ORDER OF GROWTH

The O-notation asymptotically bounds a function from above: Let f(n) and g(n) be two functions. We say that f(n) = O(g(n)) if there exist positive constants c and n_0 such that f(n) ≤ cg(n) for all n ≥ n_0 .

The Ω-notation asymptotically bounds a function from below: Let f(n) and g(n) be two functions. We say that f(n) = Ω(g(n)) if there exist positive constants c and n_0 such that f(n) ≥ cg(n) for every n ≥ n_0 .

The Θ-notation asymptotically bounds a function from above & below: Let f(n) and g(n) be two functions. We say that f(n) = Θ(g(n)) if there ∃ positive constants c_1 , c_2 and n_0 s.t. c_1g(n) ≤ f(n) ≤ c_2g(n) for all n≥n_0

Rule of Sums: Suppose that $T_1(n), T_2(n)$ are running times of two program fragments. $T_1(n)$ is $O\big(f(n)\big)$, $T_2(n)$ is $O\big(g(n)\big)$. Then running time of: $T_1(n) + T_2(n)$ is $O(\max[f(n), g(n)])$

Rule of Products: If $T_1(n)$ and $T_2(n)$ are $O\big(f(n)\big)$ and $O\big(g(n)\big)$ respectively, then $T_1(n) \cdot T_2(n)$ is $O\big(f(n) \cdot g(n)\big)$. It follows from this that $O\big(cf(n)\big) \equiv O\big(f(n)\big)$ if $c =$ const. > 0.

General Rules for Running Time Analysis:

1. Runtime of each basic **assignment, read and write statement** can be taken as $O(1)$.
2. **Sequence of statements:** the largest runtime of any statement in the sequence, with a constant factor.
3. **IF statement:** Cost of conditionally executed statements, plus time for evaluating the condition (usually $O(1)$). IF-THEN-ELSE: time to evaluate condition plus largest of execution times of different blocks corresponding to the condition.
4. **Loop:** Sum, over all loop executions, of the time to execute the body & time to evaluate the termination condition (usually $O(1)$) . Often, this is the product of number of loop iterations and largest possible time for a single execution.

Runtime w. Recursion: Can't find ordering of ∀ procedures s.t each only calls previously evaluated ones. Instead - associate unknown time function $T(n)$ w. each recursive procedure. n measures size of arguments to the procedure. We get a *recurrence* for $T(n)$, i.e. an eqn for $T(n)$ in terms of $T(k)$ for various values of k.

RECURRENCE SOLVING

Iteration Method: In iteration method we iteratively "unfold" the recurrence until we "see the pattern".

Recursion-tree Method: Convert recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. Sum the costs within each level of the tree to get a set of per-level costs. Then sum all the per-level costs to determine the total cost of all levels of the recursion. Aka number of levels * cost/level.

Master theorem: Let a ≥ 1 and b > 1 be consts., let f(n) be a function, & let T(n) be defined on nonnegative integers by recurrence: **T(n) = aT(n/b) + f(n)**. Let p=$\log_b a$. Then T(n) has the following asymptotic bounds:
<u>1.</u> If f(n) = O($n^{p-\epsilon}$) for some constant ε > 0, then T(n) = Θ(n^p).| <u>2.</u> If f(n) = Θ(n^p), then T(n) = Θ($n^p \cdot \log n$).
3. If f(n) = Ω($n^{p+\epsilon}$) for const. ε > 0, & a·f(n/b) ≤ c·f(n) for const. c < 1 & all big enough n, then T(n) = Θ(f(n)).

BINARY SEARCH. Runtime: $O(\log_2 n)$

```
BinarySearch (int n, sorted array of ints A, int x)
    min=1, max=n, found=FALSE
    while (found == FALSE and min<=max)
        mid= floor((min+ max)/2)
        if (A[mid]=x)     :   found = TRUE
        else if (x < A[mid])   :   max = mid-1
        Else   :   min=mid+1
    If (found = TRUE)    :   Return (mid)
    Else   :   Return (NOT-FOUND)
```

Search through **sorted** data.
1. Initial search region is the whole array
2. Look at value in middle of search region
3. If target found, stop
4. If target is less than middle value, new search region is lower half of data.
5. If target is greater than middle data value, new search region is higher half of data
6. Continue from step 2.

INSERTION SORT. Runtime: $O_{worstcase}(n^2)$, $\Omega_{\text{bestcase}}(n)$, $\Theta(n^2)$. $T_{is}(n) = \Theta(n^2)$

```
InsertionSort(Input: integer n, array A){
  for (j=2 to n){
     newnum=A[j]  #newnumberwewanttoinsert
     i=j-1
     while (i>0 and newnum<A[i]){ #moveallnumbers
         A[i+1]=A[i]      # > newnum by one
         i=i-1     }        #  position to the right
     A[i+1]=newnum # inserted in correct position
```

Description: Each **iteration j, consider element in A[j]**, insert it in correct position. To determine where A[j] inserts, **go through A[1…j-1] starting at A[j-1],** going toward the left, **shifting** each element **greater** than this by **one to the right**. When we find an element that isn't greater than A[j] or get to left end of the array, insert element originally in A[j] into its new pos. in array.

MERGE SORT. Runtime: $\Theta(n\log(n))$

```
MergeSort (Input: array A, integers p,r) {
    if (r<=p)    :   return
    else{
       q=floor( (p+r)/2 )
       MergeSort(A,p,q)
       MergeSort(A,q+1,r)
         Merge(A,p,q,r) #mergestwosortedsub-arrays
 }#to sort A call MergeSort(A,1,n)
```

Description: Given array A of size n- **Divide** it into 2 sub-arrays of size $n/2$. **Conquer** by sorting each of the two sub arrays recursively using mergesort. **Combine** by merging two sorted sub-arrays to produce one sorted array.
Merge($\boldsymbol{A, p, q, r}$): Merges sub arrays $A[p \dots q]$ and $A[q+1 \dots r]$, which are sorted after the recursive calls MergeSort(A,p,q) and MergeSort($A,q+1,r$). Takes $\Theta(n)$ time.

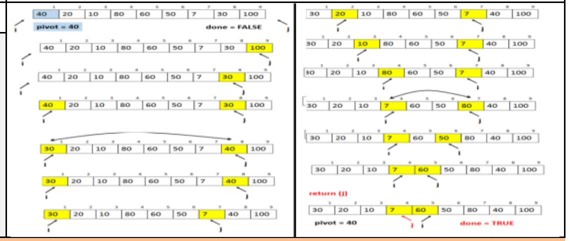
QUICKSORT. Runtime: $\Theta_{\text{worst}}(n^2)$, $\Theta_{\text{best}}(n\log(n))$ best if partitions balanced

```
QuickSort (Input: array A, integers p,r) {
    if (r==p) : return
    else
         q = Partition(A,p,r)
         QuickSort(A,p,q)
          QuickSort(A,q+1,r)
}#to sort A call QuickSort(A,1,n)
```

Divide: Parition (rearrange) array A[p…r] into two sub array A[p…q] and A[q+1…r] s..t each element of A[p…q] is ≤ to each element of A[q+1…r]
Conquer: Sort the two sub arrays by recursive calls to quicksort
Combine: As the sub-arrays are already sorted, no work to combine them: entire array $A[p \dots r]$ is sorted.

Partition. Time: linear in array size, cn (∀ inputs)

```
Partition (A,p,r)
   pivot = A[p] , i=p-1 , j=r+1 , done = FALSE
   while (done == FALSE) {
         repeat j=j-1 until A[j] <= pivot
         repeat i=i+1 until A[i] >= pivot
         if i<j then exchange A[i] with A[j]
         else done = TRUE
  return (j)
```



COMPARISON SORTS

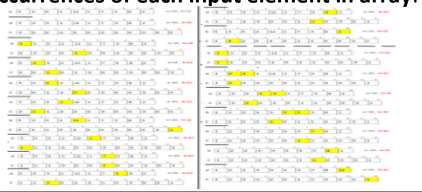
Comparison Sorts: Algorithms that determine sorted order based only on comparisons between the input elements. They don't use the values of the elements, but just their relative order. **Any comparison sort takes $\boldsymbol{\Omega(n\log(n))}$ in the worst case**. Insertion Sort, Merge Sort, QuickSort are all comparison sorts.

Comparison Sort as a Binary Decision Tree: Each node represents a set of possible orderings, consistent with comparisons that have been made among the elements. The results of the comparisons are the tree edges. Worst case number of comparisons by the sorting algorithm is equal to the depth of the deepest leaf. A decision tree to sort n elements must have $n!$ Leaves at least. A binary tree of depth d has at most 2^d leaves. If k =number of leaves then: $n! \le k \le 2^d \rightarrow n! \le 2^d \rightarrow$ tree depth $\ge \lceil \log_2(n!) \rceil$

Counting Sort. Runtime: $\theta_{\text{all cases}}(n+k)$, $\Theta_{\text{when } k=O(n)}(n)$

```
CountingSort (A,n,k) {
     for i=1 to k  :  C[i]=0  #initialize all k entries in C to 0
     for j=1 to n {
          val = A[j] #val appears in A
          C[val]=C[val]+1  #+1 counter for 'val'
     }
     j=1
     for i=1 to k { # ∀1 ≤ i ≤ k (may appear in A)
          count = C[i]  # times i appears in A
          for t=1 to count {
               A[j]=i #copy value i count (=C[i]) times into A
               j=j+1
```

Assumes **each of the $\boldsymbol{n}$ input elements is an integer** in range 1 to k, k =integer. Using another array of size k, **count number of occurrences of each input element in array**.



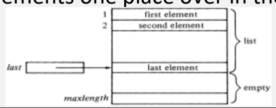
ABSTRACT DATA TYPES (ADT)

LIST ADT

Definition:	Sequence of zero or more elements of a given type. $L = [a_1, a_2, a_3, \dots, a_N]$, $\forall N \ge 0$
INSERT(x, p, L)	Insert x at pos. p in list L, moving elements at p & following positions to next higher pos.
DELETE(p, L)	Delete element at position p of list L. Undefined if L has no position p or if $p =$ END(L).
FIND(x, L)	Return the position of x on list L. If x appears more than once, then the position of the first occurs is returned. If x does not appear at all, then END(L) is returned.
RETRIEVE(p, L)	Return element at position p on list L. Undefined if $p =$ END(L) or if L has no position p.
FIRST(L)	Returns the first position on list L. If L is empty, the position returned is END(L).
LAST(L)	Returns the last element in list L. | EMPTY(L) | TRUE if the list L is empty, FALSE otherwise.

