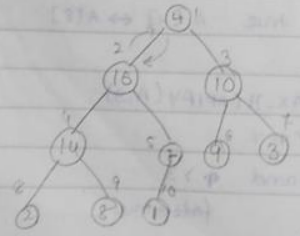
i = 5, l ← 10, r ← does not exist

if 10 ≤ 10 and 7 > 1 true

largest ← 10

if r ≤ 10 and A[r] > A[10] false

if 10 ≠ 5 then A[10] ↔ A[5]



i = 1, MAX-HEAPIFY (A,1)

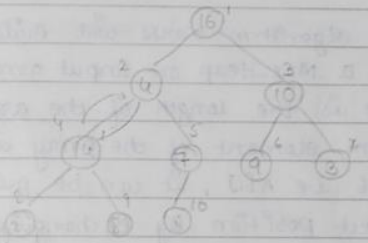
l ← 2, r ← 3

if 2 ≤ 10 and 16 > 4 true

largest ← 2

if 3 ≤ 10 and 10 > 16 false

if 2 ≠ 1 then A[2] ↔ A[1]



MAX-HEAPIFY (A,2)

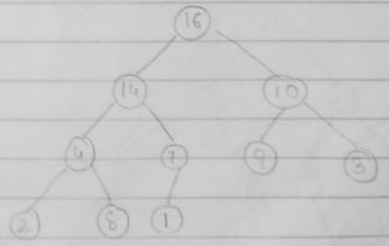
i = 2, l ← 4, r ← 5

if 4 ≤ 10 and 14 > 4 true

largest ← 4

if 5 ≤ 10 and 14 > 1 false

if 4 ≠ 2 true A[4] ↔ A[2]



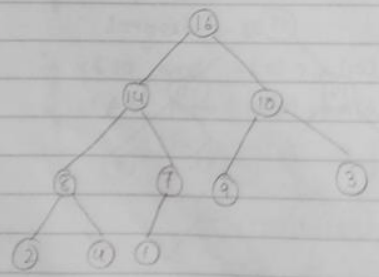
Running time of Build-Max-Heap is O(n).

```
HEAP_SORT ( A)
1. BUILD-MAX_HEAP (A)
2. for i = length [A] down to 2
3.    do exchange A[1] ↔ A[i]
4.       heap_size [A] ← heap_size [A] -1
5.       MAX_HEAPIFY (A, 1)
```

Heap_sort algorithm starts with Build_Max_Heap to build a Max_Heap on Input array A[1...n] where n is the length of the array. The maximum element of the array is stored at the root i.e A[1], it can be put to ~~eat~~ its correct position by exchanging it with A[n].

Decreave the heap_size by 1, and then call MAX_HEAPIFY (A,1) to restore the heap property of new Binary Tree created.

4	1	3	2	16	9	10	14	8	7



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

for i = 10 to 2

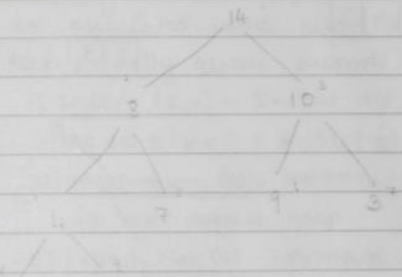
i = 10, exchange A[1] ↔ A[10]

1	14	10	8	7	9	3	2	4	16

heap_size [A] ← 9

MAX_HEAPIFY (A, 1)

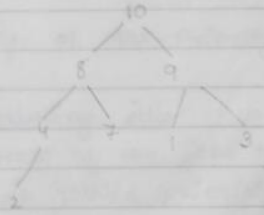
i = 2



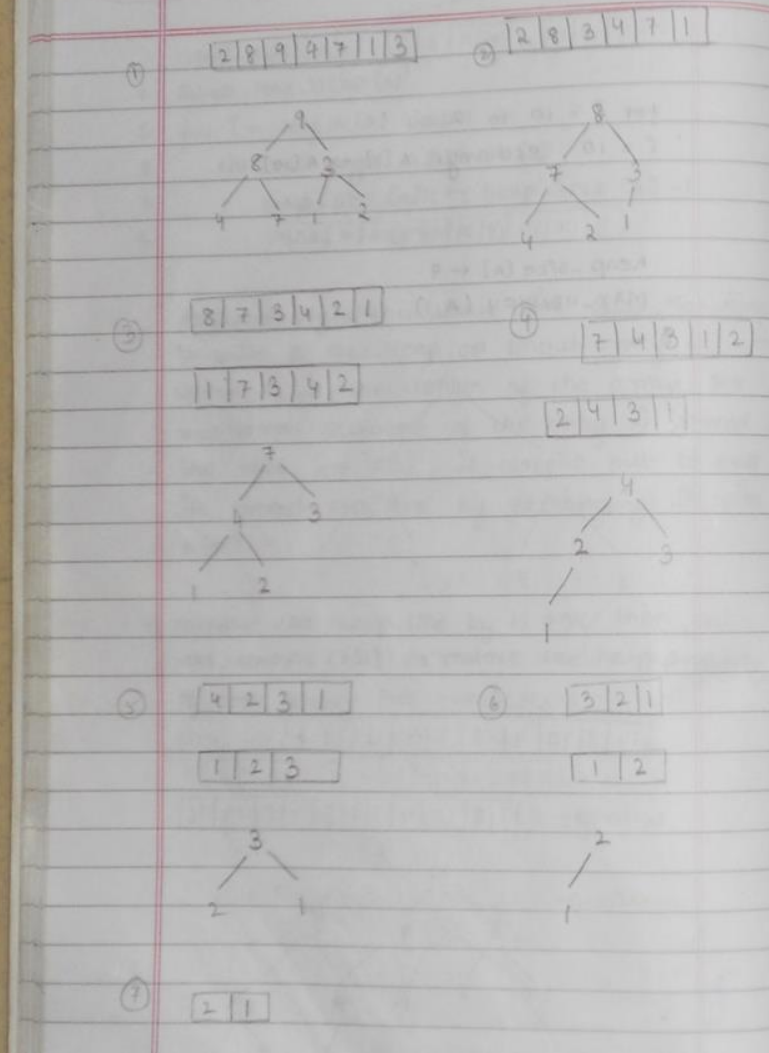
14	8	10	4	7	9	3	2	1	16

exchange

1	8	10	4	7	9	3	4	2	14	16



10	8	9	4	7	1	3	2



10/8/2013 Priority Queues

Application of heap as a priority queue: Using heap can be used

1) Max - Priority
2) Min - Priority

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a ~~heap~~ key. A Max-priority queue supports following operations

1) Insert (S, x) - Insert the element x into the set S i.e $S \leftarrow S \cup \{x\}$
2) Maximum (S) - returns the element of S with the largest heap.
3) Extract_Max (S) - removes and returns the element of S with the largest key
4) Increase key (S, x, k) - It increases the value of element x's key to the new value k which is assumed to be atleast as large as x's current key value.

Application of Max-Priority Queues.

1. In scheduling jobs
   - Keeps track of the jobs to be performed and their related priorities. When a job is finished, the highest priority job is selected from those pending using the extract-

max-procedure. operations of Min-Priority-queue are

1) Insert- ~~Minimum~~
2) Minimum - ~~extract-Min~~ (S)
3) Extract-Min (S)
4) Decrease-key

Operations of Max-Priority Queue

HEAP_MAXIMUM (A)

1. return A[1]

The above procedure, implements the maximum operation in $\Theta(1)$ time

HEAP_EXTRACT_MAX (A)

1. if heap-size [A] < 1
2. then error "heap underflow"
3. max ← A[1]
4. A[1] ← A[heap_size[A]]
5. heap_size [A] ← heap-size [A] - 1
6. MAX_HEAPIFY (A, 1)
7. return max

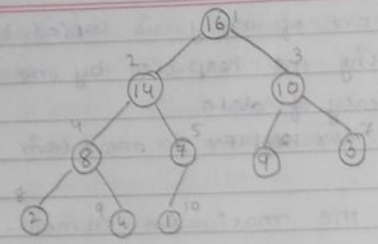
#1. to check whether elements exist in the heap or not.
#2. no elements are there in the heap.
#3. maximum element is at the first position in the array.
#4. last element of array is copied to A[1]
#5. decrementing the heap-size by one to remove the duplicacy of data.
#6. applying MAX_HEAPIFY to maintain MAX_HEAP property.
#7. returning the maximum element.

The running time of heap-extract_max procedure is $O(\log n)$ which is the time taken by max-heapify.

HEAP_INCREASE_KEY (A, i, key)

1. if key < A[i]
2. then error "Now key is smaller then current key"
3. A[i] ← key
4. while i > 1 and A[Parent (i)] < A(i)
5. do exchange A[i] ↔ A[Parent(i)]
6. i ← Parent (i)

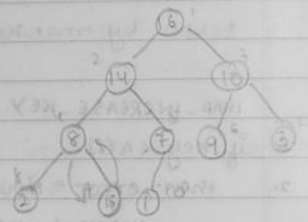
The priority queue element whose key is to be increased is identified by an index i into the array. The procedure first update the key of element A[i] to its new value. This procedure traverses a path from node i to the root and repeatedly compares an element to its parent exchanging there keys and continuing if the elements key is larger



Increase the key at index 9 by the new key value 15.

if 15 < 4 false

A[9] ← 15



while i > 1 and A[Parent(i)] < A(i)

9 > 1 true 8 < 15 true

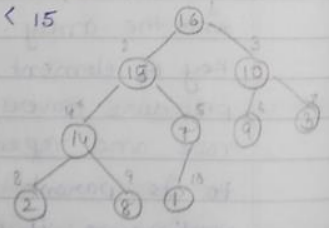
8 ↔ 15 i ← 4

while 4 > 1 and A[2] < A[4]

true 14 < 15

15 ↔ 14

i ← 2

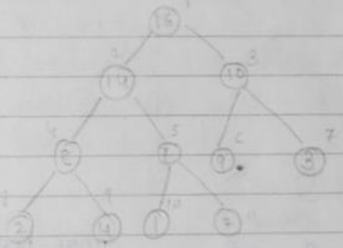


while 2 > 1 and A[1] < A[2]

false.

```
MAX-HEAP-INSERT (A, key)
1. Heap-size [A] ← Heap-size [A] + 1
2. i ← heap-size [A]
3. while i > 1 and A [Parent (i)] < key
4.    do A[i] ← A[Parent(i)]
5.       i ← Parent (i)
6. A[i] ← key
```

Given a max-heap, insert a new key value 15 into it.



Heap-Size [A] ← 10 + 1

i ← 11

while 11 > 1 and A[5] < key

true 7 < 15 true

A[11] ← A[5] i.e A[11] ← 7

i ← 5

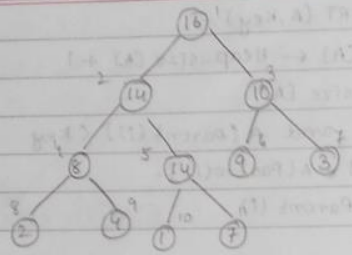
i ← 5

while 5 > 1 and A[2] < key

true 14 < 15 true

A[5] ← A[2] i.e A[5] ← 14

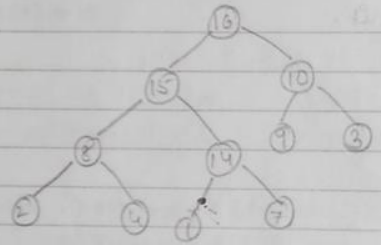
i ← 2



while $2 > 1$ and $A[1] \leq 15$

true $16 < 15$ false

$A[2] \leftarrow 15$.



The running time of ~~heap-increase-key~~ max-heap-insert on an n element heap is $O(\log n)$ because the path traced from a node to the root has length $\log n$

Quick Sort

Quick sort uses divide and conquer approach. Three steps of divide and conquer process for sorting array $A[p \ldots r]$ are

1) divide - it partitions the array $A[p \ldots r]$ into two $A[p \ldots q-1]$ and $A[q+1 \ldots r]$, such that each element of $A[p \ldots q-1]$ is less than or equals to $A[q]$ and $A[q]$ is $\leq$ each element of $A[q+1 \ldots r]$.
2) conquer - sorts the two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ by recursive calls to quicksort
3) combine - subarrays are sorted in place and ∴ no work is needed to combine them.

Stable	Inplace
- maintains the order of the array for duplicate elements	- sorts the array with the given memory space (no other arrays are created for sorting) - sorts in the array itself

```
QUICKSORT (A, p, r)
1. if p < r
2.   then q ← Partition (A, p, r)
3.      QUICKSORT (A, p, q-1)
4.      QUICKSORT (A, q+1, r)
```

The first call to quicksort procedure is QuickSort (A, 1, length [A]) i.e initially p is on the 1st element and r = sizeof (A) (at the last element).

```
PARTITION (A, p, r)
1. x ← A[r]
2. i ← p − 1
3. for j = p to r − 1
4.     do if A[j] ≤ x
5.         then i ← i + 1
6.              exchange A[i] ↔ A[j]
7. exchange A[i+1] ↔ A[r]
8. return (i+1);
```

1	2	3	4	5	6	7	8
2	8	7	1	3	5	6	4

Initially quicksort procedure is called with the values QuickSort (A, 1, 8), then p is checked against r i.e 1 < 8 which means that there are more than 1 element in the array. The array is then partitioned using partition procedure with the call Partition (A, 1, 8). Partition procedure selects an element x = A[r] as pivot element around which to partition the array A[p ... r].

Partition (A, 1, 8)

x ← 4

i ← 0

1	2	3	4	5	6	7	8
2	8	7	1	3	5	6	4

for j = 1 to 7

j = 1, if 2 ≤ 4 true

i ← 1, 2 ↔ 2

j = 2, if 8 ≤ 4 false

j = 3, if 7 ≤ 4 false

j = 4, if 1 ≤ 4 true

i ← 2, 8 ↔ 1

1	2	3	4	5	6	7	8
2	1	7	8	3	5	6	4

j = 5, if 3 ≤ 4 true

i ← 3, 7 ↔ 3

1	2	3	4	5	6	7	8
2	1	3	8	7	5	6	4

j = 6, if 5 ≤ 4 false

j = 7, if 6 ≤ 4 false

A[4] ↔ A[8]

i.e 8 ↔ 4, return 4.

1	2	3	4	5	6	7	8
2	1	3	4	7	5	6	8

QuickSort (A, 1, 3)

Partition (A, 1, 3)

1	2	3
2	1	3

x ← 3

i ← 0

for j = 1 to 2

j = 1, if 2 ≤ 3 true

i ← 1, 2 ↔ 2

j = 2, if 1 ≤ 3 true

i ← 2, 1 ↔ 1.

3 ↔ 3, return 3

QuickSort (A, 1, 2)

Partition (A, 1, 2)

$x \leftarrow A[2] \leftarrow 1$

2	1	3	4	7	5	6	8
1	2	3	4	5	6	7	8

$p \leftarrow 0$

for $j = 1$ to 1

if $A[1] \leq 1$ i.e $2 \leq 1$ false

$2 \leftrightarrow 1$

1	2	3	4	7	5	6	8
1	2	3	4	5	6	7	8

return 1.

Quicksort (A, 1, 0)

Partition (A, 1, 0)

if ($p < r$) i.e $1 < 0$ false

Quicksort (A, 4, 3)

Partition (A, 4, 3)

if $p < r$ i.e $4 < 3$ false.

Quicksort (A, 5, 8)

Partition (A, 5, 8)

if $p < r$ i.e $5 < 8$ true

The running time of Quicksort depends on whether the partitioning is balanced or not balanced or which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, the algorithm runs as slow as insertion sort.

Worst case Partitioning

Worst case occurs when partitioning procedure ~~occurs~~ produces one subproblem with $n-1$ elements & one with only 1 element.

Let this unbalanced partitioning arises at each step of the algorithm. Partition costs $\theta(n)$ time. The recursive call to an array of size one just returns, $\therefore T(1) = \theta(1)$

Recurrence equation for total running time is defined as $T(n) = T(n-1) + T(1) + \theta(n)$

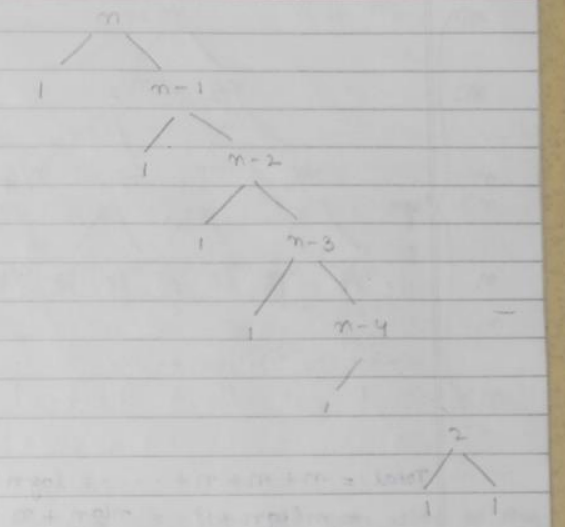
The recursion tree for worst case execution of quicksort is drawn as shown below.

$$\text{Total} = n + n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2$$

$$\text{Total} = \{1+2+3+\cdots+(n-3)+(n-2)+n\} + n - 1$$

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

$$= \frac{n^2+n}{2} \approx \theta(n^2)$$



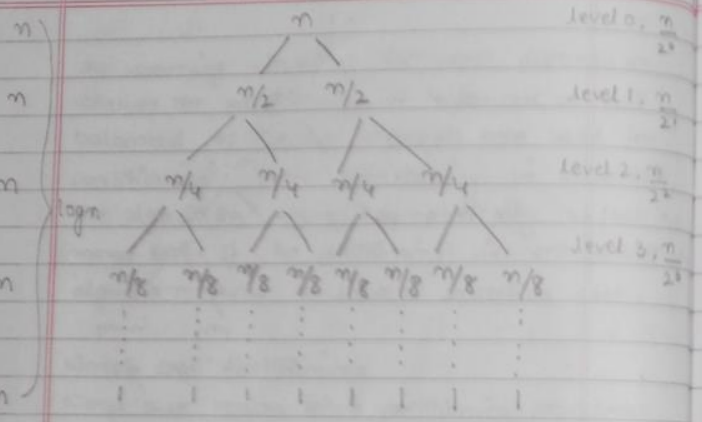
Best case Partitioning

Best case partitioning occurs when partition procedure 2 regions i.e 2 sub problems each of size not more than $\lfloor n/2 \rfloor$ and other is of size $\lceil n/2 \rceil - 1$

The recurrence equation for running time is then defined as $T(n) \leq 2T(n/2) + \theta(n)$

↓ Partition time

The recurrence tree for best case partition is drawn as shown below.



