COMP20003
Algorithms and Data Structures
Priority Queues

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

Queues

- A queue Q has the following operations:
  - `makeQ();`
  - `enQ(Q,item);`
  - `deQ(Q,first);`
  - `emptyQ(Q);`

Priority Queues

- A priority queue PQ has the operations:
  - `makePQ();`
  - `enQ(PQ,item);`
  - `deletemax(PQ);   /* or deletemin() */`
  - `emptyPQ(PQ);`
  - `changeWeight(PQ,item);`
- Also `delete(PQ,item)`, `replace(PQ,item)`.

Simple implementations of priority queue

- Unsorted array:
  - Construct:
  - Get highest priority:
- Sorted array:
  - Construct:
  - Get highest priority:

Simple implementations of priority queue

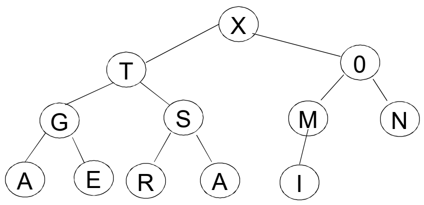
- Unsorted list:
  - Construct:
  - Get highest priority:
- Sorted list:
  - Construct:
  - Get highest priority:

A better implementation of priority queue: The Heap

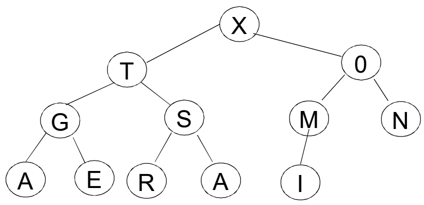
Heap data structure:
- A complete tree
  - *n.b.* a complete tree is…?
- Every node satisfies the "heap condition":
  - `parent->key >= child->key`, for all children
  - Root is therefore …?

Complete tree represented as an array:
- *n.b.* we first look at binary heaps, but
  - A heap need not be binary

Example heap



Example heap



X	T	O	G	S	M	N	A	E	R	A	I

Example heap

X	T	O	G	S	M	N	A	E	R	A	I

Example heap

The arithmetic is easier if we use A[1] as the root.

Parent:
A[i], E.g. i =1

Children of A[i]:
i = parent*2, parent*2 + 1
E.g. i = 2,3

	X	T	O	G	S	M	N	A	E	R	A	I

Example heap

Parent:
A[i], i=2

Children of A[i]:
i= parent*2, parent*2 + 1
i=4,5

X	T	O	G	S	M	N	A	E	R	A	I

Example heap

Parent:
A[i], i=3

Children of A[i]:
i= parent*2, parent*2 + 1
i=6,7

X	T	O	G	S	M	N	A	E	R	A	I

Example heap

Recall the heap condition:

- **parent->key >= child->key**, for all children

For array representation of Binary Heap, this means:

- A[i] >= A[2*i] && A[i] >= A[2*i+1]

X	T	O	G	S	M	N	A	E	R	A	I

deletemax()

1) Return highest priority item:
   - Return root

2) Fix heap:
   - Put last item into root position
   - Reduce size of PQ by one
   - Fix heap condition for root: `downheap()`

deletemax(): Excercise

- Return highest priority item:
  - Return root.
- Fix heap:
  - Put last item into root position.
  - Reduce size of PQ by one.
  - Fix heap condition for root: `downheap()`.

X	T	O	G	S	M	N	A	E	R	A	I	← Keys
1	2	3	4	5	6	7	8	9	10	11	12	← Array Index

downheap()

I	T	O	G	S	M	N	A	E	R	A	← Keys
1	2	3	4	5	6	7	8	9	10	11	← Array Index

```
downheap(int[] PQ, int k)
{
   int j,v;
   v = PQ[k];              /* value, or priority */
   while( k <= n/2 )       /* A[k] has children */
   {
      /* point to children*/
      j= k*2;
      /* j set to highest child*/
      if(j<n && PQ[j]< PQ[j+1]) j++;
      if (v>= PQ[j]) break;   /* check heap OK */
       PQ[k] = PQ[j]; k = j;  /* swap and continue */
   }
   /* final position of original A[k] value*/
   PQ[k] = v;
}
```

Complete code: https://jdoodle.com/a/70l

`deletemax()`

For a maxheap of integers:

```
int deletemax(int[] PQ)
{
        int v = PQ[1];
        PQ[1] = PQ[n--];
        downheap(1);
        return(v);
}
```