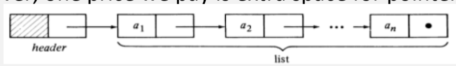
Array Implementation of a List

Elements stored in contiguous cells of an array. Array is easily traversed and appended. Inserting elements into middle, or deleting any element except the last, however, all require shifting multiple elements one place over in the array.



Pointer Implementation of a List (Linked List)

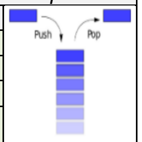
Here, list is made up of cells. Each cell contains an element of the list and a pointer to the next cell on the list. This frees us from using contiguous memory for storing a list and hence from shifting elements to make room for new ones, or close gaps created by deleting elements. However, one price we pay is extra space for pointers.



STACK:

STACK ADT (aka: Pushdown List ; Last-In First-Out [LIFO] List)

Definition:	A list in which all insertions and deletions take place at one end, called the *top*.
Reverse order:	POP each element & PUSH it into another stack. This new stack is the reverse .
PUSH(x, S)	Insert element x on top of the stack S.
POP(S)	Removes and returns the element from the top of stack S.
TOP(S)	Returns the element at the top of stack S
EMPTY(S)	Returns TRUE if stack S is empty, and FALSE otherwise.



Implementation of a Stack using Arrays:

elements[ ]: array of size MAX (elements in stack), integer **top** (indexes most recently inserted element)

```
Empty(S) {
  if (S.top == 0)
      return TRUE
  else
      return FALSE
}
```

```
PUSH(x,S) {
  if (S.top < MAX)
      S.top = S.top + 1
      S.elements[S.top] = x
  else
      return "stack full"
```

```
POP(S){
if (!EMPTY(S))
  S.top = S.top - 1
  return (S.elements[S.top+1])
else
  return "stack empty"
```

```
TOP(S){
  if (!EMPTY(S))
   return (S.elements[S.top])
  else
   return "stack empty"
}
```

Implementation of a Stack using Linked Lists

A struct containing one field: **top** – a pointer to node. Top is a pointer to the most recently inserted element to the stack.

```
Empty (S) {
  if (S.top == NULL)
     return TRUE
   Else
     return FALSE
}
```

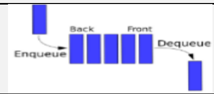
```
PUSH(x, S){
  make new node, let p be ptr. to it
  p->element = x
  p->next = S.top
  S.top = p
}
```

```
POP(S) {
  if (!EMPTY(S))
     x= S.top->element
     S.top = S.top -> next
     return (x)
  Else : return "stack empty"
}
```

```
TOP(S) {
  if (!EMPTY(S))
    return (S.top->element)
  else
    return "stack empty"
}
```

QUEUE ADT (aka First-In First-Out [FIFO] list)

Definition:	A list in which all insertions happen at one end (*the rear*), and all deletion happen at the other end (*the front*)		
EMPTY(Q)	Returns TRUE if the queue Q is empty, else FALSE	In terms of fundamental List Operations:	
DEQUEUE(Q)	Remove & return element from front of queue Q.		DELETE(FIRST(Q), Q)
FRONT(Q)	Return the first element of the queue Q.		RETRIEVE(FIRST(Q), Q)
ENQUEUE(x, Q)	Insert element x at the back of the queue Q.		INSERT$(x$, END(Q), $Q)$



Implementation of a Queue using Linked Lists: Running time: $O(1)$.

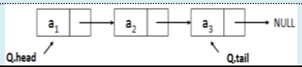
```
ENQUEUE(val, Q)
  p.element := val
  p.next := NULL
  if (Empty(Q))
     Q.head=p
     Q.tail=p
  Else
     Q.tail -> next :=p
     Q.tail := p
```

```
DEQUEUE(Q){
if (! EMPTY(Q)) {
  p := Q.head
  Q.head := p→next // Update Q.head
    If (Q.tail== p) // Had 1 elmnt, now empty lst
   Q.tail:= NULL
  return(p→element) }
else :  return(QUEUE_IS_EMPTY) }
```

```
EMPTY(Q){  //Q.head=null -> list empty
if (Q.head == NULL) :   return(TRUE)
else :   return(FALSE) }
```

```
FRONT(Q({
if (! EMPTY(Q))
  p := Q.head // Q.head points to 1 st cell
  return(p→element)
else :  return(QUEUE_IS_EMPTY) }
```

Queue=struct w. 2 fields: **head** = ptr to node, **tail** =ptr to node



In case there is no pointer to tail fo the queue (i.e. we only have **head**, but not **tail)**: EMPTY, FRONT, DEQUEUE don't change. ENQUEUE changes, new runtime is $O(n)$

```
ENQUEUE_NOTAIL(val, Q)
  p.element := val
  p.next := NULL
  if (Empty(Q))  : Q.head=p //emptlst, insert to head
  Else
     c = Q.head
     While (NULL != c.next) do : c = c.next
     c.next :=p
```

Implementation of Queue using Arrays

Queue is a struct with 3 fields: **elements[]**=an array of size MAX, int **head**, int **tail**. Head & Tail initialized to 1.

```
ENQUEUE(val, Q)
  If (Q.tail >MAX)
    Return (QUEUE_FULL)
  Else
    Q.elements[Q.tail] := val
    Q.tail = Q.tail +1
```

```
DEQUEUE(Q)
  If (!EMPTY(Q))
    Q.head := Q.head +1 //update Q.head
    Return (Q.elements[Q.head-1])
  Else
    Return(QUEUE IS EMPTY)
```

Problem: After we enqueue MAX elements to the queue, and then dequeue them, we can't use this queue anymore.
Solution: Use Cyclical Lists (doughnut queue)

GRAPH THEORY

Graph Theory:

Graph:	Finite set of vertices with edges between vertices. Formally, graph G is a pair of sets (V, E) where V is the set of vertices and E is the set of edges formed by pairs (x, y) of vertices in V. **Number of vertices**, $\|V\|$ is denoted by $\boldsymbol{n}$. **Number of edges**, $\|E\|$ is $\boldsymbol{m}$.
Adjacent Vertices:	Two vertices u and v are adjacent (are neighbors) if they are connected by an edge.
Degree of a vertex:	Degree of vertex is the number of its neighbors. Denote degree of vertex v by deg(v).
Path:	A sequence of vertices $v_1, v_2, \ldots, v_k$ such that each vertex is adjacent to the next. Means, each consecutive pair v_i, v_{i+1} are joined by an edge E. A path is **simple** if all vertices are distinct. The **length** of a path is the number of edges in that path.
Distance bn Vertices:	Length of the shortest path between then. If there is no path, distance is ∞.
Cycle:	A **closed** path. That is, we start and end at the same vertex. A cycle is **simple** if all vertices (except the first/last one) are distinct.
Connected Graph:	A graph is **connected** if any two vertices can by joined by a path. Else – disconnected.
Connected Components:	Connected parts of a **disconnected** graph. Any 2 vertices in a **connected component** are connected by paths. ∄ paths between. vertices from diff. connected components.

Representations of Graphs			Time to examine all	neighbors
Representation	**Definition (for $G = (V, E)$**	**Memory Size**	**Check if edge (u, v) in G:**	**of vertex v:**
Adjacency Matrix M_G	$n \times n$ 0/1 valued matrix. $M_G[i,j] = \begin{cases} 1 \,; (i,j) \in E \\ 0 \,; else \end{cases}$	$\Theta(n^2)$	$O(1)$ (const. time)	$\Theta(n)$
Adjacency List L_G	Array L_G of n lists, one for each vertex in V.For each vertex i, $L_G[i]$ is ptr. to list of its nghbrs.	$\Theta(m+n)$	$\Theta(\min[\deg(u), dev(v)])$	$\Theta(\deg(v))$

GRAPH TRAVERSAL (visiting, checking, updating each vertex)

Breadth-First Search (BFS). Assumes adjacency list representation(sorted up). Given graph $G = (V, E)$ & source vertex v, we will systematically visit every vertex in G that's reachable from v

```
BFS(G,v) {
 for every vertex x in V #V={1,...,n}
      visited[x] := FALSE #touch every vrtx: Θ(|V|) = Θ(n)
      print(v)
      ENQUEUE(v,Q)
      visited[v] := TRUE #v is visited
      while (not EMPTY(Q)) {  #executed once per each
         x := DEQUEUE(Q)  #reachable vrtx from v
         for every neighbor y of x { #executes deg(x) times
              if (visited[y] = FALSE) { #1st time visit y
              print(y)
              visited[y] := TRUE
              ENQUEUE(y,Q)
```

Start at source vertex v, and visit the neighbor of v level by level: First visiting all vertices that are neighbors of v (I,e, 1 distance), then vertices that are neighbors of neighbors of v (i.e. 2 distance). Then distance three, etc.

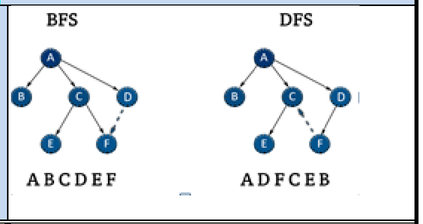
BFS Uses: Array **visited**[] of size n, to track which vertices **have already been visited,** to avoid visiting more than once. A **queue $\boldsymbol{Q}$ to maintain all visited vertices** in order to visit their unvisited neighbors in the next level.

Runtime: Connected: $\boldsymbol{O(n+m)}$, $\sum_{x \in V} \deg(x) = 2m$ **Disconnected: $\boldsymbol{\Theta(n + m(v))}$**, $\sum_{x \in C(v)} \deg(x) = 2m(v)$, $C(v)$ =vertex set of connected component of v. $m(v)$ =number of edges at the connected component of v.

Depth First Search (DFS)

```
DFS(G, v)
  For i=1 to n
    Visited[i] := FALSE
  DFS-Recursive(G,v)
DFS-Recursive (G, x)
  Print(x)
  Visited[x] := TRUE
  For every neighbor y of x
    If (not visited[y])
      DFS-Recursive (G,y)
```

DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once all of v's edges have been explored, search "backtracks" to explore edges leaving the vertex from which v was discovered. This continues until we discovered all vertices reachable from original source vertex.