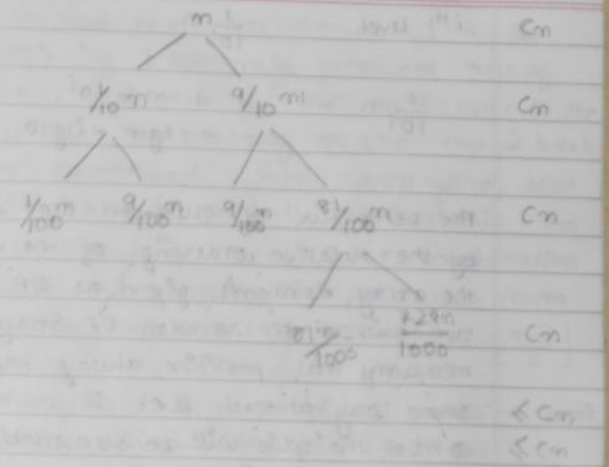
Total = $n + n + n + \cdots +$ log n times

$= n(\lg n + 1) = n\lg n + n = n\lg n.$

$\approx \Theta(n \lg n)$

Average case Partitioning

Average case ~~partitioning~~ running time quick sort is closer to the best case than to the worst case. Let us assume that partitioning algorithm always produces 9:1 proportional split which seems unbalanced at first sight ∴ the recurrence equation can be defined as $T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$



Cn is the cost involved at each level of the tree until a boundary condition is reached at depth $\log_{10} n = \Theta(\lg n)$ and then the levels have cost atmost Cn. The recursion terminates at depth ~~log10~~ $\lg^{10/9} n = \Theta(\lg n)$

∴ Total cost of quicksort algorithm is proportional to $\Theta(n \lg n)$. This running time is same as if we split the array from the middle. This is because any split of constant proportionality ¶ yields a recursion tree of depth $\log n$ where cost at each level is Cn. So total running time $= Cn \times \log n$

$= n \log n$

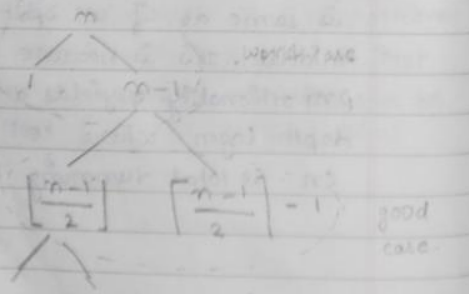
$\approx \Theta(n \log n)$

i^{th} level $\quad \frac{1}{10^i} n$

$\frac{1}{10^i} n = 1 \quad \Rightarrow n = 10^i$

$\Rightarrow \lg n = i \lg 10$

The behaviour of quicksort algo is determined by the relative ordering of the values in the array elements given as I/P. when we run quicksort on random I/P array it is not necessary that partition always happens in the same way at each level. So we assume that some of the splits will be reasonably well balanced & some will be fairly unbalanced. In the avg. case partition procedure produces a mix of good and bad splits i. in recursion tree of average case execution of partition, the good & bad splits are distributed randomly throughout the tree. Good splits are best case splits & bad splits are worst case splits.



At the root of the tree the cost is n for partitioning & subarrays produced are of size 1 and n-1 i.e the worst case. At the next level the subarray of size n-1 is best case partitioned into subarrays of size $\lfloor (n-1)/2 \rfloor$ and $\lceil (n-1)/2 \rceil - 1$. The cost is 1 for the subarray of size 1. So the combination of bad and good splits produces three subarray of size 1, $\lceil \frac{n-1}{2} \rceil - 1$ and $\lfloor \frac{n-1}{2} \rfloor$.

which yields to a combined partition cost of ~~$\Theta(1) + \Theta(n) + \Theta(n)$~~ $\Theta(1) + \Theta\left(\lceil \frac{n-1}{2} \rceil - 1\right)$

$+ \Theta\left(\lfloor \frac{n-1}{2} \rfloor\right) \simeq \Theta(n)$. Thus running time of quicksort when levels alternate b/w good & bad splits is like the running time of good splits i.e $\Theta(n \log n)$

17/8/2013 Randomised Version of Quick Sort

Randomisation is added to an algorithm in order to obtain good average case partitioning performance. Instead of using A[r] as the pivot element, we will use a randomly choosen element from the subarray A[p...r]. Then we exchange the element A[r] with an element choosen at random from A[p...r]. This modification ensures that the pivot element $x = A[r]$ is equally likely to be any of $r-p+1$ elements in the subarray. In the new partition procedure swapping is implemented before actual ~~swap~~ partitioning.

```
RANDOMISED_PARTITION (A,p,r)
1. i ← RANDOM (p,r)
2. exchange A[r] ↔ A[i]
3. return PARTITION (A,p,r)

RANDOMISED_QUICKSORT (A,p,r)
1. if p<r
2.    then q ← RANDOMISED_~~QUICKSORT~~PARTITION (A,p,r)
3.         ~~q~~ RANDOMISED_QUICKSORT (A,p,q-1)
4.         RANDOMISED_QUICKSORT (A,q+1,r)
```

17/8/2013.

SORTING IN LINEAR TIME

In this the sorted order is determined based on comparision b/w the I/p elements. There are several algorithms that can sort n nos in $O(n \lg n)$ time.

Mergesort, Heapsort achieve this upper bound in worst case & Quicksort achieve it in average case. The aim is to prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements

Sorting algorithms like counting sort, radix sort and Bucket sort runs in linear time. ∴ these algorithms use operations other than comparisons to determine the sorted order & ∴ $\Omega(n \lg n)$ does not apply to them.

In a comparison sort, we use only comparison b/w elements to gain order information about an I/P sequence $\langle a_1, a_2, \ldots, a_n \rangle$. Given two elements a_i and a_j, we perform the following tests to determine the ~~following tests~~ their relative order.

$a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$

Decision Tree Model.

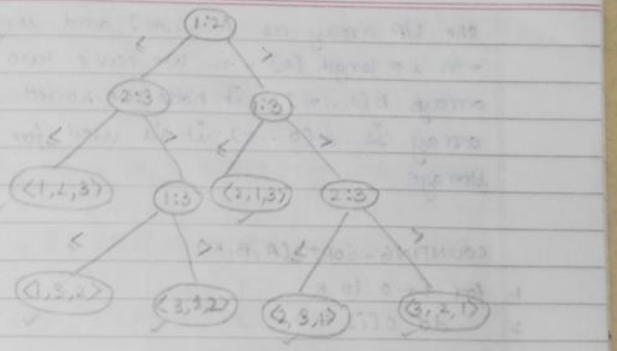
Comparison sorts can be viewed in terms of decision trees. A decision tree is a binary tree that represents the comparison b/w the elements that are performed by a particular sorting algorithm operating on an input of a given size.

In a decision tree, each internal node is represented by $i:j$ for some i and j lies in the range $1 \leq i$ and $j \leq n$. Each leaf is represented by a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$.

For example, $S = \{a, b, c\}$. So possible permutations are $3! = 6$.

abc, acb, bac, bca, cab, cba

For n elements, first element can be choosen from n ways, second element can be chosen from $n-1$, third element from $n-2$ and so on. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each node, a comparison $a_i \leq a_j$ is made. The left subtree shows the comparisons for $A[i] \leq A[j]$ and right subtree shows for $A[i] > A[j]$.



The correct sorting algorithm produces $n!$ permutations on n elements as one of the leaves of the decision tree. Each of these leaves must be reachable from the root by a path.

COUNTING SORT

counting sort assumes that each of the n input elements is an integer in the range 0 to k for some integer k. In counting sort we determine for each I/P element x, the no of elements less than x. This information is used to place element x directly into its position in the output array e.g if there are 5 elements less than x than x belongs to the 6th position in the O/P. Little modification is required if there is duplication of elements in the I/P to avoid placing them at the same position.

The I/P array as A[1...n] and length of A = n i.e length [A] = n. We have two other arrays B[1...n] - it holds the sorted o/p. second array is C[0...k] - it is used for temporary storage.

COUNTING-SORT (A, B, K)

1. for i ← 0 to k
2. do C[i] ← 0] O(k)
3. for j ← 1 to length [A]
4. do C[A[j]] ← C[A[j]] + 1] O(n)
5. ▷ C[i] now contains the no of elements equal to i
6. for i ← 1 to k
7. do C[i] ← C[i] + C[i-1]] O(k)
8. C[i] now contains the no. of elements less than or equal to
9. for j ← length [A] down to 1
10. do B[C[A[j]]] ← A[j]] O(n)
11. C[A[j]] ← C[A[j]] - 1

2	5	3	0	2	3	0	3
1	2	3	4	5	6	7	8

k = 5,

1. for i ← 0 to 5
2. do C[i] ← 0

0	0	0	0	0	0
0	1	2	3	4	5

3. for j ← 1 to 8
do C[A[j]] ← C[A[j]] + 1

j ← 1, C[2] ← C[2] + 1

0	0	1	0	0	0
0	1	2	3	4	5

j ← 2, C[5] ← C[5] + 1

0	0	1	0	0	1
0	1	2	3	4	5

j ← 3, C[3] ← C[3] + 1

0	0	1	1	0	1
0	1	2	3	4	5

j ← 4, C[0] ← C[0] + 1

1	0	1	1	0	1
0	1	2	3	4	5

j ← 5, C[2] ← C[2] + 1

1	0	2	1	0	1
0	1	2	3	4	5

j ← 6, C[3] ← C[3] + 1

1	0	2	2	0	1

j ← 7, C[0] ← C[0] + 1

2	0	2	2	0	1

j ← 8, C[3] ← C[3] + 1

2	0	2	3	0	1
0	1	2	3	4	5

6. for i ← 1 to 5
do C[i] ← C[i] + C[i-1]

i ← 1, C[1] ← C[1] + C[0]

2	2	2	3	0	1

i ← 2, C[2] ← C[2] + C[1]

2	2	4	3	0	1

i ← 3, C[3] ← C[3] + C[2]

2	2	4	7	0	1

i ← 4, C[4] ← C[4] + C[3]

2	2	4	7	7	1

i ← 5, C[5] ← C[5] + C[4]

2	2	4	7	7	8
0	1	2	3	4	5

9. for $j \leftarrow 8$ downto 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

$j \leftarrow 8$, $B[C[3]] \leftarrow 3$
$\Rightarrow B[7] \leftarrow 3$
$C[3] \leftarrow C[3] - 1$

						3	
1	2	3	4	5	6	7	8

2	2	4	6	7	8
0	1	2	3	4	5

$j \leftarrow 7$, $B[C[0]] \leftarrow 0$
$\Rightarrow B[2] \leftarrow 0$
$C[0] \leftarrow C[0] - 1$

	0					3	
1	2	3	4	5	6	7	8

1	2	4	6	7	8
0	1	2	3	4	5

$j \leftarrow 6$, $B[C[3]] \leftarrow 3$
$\Rightarrow B[6] \leftarrow 3$
$C[3] \leftarrow C[3] - 1$

	0				3	3	
1	2	3	4	5	6	7	8

1	2	4	5	7	8
0	1	2	3	4	5

$j \leftarrow 5$, $B[C[2]] \leftarrow 2$
$\Rightarrow B[4] \leftarrow 2$
$C[2] \leftarrow C[2] - 1$

	0	2		3	3		
1	2	3	4	5	6	7	8

1	2	3	5		
0	1	2	3	4	5

$j \leftarrow 4$, $B[C[0]] \leftarrow 0$
$\Rightarrow B[1] \leftarrow 0$
$C[0] \leftarrow C[0] - 1$

0	0	2		3	3		
1	2	3	4	5	6	7	8

0	2	3	5	7	8
0	1	2	3	4	5

$j = 3$, $B[C[3]] \leftarrow 3$
$B[5] \leftarrow 3$
$C[3] \leftarrow C[3] - 1$

0	0	2	3	3	3		
1	2	3	4	5	6	7	8

21/8/2013

The total time taken by counting sort is $\theta(n+k)$. Counting sort beats the lower bound of $\Omega(n \lg n)$ because it is not a comparison sort and no comparison b/w input elements occur. Counting sort is stable algorithm which means the numbers with the same value appear in the output array in the same order as they do in the input array.

Radix sort.

Radix sort works by sorting the least significant digit of a d-digit no first, then secondly significant and so on. This process continues until the no's have been sorted on all the digits. Thus only d-passes are required to sort the d-digit no's. e.g sort dates by 3 keys - year, month, day. We run sorting algorithm with a comparison function that given 2 dates, first we compare years, if their is a tie, we compare month, if another tie occurs then we compare the dates.

The procedure for radix sort assumes that each element in the n element array has d-digit.

Radix_Sort (A, d)

1. for $i \leftarrow 1$ to d
2. do use a stable sort (counting array) to sort array A on digit i

Input instance

329	720	720	329
457	355	329	355
657	436	436	436
839 ⇒	457 ⇒	839 ⇒	457
436	657	355	657
720	329	457	720
355	839	657	839

Given n d-digit no's in which each digit can take up to k possible values, radix sort correctly sort these no's in $\theta(d(n+k))$ time. The analysis of running time depends on the stable sort used as the intermediate sorting algorithm.

Bucket sort

Bucket sort assumes that the I/p lies in the interval $[0, 1)$. The idea is to divide the interval $[0, 1)$ into n-equal sized sub intervals or buckets and then distribute n-input no's into the buckets. To produce the O/P we sort the no's into each bucket & then go through the buckets in order listing the elements in each. Bucket sort assumes that I/p is n element array A & each element of $A[i] < 1$. Another array $B[0..n-1]$ of Buckets is used to sort the data.

BUCKET_SORT (A)

1. $n \leftarrow$ length [A]
2. for $i \leftarrow 1$ to n
3. do insert A[i] into list B[Ln·A[i]]
4. for $i \leftarrow 0$ to $n-1$
5. do sort list B[0..n-1] with insertion sort
6. concatenate the list B[0], B[1], B[2] .... B[n-1] in order.

	A		B
1	0.78	0	/
2	0.17	1	→ 0.17 → 0.12
3	0.39	2	→ 0.26 → 0.21 → 0.23
4	0.26	3	→ 0.39
5	0.72	4	/
6	0.94	5	/
7	0.21	6	→ 0.68
8	0.12	7	→ 0.78 → 0.72
9	0.23	8	/
10	0.68	9	→ 0.94

for $i = 1$ to 10

$i \leftarrow 1$, A[1] ~~$\leftarrow$~~ B[~~10x 0.78~~] B[L10x0.78]] $\leftarrow$ A[1]

A[1] $\leftarrow$ B[ ] B[7] $\leftarrow$ A[1]

$i \leftarrow 2$, B[L10x 0.17]] $\leftarrow$ A[2]

B[1] $\leftarrow$ A[2]

lines 2 and 3 takes $\theta(n)$ time. line 4 take $\theta(n)$ time. line 5 ~~takes~~ calls the insertion sort n times. To analyse the cost of the calls to insertion sort, let n_i be the random variable denoting the no. of elements placed in bucket B[i]. Insertion sort runs in quadratic time so running time of bucket sort is equal to

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} \theta(n_i^2)$$

$$= \theta(n)$$

22/8/2013

Medians and order statistics

The ith order statistic of a set of n elements is the ith smallest element e.g the minimum of a set of elements is the first order statistic, the maximum is the nth order statistic.

A median is the half way point of the set. When n is odd the median is unique and is at position i where $i = (n+1)/2$. When n is even there are two medians occuring at position i where $i = n/2$ and $i = (n/2)+1$

Regardless of the parity, median occurs at $i = \lfloor \frac{n+1}{2} \rfloor$ i.e lower median and $i = \lceil \frac{n+1}{2} \rceil$ i.e upper median.

We address the problem of selecting the ith order statistic from a set of n distinct elements. The selection problem can be specified formally as

- Input - is set A of n distinct nos and a no i where i lies blw 1 to n $(1 \leq i \leq n)$.
- Output - the element x belongs to A i.e larger than exactly i-1 other elements of A.

The selection problem can be solved in order of $O(n \lg n)$ time because we sort the nos using heap sort, or merge sort & then index the ith element in the output array.

Minimum and Maximum

How many comparisons are necessary to determine the minimum of a set of n elements? The upper bound is n-1 comparisons. We assume the set resides in array A.

```
MINIMUM (A)
1. min ← A[1]
2. for i ← 2 to length [A]
3.   do if min > A[i]
4.     then min ← A[i]
5. return min
```

The maximum can also determined in n-1 comparisons. The above algorithm is optimal and thus n-1 comparisons are necessary to determine the minimum. Consider the process of finding the minimum as a tournament among elements. Each element comparison is a match in the tournament in which smaller of the two element wins. Therefore, every element except the winner must lose atleast one match. Thus n-1 comparisons are required.

Simultaneous Minimum and Maximum

The minimum & maximum of n elements can be obtained using $\Theta(n)$ comparisons i.e if we find minimum and maximum indepen-

dently using $n-1$ comparisons each, then total no of comparisons are $(n-1) + (n-1)$ i.e $2n-2$
(⇓ min, ⇓ max)

But atmost $3\lfloor n/2 \rfloor$ comparisons are sufficient to find both the minimum and maximum. we compare the pair of elements from the I/P first with each other and then we compare the smaller to the current minimum and larger to the current maximum at a cost of 3 comparisons for every 2 elements. The initial values for the current minimum and maximum depends on whether n is even or odd. If n is odd we set both the minimum and maximum to the value of the first element & then process rest elements in pairs. If n is even, we perform one comparison on the first two elements to determine the initial values of minimum and maximum, the & process the rest elements in the pairs. Total no of comparisons involved in both the cases are

n is odd: $3\lfloor \frac{n}{2} \rfloor$

n is even: 1 initial comparison + $\frac{3(n-2)}{2}$ comparisons

So total no of comparisons in case n is even are $1 + 3\left[\frac{(n-2)}{2}\right] = \frac{3n-4}{2} = \frac{3n}{2} - 2$. Thus in either case, total no of comparisons are $3\lfloor \frac{n}{2} \rfloor$.

Selection in expected linear time.

DIVide and conquer algorithm is used for the selection problem. As quicksort partition the I/P array recursively and process both the sides of the partition, randomized select works only on the 1 side of the partition. The running time of quicksort $\theta(n \lg n)$ & the running time of randomized select is $\theta(n)$.

The following code for randomized select returns the ith smallest element of the array $A[p \ldots r]$.

```
RANDOMISED_SELECT (A, p, r, i)
1.  if p = r
2.     then return A[p]
3.  q ← RANDOMISED_PARTITION (A, p, r)
4.  k ← q - p + 1
5.  if i = k
6.     then return A[q]
7.  else if i < k
8.     then return RANDOMISED_SELECT (A, p, q-1, i)
9.  else return RANDOMISED_SELECT (A, q+1, r, i-k)
```

#1. only one element is there in the array, ∴ return A[p], that element was it is the smallest

#3. if $p \neq r$, then partition the array

#4. no of elements in the low side of partition, & +1 for the pivot element.

#5. if $i = k$, check $A[q]$ is the ith smallest element

#7. to check two subarrays for the ith smallest element. If $i < k$, the smallest element lies in the low exside and we will call RANDOMISED-SELECT on left side.