Exercise: construct a maxheap of pointers to struct; return a pointer to the maximum priority item

- LINK: https://jdoodle.com/a/70l

Fixing heap with `upheap()`

Inserting a new item into an *already-formed heap:*

```
void upheap(int* PQ, int k)
  {
    int v;
    v = PQ[k];
    PQ[0] = INT_MAX; /* sentinel, limits.h */
    while(PQ[k/2] <= v){ /* note integer arith */
          PQ[k] = PQ[k/2];
          k = k/2;
    }
    PQ[k] = v;
  }
```


`uphead()` *vs.* `downheap()`

- Add new item in last place in heap:
  - `upheap()`
  - *O()*
- Replace root in heap:
  - `downheap()`
  - *O()*

Heapsort

Heap suggests a method for sorting:

- Construct heap
- Swap root (max) with last element
- Remove last element from further consideration, *i.e.* decrease size of heap by 1
- Fix heap using….
  … `downheap()`
- Repeat until finished

Heapsort: Excercise

- Heap suggests a method for sorting:
  - Construct heap.
  - Swap root (max) with last element.
  - Remove last element from further consideration, *i.e.* decrease size of heap by 1.
  - Fix heap using….
    … **`downheap()`**
  - Repeat until finished.

X	T	O	G	S	M	N	A	E	R	A	I	← Keys
1	2	3	4	5	6	7	8	9	10	11	12	← Array Index

Cost of heapsort

- Construct heap   O() ?
- Sucessively move max to end and fix:
  - *n* * **`deletemax()`**:
  - *n* * *O(logn) –> O(n log n)*

Making a heap: two strategies

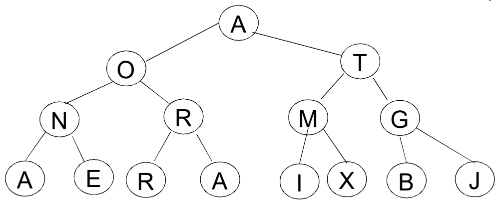
Strategy 1:
- Insert items one-by-one into the array
- **`upheap()`** as each new item is inserted

Insert *n* items into heap of size *n*:
- Each insertion: *O( )*
- How many insertions?
- Overall: *O( )*

Making a heap: two strategies

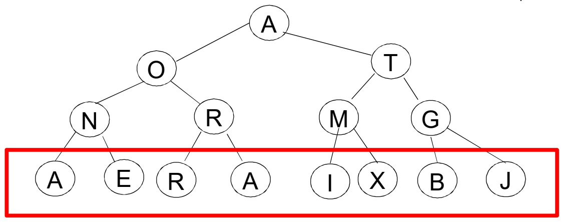
Strategy 2:
- Insert items into unordered array
- Once all items are in, **`downheap()`** for each subheap with roots from **`A[n/2]`** to **`A[1]`**

Strategy 2: How does it work?