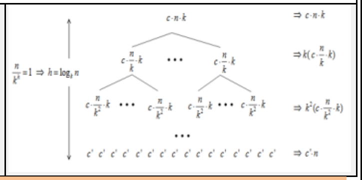
	Basic	Storing nodes in	Memory	Structure of Tree	Traversal fashion	Optimality	Application (Examines...)
BFS	Vertex Based	Queue	Inefficient	Wide & short	Oldest unvisited vertices first	Opt for shortest dist., not cost	bipartite graph, connect component & shortest path
DFS	Edge-based	Stack	Efficient	Narrow & long	Vertices along the edge first	Not optimal	Two-edge connected, strongly connected, acyclic, topologically ordered graphs.

Divide and conquer:

Divide problem into a number of sub-problems that are smaller instances of the same problem. **Conquer** sub-problems by solving them recursively. **Combine** solutions to the sub-problems into solution for original

K-MERGE SORT.

Given array A of size n, the k-merge sort does:**Divide:** Divide the array into k sub arrays of size n/k. **Conquer:** Sort each of the k sub arrays recursively using MergeSort. **Combine:** Merge k sorted sub arrays to produce one sorted array.**Recurrence:** $T(n,k) = kT(n/k\,,k) + cnk, n > 1. T(1,k) = c'$
Add **costs across each level of tree**. Level I below the top has k^i nodes, each contributing a cost of $c(n/k^i)k = cnk$. Bottom level has n nodes, each contributing a cost of c. Recursion tree has $\log_k n$ levels. **Total cost** is $cnk \cdot \log_k n + cn = \Theta(nklog_k n)$. **For k=2, MergeSort**, we get $\Theta(nlogn)$ as seen before



PRIORITY QUEUE ADT

Priority Queue ADT PRIORITY QUEUES ARE IMPLEMENTED WITH HEAPS

A **priority queue** is like a regular queue, but where additionally each element has a "priority" associated with it. In a max priority queue, an element with high priority is served before an element with low priority.

Implementation with a Linked List: If we use a linked list, we have a choice of sorting it or leaving it unsorted. If we sort the list, finding a minimum is easy – just take the first element. However, insertion requires scanning half the list on average to maintain the sorted list. On the other hand, we could leave the list unsorted, which makes insertions easy, and selection of a minimum more difficult.

Operation	Max Priority Queue	Min Priority Queue
INSERT(x,Q)	Inserts an element x with an associated priority to Q	
MAX(Q)	Returns the highest priority event in Q	Returns the lowest priority element in Q
DELETEMAX(Q)	Deletes and returns the element from the queue that has the highest priority.	Deletes and returns the element from Q that has the lowest priority.

Priority Queue Implementation with a Partially Ordered Binary Tree

In list implementation, we spend time proportional to n to implement either INSERT of DELETEMIN. In a partially ordered tree implementation, DELETEMIN and INSERT both require $O(\log(n))$ steps

- **At lowest level**, some leave might be missing, require all missing leaves be right of all present leaves.
- **The tree must be *partially ordered***: the priority of node v is no greater than the priority of the children of v, where the priority number of a node is the propriety number of the element stored at the node.
- **Implementing the functions:**
  - **DELETEMIN**: Return minimum-priority element, which must be at root. If we simply remove the root, we no longer have a tree. To maintain the properties (partially ordered, balanced, leaves @ lowest level) take the rightmost leaf at the lowest level, and temporarily put it at the root. Then push this element as far down the tree as it will go, by exchanging it with the one of its children that has a smaller priority, until the element is either at a leaf or at a position where it has priority no larger than either of its children.
    - **Running time:** DELETEMIN applied to set of n elements takes $O(\log(n))$ time, since no path in the tree has over $1+\log(n)$ nodes, and process of forcing an element down the tree takes a constant time per node. For constant c, $c(1+\log(n))$ is at most $2clog(n), \forall n \geq 2$. Thus $c(1+\log(n))$ is $O(\log(n))$
  - **INSERT**: Place new element as far left as possible on the lowest level, starting a new level if the current lowest level is all filled. If the new element has priority lower than its parent, exchange it with its parent. The new element is now at a position where it is of lower priority than either of its children, but it may also be of lower priority than its parent. In that case, we must exchange it with its parent again, and keep repeating this process until the new element is either at the root, or has larger priority than its parent.
    - **Running Time**: Time to perform an insertion is proportional to distance up the tree that the new element travels. The distance can be no greater that $1+\log(n)$, so INSERT also takes $O(\log(n))$

TREES:

B-Trees: B-Tree is a generalization of 2-3 tree. **Explanation:** Each internal node has at least t_1 children and at most t_2 children. The root has at least 2 children and at most t_2 children. All leaves are at the same distance from the root. Values of the set are kept at the leaves, sorted from smallest to largest. A 2-3 tree is a B-tree with $t_1 = 2, t_2 = 3$.
Implementation: Each node in a B-tree struct contains: **Parent** – ptr to its parent. **Num_of_children**, **min_values[]** – an array of size t_2 that contains the minval of each Subtree, means min_value[i] is the minval at the Subtree rooted by children[i]. **Children[]** – array of size t_2 contains pointers to the children. **Height:** What is max height & min height of tree if it has n leaves? $\log_{t_2} n \leq h \leq \log_{t_1} n$. **Searching for nodes:** Given x, how to find the leaf with this value x, assume $t_2 < ct_1$? In each step apply a version of binary searchto choose the correct Subtree. Runtime: $O(\log_{t_1} n \cdot \log_2 t_2) \rightarrow O(\log_2 n)$**Inserting nodes:** Given x, how can we insert a new leaf w this value x? After the split we have to make sure that the number of children in each internal node is at least t_1, so $\frac{t_2+1}{2} > t_1$. Worst case we split each internal node in the path from leaf to the root. Runtime: $O(\log_{t_1} n \cdot t_2\ )$

TREES

Definition:	Collection of elements called *nodes*, one of which is distinguished as a *root*, along with a ('parenthood') that places a hierarchical structure on nodes. Nodes can be of any type.
Null Tree:	A "tree" with no nodes, denoted by Λ. **Siblings:** Children of the same node
Tree *Path*:	If $n_1, n_2, \dots, n_k$ is a sequence of nodes in a tree s.t. n_i is the parent of n_{i+1} for $1 \leq i < k$, then this sequence is called a *path* from node n_1 to node n_k.
Path Length:	The *length* of a path is one less than the number of nodes in the path.
Descendant:	If there is a path from node a to node b, then a is an *ancestor* of b, and b is a *descendant* of a. An ancestor/descendant of a node is called *proper* if it is not the node itself.
Root:	Only the root node of a tree can have no proper ancestors.
Leaves:	Node with no proper descendants. **Subtree:** A node, together with all of its descendants.
Node Height:	The *height* of a node in a tree is the length of the longest path from the node to a leaf. The height of the entire tree is the height of the root.
Node Depth:	The *depth* of a node is the length of the unique path from the root to that node.
Analogy to List:	tree:list = label:element = node:position (not all trees are labeled)

TREE ADT

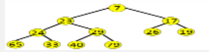
PARENT(n,T)	Return the parent of node n in tree T. If n is the root, which has no parent, Λ is returned. Λ is a "null node", used to signal that we navigated off the tree.
LEFTMOST_CHILD(n,T)	Returns the leftmost child of node n in tree T, and returns Λ if n is a leaf.
RIGHT_SIBLING(n,T)	Returns right sibling of node n in tree T, defined as node m with same parent p as n such that m lies immediately to the right of n in the ordering of the children of p.
LABEL(n,T)	Return label of node n in T. We don't require labels to be defined for every tree.
CREATE$i(v,T_1,T_2,\dots,T_i)$	Infinite family of functions, one for each value of $i = 0,1,2$ .... CREATEi makes new node r w. label v & gives it i children, which are roots of trees $T_1, T_2, \dots, T_i$ in order from left. Tree with root r is returned. If $i = 0$ then r is both a leaf and the root.
ROOT(T)	Returns the node that is the root of tree T, or Λ if T is the null tree

COMPLETE BINARY TREE

Complete Binary Tree

Definition: All levels are full (except maybe the last). The last level leaves are "pushed to the left"

Array Representation: Store tree elements in array from top to bottom, left to right. Root of the tree is $Q[1]$:



Q	1	2	3	4	5	6	7	8	9	10	11
	7	23	17	24	29	26	19	65	33	40	79

Indices of Relatives: Given the index i of a node, compute the indices of its parent and children: $LeftChild(i): 2i, RightChild(i): 2i+1\ , Parent(i): \lfloor i/2 \rfloor$. **Height:** log(n)

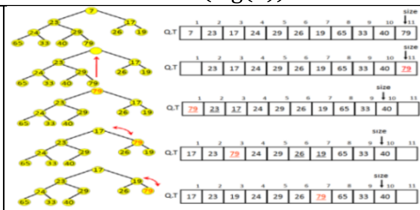
Min Heap: A complete binary tree. Values in nodes satisfy **Min-Heap property**: value of node is smaller than values of its children $\Rightarrow$ smallest element in a min-heap is stored at **root**. No ordering property between children.