24/8/2013

The behaviour of above algorithm is determined by the o/p of random no generator. After randomised partition is executed in line 3, the array is subdivided into two subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1] \leq A[q] \leq A[q+1..r]$

Selection in worst case linear time

We use select algorithm to find the desired element by recursively partitioning the I/P array. Select uses deterministic partitioning algorithm ie partition with little modification. The select algo determines the ith smallest of an I/P array of $n > 1$ elements by executing the following steps:

1. If $n = 1$, then select returns its only I/P value as the ith smallest element.
2. Divide the n elements of the I/P array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most 1 group made up of remaining $n \bmod 5$ elements.
3. Find the median of each of the $\lfloor n/5 \rfloor$ groups by first insertion sorting the elements of each group and then picking the median from the sorted list of grouped elements.
4. Use select recursively to find the median x of the $\lfloor n/5 \rfloor$ medians found in step 2.
5. Partition the I/P array around the median of medians x using the partition procedure with little modification.
6. Let k be one more than the no. of elements on the lower side of partition so that x is the kth smallest element & their are $n-k$ elements on the higher side of partition.
7. If $i = k$, the return k. Otherwise use select recursively to find the ith smallest element on the lower side if $i < k$ or $i-k$ th smallest element on the high side if $i > k$

24/8/2023

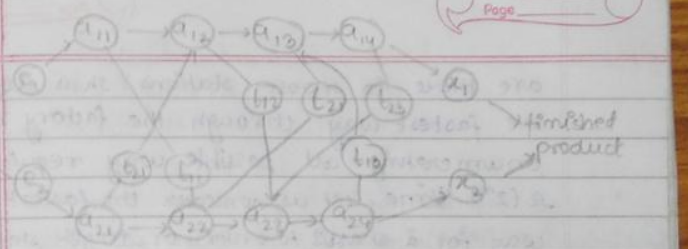
Dynamic Programming

Dynamic programming solves the problem by combining the sol^{ns} to subproblems. It is applicable when the subproblems are not independent. It solves every subproblem just once and then saves its answer in a table. It is applied to optimization problems. In such problems there can be many possible sol^{ns} but we wish to find a soln with optimal value.

Assembly Line Scheduling

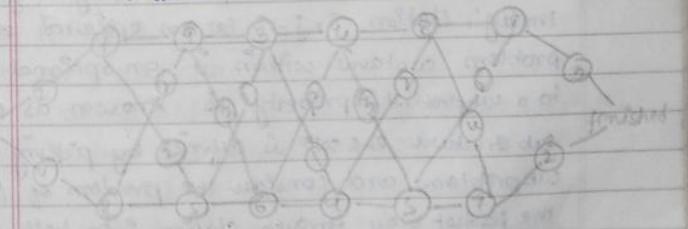
Example of dynamic programming is solving a manufacturing problem. ABC company produces automobiles in a factory that has two assembly lines. An automobile chasis enters each assemble line where parts of auto are added at a no. of stations and finished auto exits at the end of the line.

Each assembly line has n stations numbered $j = 1, 2, \ldots, n$. jth station on line i is represented as S_{ij} where i = assembly line and j = station no. The time required at each station varies even b/w stations at same position on two different lines. Assembly time required at station S_{ij} is represented as a_{ij}.



e_i is the entry time for the chassis to enter assembly line i. x_i is the exit time for the completed auto to exit assembly line i. Once the chasis enters an assembly line, it passes through that line but sometimes a partially completed auto may be switched from one line to ~~either~~ another. The time to transfer a chassis away from assembly line i after having gone through station S_{ij} is t_{ij} where $i = 1, 2$ and $j = 1, 2, \ldots n$

The problem is to determine which stations to choose from line 1 and line 2 in order to minimize the total time.



The fastest total time comes from stations 1, 3, and 6 from line 1 and stations 2, 4, 5 from line 2. There as n stations and 2 assembly lines ~~ie~~ $\therefore 2^n$ possible ways

are there to choose stations. Thus determining the fastest way through the factory by enumerating all possible way requires $\Omega(2^n)$ time. Let us consider the fastest possible way for a chassis to enter or to get start from is $S_{1,j}$. If $j = 1$, there is only 1 way to ~~arrive~~ arrive at that station. So it is easy to determine how long it takes to get through stations $S_{1,j}$. For $j = 2$ to n, there are two choices, first is the chassis could come from station $S_{1,j-1}$ and reach to station $S_{1,j}$. 2nd is the chassis could come from station $S_{2,j-1}$ and reach to station $S_{1,j}$. It involves transfer time through station $S_{1,j}$ $t_{2,j-1}$. First, if we suppose the fastest way is through station $S_{1,j-1}$, then we have considered the fastest way to get through station $S_{1,j-1}$ similarly, if we assume that fastest way through station $S_{1,j}$ is through station $S_{2,j-1}$ then we have assumed the fastest way to get through station $S_{2,j-1}$. So an optimal soln to a problem contains within it an optimal sol to a subprob. This property is known as optimal sub structure. The soln is defined by picking up subproblems and consider the problem of finding the fastest way through station j on both lines for $j = 1, 2, \cdots n$. Let $f_i[j]$ denotes the fastest possible time to get a chassis from a starting point through station S_{ij}.

	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

The total fastest time is denoted by f^* which is equal to $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

$$f_1[1] = e_1 + a_{11}$$
$$f_2[1] = e_2 + a_{21} \quad \Big\} \; j = 1$$

$$f_1[j] = \begin{cases} e_1 + a_{11}, & \forall\, j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \forall\, j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{21}, & \forall\, j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) \end{cases}$$

$f^* = 38$

31/8/2019

```
FASTEST_WAY (a, t, e, x, n)
1.  f1[1] ← e1 + a11
2.  f2[1] ← e2 + a21
3.  for j ← 2 to n
4.    do if f1[j-1] + aij ≤ f2[j-1] + t2,j-1 + aij
5.      then f1[j] ← f1[j-1] + aij
6.        l1[j] ← 1
7.      else f1[j] ← f2[j-1] + t2,j-1 + aij
8.        l1[j] ← 2
9.      if f2[j-1] + a2j ≤ f1[j-1] + t2,j-1 + a2,j
10.       then f2[j] ← f2[j-1] + a2j
11.         l2[j] ← 2
```

else $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + e_{2,j}$

$l_2[j] \leftarrow 1$

if $f_1[n] + x_1 \leq f_2[n] + x_2$

then $f^* = f_1[n] + x_1$

$l^* = 1$

else $f^* = f_2[n] + x_2$

$l^* = 2$

This algo accepts the values for a_{ij}, t_{ij}, e_i, and x_i and n (no. of stations).

Rod cutting

Dynamic programming is used to solve a simple problem in deciding where to cut the steel rods.

Problem statement: Serling Enterprises buys long steel rods and cuts them into shorter rods which it then sells. They want to know to best way to cut up the rods.

Assumption - we assume that for $i = 1, 2, \ldots n$ the price P_i in dollars that serling Enterprises charges for a rod of length i inches is given. Rod lengths are always in integers.

Given a rod of length n inches and a table of prices for $i = 1, \ldots, n$ we have to determine the maximum revenue R_n obtainable by cutting up the rod and selling the pices. We can cut up a rod of length n inches in 2^{n-1} different ways. e.g a rod of length 4 inches can be cut in 8 possible ways

length i	1	2	3	4
Price P_i	1	5	8	9

First possible way

Second possible way

Third possible way

Fourth Possible Way

Fifth Possible Way

Sixth Possible Way

Seventh Possible Way

Eight Possible way

Let the optimal solution cuts the rod into k pieces for some k lies b/w 1 to n, then an optiful optimal decomposition is written as $n = i_1 + i_2 + i_3 + \cdots + i_k$ and revenues is

written as $r_n = p_{i_1} + p_{i_2} + p_{i_3} + \cdots + p_{i_k}$

length i	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	20	24	30

$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1)$

$r_1 = \max(1, 1+0) = 1$

$r_2 = \max(p_2, r_1 + r_1) = \max(5, 1+1) = 5$

$r_3 = \max(p_3, r_1 + r_2, r_2 + r_1) = \max(8, 6, 6) = 8$

$r_4 = \max(p_4, r_1 + r_3, r_2 + r_2, r_3 + r_1)$

$= \max(9, 9, 10, 9) = 10$

$r_5 = \max(p_5, r_1 + r_4, r_2 + r_3, r_3 + r_2, r_4 + r_1)$

$= \max(10, 1+10, 5+8, 8+5, 10+1) = 13$

$r_6 = \max(p_6, r_1 + r_5, r_2 + r_4, r_3 + r_3, r_4 + r_2, r_5 + r_1)$

$= \max(17, 14, 15, 16, 15, 14) = 17$

$r_7 = \max(p_7, r_1 + r_6, r_2 + r_5, r_3 + r_4, r_4 + r_3, r_5 + r_2, r_6 + r_1)$

$= \max(20, 18, 18, 18, 18, 18, 18) = 20$

$r_8 = \max(p_8, r_1 + r_7, r_2 + r_6, r_3 + r_5, r_4 + r_4, r_5 + r_3, r_6 + r_2, r_7 + r_1)$

$= \max(24, 21, 22, 21, 20, 21, 22, 21) = 24$

$r_9 = \max(p_9, r_1 + r_8, r_2 + r_7, r_3 + r_6, r_4 + r_5, r_5 + r_4, r_6 + r_3, r_7 + r_2, r_8 + r_1)$

$= \max(30, 25, 25, 25, 19, 19, 25, 25, 25)$

length, i	1	2	3	4	5	6	7	8	9
Revenue, r_n	1	5	8	10	13	17	20	24	30

```
CUT-ROD(p,n)
1. if n == 0
2.     return 0
3. q = -∞
4. for i = 1 to n
5.     q = max(q, p[i] + CUT-ROD(p, n-i))
6. return q.
```

The CUT-ROD procedure takes an input an array p[1...n] of prices and then an integer n i.e the length of rod and it returns the maximum revenue possible for a rod of length n. If the I/P size becomes moderately large, the program would take a long time to run & thus degrades its performance. ∴ it is inefficient algo.

Using Dynamic Programming for optimal Rod cutting

Problem statement - the recursive soln is inefficient because it solves the same subproblem repeatedly.

Solution - To solve the subproblem only once & saving its soln in a table

The dynamic programming uses additional memory to save the computational time. There

two solns that uses dynamic programming
1) top-down memorization
2) Bottom-up Approach

For top-down approach
In this the procedure is written recursively in a natural manner but modified to save the result of each subproblem. The procedure first checks to see whether it has stored result, if so it returns the same value, else calculates it. Thus we say that recursive procedure is memoized it i.e it remembers what it has computed previously.

For Buttom-up Approach
In this, we sort the subproblems by size and solve them in size order (smallest first)

```
MEMOIZED_CUT_ROD (p,n)
1. let r[0...n] be a new array
2. for i = 0 to n
3.     r[i] = -∞
4. return MEMOIZED_CUT-ROD-AUX (p,n,r)

MEMOIZED_CUT-ROD-AUX (p,n,r)
1. if r[n] >= 0
2.     return r[n]
3. if n == 0
4.     q = 0
5. else q = -∞
6. for i = 1 to n
7.     q = max(q, p[i] + MEMOIZED_CUT-ROD-AUX(p,n-i,
                                              r))
8. return r[n] = q
9. return q.
```

4/9/2013 Matrix Chain Multiplication

Given a sequence of $\langle A_1, A_2, A_3, \ldots, A_n \rangle$ of n matrices to be multiplied, we have to calculate a product $A_1 \times A_2 \times A_3 \times \ldots \times A_n$.

Matrix multiplication is associative and so we use parenthesis to show how pair of matrices are multiplied e.g. $\langle A_1, A_2, A_3, A_4 \rangle$ matrix sequences are given and we have five different ways to fully parenthesise the problem.

$(A_1(A_2(A_3 A_4)))$
$(A_1((A_2 A_3) A_4))$
$((A_1 A_2)(A_3 A_4))$
$((A_1(A_2 A_3)) A_4)$
$(((A_1 A_2) A_3) A_4)$

Parenthesisation have a dramatic impact on the cost of evaluating the product. First we multiply two matrices using Matrix Multiply procedure

```
MATRIX-MULTIPLY (A,B)
1. if columns [A] ≠ rows [B]
2.     then error "Incompatible"
3. else for i ← 1 to rows [A]
4.          do for j ← 1 to columns [B]
5.                do C[i,j] ← 0
6.                   for K ← 1 to columns[A]
7.                      do C[ij] ← C[ij] + A[ik].B[kj]
8. return C
```

Two matrices are multiplied only if the no. of columns of A must be equals to the no. of rows of B. The no. of multiplications. $= p \times q \times r$, so the time to compute C is dependent on the no of multiplications ($p \times q \times r$) i.e. we express the cost in terms of multiplications. To identify the different costs incurred by different parenthesisation of a matrix ~~sets~~, we consider an example.

$A_1 = [\ ]_{10 \times 100}$ $B = [\ ]_{100 \times 5}$

$C = [\ ]_{5 \times 50}$

- $(A_1 A_2) A_3) = (10 \times 5) \times 50 = 2500$
  $= A_1$
- $((A_1 A_2) A_3)$ $A_1 A_2 = 10 \times 100 \times 5 = 5000$
  $(A_1 A_2) A_3 = 10 \times 5 \times 50 = 2500$
  $\therefore ((A_1 A_2) A_3) = 7500$
- $(A_1 (A_2 \times A_3))$ $A_2 A_3 = 100 \times 5 \times 50 = 25000$
  $A_1 (A_2 A_3) = 10 \times 100 \times 50 = 50000$
  $\therefore (A_1 (A_2 A_3)) = 75000$

Thus 1st parenthesisation is better than 2nd as it is faster & involves less no of calculations.

Matrix chain multiplication problem can be stated as : Given a chain $\langle A_1, A_2, A_3, \cdots, A_n \rangle$ of n matrices where for $i \leftarrow 1$ to n, matrix A_i has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1 \times A_2 \times A_3 \cdots A_n$ in a way that minimizes the scalar multiplication.

Counting the no of parenthesis.

When $n = 1$, there is only 1 matrix and thus there is only one way to fully parenthesize. But if $n \geq 2$, the fully parenthesize the matrix product is the product of 2 fully parenthesized matrix subproducts & the split b/w two subproducts occur b/w K^{th} and $(K+1)^{th}$ matrices for $K = 1, 2, 3, \ldots, n-1$. The recurrence relation for the parenthesization problem is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{K=1}^{n-1} P(K)\, P(n-K) & \text{if } n \geq 2 \end{cases}$$

1. Find the optimal substructure

   $A_i \ldots A_j$ where $i \leq j$

   let $i \leq k < j$

   $A_i \ldots A_j = (A_i \ldots A_K)(A_{K+1} \cdots A_j)$

2. Divise a recursive solution .ie optimal sol^n is calculated using optimal sol^n to subproblems

   $A_i\, A_{i+1}\, A_{i+2} \cdots A_j \quad \forall \; 1 \leq i \leq j \leq n$

   is a chain of matrices.

Let $m[i,j]$ be minimum no of scalar multiplications to compute the product of A_i to A_j.

$m[i,j] = 0$ if $i = j$

$m[i,i] = 0$

If $i > j$ we compute $m[i,j]$ as : we split the product A_i to A_j b/w A_i to A_K and A_{K+1} to A_j.

$$m[i,j] \qquad (A_i \ldots A_K)(A_{K+1} \ldots A_j) \qquad i \leq K < j$$

computing matrix product of $A_i \ldots A_K A_{K+1} \ldots A_j$ takes $P_{i-1} \times P_K \times P_j$ scalar multiplications.

$(A_i)_{P_{i-1} \times P_i} \qquad (A_{i+1})_{P_i \times P_{i+1}}$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \} & \text{if } i < j \end{cases}$$

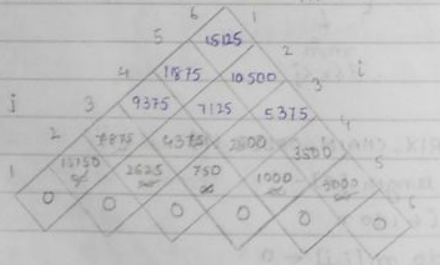
MATRIX_CHAIN_ORDER (P)

1. $n \leftarrow$ length $[P] - 1$
2. for $i \leftarrow 1$ to n
3. do $m[i,i] \leftarrow 0$
4. for $l \leftarrow 2$ to n
5. do for $i \leftarrow 1$ to $n - l + 1$
6. do $j \leftarrow i + l - 1$
7. $m[i,j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j - 1$

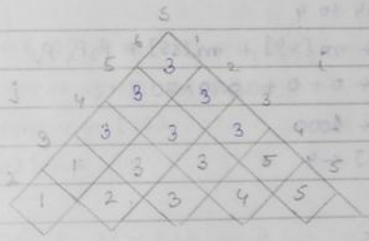
```
9.      do q ← m[i,k] + m[k+1, j] + P_{i-1} P_k P_j
10.        if q < m[i,j]
11.           then m[i,j] ← q
12.                s[i,j] ← k
13.  return m and s
```

8. $(A_1)_{30 \times 35}$ $(A_2)_{35 \times 15}$ $(A_3)_{15 \times 5}$ $(A_4)_{5 \times 10}$ $(A_5)_{10 \times 20}$ $(A_6)_{20 \times 25}$

Matrix	Dimensions
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25



1. $n \leftarrow$ length[P] -1 i.e. $n \leftarrow 6+1-1 \leftarrow 6$
2. for $i \leftarrow 1$ to 6
3. $\quad m[i,i] \leftarrow 0$
4. for $l \leftarrow 2$ to 6
5. $l = 2$ (for tracking row no's)
6. for $i \leftarrow 1$ to $6-2+1$ $(=5)$ i.e. 1 to 5
7. $\quad j \leftarrow i+2-1$ i.e. $j \leftarrow 2$ (because 2nd row starts from $j=2$)
8. $\quad m[i,j] \leftarrow \infty$
9. $\quad$ for $k \leftarrow 1$ to $2-1$ $(=1)$
10. $\quad\quad i=1, j=2, k=1$
11. $\quad\quad q \leftarrow m[1,1] + m[2,2] + P_{i-1} P_i P_j$
12. $\quad\quad q \leftarrow 0 + 0 + 30 \times 35 \times 15 \leftarrow 15750$
13. $\quad\quad q < m[i,j]$ i.e. $15750 < \infty$



5. $i \leftarrow 2$,
6. $j \leftarrow 2+2-1 \leftarrow 3$
7. $\quad m[2,3] \leftarrow \infty$
8. $k \leftarrow 2$ to 2
9. $q \leftarrow m[2,2] + m[3,3] + P_1 P_2 P_3$
   $q \leftarrow 0 + 0 + 35 \times 15 \times 5 \leftarrow 2625$
10. $q < m[2,3]$ i.e. $2625 < \infty$ true

5. $i \leftarrow 3$
6. $j \leftarrow 3+2-1$ i.e $j \leftarrow 4$
7. $m[34] \leftarrow \infty$
8. $k \leftarrow 3$ to 3
9. $q \leftarrow m[33] + m[44] + P_3 P_3 P_4$
   $q \leftarrow 0 + 0 + 15 \times 5 \times 10$
   $q \leftarrow 750$
12. $s[34] \leftarrow 3$