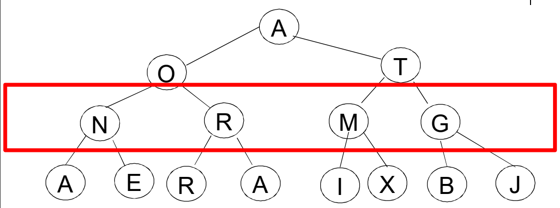
A	O	T	N	R	M	G	A	E	R	A	I	X	B	J

Strategy 2: How does it work?
`downheap()` bottom row



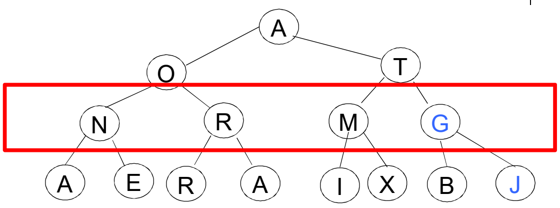
A	O	T	N	R	M	G	A	E	R	A	I	X	B	J

Strategy 2: How does it work?
`downheap()` next row up



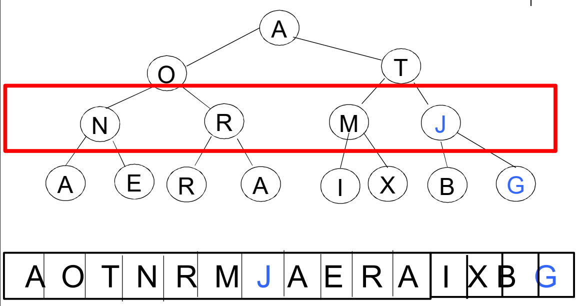
A	O	T	N	R	M	G	A	E	R	A	I	X	B	J

Strategy 2: How does it work?
`downheap()` next row up



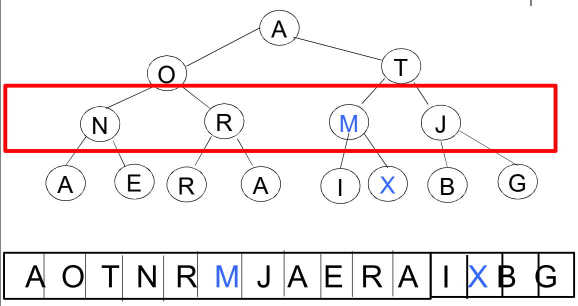
A	O	T	N	R	M	G	A	E	R	A	I	X	B	J

Strategy 2: How does it work?
`downheap()` next row up



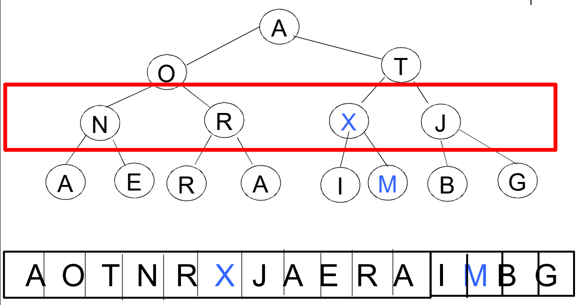
COMP 20003 Algorithms and Data Structures 1-29

Strategy 2: How does it work?
`downheap()` next row up



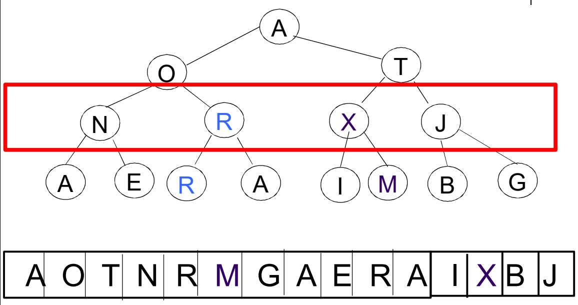
COMP 20003 Algorithms and Data Structures 1-30

Strategy 2: How does it work?
`downheap()` next row up

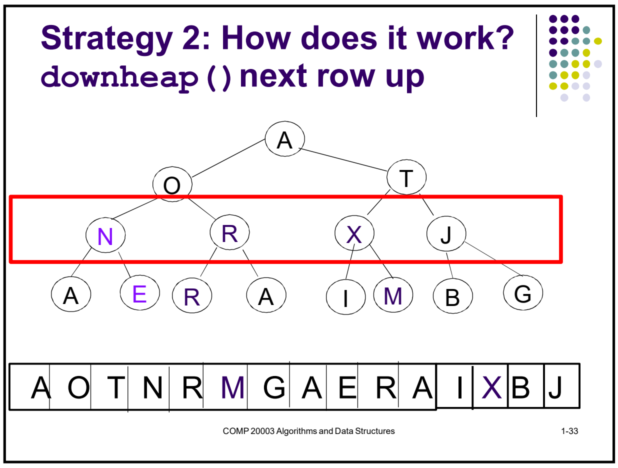


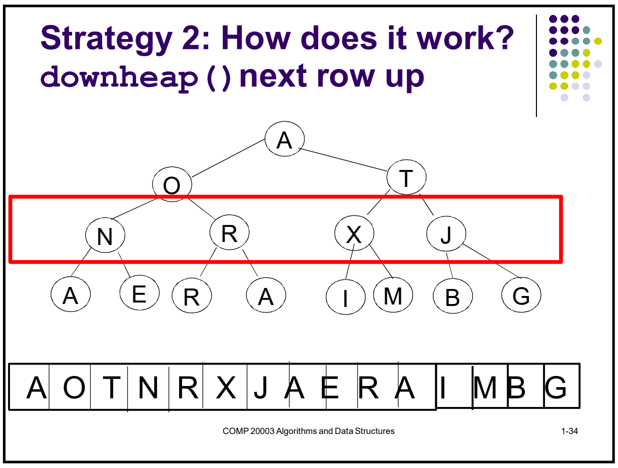
COMP 20003 Algorithms and Data Structures 1-31