PRIORITY QUEUE

Definition	A struct w. 2 fields: **array *T*** represents the heap. Integer **size** – number of elements in heap.
Find Minimum:return Q.T[1]	**Make Heap:**$O(n)$,**Insert:** $O(\log n)$,**Deletemin:**$O(\log n)$,**Min:** $O(1)$

Delete the Minimum. Runtime: $O(\log(n))$

```
DeleteMin(Q) {
  if (EMPTY(Q))
    return (PQ-EMPTY)
  min = Q.T[1] #save min value
  if (Q.size==1) #special case, 1 elmnt.
    Q.size = Q.size - 1
  else
    Q.T[1]=Q.T[Q.size]#lastleaf.valputinroot
    Q.size = Q.size – 1 #remove r-most leaf
    Heapify-down(Q.1) #"fix" heap
  return (min)
```



Complexity: Constant # operations per recursive call. Number of recursive calls = depth of tree. Depth of tree = $O(\log(n))$

Inserting an Item. Runtime $O(log(n))$

```
Insert(x,Q) {
  if (Q.size = MAXSIZE) : return(OVERFLOW) #queue is full
  Q.size = Q.size + 1 #add new leaf
  Q.T[Q.size] = x #put x in new leaf
  Heapify-up(Q,Q.size)
```

Complexity: Constant # of operations per iteration. Number of iteration = depth of tree. Depth of tree = $O(\log(n))$

HEAP OPERATIONS

Heapify Down. Runtime: $O(n)$

```
Heapify-down(Q,i) {
   if (2  > P.size) return() #reached  leaf
   if (2 +1 > P.size)
        if (P.T[i] > P.T[2 ] #value>Lchild
             Swap(P.T[i], P.T[2]) #swap vals w. Lchild
        return()
   else #reached vertex w 2 children
        if (P.T[2 ] < P.T[2 +1] #select smaller child
             j = 2
        else j = 2 +1
        if (P.T[i] > P.T[j])
             P.T[i] = P.T[j]
             Heapify-down(P,j) #call recursively on child
   else return() #can stop fixing since<=both children
```

Heapify Up.

```
Heapify-up(Q,i) {
 while (i>1 and Q.T[i] < Q.T[parent(i)] {
 #aslongasrootisnotreachedandvalue<thanparent
   swap(Q.T[i],Q.T[parent(i)] #swap val w. prnt
   i = parent(i) #update position
  }
```

Heap Sort. Runtime: $O(n\,log(n))$

```
Heap-Sort(A,n) {
   for (i=1 to n) // this loop O(nlogn)
        INSERT(A[i], Q)
   for (i=1 to n) // this look O(nlogn)
        A[i] = DELETEMIN(Q)
}
```

Can replace 1st loop with: Array-to-Heap(A,n)

Make Array into Heap. Runtime: $O(n)$, since $\sum_{k=1}^{h} 2^{-k} \cdot k < 2$, and $O(k) = O(n)\sum_{k=1}^{h} 2^{-k} \cdot k$

```
Array-to-Heap(A,n) #A=array w. n numbers
   for (i=1 to n)
        Q.T[i] = A[i] #copy entries of A into heap
   Q.size = next
   for (i = floor(Q.size / 2) downto 1) #start @ parent of rightmost leaf & go up the tree
        Heapify-down(Q,i) #when call Heapify-down(Q,i),subtrees rooted @ kids of i are heaps.
```

(1): put the elements in an array - $\Theta(n)$. (2) Make array into a heap: (k=h-L) $\sum_{l=0}^{h-1} 2^l\, O(h-l)$

K min vals in array: (1) Heap from array w. Array-To-Heap ($\Theta(n)$). (2) call deleteMin k times to get k min vals ($\Theta(klogn)$

THE DICTIONARY ADT

The Dictionary ADT -. All operations take $O(1)$ time in the worst case.

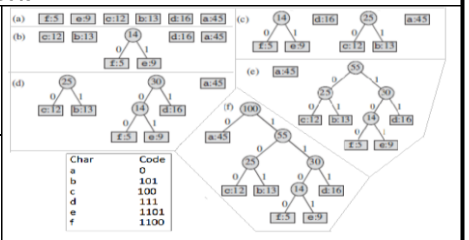
INSERT(x, D)	**DELETE(x, D)**	**SEARCH(x, D)**
Inserts x to D	Deletes x from D	Returns TRUE if x in D, and FALSE otherwise.

INSERT(x, D) {
D[x]=1
}
	DELETE(x, D) {
	D[x]=0
	}
		SEARCH(x, D) {
		If D[x] == 1 : return TRUE
		Else : return FALSE }

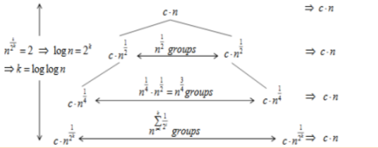
HUFFMAN CODING

Huffman Coding

Huffman Code:	Greedy algorithm that constructs an optimal prefix code. Based on frequency of occurrence of source values. Common symbols are represented using less bits than uncommon symbols.
Building a Tree:	Algorithm builds tree T corresponding to the optimal code in a bottom up manner. Begins with a set of $\lvert C\rvert$ leaves – a leaf for each character c in alphabet C. Then it performs a sequence of $\lvert C\rvert - 1$ "merging" operations to create the final tree. Uses a **min-priority queue** Q, keyed on the **freq** attribute, to identify the **two least-frequent** objects to merge. When we **merge** two objects, result is a **new** object whose frequency is the **sum** of the frequencies of the merged objects
While heap contains two or more nodes:	Create new node. DeleteMin node, make it left Subtree. DeleteMin next node, make it right Subtree. Frequency of new node = sum of frequency of left and right children. Enqueue new node back into queue.
Obtaining Codewords from the Tree:	Perform a **traversal** of the tree to obtain **new code words.** Going **left** is a **0**, Going **right** is a **1.** Code word is only **completed** when a **leaf** node is reached.
Memory Usage:	$\forall c \in C$, let freq(c) = frequency of c in file, and let d(c) be depth of c's leaf in tree, which is also the length of codeword for character c. # of bits to encode: $\mathcal{B} = \sum_{c\in C}$ freq$(c) \cdot$ d(c)
Encoding:	Rescan the text & encode using new Codewords (after tree is made).
Decoding:	Once receiver has the tree, scan bitstream (encoded file). 0=go left, 1=go right in tree.



Recurrences with recursion tree; Can't use Master Theorem because a and b are not constants. Use recursion trees. The tree has $loglog(n)$ levels, each costing cn. The total cost is $\theta(nloglogn)$. This is for a recursion $T(n) = \sqrt{n}T(\sqrt{n}) + cn, T(2) = c$



THE DYNAMIC-SET ADT

SETs as Abstract Data Types

Definition of a Set:	A set is an ADT that can store certain values, without any particular order, and no repeated values.
Implementation	Implemented using a Binary Search Tree
UNION(A, B, C)	Take set valued arguments A and B, assign the result $A \cup B$ to the set variable C
INTERSECTION(A, B, C)	Take set valued arguments A and B, assign the result $A \cap B$ to the set variable C
DIFFERENCE(A, B, C)	Take set valued arguments A and B, assign the result $A - B$ to the set variable C
MEMBER(x, A)	Takes set A and object x, whose type is the type of elements of A, and returns a Boolean value - TRUE if $x \in A$, and FALSE if $x \notin A$
INSERT(x, A)	A is a set valued variable, and x is an element of the type of A's members. This makes x a member of A. That is, the new value of A is $A \cup \{x\}$. Not that if x is already a member of A, then this does not change A.
DELETE(x, A)	Removes x from A, i.e. A is replaced by $A - \{x\}$. If x is not in A originally, then this does not change A.
MIN(A)/ MAX(A)	Returns the least/largest element in set A.
SUCCESSOR(x, S) / PREDECESSOR(x, S)	Returns the successor/predecessor of x in S
SEARCH(x, S)	Returns TRUE if x exists in S, and FALSE otherwise.

BINARY SEARCH TREE. If tree balanced, running time in worst case is $\Theta(\log(n))$

Binary Tree must satisfy:	Let v be node in binary search tree. Each value in **left sub-tree** of v is **smaller** than the value of v. Each val in **right sub tree** of v is **larger** than value of v. Must hold $for \forall$ nodes in tree.		
Representation of a Binary Search Tree	Represent as a linked data structure where each node is a struct w PAR, VAR, LC, RC **Relative missing** → corresponding attribute NULL. Root is only node w. NULL. parent	val : value of the node LC : a pointer to the left child RC : a pointer to the right child PAR : a pointer to the parent.	PAR val LC RC
Height of Tree:	Maximum case: we have a long stick of children of one type. Minimum case is when we have a perfectly balanced tree. **Maximum height**: n | **Minimum Height**: $\log_2(n)$		

SEARCHING: Given **pointer to root** of tree, & **key k,** SEARCH returns **TRUE** if k **exists** in tree; else **FALSE**

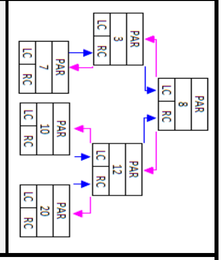
(1) Start @ root node as current node. (2) If the search key's value matches the current node's key, then match found. (3) If search key's value is greater than current node's: (3.1) If the current node has a right child, search in the right Subtree. (3.2) Else, no matching node in the tree. (4) If search key is *less* than the current node's: (4.1) If current node has left child, search in left Subtree. (4.2)Else, no matching node in tree

Binary Tree Search (iterative)

```
SEARCH(x,T)
  pnode := T ;  found := FALSE
  while (!found and (pnode != NULL))
      if (x= (pnode.val)) #"."="->"
          found := true
      else if (x < (pnode.val))
          pnode := (pnode.LC)
      else
          pnode := (pnode.RC)
  return(found)
```

Binary Tree Search (recursive)

```
SEARCH(x, pnode) {
   if (pnode = NULL)
        return (FALSE)
   if (x = pnode.val)
        return(TRUE)
   if (x > pnode.val)
        return SEARCH(x, pnode.RC)
   Else
        return SEARCH(x, pnode.LC)
}#initially call w. SEARCH(x,T)
```



MINIMUM VALUE: Traverse node from root to **left** recursively till left child is NULL. Node with null LC is the minimum.

```
MIN(pnode) { #assum pnode != NULL
    if (pnode.LC != NULL) :
        return (MIN(pnode.LC))
    Else :  return (pnode.val
} #initiall call w.MIN(T), T ptr to root.
```

MAXIMUM VALUE: Traverse node from root to **right** recursively until right child is NULL. Node with null RC is max.

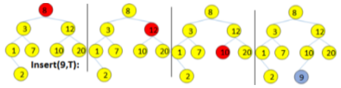
```
MAX(pnode) {
    if (pnode.RC != NULL)
        return (MAX(pnode.RC))
    Else : return (pnode.val)
} #initial call w. MAX(T),T ptr to root.
```

RUNNING TIME: Both of these procedures run in $O(h)$ time on a tree of height h. As in SEARCH, the sequence of nodes encountered forms a simple path downward from the root.