5. $i \leftarrow 4$
6. $j \leftarrow 5$
7. $m[45] \leftarrow \infty$
8. $k \leftarrow 4$ to 4
9. $q \leftarrow m[44] + m[55] + P_3 P_4 P_5$
   $q \leftarrow 0 + 0 + 5 \times 10 \times 20$
   $q \leftarrow 1000$
12. $s[45] \leftarrow 4$

5. $i \leftarrow 5$
6. $j \leftarrow 6$
7. $m[56] \leftarrow \infty$
8. $k \leftarrow 5$ to 5
9. $q \leftarrow m[55] + m[66] + P_4 P_5 P_6$
   $q \leftarrow 0 + 0 + 10 \times 20 \times 25$
   $q \leftarrow 5000$
12. $s[56] \leftarrow 5$

4. $L \leftarrow 3$ (for filling 3rd row)
5. $i \leftarrow 1$ to $6-3+1$ i.e $i \leftarrow 1$ to 4
   $i \leftarrow 1$
6. $j \leftarrow 1+3-1 \leftarrow 3$
7. $m[13] \leftarrow \infty$
8. $k \leftarrow 1$ to 2
9. $q \leftarrow m[11] + m[23] + P_0 P_1 P_2$
   $q \leftarrow 0 + 0 + 2625 + 5250$
   $q \leftarrow 7875$
10. $7875 < m[13]$
12. $s[13] \leftarrow 1$

8. $k \leftarrow 2$
9. $q \leftarrow m[12] + m[33] + P_0 P_2 P_3$
   $q \leftarrow 2250 + 15750 \leftarrow 18000$
10. $18000 < m[13]$
12. $s[13] \leftarrow 2$

5. $i \leftarrow 2$
6. $j \leftarrow 4$
8. for $k \leftarrow 2$ to 3
9. $q \leftarrow m[22] + m[34] + P_1 P_2 P_4$
   $q \leftarrow 0 + 750 + 5250$
   $q \leftarrow 6000$

5 $i \leftarrow 2$
6 $j \leftarrow 4$
8 $k \leftarrow 3$
9. $q \leftarrow m[23] + m[44] + P_1 P_3 P_4$
   $q \leftarrow 2625 + 0 + 1750$
   $q \leftarrow 4375$

5. $i \leftarrow 3$
6. $j \leftarrow 5$
8. $k \leftarrow 3$ to 4
9. $q \leftarrow m[35] + m[45] + P_2 P_3 P_5$
   $q \leftarrow 0 + 1000 + 3000 \leftarrow 4000 \leftarrow 0 + 1500 + 1000$
   $q \leftarrow 2500$

5. $i \leftarrow 3, j \leftarrow 5$
8. $k \leftarrow 4$
9. $q \leftarrow m[34] + m[55] + P_2 P_4 P_5$
   $q \leftarrow 750 + 0 + 3000 \leftarrow 3750$

5. $i \leftarrow 4, j \leftarrow 6$
8. $k \leftarrow 4$ to 5
9. $q \leftarrow m[44] + m[56] + P_3 P_4 P_6$
   $q \leftarrow 0 + 5000 + 1250 \leftarrow 6250$

5. $i \leftarrow 4, j \leftarrow 6$
8. $k \leftarrow 5$
9. $q \leftarrow m[45] + m[66] + P_3 P_5 P_6$
   $q \leftarrow 1000 + 0 + 2500 \leftarrow 3500$

4. $i \leftarrow 4, i \leftarrow 1$ to $3, j \leftarrow 4$
8. $k \leftarrow 1$ to 3
9. $q \leftarrow m[14] + m[24] + P_0 P_1 P_4$
   $q \leftarrow 0 + 4375 + 5250 \leftarrow 9625$
8. $k \leftarrow 2$
9. $q \leftarrow m[1$

4. $L \leftarrow 4$
8. $i \leftarrow 1$ to 3, $i \leftarrow 1$
6. $j \leftarrow 4$
8. $k \leftarrow 1$ to 3, $k \leftarrow 1$
9. $q \leftarrow m[11] + m[24] + P_0 P_1 P_4$
   $q \leftarrow 0 + 4375 + 10500 \leftarrow 14875$

4. $L \leftarrow 4,$
5. $i \leftarrow 1,$
6. $j \leftarrow 4$
8. $k \leftarrow 2$
9. $q \leftarrow m[12] + m[34] + P_0 P_2 P_4$
   $q \leftarrow 15750 + 750 + 4500 \leftarrow 21000$

4. $L \leftarrow 4,$
5. $i \leftarrow 1$
6. $j \leftarrow 4$
8. $k \leftarrow 3$
9. $q \leftarrow m[13] + m[44] + P_0 P_3 P_4$
   $q \leftarrow 7875 + 0 + 1500 \leftarrow 9375$

4. $L \leftarrow 4, i \leftarrow 2, j \leftarrow 5, k \leftarrow 2$ to 4
9. $q \leftarrow m[22] + m[35] + P_1 P_2 P_5$
   $q \leftarrow 0 + 2500 + 10500 \leftarrow 13000$

4. $L \leftarrow 4, i \leftarrow 2, j \leftarrow 5, k \leftarrow 3$
9. $q \leftarrow m[23] + m[45] + P_1 P_3 P_5$
   $q \leftarrow 2625 + 1000 + 3500 \leftarrow 7125$

4. $l \leftarrow 4,\ i \leftarrow 2,\ j \leftarrow 5,\ k \leftarrow 4$
9. $q \leftarrow m[24] + m[55] + P_1 P_4 P_5$
$q \leftarrow 4375 + 0 + 35 \times 10 \times 20 \leftarrow 11375$

4. $l \leftarrow 4,\ i \leftarrow 3,\ j \leftarrow 6,\ k \leftarrow 3$ to 5
9. $q \leftarrow m[33] + m[46] + P_2 P_3 P_6$
$q \leftarrow 0 + 3500 + 1875 \leftarrow 5375$

4. $l \leftarrow 4,\ i \leftarrow 3,\ j \leftarrow 5,\ k \leftarrow 4$
9. $q \leftarrow m[34] + m[56] + P_2 P_4 P_6$
$q \leftarrow 750 + 5000 + 3750 \leftarrow 9500$

4. $l \leftarrow 4,\ i \leftarrow 3,\ j \leftarrow 6,\ k \leftarrow 5$
9. $q \leftarrow m[35] + m[66] + P_2 P_5 P_6$
$q \leftarrow 2500 + 0 + 7500 \leftarrow 1000$

4. $l \leftarrow 5,$
5. $i \leftarrow 1$ to $2,\ i \leftarrow 1$
6. $j \leftarrow 5$
8. $k \leftarrow 1$ to $4,\ k \leftarrow 1$
9. $q \leftarrow m[11] + m[25] + P_0 P_1 P_5$
$q \leftarrow 0 + 7125 + 21000 \leftarrow 28125$

4. $l \leftarrow 5,\ i \leftarrow 1,\ j \leftarrow 5,\ k \leftarrow 2$
9. $q \leftarrow m[12] + m[35] + P_0 P_2 P_5$
$q \leftarrow 15750 + 2500 + 9000 \leftarrow 27250$

4. $l \leftarrow 5,\ i \leftarrow 1,\ j \leftarrow 5,\ k \leftarrow 3$
9. $q \leftarrow m[13] + m[45] + P_0 P_3 P_5$
$q \leftarrow 7875 + 1000 + 3000 \leftarrow 11875$

4. $l \leftarrow 5,\ i \leftarrow 1,\ j \leftarrow 5,\ k \leftarrow 4$
9. $q \leftarrow m[14] + m[55] + P_0 P_4 P_5$
$q \leftarrow 9375 + 0 + 6000 \leftarrow 15375$

4. $l \leftarrow 5$
5. $i \leftarrow 2$
6. $j \leftarrow 6$
8. $k \leftarrow 2$ to $5,\ k \leftarrow 2$
9. $q \leftarrow m[22] + m[36] + P_1 P_2 P_6$
$q \leftarrow 0 + 5375 + 13125 \leftarrow 18500$

4. $l \leftarrow 5,\ i \leftarrow 2,\ j \leftarrow 6,\ k \leftarrow 3$
9. $q \leftarrow m[23] + m[46] + P_1 P_3 P_6$
$q \leftarrow 2625 + 3500 + 4375 \leftarrow 10500$

4. $l \leftarrow 5,\ i \leftarrow 2,\ j \leftarrow 6,\ k \leftarrow 4$
9. $q \leftarrow m[24] + m[56] + P_1 P_4 P_6$
$q \leftarrow 4375 + 5000 + 8750 \leftarrow 18125$

4. $l \leftarrow 5,\ i \leftarrow 2,\ j \leftarrow 6,\ k \leftarrow 5$
9. $q \leftarrow m[25] + m[66] + P_1 P_5 P_6$
$q \leftarrow 7125 + 0 + 17500 \leftarrow 24625$

4. $l \leftarrow 6,\ i \leftarrow 1$ to $1,\ j \leftarrow 6,\ k \leftarrow 1$ to $5,\ k \leftarrow 1$
9. $q \leftarrow m[11] + m[26] + P_0 P_1 P_6$
$q \leftarrow 0 + 10500 + 26250 \leftarrow 36750$

4. $l \leftarrow 6,\ i \leftarrow 1,\ j \leftarrow 6,\ k \leftarrow 2$
6. $q \leftarrow m[12] + m[36] + P_0 P_2 P_6$
$q \leftarrow 15750 + 5375 + 11250 \leftarrow 32375$

4. $l \leftarrow 6, i \leftarrow 1, j \leftarrow 6, k \leftarrow 3$

9. $q \leftarrow m[13] + m[46] + P_0 P_3 P_6$

$q \leftarrow 7875 + 3500 + 3750 \leftarrow 15125$

4. $l \leftarrow 6, i \leftarrow 1, j \leftarrow 6, k \leftarrow 4$

9. $q \leftarrow m[14] + m[56] + P_0 P_4 P_6$

$q \leftarrow 9375 + 5000 + 7500 \leftarrow 21875$

4. $l \leftarrow 6, i \leftarrow 1, j \leftarrow 6, k \leftarrow 5$

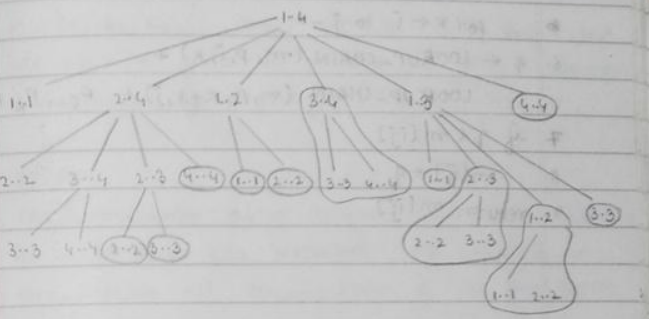
9. $q \leftarrow m[15] + m[66] + P_0 P_5 P_6$

$q \leftarrow 11875 + 0 + 15000 \leftarrow 26875$

7/9/2013

```
RECURSIVE_MATRIX_CHAIN (P,i,j)
1.  if i == j
2.      return 0
3.  m[i,j] = ∞
4.  for k ← i to j-1
5.      q = RECURSIVE_MATRIX_CHAIN (P,i,k) +
                                   (P,k+1,j) +
                                   (P_{i-1}, P_k, P_j)
6.  if q < m[i,j]
7.      m[i,j] = q
8.  return m[i,j]
```

Recursive tree procedure reduces a recursion tree by the call RECURSIVE_MATRIX_CHAIN (P,1,4). Each ~~vat~~ node is labelled by the values of the parameter i and j.



Memoization of Matrix chain Multiplication.

A memoized recursive algorithm maintains an entry in a table for the soln to each subproblem. Each table entry initially contains a special value to indicate that the entry has not yet been filled (still optimization is required).

```
MEMOIZED_MATRIX_CHAIN (P)
1.  n = P.length - 1
2.  let m[1...n, 1...n] be a new table
3.  for i ← 1 to n
4.      for j ← i to n
5.          m[i,j] ← ∞
6.  return LOOPUP_CHAIN (m, P, 1, n)

LOOPUP_CHAIN (m, P, i, j)
1.  if m[i,j] < ∞
2.      return m[i,j]
3.  if i == j
```

```
4.    m[ij] ← 0
5.  else
6.     for K ← i to j - 1
6.  q ← LOOKUP_CHAIN (m, p, i, K) +
        LOOKUP_CHAIN (m, p, K+1, j) + p_{i-1} p_K p_j
7.  if q < m[ij]
8.     m[ij] ← q
9.  return m[ij]
```


Longest common subsequence

In the LCS problem, we are given two sequences $X = \langle x_1, x_2, x_3, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, y_3, \ldots y_n \rangle$ we have to find the maximum length common subsequence of x and y.

STEPS

1) characterisation of a longest common subsequence - To solve LCS problem, one way is to enumerate all subsequences of x and check each subsequence to see if it is also a subsequence of y, keeping in track of the longest subsequence found. Each subsequence of x has the subset of indices $\{1, 2, \ldots m\}$. There are 2^m subsequences of x and thus this approach requires exponential time.

Let $x = \langle x_1, x_2, x_3, \ldots, x_m \rangle$ and $y = \langle y_1, y_2, y_3, \ldots, y_n \rangle$ be sequences and let $z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of x and y.

1. If $x_m = y_n$, then
   $z_k = x_m = y_n$ and
   z_{k-1} is an LCS of x_{m-1} and y_{n-1}

2. If $x_m = y_n$, then
   $z_k \neq x_m$ implies that z is an LCS of x_{m-1} and y_n

3. If $x_m \neq y_n$ then
$z_k \neq y_n$ implies that z is an LCS of x_m and y_{n-1}

2) Deriving a recursive solution
Let us define c[i,j] to be the length of an LCS of the sequences of x_i and y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0 which implies LCS has length 0.

The optimal substructure of LCS subproblem gives the recursive formula as

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

3) Computing the length of an LCS
Procedure LCS takes two sequences $x = \langle x_1, x_2 \ldots x_m \rangle$ and $y = \langle y_1, y_2 \ldots, y_n \rangle$ as input. It stores the c[i,j] values in a table c[0..m, 0..n] whose entries are computed in row major order. It also maintains a table B[1..m, 1..n] to simplify the construction of an optimal sol^n. B[i,j] points to the table entry corresponding to optimal subproblem sol^n chosen when computing c[i,j].

```
LCS-LENGTH (X, Y)
1.  m ← length[X]
2.  n ← length[Y]
3.  for i ← 1 to m
4.      do c[i,0] ← 0
5.  for j ← 0 to n
6.      do c[0,j] ← 0
7.  for i ← 1 to m
8.      do for j ← 1 to n
9.          do if x_i = y_j
10.             then c[i,j] ← c[i-1, j-1] + 1
11.                 b[i,j] ← "↖"
12.         else if c[i-1, j] ≥ c[i, j-1]
13.             then c[i,j] ← c[i-1, j]
14.                 b[i,j] ← "↑"
15.         else c[i,j] ← c[i, j-1]
16.             b[i,j] ← "←"
17. return c and b
```

i		j→ 0	1	2	3	4	5	6
	y_j	B	D	C	A	B	A	
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ (1)	← 1	← 1	← 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ (2)	← 2	↑ 2	↑ 2
4	B	0	↖ 1	← 1	↑ 2	↑ 2	↖ (3)	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑	↑ 2	↑ 2	↖ 3	↑ 3	↖ (4)
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

m = 7, n = 6

BCBA

7. $i \leftarrow 1$ to 7
8. $j \leftarrow 1$ to 6
9. if $x_1 = y_1$ false
12. if $c[0,1] > c[1,0]$ true
13. $c[1,1] \leftarrow c[0,1]$

7. $i \leftarrow 1$ to 7
8. $j \leftarrow 2$
9. if $x_1 = y_2$ false
12. if $c[0,2] > c[1,1]$

Greedy Algorithms.

A greedy algorithm makes the choice that looks best at the moment. Greedy algorithms does not always yields the optimal solution.

Activity Selector Problem.

The problem is to schedule several competing activities which requires exclusive use of a common resource with a goal of selecting a maximum size set of mutually compatible activities.

We have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities that wish to use a resource. Each activity has its start time denoted by s_i and a finishing time denoted by f_i where $0 \leq s_i < f_i < \infty$. If selected activity a_i takes place during the half open time interval $[s_i, f_i)$. Activity a_i and a_j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap i.e $s_i \geq f_j$ or $s_j \geq f_i$

Example - consider the set S of activities sorted in increasing order of finishing times.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

* Difference b/w Greedy and Dynamic Approach

```
GREEDY_ACTIVITY_SELECTOR (s, f)
1. n ← length [s]
2. A ← {a1}
3. i ← 1
4. for m ← 2 to n
5.    do if sm ≥ fi
6.       then A ← A ∪ {am}
7.            i ← m
8. return A
```

The above algorithm assumes that input activities are ordered by monotonically increasing finishing times. It collects selected activities into a set A and then returns this set.

```
RECURSIVE_ACTIVITY_SELECTOR (s, f, k, n)
1. m ← k+1
2. while m ≤ n and sm < fk
3.    do m ← m+1
4. if m ≤ n
5.    then return {am} ∪ RECURSIVE_ACTIVITY_SELECTOR
                         (s, f, m, n)
6. else return ∅
```

1. R_A_S (s, f, 0, 11)
2. RAS (s, f, 1, 11) m = 4
3. R_A_S (s, f, 4, 11) m = 8

{a1, a4, a8, a11} are the compatible activities.

K	s_k	f_k
0	...	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13

Greedy choice Property

The first key ingredient is the greedy choice property - a globally optimal sol^n can be acheived at by making a locally optimal (greedy) choice. When we are considering which choice to make, we make choice that looks best in the current problem without considering results from the subproblems. In Dynamic Programming, we make a choice at each step but the choice usually depends on the sol^{ns} to subproblems. Dynamic problems are solved in bottom up manner. In greedy, we make whatever choice seems best at the moment and the solve the subproblem arising after choice is made (top-down approach)

18/9/2013 Elements of Greedy strategy.

A greedy algorithm obtains optimal sol^n to a problem by making a sequence of choices. The choice that seems best at the moment is chosen

optimal substructure - a problem exhibits optimal substructure if an optimal sol^n to the problem contains within it optimal sol^{ns} to subproblems. This property is applicable to both dynamic and greedy algorithms e.g if an optimal sol^n to subproblem S_{ij} includes an activity a_k then it must also contain optimal sol^{ns} to the subproblem S_{ik} and S_{kj}

Greedy vs Dynamic Programming

0-1 knapsack problem - A thief robbing a store finds n items, the ith item is worth v_i $ and weights w_i pounds where v_i and w_i both are integers.

Problem - A thief wants to take as valuable a load as possible but he can carry atmost W pounds in his knapsack.

This is called 0-1 knapsack problem because each item must either be taken or must leave left behind. The thief cannot take a fractional amount of an item or take an item more than once.

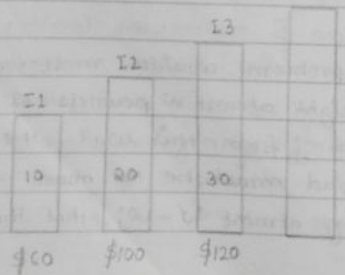
Fractional knapsack problem

In this the thief can take fractions of items rather than having to make a binary choice for each item.