Inserting and Deleting Values into the Binary Search Tree. Runtime: worst case $\Theta(n)$

```
INSERT(x, pnode)
if (x > pnode.val)
    if (pnode.RC != NULL)  :  INSERT(x, pnode.RC)
    else
     create new node, let newpnode be ptr to this node
     (newpnode.val) := x
     (newpnode.PAR) := pnode
     (pnode.RC) := newpnode
else
    if (pnode.LC != NULL) :  INSERT(x, pnode.LC)
    else {
       create new node, let newpnode be ptr to this node
       (newpnode.val) :=xrange
       (newpnode.PAR) := pnode
       (pnode.LC) := newpnode
```

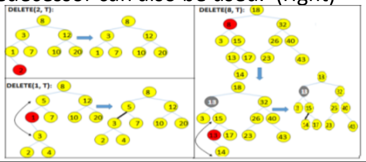
1. Always insert new node as a leaf node.
2. Start at root node as the current node
3. If new node's key < current node's key
a. If current node has a left child, search in the left Subtree
b. Else add new node as current's left child
4. If new node's key >current's key
a. If current node has a right child, search in the right Subtree
b. Else add new node as current's right child.



Runtime: $O(h)$, h =tree height.

```
DELETE(x, pnode)  // Running Time: O(h), h=tree height
   pnode := FIND(x, T)
   if (pnode == NULL)  :  return (NOT-FOUND)
   if (pnode == ((pnode.PAR).LC)
        side := LEFT #pnode is left child of its parent
   Else  :   side := RIGHT
   if (pnode.LC == NULL and pnode.RC == NULL) #pnode is leaf
        if (side == LEFT)  :  (pnode.PAR).LC := NULL
        Else :  (pnode.PAR).RC := NULL
        free (pnode) #free the space used by record of pnode
   else if (pnode.LC == NULL) #pnode only has right child
        if (side == RIGHT)#connect child as right child of parent
             (pnode.PAR).RC := pnode.RC
        else #connect child as left child of parent
             (pnode.PAR).LC := pnode.RC
        (pnode.RC).PAR := pnode.PAR #connect parent to child
        free (pnode)
   else if (pnode.RC == NULL) #pnode has only left child
        if (side == RIGHT)  :  (pnode.PAR).RC := pnode.LC
        Else :  (pnode.PAR).LC := pnode.LC
        free (pnode)
   else #pnode has two children
        min_val := MIN(pnode.RC) #find min val in subtree of RC
        DELETE(min_val, pnode.RC) #delete node w that value
          pnode.VAL := min_val #put value in pnode
```

The Algorithm: We want to remove x. Search for the node storing x. 3 cases:
1. Node to be deleted is a leaf : simply delete it from the tree (top left)
2. Node to be deleted has only one child: Cut the node from the tree and link the single child (and its Subtree, if ∃) directly to parent of the deleted node. (bot left)
3. Node to be deleted has two children: Find the successor of the node. Copy content of the successor to the node, and delete the successor. Note that predecessor can also be used. (right)

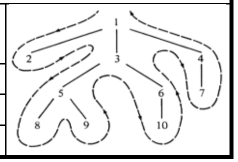


Height of Binary Search Tree of n Items:
Maximum case: long stick of children of one type. Minimum case: perfectly balanced tree.
Max h: n | Min h: $\log_2(n)$

BINARY TREE TRAVERSAL. Let T be a tree with root n, and subtrees $T_1, T_2, \dots, T_k$...

PREORDER(pnode)	INORDER(pnode)	POSTORDER(pnode)
if (pnode != NULL) visit(pnode) PREORDER(pnode.LC) PREORDER(pnode.RC)	if (pnode != NULL) INORDER(pnode.LC) visit(pnode) INORDER(pnode.RC)	if (pnode != NULL) POSTORDER(pnode.LC) POSTORDER(pnode.RC) visit(pnode)
Preorder traversal of nodes of T is the root n of T, followed by nodes of T_1 in preorder, then nodes of T_2 in preorder, and so on.	*Postorder listing* of nodes of T is nodes of T_1 in postor, nodes of T_2 in postor, and so on, upto T_k, all followed by node n.	*Inorder listing* of the nodes of T is nodes of T_1 in inorder, then by node n, followed by nodes of $T_2, \dots, T_k$, each group of nodes in inorder.

Trick for Producing the orderings: "Walk" around the outside of the tree, starting at the root, moving counterclockwise, & staying as close to the tree as possible.



PRE: List a node the first time we pass it. 1,2,3,5,8,9,6,10,4,7

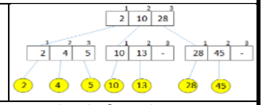
POST: List node the last time we pass it, as we move up to its parent. 2,8,9,5,10,6,3,7,4,1

IN: List leaf first time we pass it, list interior node second time we pass it. 2,1,8,5,9,3,10,6,7,4

2-3 Trees

2-3 Trees

Properties: Each internal node has either **2 or 3 children**.
All leaves are at the **same distance from the root**.
Values of the set are kept at the leaves, sorted **smallest to largest**.
If there are 2 children, they will be the left child and the middle child.



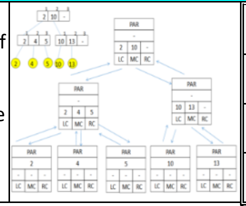
Internal Nodes: Each internal node keeps data to direct the search:. $\min_1$ is the min value in the left Subtree. $\min_2$ is the min value in the middle sub tree. $\min_3$ is the min value in the right sub tree.

Tree Height:
All values are kept at the leaves. So there are n leaves total.
There are at least 2^h leaves. Therefore $n \geq 2^h \rightarrow \boldsymbol{h \geq \log_2 n}$
There are at most 3^h leaves. Therefore $n \leq 3^h \rightarrow \boldsymbol{h \geq \log_3 n}$

Tree Running Time:
All operations $\boldsymbol{O(h) = O(\log n)}$ on a tree of height h since the sequence of nodes forms a simple path down from root.

Implementation as a Linked Data Structure:

Inserting a value to a 2-3 tree:
Search for appropriate PAR of x (par=P). If parent node P has 2 children, just add x.
If P has 3 children, split P & add x.
Thus, we get two nodes of size 2.Continue up the tree, splitting full nodes until reaching a node w. 2 children. If reached root & it's full, split it & create new root.



PAR		
a pointer to the parent		
VAL		
value of the node (only for leaves)		
MIN1	MIN2	MIN3
minval in L. Subtree	*In mid Subtree*	*in R. subtree*
LC	MC	RC
Ptr. to left child	*Ptr. to mid kid*	*Ptr. R. child*

```
Search(x, pnode) { #pnode is ptr to a node
if (pnode.LC == NULL) #node=leaf
     if (pnode.VAL == x) #x is found
          return (TRUE)
     else return (FALSE) #x not in tree
else #node is internal, so go down the tree
     if (x < pnode.MIN1)
          return (FALSE) #x not in tree since smaller than min values
     if (pnode.MIN1 <= x < pnode.MIN2)
          return (Search(x, pnode.LC)) #cont. to LC
     else
          if (pnode.RC == NULL) #no RC
               return (Search(x, pnode.MC)) #cont. to MC
          else #pnode has RC
               if (pnode.MIN2 <= x < pnode.MIN3)
                    return (Search(x, pnode.MC))
               else return (Search(x, pnode.RC))
```

Searching in a 2-3 Tree:
If $x < \min_1$ then x doesn't appear in T, return FALSE.
Else, the search will be directed to the correct Subtree, depending on x:
$\min_1 \leq x < \min_2$: go to the left Subtree
$\min_2 \leq x < \min_r$: go to the mid Subtree
$\min_3 \leq x$: go to the right Subtree
Eventually we get to a leaf. If leaf contains value x ,return TRUE. Else,$FALSE$.

Deleting a Value from a 2-3 Tree:
Search for parent of x (i.e P) & remove x.
If P remains with 2 children, stop.
If P remains w. single k, look @ P's sibling.
If P's sibling has 3 kids, move 1 of them to P
If P's sibling has 2 children, merge P w. P's sibling.

TRUE/FALSE BOIS

Minval in bin. search tree always appears at leaf.	FALSE. Minval doesn't have LC, but may have RC thus not leaf
x =a node in binary search tree that has RC. Then node that contains the successor value, doesn't have LC.	TRUE. if x has RC, then successor is the min val in right Subtree. This node doesn't have LC, otherwise the LC would be smaller.
X= a node in binary search tree that doesn't have LC. Then predecessor of x appears In its parent.	FALSE. If x is a RC of its par, then claim is true. If x is LC of its par, its par's val is bigger, thus claim false.
For each internal node v in a binary search tree, the successor of v appears at its Subtree. FALSE	
For each internal node v in a binary search tree, the successor of v appears at its ancestor. FALSE	
For every connected undirected graph G=(V,E) and for each vertex s in V, running BFS from s will visit the vertices in the same order as running DFS from s.	FALSE, obviously.
For every P, minimium priority queue, and for any two different values x,y: If we insert into P the value x first and then the value y, we get the same priority queue as if we insert into P the value y first and then the value x.	FALSE. For a minimum priority queue P.T=[1], containing only 1 element 1. P.size=1. Say x=2, y=3. Inserting x first: P.T[1,2,3], P.size=3. Inserting y first: P.T[]=[1,3,2], P.size=3
For every P, minimum priority queue, the maximum value in P sits at position P.size, i.e. P.T[P.size] is the max value in P.	FALSE. For example: P.T[1,3,2], P.size=3 is a valid minimum priority queue, but P[3] is not the max.
For every bin. search tree T that contains diff. values, & for each value x in tree, if we delete value x from T , then insert it, we will get the same tree T as it was before the 2 operations.	FALSE. Suppose we have (1)\(2). Delete 1: (2). Insert 1: (1)/(2), which is not the same.
For every 2-3 tree T that contians different values, if we insert the value x and then we insert the value y we will get the same tree as if we first insert the value y and then insert the value x.	FALSE: if we have a tree with leaves (1,2,4), and we insert 3 then 5: (1,2)/\(3,4,5). If insert 5 then 3: (1,2,3)/\(4,5). NOT THE SAME

Dynamic programming

Overview: Divide problem into a reasonable number of sub-problems, s.t. we can use optimal solutions to sub-problems to give us optimal solutions to larger one. A DP algorithm solves each subproblem once, saving the answer in a table, thus avoiding work of recomputing answer every time it solves each subproblem.