For the 0-1 problem, consider most valuable load that weighs atmost W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing atmost $W - w_j$ that the thief

can take from n-1 original items excluding j. For the fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load considering that the thief can take fractional part of an item.

Solution for Fractional Knapsack Problem.
To solve the fractional knapsack problem, we first compute the value per pound v_i/w_i for each item. Following Greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound and so on until he cannot carray any more. Thus by sorting the items by value per pound, the greedy algo runs in O(n lgn) time.



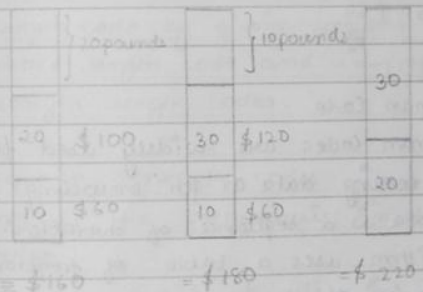
Value per pound $\frac{v_i}{w_i}$

Item 1 $= \frac{v_i}{w_i} = \frac{60}{10} = \$6$

Item 2 $= \frac{v_i}{w_i} = \frac{100}{20} = \$5$

Item 3 $= \frac{v_i}{w_i} = \frac{120}{30} = \$4$

Greedy Strategy



FRactional Problem taking item 1 does not work in the 0-1 problem because the thief is unable to fill its knapsack to its capacity & the empty space lowers the effective value per pound in his load. So in this case, we need to consider all the possible alternatives and choose amongst best which involves dynamic programming.

But in case of fractional Knapsack problem, the thief starts with Item 1, then Item 2 which occupies 30 pounds out of 50. Now he is left with 20 pounds so he tries to take fractional part of Item 3.

20	$80
20	$100
10	$60

Huffman Code

Huffman codes are widely used for compressing data or for encoding purposes. The data is a sequence of characters. Huffman algorithm uses a table of frequencies of occurences of the characters to build up an optimal way of representing each character as a binary string.

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5

Our problem is to design a binary code where each character is represented with a unique binary string

Fixed Code

6 characters = 3 bits

a = 000 c = 010 e = 100

b = 001 d = 011 f = 101

suppose there are 1 lakh characters in a file $6 \times 3 = 18$

$100000 \times 3 = 300000$

Variable length Code

Variable length code can do better than fixed length code by giving frequent characters a short length code and in-frequency characters longer length codes.

Steps to generate codes

1. Arrange the frequencies in descending order

a = 45, d = 16, b = 13, c = 12, e = 9, f = 5

2. start from the lowest frequency value and give the bottom value a binary code 0 & a value above a binary code 1.

3. start creating a tree by following a reverse order placing zero binary values on left and on right hand side.

a = 45
d = 16
b = 13
c = 12
e = 9] 1
f = 5] 0

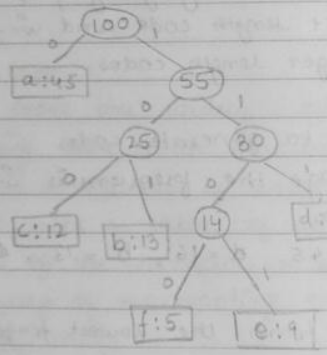
a = 45
d = 16
= 14
b = 13] 1
c = 12] 0

a = 45
= 25
d = 16] 1
= 14] 0

a = 45
= 30
= 25

step1

a = 45
= 30] 1
= 25] 0
= 55] → 100
a = 45]



	a	b	c	d	e	f
Binary code	0	101	100	111	1101	1100

Total no of bits required
= (45×1) + (13×3) + (12×3) + (16×3) + (9×4) + (5×

HUFFMAN (C)

1. n ← |C|
2. Q ← C
3. for i ← 1 to n-1
4. do allocate a new node z
5. left [z] ← x ← EXTRACT_MIN (Q)
6. right [z] ← y ← EXTRACT_MIN (Q)
7. f [z] ← f [x] + f [y]
8. INSERT (Q, z)
9. return EXTRACT-MIN (Q)

C is a set of n characters where each character c ∈ C is an object with a defined frequency f [c]. Algo builds tree T in a bottom up manner. It begins with a set of |C| leaves and perform a sequence of |C| - 1 merging operations. A minimum priority queue Q keyed on f is used to identify the two least frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of frequencies of two objects that were merged.

19/9/2013 Data Structures for Disjoint Sets.

A disjoint set data structure maintains a collection $S = \{S_1, S_2, S_3, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a representative which is some member of the set. The representative can be chosen by different methods depending upon the application.

Let us suppose each element of a set is represented by an object x, so the following operations are supported.

1) MAKE_SET(x). It creates a new set whose only member is x and thus x is a representative of a set.

2) UNION(x,y) It unites the dynamic sets that contain x and y i.e S_x and S_y into a new set i.e the union of these two sets. The representative of the resulting set is any member of $S_x \cup S_y$.

3) FIND_SET(x). It returns a pointer to the representative of the unique set containing x.

Running Time. The running time of disjoint set data structure is analysed in terms of two parameters n and m where n = no of make_set operations and m = total no of operations (MAKE_SET, UNION, FIND_SET).

The sets are disjoint and each union operation reduces the no. of sets by one. Let us consider an example $S = \{S_1, S_2, S_3, S_4, S_5\} = 5$

$\{S_1 \cup S_2\} = \{S_{12}\}$ $S = \{S_{12}, S_3, S_4, S_5\} = 4$

$\{S_{12} \cup S_3\} = \{S_{123}\}$ $S = \{S_{123}, S_4, S_5\} = 3$

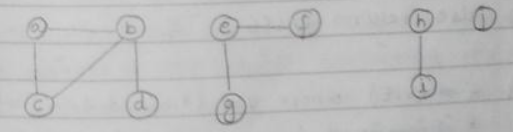
$\{S_{123} \cup S_4\} = \{S_{1234}\}$ $S = \{S_{1234}, S_5\} = 2$

$\{S_{1234} \cup S_5\} = \{S_{12345}\}$ $S = \{S_{12345}\} = 1$

After n-1 union operations only one set remains ∴ No of union operations that can be applied is at most n-1. MAKE_SET operations are included in total no. of operations i.e m. so we can say that $m \geq n$. We assume that the n makeset operations are the first n operations performed.

Application of Disjoint set Data Structure

An application of disjoint set data structure arises in determining the connected components of an undirected graph



CONNECTED_COMPONENTS (G)

1. for each vertex $v \in V[G]$
2. do MAKE_SET (V)
3. for each edge $(u,v) \in E[G]$

4. do if FIND-SET(U) ≠ FIND-SET(V)
5. then UNION (u,v)

V[G] = {a, b, c, d, e, f, g, h, i, j}

{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

E = {(a,b), (b,c), (c,a), (b,d) ..... }

We are considering the element

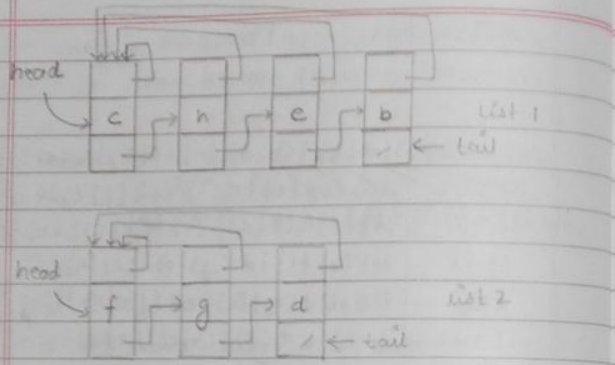
The connected-component procedure uses disjoint operations to compute the connected components of a graph. Once the C_C(G) procedure has been run the same component procedure identifies whether two p vertices are in the same connected component or not.

SAME_COMPONENT (u,v)
1. if FIND_SET(u) = FIND-SET(v)
2. then return TRUE
3. else return FALSE

Q. G = (V,E) where V = {a,b,c,d,e,..., k} and E is processed in the following order.
(d,i) (f,k), (g,i) (b,g) , (a,h) (i,j) (d,k) (b,j) (d,f) (g,j) (a,e) (i,d)

Edges Processed	
Initial set	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j} {k}
(d,i)	{a} {b} {c} {d,i} {e} {f} {g} {h} {j} {k}
(f,k)	{a} {b} {c} {d,i} {e} {f,k} {g} {h} {j}
(g,i)	{a} {b} {c} {d,i,g} {e} {f,k} {h} {j}
(b,g)	{a} {b,d,i,g} {c} {e} {f,k} {h} {j}
(a,h)	{a,h} {b,d,i,g} {c} {e} {f,k} {j}
(i,j)	{a,h} {b,d,i,g,j} {c} {e} {f,k}
(d,k)	{a,h} {b,d,i,g,j,f,k} {c} {e}
(b,j)	NO UNION OP
(d,f)	"
(g,j)	"
(a,e)	{a,h,e} {b,d,i,g,j,f,k} {c}
(i,d)	NO UNION OP

Linked List Representation.
A simple way to implement Disjoint set Data structure is to represent each set by a LL. The 1st object in each LL serves as its set representative. Each object in the LL contains a set member, a pointer to the object containing the next set member & a pointer back to the representative. Each list maintains pointers head to the representative & tail to the last object in the list.



Within each LL the objects may appear in any order. We assume that the 1st object of each list is the representative of the list. The MAKE_SET and FIND_SET operations become easy and require O(1) time i.e constant time. To carry out MAKESET (x) we, create a new LL whose only object is x. For findset (x) we return the pointer from x back to the representative.

A Simple Implementation of UNION

The UNION operation using LL takes more time than MAKESET and FIND_SET.

UNION (x,y) appends x list to the end of y list. The tail pointer for y list is used to identify where to append the x list.

A sequence of m operations requires $O(m^2)$ time, we let $n = \lceil m/2 \rceil + 1$ and $q = m - n$, n = no of MAKE_SET operations and m = total no of operations.

$$q = m - n = m - \left(\frac{m}{2} + 1\right) = m - \frac{m}{2} - 1$$

$$q = \frac{2m - m}{2} - 1 = \frac{m}{2} - 1 = \left\lfloor \frac{m}{2} \right\rfloor - 1$$

Suppose we have $x_1, x_2, x_3, \ldots, x_n$ as objects, so we execute a sequence of $m = n + q$ operations i.e. n MAKE_SET operations and (n-1) UNION operations.

Operations	No. of objects updated.	
MAKE_SET (x_1)	1	$\Theta(n)$
MAKE_SET (x_2)	1	
MAKE_SET (x_3)	1	
⋮	⋮	
MAKE_SET (x_n)	1	
UNION (x_1, x_2)	1	$\Theta(m^2)$
UNION (x_2, x_3)	2	
⋮	⋮	
UNION (x_{n-1}, x_n)	n-1	

So we will spend $\Theta(n)$ time performing the n MAKE_SET operations. Because union operation updates i objects the total no. of objects updated by all n-1 UNION operations is written as

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + n - 1 = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \approx O(m^2)$$

Total time = $\theta(m) + \theta(n^2)$

$= \theta(m+n^2)$

$= \theta(n^2)$

Thus, on an average each operation requires $\theta(n)$ time.

Weighted UNION Heuristic

Previously we have studied that when we append a longer list to a shorter list it takes more time and each union operation takes $\theta(n)$ time. Suppose each list also includes the length of the list field and we always append shorter list to the longer list, then the sig single union operation can take $\Omega(n)$ time. If both sides have $\Omega(n)$ members.

Theorem: A sequence of operations i.e m (MAKE-SET, FIND_SET, UNION) out of which n are MAKE-SET operations which takes $O(m + n\lg n)$ time.

MAKE-SET OPERATIONS

→ Each operation takes $\theta(1)$ time.

→ So n MAKE_SET operations take $\theta(n)$ time.

→ Since $m \geq n$, n = MAKE_SET operations which take $\theta(m)$ time.

UNION OPERATIONS

→ Using weighted union heuristic, we always append the smaller list onto the larger list.

What is the upper bound on the number of times that an element's representative must be updated.

Consider a fixed element x:

1) the first time x's representative is updated: x must have started in the smaller set, so the new set has atleast 2 members.

2) The second time x's representative is updated: x must have started in the smaller set, so the new set has atleast 4 members.

3) The k^{th} time x's representative is updated where $k \leq m$: x must have started in the smaller set, so the new set has atleast 2^k members

Therefore after x's representative pointer is has been updated $\lg k$ times, the new set has atleast k members.

Since there are n MAKE-SET operations. The largest set can atmost have n members. ∴ x's representative is updated atmost $\lceil \lg n \rceil$ times, so the total updated time for all n elements is $O(n\lg n)$ Notice that updating head and tail pointers takes $O(1)$ time per operation. So total time to update pointers over atmost m UNION operations is $O(m)$.

But we note that there can be atmost n union operations, so the total time to update pointers is actually O(n)

FIND_SET OPERATION

→ Each FIND_SET operation takes O(1) time. So atmost m FIND_SET operations takes O(m) time, ∴ in total all the operations take O(m + nlgn) time.

25/9/2023. Minimum Spanning Tree.

Given an undirected graph G = (V, E) where V is the set of vertices and E is the set of possible interconnections btw a pair of vertices. and for each edge (u, v) ∈ E we have a weight w (u, v) that specifies the cost to connect u and v. We have to find out an acyclic subset T ⊆ E that connects all the vertices and has total weight represented as $w(T) = \sum_{(u,v)\in T} w(u,v)$ and which should be minimum.

The subset T is acyclic and connects all of the vertices and thus forms a tree known as spanning tree & the problem is known as minimum spanning tree. There are two algorithms to solve minimum spanning tree problem

1) Kruskal's Algorithm.
2) Prim's Algorithm

Both the algorithms follow greedy strategy

Growing a minimum spanning tree.
Given an undirected graph G(V, E). The generic algorithm grows a minimum spanning tree one edge at a time. The algo manages a set of edges A maintaining the loop invariant: prior to each iteration A is a subset of minimum spanning tree At each step an edge (u, v) is determined that can be added to

$A \cup \{(u,v)\}$ is also a subset of MST. and that edge is called a safe edge for A.

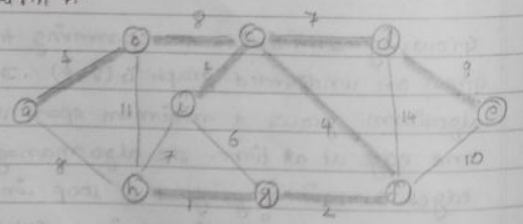
GENERIC_MST (G, w)

1. $A \leftarrow \emptyset$
2. while A does not form a spanning tree
3. do find an edge (u,v) that is safe for A
4. $A \leftarrow A \cup \{(u,v)\}$
5. return A.

Kruskal's Algorithm.

MST_KRUSKAL (G, w)

1. $A \leftarrow \emptyset$ — O(1)
2. for each vertex $v \in V[G]$ } V make set operations
3. do MAKE_SET(v)
4. Sort the edges of E in increasing order of weight — O(ElgE)
5. for each edge $(u,v) \in E$ taken in increasing order
6. do if FIND_SET (u) $\neq$ FINDSET (v) } O(E)
7. then $A \leftarrow A \cup \{(u,v)\}$
8. UNION (u, v) — |V|−1 max. union op.
9. return A.



{a} {b} {c} {d} {e} {f} {g} {h} {i}

(h,g) = 1 ✓
(g,f) = 2 ✓
(i,c) = 2 ✓
(a,b) = 4 ✓
(c,f) = 4 ✓
(i,g) = 6
(c,d) = 7 ✓
(h,i) = 7
(b,c) = 8 ✓
(a,h) = 8
(d,e) = 9 ✓
(e,f) = 10
(b,h) = 11
(d,f) = 14

{a} {b} {c} {d} {e} {f} {g} {h} {i}
{a} {b} {c} {d} {e} {fgh} {i}
{a} {b} {ci} {d} {e} {fhg}
{ab} {ci} {d} {e} {fhg}
~~{ab} {ci}~~ {abcifhg} {d} {e}
{abcifhgd} {e}
{abcifhgde}

Running time

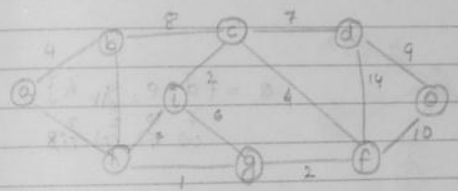
The running time of Kruskal's algorithm for a graph G(V, E) depends on the implementation of disjoint set data structure. Initializing the set A in line 1. takes O(1) time. Time to sort the edges in line 4 is is ElgE. The cost of |V| make set operations in for loop of line 2 is included. The loop of line 5 to 8 performs O(E) find-set and UNION operations along with |V| − 1 make-set operations. so total time is $O((V+E)\cdot\alpha(V))$ where α is a slowly growing function dependent on the no. of functions. The graph G is

connected so the no of edges is greater than or equal to the no of vertices - 1. Disjoint operations takes $O(E \cdot \alpha(V))$ time where $\alpha(V)$ is dependent upon no of vertices $\alpha(V) = |V|$ so we can write it like $O(\lg V) = O(\lg E)$. So running time for Kruskal is $O(E \lg V)$

Prim's Algorithm.

During the execution of Prim's algo all vertices that are not in a tree resides in a min-priority queue Q based on key field. For each vertex v, key of v is a minimum weight of any edge connecting v to a vertex in the tree. Key[v] = ∞ if there is no such edge. $\pi[v]$ is the parent of v in the tree. When algo terminates the min-priority queue Q is empty.

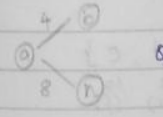
```
MST_PRIM (G, w, r)
1. for each u ∈ V[G]
2.     do key[u] ← ∞
3.        π[u] ← NIL
4. key[r] ← 0
5. Q ← V[G]
6. while Q ≠ ∅
7.     do u ← EXTRACT_MIN(Q)
8.        for each v ∈ Adj[u]
9.            do if v ∈ Q and w(u,v) < key[u]
10.               then π[v] ← u
11.                  key[u] ← w(u,v)
```



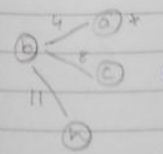
root = (a)

Q = { a , b , c , d , e , f , g , h , i }
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

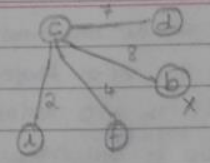
Q = { a , b , c , d , e , f , g , h , i }
0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞



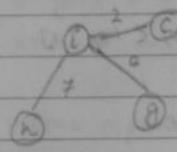
Q = { b , c , d , e , f , g , h , i }
4 ∞ ∞ ∞ ∞ ∞ 8 ∞
(a) (a)



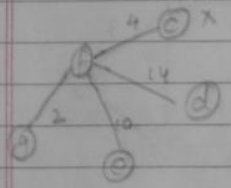
Q = { c , d , e , f , g , h , i }
8 ∞ ∞ ∞ ∞ 8 ∞
(b) (a)



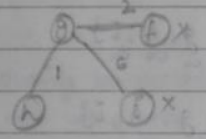
Q = { d , e , f , g , h , i }
7 ∞ 4 ∞ 8 2
(c) (c) (a) (c)



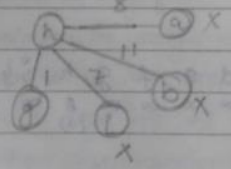
Q = { d , e , f , g , h }
7 ∞ 4 6 7
(c) (c) (i) (i)



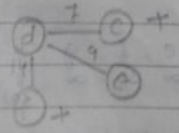
Q = { d , e , g , h }
7 10 2 7
(c) (f) (f) (i)



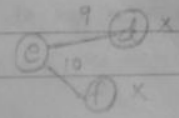
Q = { d , e , h }
7 10 1
(c) (f) (g)



Q = { d , e }
7 10
(c) (f)



Q = { e }
9
(f)



Q = { ϕ }

classmate
Date 3/9/2013
Page

Elementary Graph Algorithms

Representations of Graph

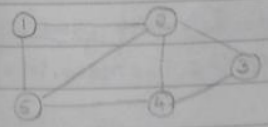
There are two ways of representing graph $G = (V, E)$

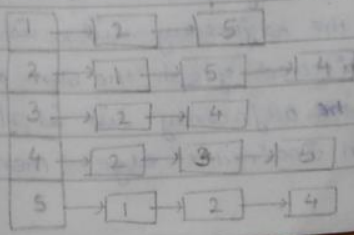
1) As an adjacency list representation
2) As an adjacency matrix representation.

The adjacency list representation is preferred because it provides a compact way to represent sparse graphs - those for which no. of edges $|E|$ is much less than $|V|^2$.

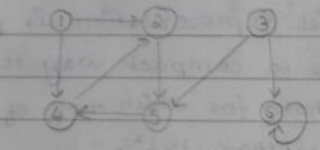
The adjacency matrix representation is preferred when the graph is dense i.e $|E|$ is close to $|V|^2$.

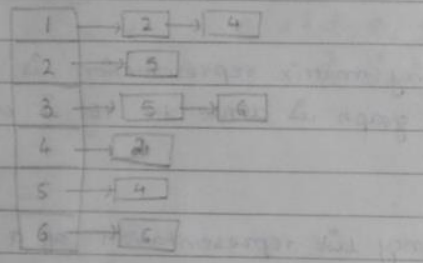
The adjacency list representation of a graph $G = (V, E)$ consists of an array adj. of V. For each $u \in V$, the adjacency list of u, contains all the vertices v such that there is an edge $(u, v) \in E$.





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$. If G is an undirected graph, the sum of the lengths of all the adjacency list is $2|E|$ because if (u, v) is an undirected edge, then u appears in v's adjacency list and vice-versa.

Adjacency list can be adopted to represent weighted graphs i.e graphs for which each edge has an associated weight given by a weight function w. Adjacency list representation robust in the sense that it can be modified to support other graph variants.

The disadvantage of adjacency list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list of u i.e adjoint of u.

This limitation of adjacency list representation is overcome by adjacency matrix representation but at the cost of using more memory space. For the adjacency matrix representation of graph $G = (V, E)$ we assume that the vertices are numbered $1, 2, 3, \ldots, |V|$ in some arbitrary manner. Then the adjacency matrix representation of graph G consists of $|V| \times |V|$ matrix $A = a_{ij} = \begin{cases} 0 & ; (i,j) \notin E \\ 1 & ; \text{otherwise.} \end{cases}$

Adjacency matrix representation requires $\Theta(|V|^2)$ memory independent of the no of Edges in the graph. Adjacency list representation require $\Theta(V + E)$. The transpose of a matrix $A = a_{ij}$ to be the matrix $A^T = (a_{ij})^T$ given

$(a_{ij})^T = a_{ji}$. In an undirected graph (u, v) and (v, u) represents the same edge so adjacency matrix A of an undirected graph is its own transpose $A = A^T$. Adjacency matrix representation can be used for weighted graph with only little modification.

Breadth First Search

It is the simplest algorithm for searching a graph. Given a graph $G_1(V, E)$ and a distinguished source vertex S, BFS search systematically and explorers the edges of a graph to discover every vertex i.e reachable from S. It computes the distance (smallest no. of edges) from S to each reachable vertex. It also produces BF Tree with root S that contains all reachable vertices. This algorithm works on both directed and undirected graphs. To keep a track of progress, BFS colors each vertex white, gray or black. All vertices are initially white and later on becomes gray & then becomes black.

A vertex is discovered the first time when it is encountered during the search operation & then it becomes non-white. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black i.e. all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices. Whenever white vertex v is discovered in the process of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. Then we say that u is the predecessor or parent of v in the BF Tree.

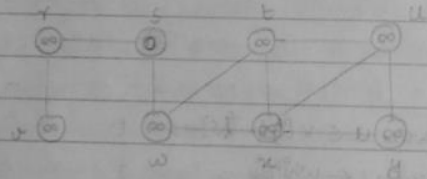
Ancestor and descendent relationship in the BFT are defined relative to the root s as - If u is on a path in the tree from the root s to a vertex v, then u is an ancestor of v and v is a descendent of u.

5/10/13

```
BFS (G, s)
1.  for each vertex u ∈ V[G] - {s}
2.      do color[u] ← white
3.         d[u] ← ∞
4.         π[u] ← NIL
5.  color[s] ← GRAY
6.  d[s] ← 0
7.  d[s] ← NIL
8.  Q ← ∅
9.  ENQUEUE (Q, s)
10. while Q ≠ 0
11.   do u ← DEQUEUE (Q)
12.      for each v ∈ Adj[u]
13.         do if color[v] = white
14.               then color[u] ← GRAY
15.                    d[v] ← d[u] + 1
```

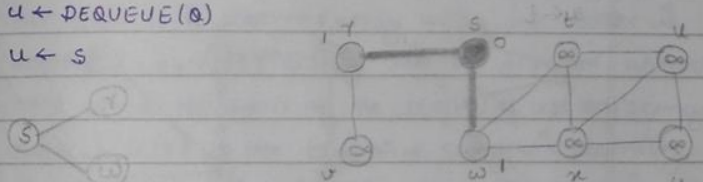
16. $\pi[v] \leftarrow u$
17. ENQUEUE(Q, v)
18. color[v] ← Black.

BFS assumes that the I/p graph $G(V, E)$ is represented using adjacency lists. The color of each vertex $u \in V$ is stored in color[u]. The predecessor of u is stored in $\pi[u]$. If u has no predecessor, then $\pi[u] \leftarrow$ NIL. The distance from source vertex to the vertex u is stored in d[u]. The FIFO Queue Q is used to manage the set of gray vertices.



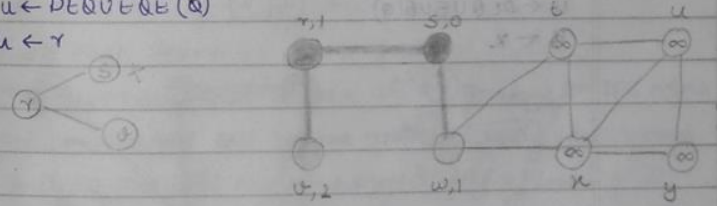
→ Line 1 to 4 paint every vertex white and set d[u] ← ∞ and set the parent of every vertex to NIL
→ Line 5 paint the source vertex to GRAY because it is considered to be discovered.
→ Line 6 initializes the distance of source with 0
→ Line 7 sets the predecessor of source to be NIL.
→ Line 8 and 9 initialize Queue to contain the the vertex s
→ The while loop of lines 10 to 18 iterates as long as there remains gray vertices which are discovered vertices that have not yet had their adjacency lists fully examined.
→ Line 10 : while Q ≠ 0 is true, so we enter into the loop and dequeue the Queue (remove s from Queue)
→ Line 12 : the for loop in line 12 works for all the adjoint vertices of u i.e s (1st time). We check if color of adjoint vertex is white, then we discover that vertex and change its color to gray and set the distance of v with distance of u+1.

u ← DEQUEUE(Q)
u ← s



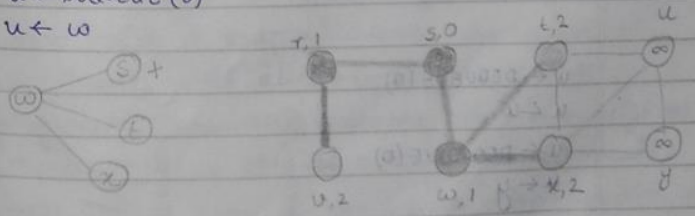
Q | r | w |

u ← DEQUEUE(Q)
u ← r



Q | w | v |

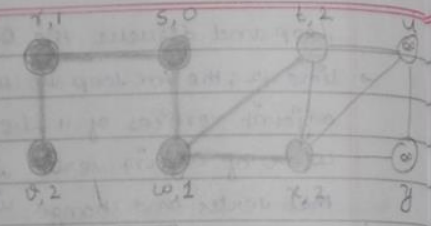
u ← DEQUEUE(Q)
u ← w



Q | v | t | x |

$u \leftarrow$ DEQUEUE(Q)

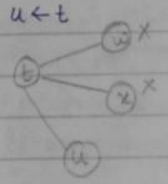
$u \leftarrow v$

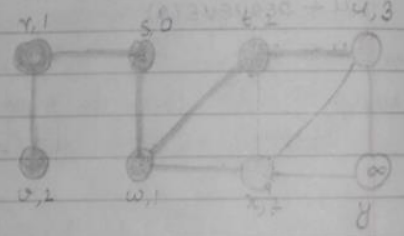


Q | t | x |

$u \leftarrow$ DEQUEUE(Q)

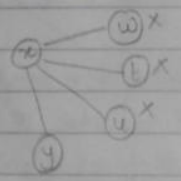
$u \leftarrow t$



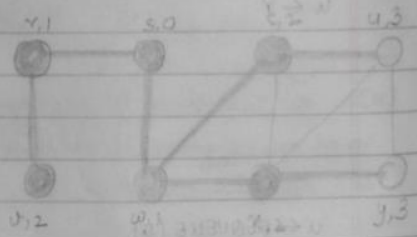


$u \leftarrow$ DEQUEUE(Q)

$u \leftarrow x$



Q | x | u |

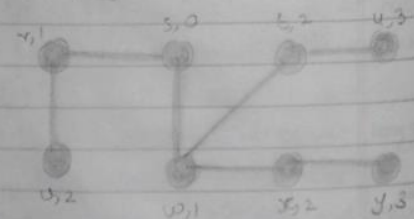


$u \leftarrow$ DEQUEUE(Q)

$u \leftarrow w$

$u \leftarrow$ DEQUEUE(Q)

$u \leftarrow y$



Analysis

After initialization, no vertex is ever whitened and thus the test in line 13 ensures that each vertex is enqueued atmost once and hence dequeued atmost once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned atmost once. Since the sum of the length of all adjacency lists is $O(E)$ so the total time spent in scanning the adjacency lists is $O(E)$. ∴ the total running time for BFS is $O(V+E)$.

Depth First Search

DFS searches more deeper in the graph. In DFS edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of the v's edges have been explored the search backtracks to explore edges leaving the vertex from which v was discovered. This process continues until all the vertices have been discovered that are reachable from the original source vertex. If any discovered vertices remain, then one of them is selected as a next source and the search is repeated from that source. Predecessor subgraph produced by a DFS may be composed of several trees because search may be repeated from multiple sources. DFS ~~from~~ form DF Forest composed of several depth first trees.

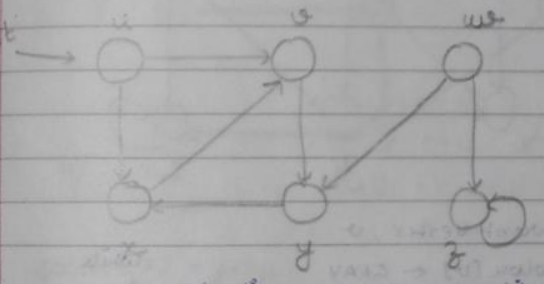
17/10/2013

DFS Timestamps each vertex. Each vertex v has two timestamps -

1) First timestamp $d[v]$ records when v is first discovered (and grayed) and second
2) timestamp records when the search finishes examining v adjacency list and blackens the v.

These timestamps are integers b/w 1 and $2|V|$. There is one discovery event and one finishing event for each of the $|V|$ vertices. For each vertex u, $d[u] < f[u]$. Vertex u is white before time $d[u]$, grayed b/w $d[u]$ and $f[u]$ and black after $f[u]$.

```
DFS (G)
1.  for each vertex u ∈ V[G]
2.      do color [u] ← WHITE
3.         π [u] ← NIL
4.  time ← 0
5.  for each vertex u ∈ V[G]
6.      do if color [u] = WHITE
7.         then DFS_VISIT (u)

DFS_VISIT(u)
1.  color [u] ← Gray
2.  time ← time + 1
3   d [u] ← time
4.  for each v ∈ Adj[u]
5.  do if color [v] = White
6.      then π [v] ← u
7.          DFS_VISIT (v)
8.  color [u] ← Black
9.  f[u] ← time ← time +1
```



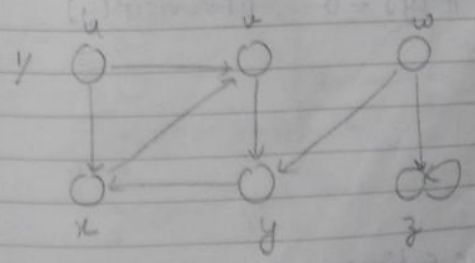
white white white white white white
$V[G] = \{ u, v, w, x, y, z \}$
NIL NIL NIL NIL NIL NIL

time = 0

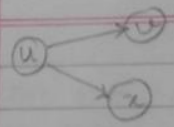
u, if color [u] = white ✓

DFS-VISIT(u)

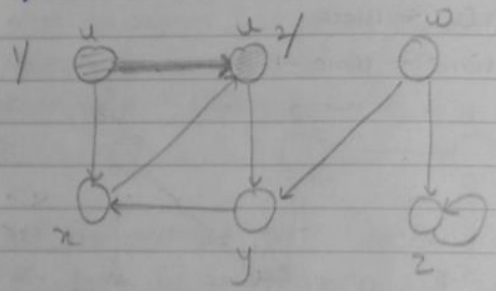
Gray white white white white white
$V[G] = \{ u, v, w, x, y, z \}$
NIL NIL NIL NIL NIL NIL



d[u] = 1



if color [v] = white ✓ , π [v] ← u , DFS_VISIT(v)



current vertex , v

color [v] ← GRAY

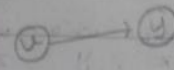
V [G] = { u , v , w , x , y , z }

u, v : GRAY — Parent-child

w, x, y, z : White — NIL

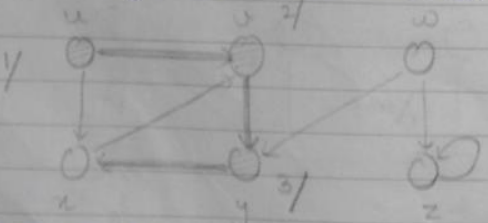
time ← time + 1 ← 1 + 1 ← 2

d [v] = 2

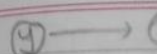


if color [y] = white

π [y] ← v DFS_VISIT (y)



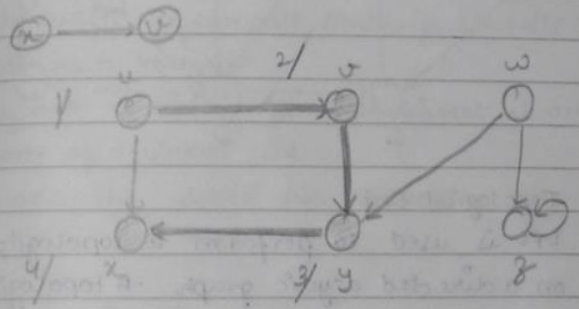
time ← time + 1 ← 2 + 1 ← 3



if color [x] = white ✓

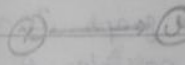
π [x] = y

DFS_VISIT (x)



color [x] = gray

time = time + 1 = 3 + 1 = 4

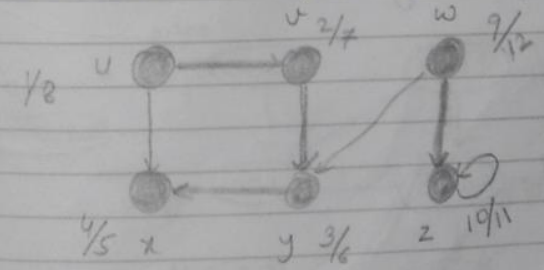


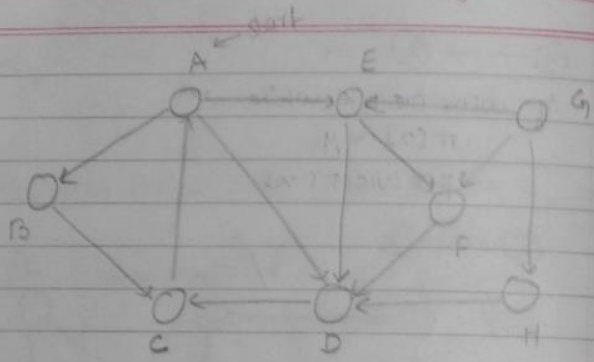
color [x] ← Black , f [x] ← 5

color [y] ← Black , f [y] ← 6

color [v] ← Black , f [v] ← 7

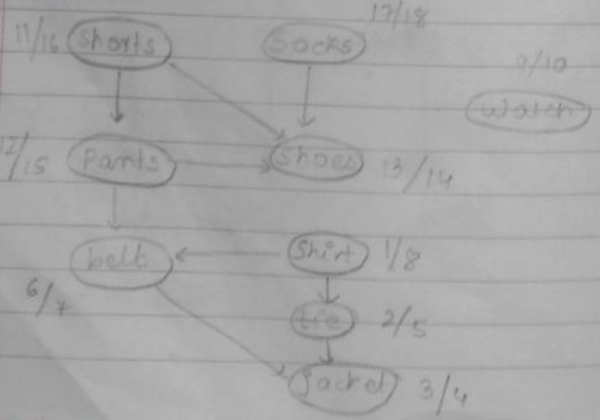
color [u] ← Black , f [u] ← 8

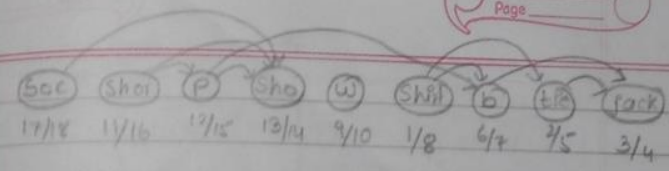




Topological Sort

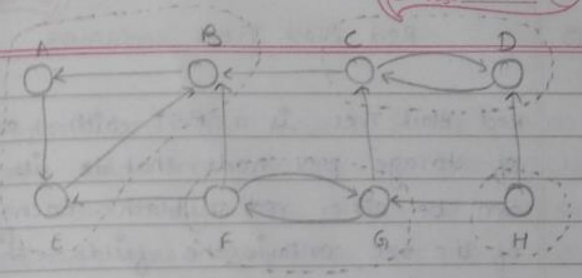
DFS is used to perform a topological sort on a directed acyclic graph. A topological sort of a DAG $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) than u appears before v in the ordering. If the graph is not acyclic, then no linear ordering is possible. A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.