MAXIMUM SUM OF NON-CONSECUTIVE ELEMENTS:

Task: Given an array A of positive integers, find non-consecutive elements (consecutive in the index, not value) from this array, which when added together produce the maximum sum.

2 Problems: (1) Find maximum sum of non-consecutive elements in A. (2) Produce the actual subsequence

Solving for Maximum Sum of Non-Consecutive Elements:
We define $V[i]$ as the maximum sum of non-consecutive elements from $A[1,\dots,i]$. We want to solve for $V[i]$ in terms of the V's of the smaller problems.
$V[i]$ = maximum sum of non-consecutive elements from $A[1,\dots,i]$. Thus:
$V[i]=A[1]$, i.e the maximum sum of non-consecutive elements from A[1] is the value A[1] itself.
$V[2]=\max\{A[1],A[2]\}$, i.e. max val of non-consec. elements from $A[1,2]$ is the max between $A[1]$ & $A[2]$.
So, in general: $V[i]=\max\{V[i-1],V[i-2]+A[i]\},\forall i>2$, and we have 2 cases:

Case 1: An optimal subsequence of non-consecutive elements from $A[1,..i]$ does not include the element $A[i]$. In this case – an optimal subsequence will be chosen from the subaray $A[1,\dots,i-1]$, and therefore its maximum sum will be equal to $V[i-1]$, and thus we use $V[i]=\max\{V[i-1],V[i-2]+A[i]\},\forall i>2$.

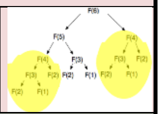
Case 2: The optimal subsequence includes the element A[i]. In this case, the optimal subsequence can't include A[i-1] (because of non-consecutive requirement), and thus the max sum is composed of A[i] and the maximum subsequence from A[1,…,i-2], i.e $V[i]=\max\{V[i-1],V[i-2]+A[i]\},\forall i>2$

Therefore, our expression for V[i] is able to appropriately select the better of the 2 cases.

Recursive Top-Down Implementation (Naïve Approach) - exponential time

```
MSWN(A,n) // maximum series w/o neighbors
   If n==1 : Return A[1] // this algo unfolds recursively, w. exp time w n
   Else if n==2 :        Return max(A[1].A[2])
   Else :    Return max(MSWN(A,n-1) , MSWN(A,n-2) + A[n] )
```

Inefficient. Repeatedly calls itself with same parameter values, re-solving same sub-problems.



Bottom Up Approach for MSWN:
1. Evaluate function V starting at smallest possible argument value
2. Stepping through possible values, gradually increase argument value. Store computed values in an array.
3. As larger arguments are evaluated, pre-computed values for smaller arguments can be retrieved.
4. So, we just do one loop over values of i from 1 to n, filling in V as we go.

Bottom Up MSWN (without generating subsequence – just sum)

```
MSWN(A,n) // maximum series w/o neighbors
   V[1] = A[1] // initialize first item in V
   V[2] = max{A[1], A[2]}
   For i=3 to n // run algo in bottom up way
      V[i] = max{V[i-1] , V[i-2] + A[i]}
   Return V[n]  // V[n]=max value of legal subsequence
```

With the "for" loop, we fill out the array V entry by entry, doing constant amount of work per entry, taking $O(n)$ time in total. This algorithm returns the maximum value of the sum, but not the sequence itself.

Bottom Up MSWN Print the Actual Subsequence:

```
Print_MSWN(A, V, n)
   i=n
   while (i ≥ 1) // we run backwards
      if (i==1)  // if we reached the first item
         print A[1]
         return
      if (i==2)  // if we reached the first two items
         print max{A[1], A[2]}
         return
      else if (V[i] ≠ V[i-1])  // if current val is diff from previous
         print A[i] // current item (by def of V must ∈ subseq)
         i = i-2  // If A[i] was selected, A[i-1] is not an option
      else  // go to next (I,e previous) item
         i = i-1
```

Observe the recursive equations for V. Notice each cell $V[i]\ \forall i\geq 2$ may be equal to either $V[i-1]$ or $V[i-1]+A[i]$. If we have $V[i]=V[i-1]$ then an optimal subsequence from $A[1,\dots,i]$ doesn't contain A[i]. In the other case, it does contain A[i]. <u>To find the subsequence, we just walk backwards through the array V, starting at i=n and going down to i=1.</u> If $V[i]=V[i-1]$ then the optimal sequence does not contain A[i], so just continue to "i-1".If $V[i]=V[i-2]+A[i]$ then optimal sequence contains A[i], so we print A[i] and continue to "i-2".

LONGEST INCREASING SUBSEQUENCE

Task: Find subsequence of a given sequence where: (1) The subsequence's elements are in sorted ascending order. (2) Subsequence is as long as possible. (3) Subsequence does not have to be contiguous or unique.

2 Problems: (1)Find the length of the longest increasing subsequence in A. (2)Produce the subsequence itself

Problem (1): Find length of longest subsequence of a sequence A[1,…,n].
To solve this, we need to solve a slightly different, but related problem for each $1\leq i\leq n$, defined as follows: $L[i]$ = the length of a longest increasing subsequence from A[1,…,i] which includes element A[i] as its last element.

Once we solve this L[i] problem, ans. to original qstn: $\max\{L[i]\ :1\leq i\leq n\}$.
Then, we come up with a recursive formula for computing $L[i]$:

$$L[1]=1\ ;\quad L[i]=\begin{cases}1\ ; if\ A[j]\geq A[i]\ \ \forall\, 1\leq j<i\\ 1+\max\{L(j):\ 1\leq j<i \text{ and } A[j]<A[i]\}\ \ ;\text{otherwise}\end{cases}$$

Case 1: For every j, $1\leq j<i$, $A[j]\geq A[i]$.
The longest increasing subsequence of A that ends in position i consists of just $A[i]$, so it has length 1.

Case 2: For some j, $1\leq j<i$, $A[j]<A[i]$. Let S be a LIS of A that ends in position i. Therefore $S=S'\cup A[i]$ for some sequence S'. S' is an increasing subsequence of A ending at some j, $1\leq j<i$, s.t. $A[j]<A[i]$. S can't be an increasing subseq. of A unless this is satisfied, thus it is true.

Proof of optimality of Case2: S' is the longest among all "longest increasing subsequences" of A that end at some position j, $1\leq i<j$, such that $A[j]<A[i]$. If S' is not the longest, then there is an increasing subsequence S'' of A that ends at some j, $1\leq i<j$ such that $A[j]<A[i]$, and S'' is longer than S'. But then $S''\cup A[i]$ is an increasing subsequence of A that ends at i, that is longer than $S'\cup A[i]=S$, contradicting the definition of S.

Longest Increasing Subsequence (Just the length)

```
LIS(A, n) // Longest Increasing Subsequence
   L[1] = 1 // len of longest ascending subseq. that ends at A[1] is 1
   For i=2 to n
         L[i] = 1 // len(longest ascending subseq that ends at A[i])≥ 1
      For (j=1 to i-1) // Find 1 + max{L(j): 1 ≤ j < i and A[j] < A[i]}
         If (A[j] < A[i])
            New-length = L[j]+1
            If (new-length > L[i])
               L[i] = new-length
   Max-length = L[1] // compute max{L[1], L[2], …, L[n]}
   For (i=2 to n)
      If (L[i] > max-length)
         Max-length = L[i]
   Output(max-length)
```

Time: $\sum_{i=2}^{n}c(i-1)=c\sum_{i=1}^{n-1}\frac{c(n-1)n}{2}=\Theta(n^2)$

This code returns the value of an optimal solution, but not the actual solution – the subsequence of elements.
We extend the algorithm to record not only the optimal value that was computed in each subproblem, but also the choice that led to the optimal value. This is shown in the next section.

Longest Increasing Subsequence with actual sequence generation

```
LIS(A, n) // Longest Increasing Subsequence
   L[1] = 1; Prev[1] = 0
   For i=2 to n
      L[i] = 1 ; Prev[i] = 0
      For (j=1 to i-1) // Compute 1 + max{L(j): 1 ≤ j < i and A[j] < A[i]}
         If (A[j] < A[i])
            New-length = L[j]+1
            If (new-length > L[i])
               L[i] = new-length ; Prev[i] = j //j comes b4 I in LAS ending at i
   Max-length = L[1] ; best-last = 1
   For (i=2 to n)
      If (L[i] > max-length)
         Max-length = L[i] ; best-last = i
   ~~Output(max-length)~~ Print-LIS (best-last, Prev)

Print-LIS (Input: integer last, array of integers Prev[])
   If (Prev[last] > 0)
      Print-LIS(Prev[last], Prev)
   Print(last) // or print A[last] if want the values in the subsequence
```

Computing actual sequence:
For each i we computed $L[i]$.
In order to find the subsequence, we also maintain Prev[]. Prev[i] will be the index of the predecessor of A[i] in a longest running subsequence that ends in A[i].
By following the Prev[i] values we can reconstruct the whole sequence in linear time.
LAS = Longest ascending subsequence

MAX SUBSET SUM

Task: Given: (1) An integer bound B. (2) A collection of n integers $s_1, \dots, s_n$. Find subset that has maximum sum, w/o exceeding B. Formally: Find $T \subseteq \{1, \dots, n\}$ of items s.t: maximizes $\sum_{j \in T} s_j$ while keeping $\sum_{j \in T} s_j \le B$

Sub-Problem: W/o finding the actual set of integers, simply find the value of maximum possible subset sum.

Solving the Sub-Problem:
To solve the problem, define a matrix M of size $(n+1) \times (B+1)$ as follows:
$\forall\ 0 \le h \le B, 0 \le i \le n$: M[i][h] will keep the sum of a subset $T \subseteq \{1, \dots, i\}$ of indices such that maximizes $\sum_{j \in T} s_j$ while keeping $\sum_{j \in T} s_j \le h$. Notice that M[n][B] is the value of the optimal solution.