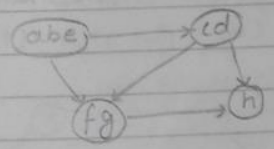
Topological_Sort (G)

1) Call DFS(G) to compute finishing times $f[v]$ for ~~call~~ each vertex v
2) as each vertex is finished insert it onto the front of a linked list
3) return the linked list of vertices.



- STRONGLY - CONNECTED - COMPONENT (G)

1. call DFS (G) to compute finishing times $f[u]$ for each vertex u
2. compute G^T
3. call DFS (G^T), but in the main loop of DFS consider vertices in order of decreasing o/p the vertices of each tree in the Depth first forest formed in line 3 as a separate strongly connected comp.



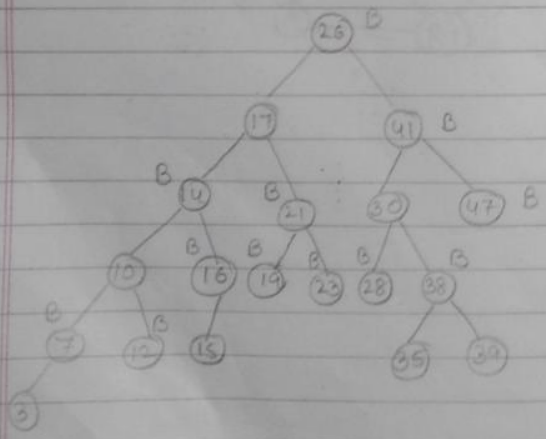
19/10/13

Red Black Trees

Red Black Tree is a BST with 1 extra bit of storage per node that is its colour which can be either red or black. Each node of the tree contains the fields - ~~child~~ color, key, left, right, and parent. If any value does not exist, its pointer field value is NIL.

Imp * A BST is a red-black tree if it satisfies following red black properties.

1) Every node is either red or black.
2) The root should be always black
3) Every leaf NIL is black.
4) If a node is red, then its both children should be black.
5) For each node all paths from the node to the descendent leaf contains the same no of black nodes.



Rotations

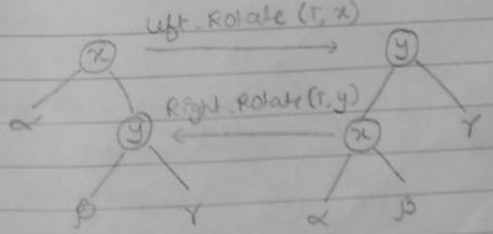
The operations like insert, delete when run on red black tree with n keys take O(lgn) time and may violate red-black properties. To restore these properties we can change the colour of ~~some~~ some of nodes or change the pointer structure.

Pointer structure is changed using rotation operation which helps in preserving BST properties.

There 2 types of rotations -

1) left rotation
2) Right rotation.

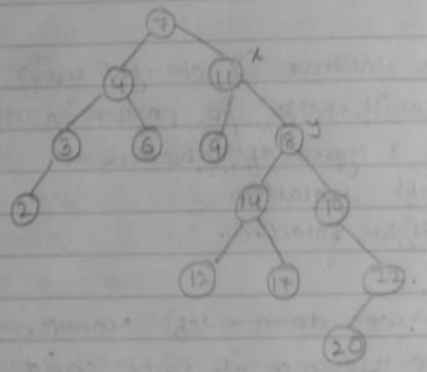
When we do a left rotation on a node x we assume its right child ~~is~~ y is not NIL[T] i.e right[x] ≠ NIL[T].



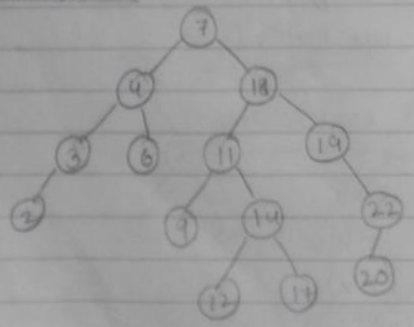
Left_Rotate(T,x)

1. y ← Right[x]
2. right[x] ← Left[y]
3. P[left[x]] ← x
4. P[y] ← P[x]
5. if P[x] = NIL[T]

6. then root [T] ← y
7. else if x = left [P[x]]
8. then left [P[x]] ← y
9. else Right [P[x]] ← y
10. left [y] ← x
11. ~~right~~P[x] ← y



Left Rotation



13/10/13

Insertion in a Red Black Tree

Insertion in a red black tree takes O(lgn) time. We insert node z into the tree T and then color it red. To preserve the properties of red-Black tree we use R-B_Insert_Fixup Procedure to recolor the nodes and perform the rotations. The RBINSERT(T,z) insert node z whose key field is assumed to have already being filled in

RB_INSERT (T, z)
1. y ← NIL [T]
2. x ← Root [T]
3. while x ≠ NIL [T]
4. do y ← x
5. if key [z] < key [x]
6. then x ← left [x]
7. else x ← Right [x]
8. P[z] ← y
9. if y = NIL[T]
10. then Root [T] ← z
11. else if key[z] < key[y]
12. then left [y] ← z
13. else right [y] ← z
14. left [z] ← NIL[T]
15. Right[z] ← NIL[T]
16. color [z] ← RED
17. RB_INSERT_FIXUP (T, z)

```
RB_INSERT_FIXUP (T,z)
1. while color [P[z]] = RED
2.   do if P[z] = left [P[P[z]]]
3.       then y ← Right [P[P[z]]]
4.       if color [y] = RED
5.       then color[P[z]] ← Black
6.            color [y] ← Black
7.            color[P[P[z]]] ← Red
8.            z ← P[P[z]]
9.       else if z = Right [P[z]]
10.           then z ← P[z]
11.                Left_Rotate (T,z)
12.      else { color [P[z]] ← Black
13.           { color [P[P[z]]] ← Red
14.           { Right_Rotate (T, P[P[z]])
15.  else (same as then clause with left & right
                    exchanged.)
16. color [root[T]] ← Black
```

Which of the Red-Black properties can be violated upon the call to RB_INSERT_FIXUP(T,z)?

RB_INSERT_FIXUP

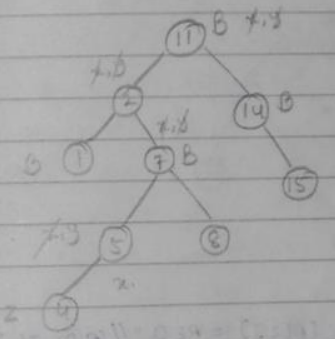
→ Property 1 and 3 holds, both children of newly inserted node are the sentinal NIL.

→ Property 5 which says that no. of black nodes is the same in every path from a given node also holds true because node z replaces the black sentinal node and node z is red with sentinal children.

→ Property two is violated which says that the root to be black. ✗

→ Property 4 is also violated which says that a red node cannot have a red child.

→ Property 2 is violated if z is the root and property 4 is violated if z's parent is red.



In the above figure node z is inserted which is red in color and its parent is also red, thus it violates property 4. RB_INSERT_FIXUP handles three invariants at the start of each iteration.

1st invariant - node z is red in color

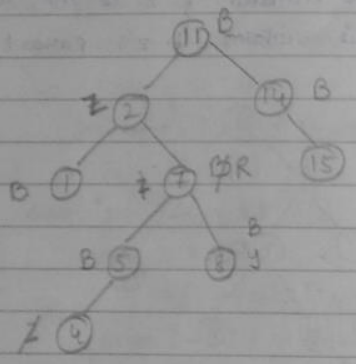
2nd invariant - if parent of z is the root, the parent of z is black in color.

3rd invariant - if red-black properties are violated then, there is atmost 1 violation and it is of either property 2 and or property 4.

There are two possible outcomes of each iteration

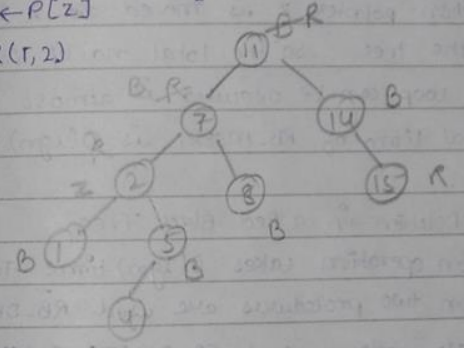
of the loop

1) pointer z moves up in the tree
2) some rotations are performed & the loop terminates

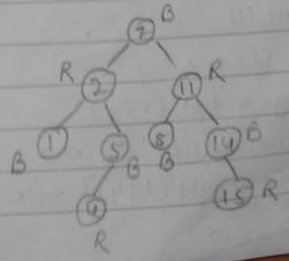


→ while color [P[z]] = RED // color of 5 is Red
→ do if P[z] = left [P[P[z]]] // True
→ then y ← Right [P[P[z]]]
y ← 8
→ if color [y] = RED // color of 8 = red, true.
→ then color [P[z]] ← Black // color of 5 - Black
→ color [y] ← Black // color of 8 = Black
→ color [P[P[z]]] ← Red // color of 7 - Red.
→ z ← P[P[z]] // z = 7
→ 10 to 15 lines not executed, and control goes to the while loop.
→ P[z] = Red. // color of 2 is ~~Black~~ Red, true

do if P[z] = left [P[P[z]]] // true
y ← Right [P[P[z]]] // y ← 14
if color [y] = Red // color [14] ≠ Red.
if z = Right [P[z]] // 7 is the right child of 2, true
z ← P[z]
L-R(T, 2)



if P[z] = left [P[P[z]]]
true.
y ← Right [P[P[z]]] // y = 14
if color [y] = red. // false
if z = Right [P[z]] // false
10 and 11 lines are not executed.
color [P[z]] ← Black // color of 7 = Black
color [P[P[z]]] ← Red // color of 11 = Red.
Right_Rotate (T, P[P[z]]



Analysis of Insertion

Since the height of red black tree on n nodes is lgn, so line no. 1 to 16 of RB-INSERT takes O(lgn) time. In RB_INSERT_FIXUP while loop repeats only if case 1 is executed and then pointer z is moved 2 levels up in the tree. So the total no. of times the while loop can be executed is atmost O(lgn)
∴ total time of RB_INSERT is O(lgn).

24/10/2013

Deletion in a Red-Black Tree

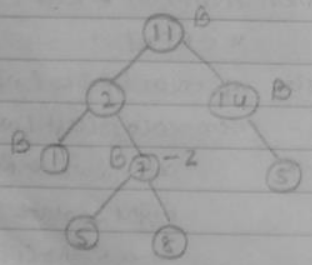
Deletion operation takes O(lgn) time. To perform deletion two procedures are used RB_DELETE - to delete node z and RB_DELETE_FIXUP - to change the colors and perform rotations to restore Red-Black properties.

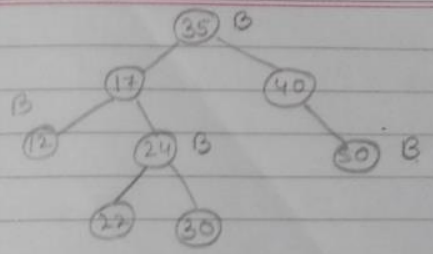
```
RB-DELETE (T,z)
1.  if left[z] = NIL[T] or right[z] = NIL[T]
2.     then y ← z
3.     else y ← Tree_Successor(z)
4.  if left[y] ≠ NIL[T]
5.     then x ← left[y]
6.     else x ← right[y]
7   p[x] ← p[y]
8.  if P[y] = NIL[T]
9.     then Root[T] ← x
10.    else if y = left[P[y]]
11.           then left[P[y]] ← x
12.           else right[P[y]] ← x
13. if y ≠ z
14.    then key[z] ← key[y]
15.    copy y's data into z
16. if color[y] = Black
17.    then RB-Delete-Fixup(T,x)
18. return y.

RB_DELETE_FIXUP (T,z)
1. while x ≠ Root[T] and color[x] = Black
2.    do if x = left[P[x]]
3.       then w ← Right[P[x]]
4.          if color[w] = RED
5.             then color[Pw] ← Black
6.                  color[P[x]] ← Red
7                   left_rotate(T, P[x])
8.                  w ← Right[P[x]]
9.          if color[left[w]] = Black and color[Right(w)]
                                        = Black
10.            then color[w] ← Red
11.                 x ← P[x]
12.         else if color[Right[w]] = Black
13.            then color[left[w]] ← Black
14.                 color[w] ← Red.
15.                 Right_Rotate(T,w)
16.                 w ← Right[P[x]]
17.           color[w] ← color[P[x]]
18.           color[P[x]] ← Black
19.           color(right[w]] ← Black
20.           left_rotate(T, P[x])
```

```
21.        x ← root[T]
22.    else (same as then clause with right & left
                exchanged)
23. color[x] ← Black.
```

```
Tree_Successor (x)
1. if right[x] ≠ NIL
2.    then return TREE_MINIMUM (Right(x))
3.    y ← P[x]
4.  while y ≠ NIL and x = Right[y]
5.      do x ← y
6.         y ← P[y]
7. return y
```

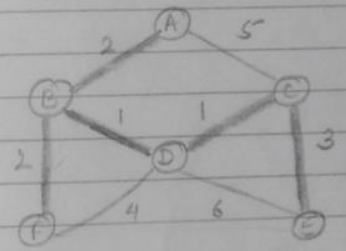
```
TREE_MINIMUM (x)
1. while left[x] ≠ NIL
2.    do x ← left[x]
3. return x.
```





Q. Insert node 27 in the given red-black tree?

Q. Using Kruskal algorithm give the MST for the following graph?



24/10/13

String Matching

We study the problem of detecting the occurrence of a particular substring called a pattern in another string called text. Let P be the pattern and T be the text, m be the length of P (i.e the no. of characters in a pattern) and n be the length of the Text (i.e the text file to be used). (n > m).

The Straight Forward Solution

- We start from the beginning of each string and compare the characters one after the ~~another~~ until either the pattern is exhausted or a mismatch is found. In first case if pattern is exhausted, we are done, a copy of pattern has been found in the text. In the latter case, we start again comparing the first pattern character with the 2nd text character.

In general, when a mismatch is found we slide the pattern one more place forward over the text and start again. Using straight forward method of string matching, compare the pattern 'ababc' with the text 'ababab cca'.

P : ABABC

T : ABABAB CCA

P : A B A B C
T : A B A B A B C C A

P : A B A B C
T : A B A B A B C C A

P : A B A B C
T : A B A B A B C C A

successful match.

Straight Forward String Matching Algorithm

```
function_match (P, T : string)
var
  i, j, K : index,
  { i is the current guess where P begins in T
    j is the index of the current character in T
    K is the index of the current character in P
  }
begin
  i = 1, j = 1, K = 1;
  while j ≤ T.length and K ≤ P.length do
    { if tj = PK then
        j = j+1;
        K = K+1;
      else
      { -- slide pattern forward and start over
          i = i+1;
          j = i;
          K = 1;
      }
      if K > P.length then Match = i (match found)
      else Match = j (match not found)
```

Knuth_Morris Pratt Algorithm

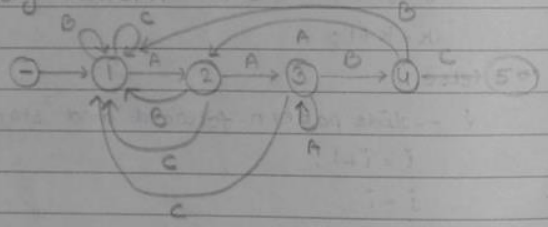
1. Pattern Matching with Finite Automata

Given a pattern P, it is possible to construct a finite automata that can be used to scan the text for a copy of pattern P very quickly. Finite automata is a machine or flowchart, where Σ be the alphabet set / set of characters

from which the characters in P and T be choosen. The finite automata has three types of nodes

1) starting node ($\vec{-}$)
2) stop node ($\vec{*}$) - reached when match is found.
3) read node - it means we have to read the next character. If there are not further characters in the text string, it will haullt and there is no match.

e.g Pattern AABC



2) KNuth Morris Pratt Flow chart

KMP algorithm constructs a flow chart to be used to ~~stan~~ scan the text. KMP flow chart contains arrows i.e the once to follow if the desired character is read from the text (success links) and other arrows are called failure links.

Difference b/w Finite Automata and KMP flow chart

i) the character labels of the KMP flow chart are on the nodes rather than on the arrows.

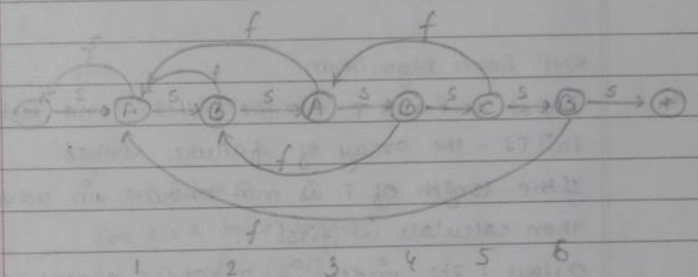
ii) The next character from the text is read only after a success link has been followed.

iii) The same text character is reconstructed if a failure link is followed.

iv) if the end of the text is reached elsewhere in the flow chart the scan terminates unsuccessfully.