Finding the Matrix: How do we compute M[i][h] in terms of solutions to smaller sub problems?

$$M[i][h] = \begin{cases} 0\ ;\ i = 0 \text{ or } h = 0 \\ M[i-1][h]\ ;\ s_i > h \\ \max\left\{ \underbrace{M[i-1][h]}_{\text{Val of opt. sol w/o i'th elmnt}},\ \underbrace{s_i + M[i-1][h-s_i]}_{\text{Val of opt. sol w. i'th elmnt}} \right\}\ ;\text{ow} \end{cases}$$

Naïve Recursive Approach:

```
Max-Subset-Sum-REC (Input: integers B, n, array of n integers s[], i, h) // Runtime: Exponential in n
    If (i=0 or h=0) :      Return(0)
    If (s[i] > h) :        Return (Max-Subset-Sum-REC(B,n,s,i-1,h)
    Else :                 Return(max(Max-Subset-Sum-REC(B,n,s,i-1,h) , s[i] + Max-Subset-Sum-REC(B,n,s,i-1, h-s[i])))
```

Max Subset Sum Bottom Up

```
Max-Subset-Sum-BU (Input: integers B, n, array of n integers s[])
    For (i=0 to n) // initialization
        M[i][0] = 0
    For (h=0 to B)
        M[0][h] = 0
    For (i=1 to n)
        For (h=1 to B)
            If (s[i]>h) //subset with upper bound h on sum cannot include item i
                M[i][h] = M[i-1][h]
            Else
                M[i][h] = max(M[i-1][h] , M[i-1][h-s[i]]) //max btwn sub-sum w. and w/o i
    ~~Return (M[n][B])~~ Print-Subset(n, B, s, M)

Print-Subset(int n, B, array of n ints s[], (n+1)×(B+1) integer array M[][]
    i=n ; h = B
    while i>0 and h>0 // backtrace, starting from item n (and upper bound B on sum)
        if s[i] ≤ h
            if M[i][h] > M[i-1][h]  // optimal choice necessarily includes item i.
                print(i)
                h = h-s[i]
        i = i-1
```

Runtime: $\Theta(nB)$
Initialize first row and first column with 0. Fill in the matrix row by row (left to right) according to the formula for M. The output will be M[n][B]

Example:
For $B = 6, s_1 = 2, s_2 = 3, s_3 = 2, s_4 = 1$

$i \backslash h$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2
2	0	0	2	3	3	5	5
3	0	0	2	3	4	5	5
4	0	1	2	3	4	5	6

Max-Subset-Sum Memoization

```
M[i][h] = [][] // global matrix
Max-Subset-Sum (Input: integers B, n, array of n integers s[])
    For i=0 to n :  M[i][0] = 0    // initializations
    For h=0 to B : M[0][h] = 0
    For i=1 to n
        For h=1 to B :  M[i][h] = -1
    Return(Max-Subset-Sum-MEM(n, B, s, M) //call memorization procedure

Max-Subset-Sum-MEM (integers I, h, array of n integers s[], (n+1)×(B+1) array of integers M) // Time: Θ(nB)
    If M[i-1][h] = -1 //M[i-1][h] was not yet computed (it is needed to find M[i][h]
        M[i-1][h]  := Max-Subset-Sum-MEM (i-1, h, s, M)
    If s[i] > h
        M[i][h] := M[i-1][h]
    Else
        If M[i-1][h-s[i]] = -1 // subset with upper bound h on sum can't include item i.
            M[i-1][h-s[i]] := Max-Subset-Sum-MEM (i-1, h-s[i], s, M) // M[i-1][h-s[i]] not yet found, but needed.
        M[i][h] := max(M[i-1][h], s[i]+M[i-1][h-s[i]]) // find max between best subset sum w. & w/o item i
    Return(M[i][h])
```

LONGEST COMMON SUBSEQUENCE

Definition of Subsequence: A subsequence of string X is a string which can be obtained by deleting some of the characters from X. This is not the same as a substring. For example: X=ABCDEFGHIJK. A subsequence of X is: ACEGIJK, or DFGHK, but DAGH is NOT a subsequence

Task: Given two strings $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$. Find the longest string Z which is a subsequence of both X and Y. Ex., X=AAACCGTGAGTTATTCGTTCTAGAA, Y=CACCCCTAAGGTACCTTTGGTTC, LCS=Z=ACCTAGTACTTG.

Brute Force Solution: Enumerate all subsequences of X. Test which ones are also subsequences of Y. Pick longest one. If X is of length n, then it has 2^n subsequences, i.e. this is an exponential-time algorithm.

2 Problems: (1) Look at just the length of an LCS. (2) Extend algorithm to get LCS sequence itself

Defining the Subproblem: $C[i,j]$ is the length of the LCS of strings $X[1, \dots, i]$ and $Y[1, \dots, j]$. Answer of the problem can thus be found in $C[m,n]$.

Developing the Recurrence: Let Z[1,…,k] be the LCS of X[1,…,i] and Y[1,…,j].
If $x_i = y_j = c$ then $z_k = x_i = y_j = c$ and Z[1,…,k-1] is the LCS of X[1,…,i-1] and Y[1,…,j-1]
If $x_i \ne y_j$ then: Case 1: Either $z_k \ne x_i$, implying that Z[1,…,k] is the LCS of X[1,…,i-1] and Y[1,…,j] Case 2: Or $z_k \ne y_j$ implying that Z[1,…,k] is the LCS of X[1,…,i] and Y[1,…,j-1].
From the definition of C[i,j], if either $i = 0$ or $j = 0$, one of the sequences has length 0 and so the LCS has length 0. Thus $C[0,j] = C[i,0] = C[0.0] = 0$.

$$C[i,j] = \begin{cases} 0\ ; i = 0\ or\ j = 0 \\ C[i-1, j-1] + 1\ ; i,j > 0\ and\ x_i = y_j \\ \max\{C[i, j-1], C[i-1, j]\}\ ; i,j > 0\ and\ x_i \ne y_j \end{cases}$$

Order of Evaluation of C[i][j]: To find C[I,j] we need to know C[i-1,j] , C[I, j-1] , C[i-1, j-1] . Computing the cells from the top most corner by rows will ensure that these values will be ready before computing C[I,j].
To be able to recover the solution from our C[I,j] – we need to keep track of "which case" of C led us to each particular value. We do this with a matrix D, defined as: $D[i,j] \begin{cases} \text{upleft}\ ; if\ C[i,j] = C[i-1, j-1] + 1 \\ \text{up}\ ; \text{if } C[i,j] = C[i-1, j] \\ \text{left}\ ; if\ C[i,j] = C[i, j-1] \end{cases}$

Longest Common Subsequence

```
LCS_VAL(Int m,n, Str X with length m, String Y with length n)
    For i=0 to m do C[I,0]=0 // base cases
    For j=0 to n do C[0,j]=0
    For i=1 to m: // filling the matrices
        For j=1 to n:
            If X[i]=Y[j]:
                C[I,j] = C[i-1, j-1] +1
                D[I,j] = upleft
            Else
                If C[i-1, j] ≥ C[I, j-1]
                    C[I,j] = C[i-1, j]
                    D[I,j] = up
                Else
                    C[I,j] = C[I,j-1]
                    D[I,j] = left
    ~~Return (C, D)~~ Print_LCS (X, Y, n, m, D)

Print_LCS (X, Y, n, m, D)
    Row = m ; col = n
    While row>0 and col>0
        If D[row, col] = upleft // X[row] = Y[col]
            Print(X[row])
            Row := row-1 ; col := col-1
        Else if D[row,col] = up
            Row := row-1
        Else if D[row,col] = left
            Col := col-1
```

Example: X=b,a,c,b,f,f,c,b. Y=d,a,b,e,a,b,f,b,c.

	0	1 d	2 a	3 b	4 e	5 a	6 b	7 f	8 b	9 c
0	0	0	0	0	0	0	0	0	0	0
1 b	0	0 ↑	0 ↑	1 ↖	1 ←	1 ↑	1 ↖	1 ←	1 ↖	1 ←
2 a	0	0 ↑	1 ↖	1 ↑	1 ↑	2 ↖	2 ←	2 ←	2 ←	2 ←
3 c	0	0 ↑	1 ↑	1 ↑	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	3 ↖
4 b	0	0 ↑	1 ↑	2 ↖	2 ←	2 ↑	3 ↖	3 ←	3 ↖	3 ↑
5 f	0	0 ↑	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ←
6 f	0	0 ↑	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ↑	4 ↑
7 c	0	0 ↑	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	4 ↑	5 ↖
8 b	0	0 ↑	1 ↑	2 ↖	2 ↑	2 ↑	3 ↖	4 ↑	5 ↖	5 ↑

We can write down an LCS by starting in the lower right corner and following the arrows backwards. Whenever we hit an "upleft" arrow, the corresponding characters should be equal in both X and Y, and we print the character.
The resulting sequence will be printed in reverse order, i.e we will get c,f,b,a,b, but the correct answer is b,a,b,f,c.

CHAIN MATRIX MULTIPLICATION

Dynamic Programming Design Warning: When designing a DP algorithm ,there are 2 parts: (1) Finding an appropriate optimal substructure property and the corresponding recurrence relation on table items. The $m[i,j]$ formula is one such example. (2) Filling the table properly. This requires finding an ordering of the table elements so that when a table item is calculated using the recurrence relation, all the table values needed by the recurrence relation have already been calculated and are available. In the Chain Matrix problem this means that by the time we calculate $m[i,j]$ we must already know $m[i,k], m[k+1,j]$.

Matrix Definition: $n \times m$ matrix $A = [a[i,j]]$ is a 2D array: $A = \begin{bmatrix} a[1,1] & a[1,2] & \cdots & a[1,m-1] & a[1,m] \\ a[1,2] & a[2,2] & \cdots & a[2,m-1] & a[2,m] \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a[n,1] & a[n,2] & \cdots & a[n,m-1] & a[n,m] \end{bmatrix}$.

Matrix Multiplication: The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix given by: $c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$, for $a \le i \le p$ and $1 \le j \le r$.

- If AB defined, that does not mean BA has to be defined (it may or may not). It is possible that $AB \neq BA$.
- Multiplication is recursively defined by: $A_1A_2A_3 \dots A_{s-1}A_s = A_1\left(A_2\left(A_3 \dots (A_{s-1}A_s)\right)\right)$
- Multiplication is associative, i.e. $A_1A_2A_3 = (A_1A_2)A_3 = A_1(A_2A_3)$, i.e. parenthesis don't change result.