P: ABABCB

T: ACABAABABA



Text T: 1 A, 2 C, 3 A, 4 B, 5 A, 6 A, 7 B, 8 A, 9 B, 10 A

KMP P: 1 A, 2 B, 3 A, 4 B, 5 C, 6 B

KMP call no.	Text character Index	character (Text)	Success / failure.
1	1	A	s
2	2	C	f
1	2	C	f
0	2	C	get next char
1	3	A	s
2	4	B	s
3	5	A	s

4	6	A	f
2	6	A	f
1	6	A	s
2	7	B	s
3	8	A	s
4	9	B	s
5	10	A	f
3	10	A	s
4	11	NONE	FAILURE

KMP Scan Algorithm

Input : P and T are the pattern and text strings
fail[] - the array of failure links
If the length of T is not known in advance then calculate it first.
Output : the index in T where a copy of P begins. The index will be T.length +1 if no match for P is found.

```
Function - KMPmatch (P, T : String ; fail : IndexArray)
var
    j, k : index;
    (j indexes text characters
     k  "    Pattern  "   & fail array)
begin
    j = 1, k = 1
    while j ≤ T.length & k ≤ P.length do
    { if k = 0 or tj = Pk then
        j = j+1
      k = k+1
    else (follow fail arrows)
        k = fail[k]
    }
    if k > P.length then
        KMP Match - j - P.length (match found)
    else KMPmatch - j (no match found)

Procedure KMP setup (P: string, fail : array)
{
    var
        k, r : Integer;
    begin
        fail[1] = 0
        for k = 2 to P.length
        {   r = fail[k-1]
            while r > 0 & Pr ≠ Pk-1 do
            {   r = fail[r]
            }
            fail[k] = r+1;
        }
}
```

3 copies

30/10/2018

Q. Give the fail indexes used by KMP algorithm for patterns

(a) AAAB

(b) AABAACAABABA

a)

fail	0	1	2	3
	1	2	3	4

fail[1] = 0

for k = 2 to length[P] // 2 to 4

k=2, r = fail[k-1] // r = 0

while r > 0 & $P_r \neq P_{k-1}$ do // false

fail[k] = r+1 // fail[2] = 1

k=3, r = fail[3-1] // k=1

while r > 0 & $P_1 \neq P_2$ do // false

fail[3] = 2

k=4, r = fail[3] // r = 2

while 2 > 0 and $P_2 \neq P_3$ false

(true) (false)

fail[4] = 3

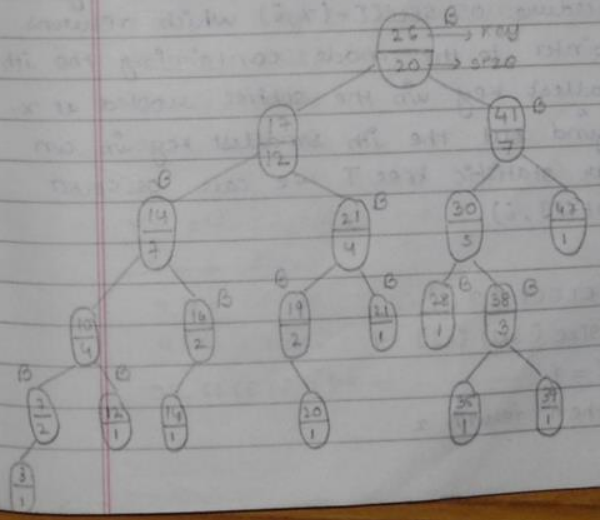
Augumenting Data Structures

Aim is to develop a data structure that supports general order statistics operations on a dynamic set and find the ith smallest no. in the set or the rank of the element in the total ordering of the set. The data structure is designed for maintaing a dynamic set of interval such as time intervals.

Dynamic Order Statistics

The ith order statistic of a set of n elements where i belongs to 1 to n i.e $i \in \{1,2,\ldots,n\}$ i.e the element in the set with ith smallest key. Any order statistic can be retrieved in O(n) time from an unordered set. Now, we will study how red-black trees are modified so that any order statistic can be determined in O(lgn) time. and how the rank of an element i.e its possible position in linear order of the set can be determined in O(lgn) time.

A data structure that can support fast order statistic operation is Red-Black Tree with little modification. e.g.



Each node x has a field size[x] which is the number of nodes in the subtree rooted at x including itself. An order statistic tree T is a RB Tree with additional property stored at each node. Each node now contains information as key[x], color[x], p[x], left[x], right[x] and size[x]. Sentinal nil size is 0. So we can have the identity

size[x] = size[left[x]] + size[right[x]] + 1.

In case of duplicate keys, the ambiguity of rank of an element is removed by defining the rank of an element as the position at which it would be printed in an in order walk of the tree.

Retrieving an element with a given rank.
An operation that retrieves an element with a given rank is performed by procedure OS_SELECT (x, i) which returns a pointer to the node containing the ith smallest key in the subtree rooted at x. To find out the ith smallest key in an order statistic tree T we call OS_SELECT (ROOT[T], i).

```
OS_SELECT (x, i)
1. r ← size[left[x]] + 1
2. if i = r
3.     then return x
4. else if i < r
5.     then return OS_SELECT (left[x], i)
6.     else return OS_SELECT (right[x], i - r)
```

#1 Rank of node x within the subtree rooted at x.
#2 x is the ith smallest element
#4 ith smallest element is in the x's left subtree
#6 ith smallest element is in the x's right subtree.

The value of size[left[x]] is the no. of nodes that comes before x in an in order tree walk of the subtree rooted at x.

Q. Search for the 17th smallest element in the order statistic tree give.

$i = 17$, key = 26 = x

$r \leftarrow$ size[left[x]] + 1 ← 13

$17 \neq 13$ false

$17 < 13$ false

OS_SELECT (41, 4)

$r \leftarrow$ size[left[x]] + 1 ← 6

~~$17 \neq 6$ false~~

$4 \neq 6$ false

$4 < 6$ true.

OS_SELECT (30, 4)

$x \leftarrow 30, i \leftarrow 4$

$r \leftarrow size[left[x]] + 1 \leftarrow 2$

$4 \neq 2$ false

$4 < 2$ false

OS_SELECT (right[x], i-r)

OS_SELECT (38, 2)

$x \leftarrow 38, i \leftarrow 2$

$r \leftarrow size[left[x]] + 1 \leftarrow 2$

$i = r$ true

return 38

Determing the rank of an element

OS_RANK (T, x)

1. $r \leftarrow size[left[x]] + 1$
2. $y \leftarrow x$
3. while $y \neq Root[T]$
4. do if $y = right[P[y]]$
5. then $r \leftarrow r + size[left[P[y]]] + 1$
6. $y \leftarrow P[y]$
7. return r

OS_RANK (T, 38)

$r \leftarrow 2$

$y \leftarrow 38$

while $y \neq Root[T]$ ✓

$y = y$ ✓

$r \leftarrow 2 + 2 \leftarrow 4$

$y \leftarrow 30$

while $y \neq Root[T]$ ✓

$y \neq 47$ ✗

$y \leftarrow 41$

key	rank
38	2
30	4

while $y \neq Root[T]$ ✓

$y = y$ ✓

$r \leftarrow 4 + 12 + 1 \leftarrow 17$

while $y \neq Root[T]$ ✗

return 17

Maintaining subtree sizes.

The subtree sizes has to be maintained for both insertion and deletion operation on red-black trees. This can be done without affecting the running time of either operation.

Insertion operation phases.

Insertion operation takes two phases

1) The first phase goes down the tree from the root, inserting the node as child of the existing node.
2) The second phase goes up the tree changing colors and performing rotations to maintain the red black tree properties.

solution for 1st phase

To maintain the subtree sizes in the first phase we increment size of x for each node on the path traversed from the root

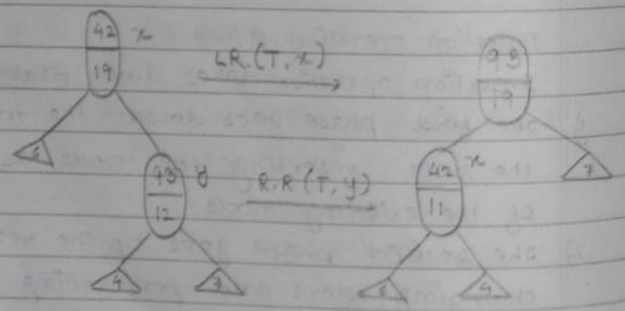
down towards the leaves where the x node lies. Thus the new node added gets a size of 1.

6/11/13

Solution for 2nd phase

In the second phase, the structural changes to RB trees are caused by rotations. There can be atmost two rotations. Rotations causes invalidation of sizes of two nodes to maintain the property two lines are added to the left rotate procedure.

12. size[y] ← size[x]
13. size[x] ← size[left[x]] + size[right[x]] + 1



Deletion operation Phases

Deletion operation involves 2 phases. The 1st phase slices out 1 node y.

Solution

To update the subtree sizes we simply traverse a path from node y up to the root decrementing the size field of each node on the path. The path has length O(lgn) in an n node RB tree. So additional time spent on maintaining size field is O(lgn).

The 2nd phase causes atmost 3 rotations and can be handled in the same way as in insertion, so the total time for deletion operation remains the same i.e O(lgn).

How to augument a data structure

Augumenting a data structure can be broken into 4 steps.

1) choosing an underline data structure
2) determining additional information to be ~~mati~~ maintainged in the underlining DS.
3) verifying that the additional information can be maintained for the basic modifying operations on the underline DS.
4) Developing new operations

To design Order Statistic Trees

step 1) We have choosen red black trees as underlining data structure

Step 2) The additional field size is choosen such that each node x stores the size of the subtree rooted at x.

step 3) we have ensured that insertion and deletion operations could maintain the size field

step 4) new operations are developed; OS_SELECT & OS_RANK.

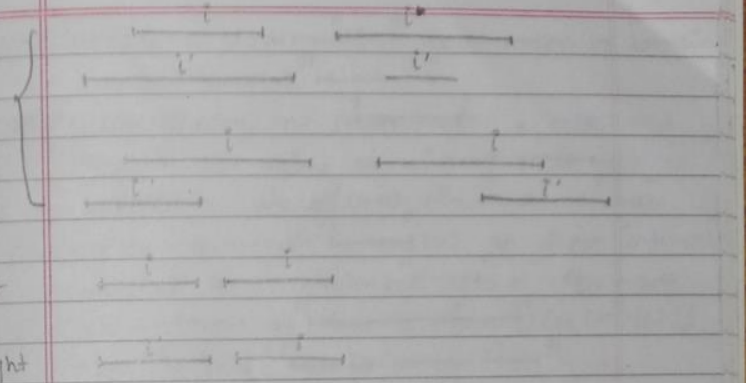
Interval Tree

RB trees are used to support operations on dynamic set of intervals. A closed interval is an ordered pair of real numbers $[t_1, t_2]$ with $t_1 < t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in R : t_1 \leq t \leq t_2\}$. Intervals are assumed to be closed. We can represent an interval $[t_1, t_2]$ as an object i with fields low[i] = t_1 and high[i] = t_2. We say that intervals i and i' overlap if $i \cap i' \neq 0$. i.e low[i] $\leq$ high[i'] or low[i'] $\leq$ high[i]. Any two intervals i and i' satisfy interval tripotomy (exactly one of the three property holds)

Prop1 : i and i' overlap

Prop2: i is to the left of i' (high[i] < low[i'])

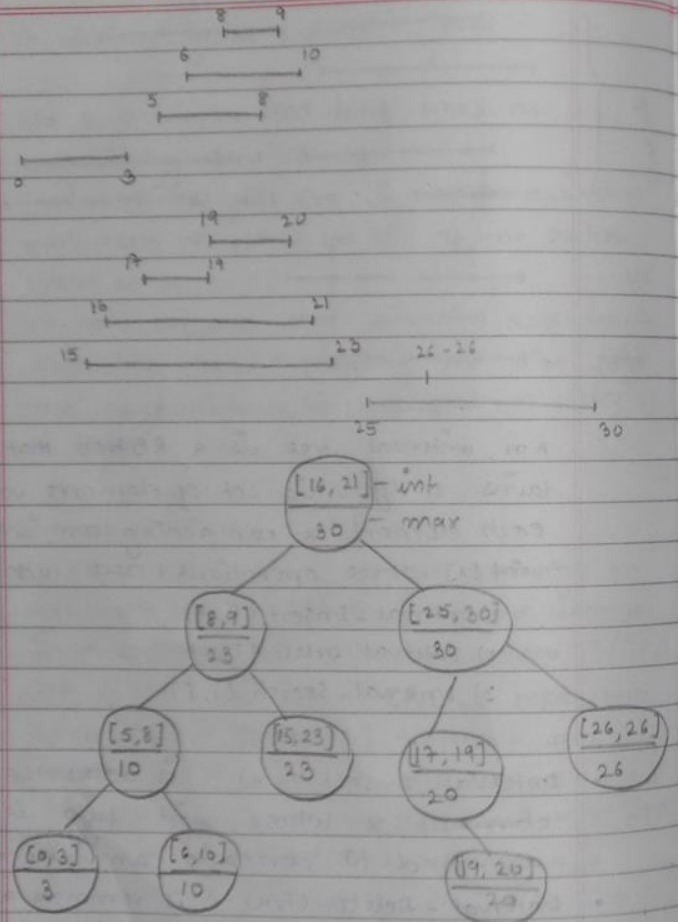
Prop3: i is to the right of i' (high[i'] < low[i])



An interval tree is a RB tree that maintains a dynamic set of elements with each element x containing an interval int[x]. Three operations are supported.

1) Interval_Insert (T, x)
2) Interval_Delete (T, x)
3) Interval_Search (T, i)

- Interval_Insert (T, x) - it adds the element x whose int field is assumed to contain an interval.
- Interval_Delete (T, x) - it removes an element x from the interval tree T.
- Interval_Search (T, i) - returns a pointer to an element x such that int[x] overlaps interval i or the sentinal nil[T] if no such element is in the set.



4 steps of Augumentation.

step1) Underline data structure - In RB tree, each node x contains an interval int[x] and key[x] is low end point i.e low[int[x]]. Thus inorder tree walk of the DS lists the intervals in sorted order by low end point.

step2) Additional Information - Each node x contains a value max[x].

step3) Maintaining the Information - Insertion & deletion operations on interval trees can be performed in O(lgn) time on n nodes. We can determine max[x] of given interval int[x] and maximum value of node x's children as max[x] = max(high[int[x]], max[left[x]], max[right[x]])

step4) Developing new operations - Interval search (T, i) finds a node in the tree T whose interval ~~op~~ overlaps interval i.

Interval_search (T, i) O(lgn)

1. $x \leftarrow$ Root[T]
2. while $x \neq$ NIL[T] and i does not overlap int[x]
3. do if left[x] ≠ nil[T] and max(left[x]) ≥ low(i)
4. then $x \leftarrow$ left[x]
5. else $x \leftarrow$ right[x]
6. return x.

Amortized Analysis.

In an Amortized Analysis, the time required to perform a sequence of DS operations is averaged over all the operations performed.

Three types of analysis methods -

1) Aggregate analysis - In this we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operations is thus $T(n)/n$. This average cost ~~an~~ is taken as amortized cost of each operation so that all operations have the same amortized cost.

2) Accounting method - In this we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost.

3) Potential method - It is like the accounting method, in which we determine the amortized cost of each operation and may overcharge operations early onto compensate for undercharges later.

Two examples are used to examine 3 methods
1) Stack - with additional operation MULTIPOP (which pops several objects at once.)
2) Binary counter - that counts up from 0 by means of single operation increment.

1) Aggregate Method - In aggregate analysis we show that for all n, a sequence of n operations takes worst case time $T(n)$ in total. In the worst case, the average cost or amortized cost per operation is $\therefore T(n)/n$. This amortized cost is applied to each operation even when there are several ~~time~~ types of operations in the sequence.

Stack operations

The stacks are analysed that have been augumented with a new operation. The two fundamental operations of stack takes $O(1)$ time

1. PUSH (S, x) - pushes object x onto the stacks
2. POP (S) - pops the top of stack S and returns the popped object.

The time taken by PUSH operation and POP operation is $O(1)$.

```
MULTIPOP(S, K)
1. while not STACK_EMPTY(S) and K>0
2.     do POP(S)
3.        K = K-1
```

Multipop(S,K) removes the K top elements from the stack S or pops the entire stack if it contains fewer than K objects.

STACK_EMPTY(S) returns true if there are no objects currently on the stack. Let us consider a sequence

of n PUSH, POP, MULTIPOP operations. The worst case cost for multipop in the sequence is O(n) since the stack size is atmost n. Thus the cost of the sequence is $O(n^2)$

** A sequence of n operations on an initially empty stack cost atmost O(n). Each object can be popped only once for each time it is pushed. ∴ the no. of times the pop can be called on a non empty stack including the calls to MULTIPOP is atmost the no. of PUSH operations which is atmost n. For any value of n, any sequence of n push, pop and multipop operations takes a total time O(n), so the average cost of operation is O(n)/n. = O(1). ∴, the amortized cost in aggregate analysis is equal to the average cost which is O(1).

Incrementing a Binary Counter

An array A[0... k-1] of bits are considered where length[A] = k. A binary no. x i.e stored in the counter has its lowest order bit in A[0] and highest order bit in A[k-1]

```
INCREMENT (A)
1. i ← 0
2. while i < k and A[i] = 1
3.     do A[i] ← 0
4.        i ← i+1
5. if i < k
6.   then A[i] ← 1
```

A single execution of increment takes O(k) in the worst case (when A contains all once). So a sequence of n executions takes O(nk) in the worst case (suppose initial counter is zero). Here we observe the running time in terms of no. of flips. But not all the bits flip each time increment is called.

counter Value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	0	0	0	0	0	31

A[0] flips every time, total n times
A[1] flips every other time, $\lfloor n/2 \rfloor$ times
A[2] flips every fourth time, $\lfloor n/4 \rfloor$ times, and so on for $i = 0, 1, \ldots, k-1$. A[i] flips $\lfloor \frac{n}{2^i} \rfloor$ times.

Thus total no of flips is $\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$ which is equal to $2n$.

Thus the worst case running time for a sequence of n increments is $O(n)$. The average cost is equals to $O(n)/n$ which is the amortized cost per operation equals to $O(n)$.

2) Accounting Method.

In accounting method, we assign differing charges to different operations with some operations charged more or less than they actually cost. The amount we charge for operation is called the amortized cost. The amortized cost is more or less than the actual cost. When amortized cost is greater than the actual cost the difference is assigned to specific objects as credits. The credits can be used by later operations whose amortized cost is less than the actual cost.

As a comparison, in aggregate analysis all operations have the same amortized cost. The amortized cost is split b/w actual cost and the credits

Different operations may have different amortized cost. Suppose the actual cost is c_i for the i^{th} operation in the sequence and amortized cost is $\hat{c}_i$. We require $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

should hold since we want to show that the average cost per operation is small using amortized cost, we need the total amortized cost as a upper bound of total actual cost.

Total credits stored in the DS is the difference b/w the total amortized cost & the total actual cost, which should be non-ve.

stack operations

The actual cost of stack operations is

PUSH	1
POP	1
MULTIPOP	min (s, k)

Let us assign the following amortized cost

PUSH	2
POP	0
MULTIPOP	0

Here all three amortized costs are $O(1)$, i.e constant. This concept is similar to a stack of plates in a cafetaria. Suppose $1 represents a unit cost. When we push a plate on the stack we use $1 to pay the actual cost of push operation & leave $1 on the plate as

credit. Whenever popping a plate the $1 on the plate is used to pay the actual cost of the pop. By charging PUSH a little more, we do not charge POP and MULTIPOP. At any point in time, every plate on the stack has a $ of credit on it. The $ stored on the plate is prepayment for the cost of popping it from the stack. When we execute a pop operation, we charge the operation nothing & pay its actual cost using the credit stored in the stack. Similarly for the MULTIPOP we do not charge anything and pay its actual cost. The total amortized cost for n push, pop and multipop is $O(n)$. The average amortized cost is $O(n)/n = O(1)$ for each operation.

Binary Counter

Incrementing a binary counter

The running time of this operation is proportional to the no. of bits flipped. Let us use a $ bill to represent each unit of cost (i.e the flip of 1 bit). For amortized analysis, we charge amortized cost of $2 to set a bit to 1. Whenever a bit is set we use $1 to pay the actual cost out of $2 and store $1 as credit on th bit. When a bit is reset, the stored $1 pays the cost of flipping it back to zero. At any point, a one in the counter stores $1. The no. of 1's is never -ve. So the total credit can also be non-ve. Atmost 1 bit is set in each operation, so the amortized cost of an operation is atmost $2's Thus total amortized cost of n operations is $O(n)$ and average cost is $O(1)$.