Direct Matrix Multiplication: Given a $p \times q$ matrix A, and a $q \times r$ matrix B, the direct way of multipltying $C = AB$ is to compute each $c[i,j]$ by the multiplication definition formula. COMPLEXITY: Note that C contains pr entries, and each entry takes $\Theta(q)$ time to compute, so the total procedure takes $\Theta(pqr)$ time.

Given a $p \times q$ matrix A, $q \times r$ matrix B, and $r \times s$ matric C, then ABC can be found in 2 ways: $(AB)C$ and $A(BC)$. The number of multiplications needed is: multnum$[(AB)C] = pqr + prs$, multnum$[A(BC)] = qrs + pqs$. Depending on the values of p, q, r, s these ways have vastly different numbers of multiplication. This implies that the parenthization is important.

Optimal Substructure Property: If the final "optimal" solution of $A_{i\dots j}$ involves splitting into $A_{i\dots k}$ and $A_{k+1\dots j}$ at the final step, then the parenthesization of $A_{i\dots k}$ and $A_{k+1\dots j}$ in the final optimal solution must also be optimal for the sub problems when they are "standing alone".

Task: Given dimensions $p_0, p_1, \dots, p_n$, corresponding to matrix sequence $A_1, A_2, \dots, A_n$, where A_i has dimension $p_{i-1} \times p_i$. Determine the "multiplication sequence" that minimizes the number of scalar multiplications in computing $A_1A_2 \dots A_n$. I.e. – determine how to parenthesize the multiplications. An exhaustive search would take $\Omega(4^n/n^{3/2})$. Dynamic programming offers a much better approach.

Developing the DP Algorithm:

Step 1: Determine the structure of an optimal solution (here – the parenthesization) :
Decompose problem into sub problems: For each pair $1 \le i \le j \le n$ determine the multiplication sequence for $A_{i\dots j} = A_iA_{i+1} \dots A_j$ that minimizes the number of multiplications. Clearly, $A_{i\dots j}$ is a $p_{i-1} \times p_j$ matrix.
High Level Parenthesization for $A_{i\dots j}$: For any optimal multiplication sequence, at the last step you would be multiplying two matrices $A_{i\dots k}$ and $A_{k+1\dots j}$, for some k. That is, $A_{i\dots j} = (A_i \dots A_k)(A_{k+1} \dots A_j) = A_{i\dots k}A_{k+1\dots j}$. For example, $A_{3\dots 6} = \left(A_3(A_4A_5)\right)(A_6) = A_{3\dots 5}A_{6\dots 6}$, and here $k = 5$.
Thus, the problem of determining the optimal sequence of multiplications is broken down into 2 questions:

2 Problems: (1) How do we decide where to split the chain (what is k)? – Search all possible values of k. (2) How do we parenthesize the subchains $A_{i\dots k}$ and $A_{k+1\dots j}$? – Problem has optimal substructure property that $A_{i\dots k}, A_{k+1\dots j}$ must be optimal so we can apply the same procedure recursively)

Step 2: Recursively define the value of an optimal solution.
We will store the solutions to the sub problems in an array. For $1 \le i \le j \le n$ let $m[i,j]$ denote the minimum number of multiplications needed to compute $A[i,j]$. The optimum cost can be described by the following recursive definition: $m[i,j] = \begin{cases} 0 \ ; \ i = j \\ \min_{i \le k \le j}\left(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\right) \ ; \ i < j \end{cases}$

Proof: Any optimal sequence of multiplication for $A_{i\dots j}$ is equivalent to some choice of splitting $A_{i\dots j} = A_{i\dots k}A_{k+1\dots j}$ for some k, where the sequences of multiplications for $A_{i\dots k}$ and $A_{k+1\dots j}$ are also optimal. Hence $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$. Now, we know that for some k we have $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$, but we don't know what k is! There are only $j - i$ possible values of k, so we can check them all and find the one that returns the smallest cost. That is why we have a "min" in $m[i,j]$

Step 3: Compute value of an optimal solution in a BU fashion.
Our table: $m[1\dots n, 1\dots n]$, and $m[i,j]$ is only defined for $i \le j$. The important point is that when using the $m[i,j]$ equation to calculate $m[i,j]$, we need to already know $m[i,k]$ and $m[k+1,j]$. For both cases – the corresponding length of the matrix chain are both less than $j - i + 1$. Hence the algorithm should fill the table in increasing order of the length of the matrix chain. That is, we calculate in the order:

$$
\begin{gathered}
m[1,2]\,, m[2,3]\,, m[3,4]\,, \dots, m[n-3,n-2]\,, m[n-2,n-1]\,, m[n-1,n] \\
m[1,3]\,, m[2,4], m[3,5], \dots, m[n-3,n-1], m[n-2,n] \\
m[1,4], m[2,5], m[3,6] \dots, m[n-3\,,n] \\
\vdots \\
m[1,n-1], m[2,n] \\
m[1,n]
\end{gathered}
$$

Step 4: Construct an optimal solution from computed information – extract the actual sequence.
Modus Operandi: Maintain an array $s[1\dots n, 1\dots n]$ where $s[i,j]$ denotes k for the optimal splitting in computing $A_{i\dots j} = A_{i\dots k}\,A_{k+1\dots j}$. The array $s[1\dots n, 1\dots n]$ can be used recursively to recover the multiplication sequence.
How to recover the multiplication sequence? Do this recursively until multiplication sequence is determined:

$$
\begin{gathered}
s[1,n] \implies (A_1 \dots A_{s[1,n]})(A_{s[1,n]+1} \dots A_n) \\
s[1, s[1,n]] \implies \left(A_1 \dots A_{s[1,s[1,n]]}\right)\left(A_{s[1,s[1,n]]+1} \dots A_{s[1,n]}\right) \\
s[s[1,n]+1\,, n] \implies \left(A_{s[1,n]+1} \dots A_{s[s[1,n]+1\,,n]}\right)\left(A_{s[s[1,n]+1,n]+1} \dots A_n\right)
\end{gathered}
$$

Example, $n = 6$, and assume we know $s[1\dots 6, 1\dots 6]$. The multiplication sequence is recovered as follows:
Optimal for 2 blocks: up to 3, and past 3: $s[1,6] = 3 \implies (A_1A_2A_3)(A_4A_5A_6)$ | Opt for "up to 3" part: $s(1,3) = 1 \implies (A_1(A_2A_3))$
Optimal for the "past 3" part of the above: $s[4,6] = 5 \implies ((A_4A_5)A_6)$ | The final multiplication sequence: $(A_1(A_2A_3))((A_4A_5)A_6)$

Matrix Chain Dynamic Algorithm

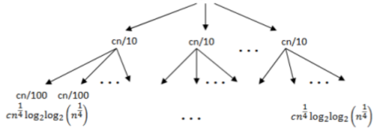
```
Matrix-Chain(p,n)
   For i=1 to n : m[i,i]=0
   For l=2 to n
      For i=1 to n − l + 1
         J=i+l-1
         m[i,j]=∞
         For k=i to j-1
            q=m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
            If q<m[i,j]
               m[i,j]=q
               s[i,j]=k
   Return m and s // optimum in m[1,n]
Mult(A, s, i, j)
   If i<j
      X = Mult (A, s, i, s[i,j]// X is now A_1 … A_k where k is s[i,j]
      Y= Mult(A, s, s[i,j]+1, j) // Y is now A_{k+1} … A_j
      Return X*Y // multiply matrices X and Y
   Else : return A[i] // To compute A_1A_2 … . A_n call Mult(A, s, 1, n)
```

Complexity: The loops are nested three deep. Each loop index takes on at most n values. Hence the time complexity is $O(n^3)$, space complexity $\Theta(n^2)$.
Mult uses the $s[i,j]$ value to decide how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1\dots n]$, and that $s[i,j]$ is global to the recursive Mult procedure. Returns a matrix.
Constructing an optimal solution (example): Want to compute $A_{1\dots 6}$. Using same example from before: $n = 6$, and array $s[1\dots 6, 1\dots 6]$ is known. The mult. Sequence is recovered as:

Mult$(A,s,1,6), s[1,6]=3, (A_1A_2A_3)(A_4A_5A_6)$
Mult$(A,s,1,3), s[1,3]=1, ((A_1)(A_2A_3))(A_4A_5A_6)$
Mult$(A,s,4,6), s[4,6]=5, ((A_1)(A_2A_3))((A_4A_5)(A_6))$
Mult$(A,s,2,3), s[2,3]=2, ((A_1)((A_2)(A_3)))((A_4A_5)(A_6))$
Mult$(A,s,4,5), s[4,5]=4, ((A_1)((A_2)(A_3)))(((A_4)(A_5))(A_6))$

Hence, final product: $\left(A_1(A_2A_3)\right)\left((A_4A_5)A_6\right)$

Master Theorem More Notes:
The recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ for some function f splits a problem of size *n* into ***a*** pieces. Every piece has size ***n/b*** (can be floored), and it takes *f(n)* to divide the problem and assemble the solutions to the pieces. This recurrence defines a tree of **depth** $\log_b(n)$, and **leaves** $L = n^{\log_b a} = a^d$. Relating f(n) to L we find that T(n) depends on L: Second case: f is $\Theta(L) \to T(n)$is $\Theta(L \log_2 n)$



Ex: $T(s) = 10T\left(\frac{n}{10}\right) + cn$ for $s > n^{\frac{1}{4}}$. $T(s) = cslog(\log(s))$ for $s = n^{1/4}$ where s is the recursively passed parameter.. Each level has 10^i nodes. Each of those nodes contributes $cn/10^i$. Each level in tree thus contributes cn. Height of tree is: $\frac{n}{10^h} = n^{\frac{1}{4}} \to h = \log_{10} n^{\frac{3}{4}}$.In total, all levels except leaves contribute $cnlog_{10}(10^{3/4})$. Size of each leaf is $n^{1/4}$, thus each leaf contributes $cn^{1/4}\log(\log(n^{1/4})$. In total, leaves contribute $cn^{3/4}n^{1/4}\log\left(\log\left(n^{\frac{1}{4}}\right)\right) = cnlog\left(\log(n^{1/4})\right)$. SO in total we get $cn \cdot \log_{10} n^{\frac{3}{4}} + nlog\left(\log(n^{1/4})\right) = \Theta(nlog(n))$