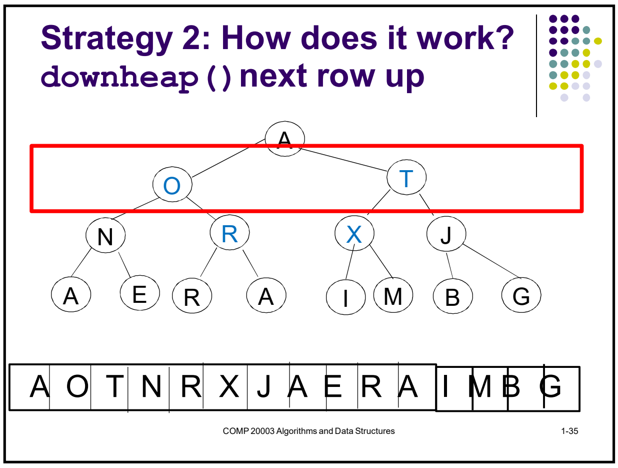
Strategy 2: How does it work?
`downheap()` next row up

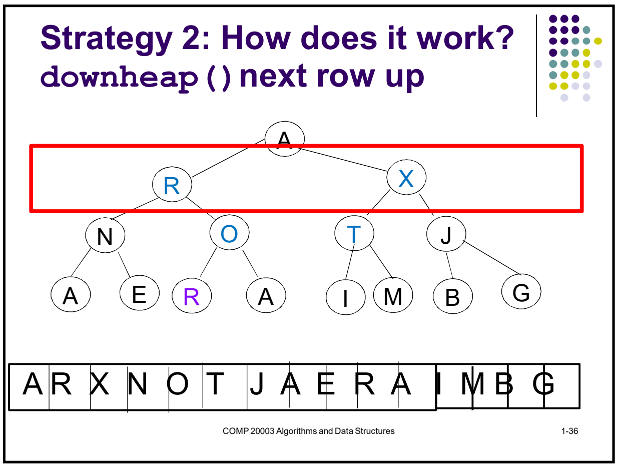


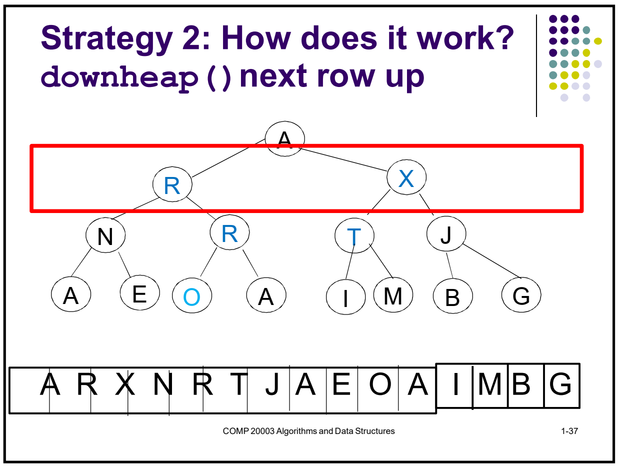
COMP 20003 Algorithms and Data Structures 1-32

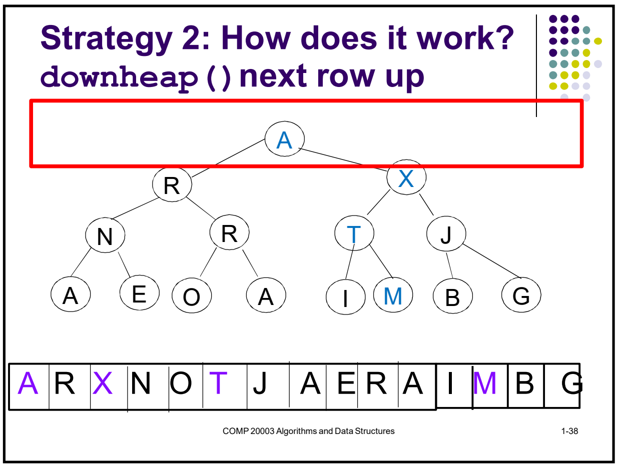

Strategy 2: How does it work?
downheap() next row up
A O T N R M G A E R A I X B J
COMP 20003 Algorithms and Data Structures
1-33

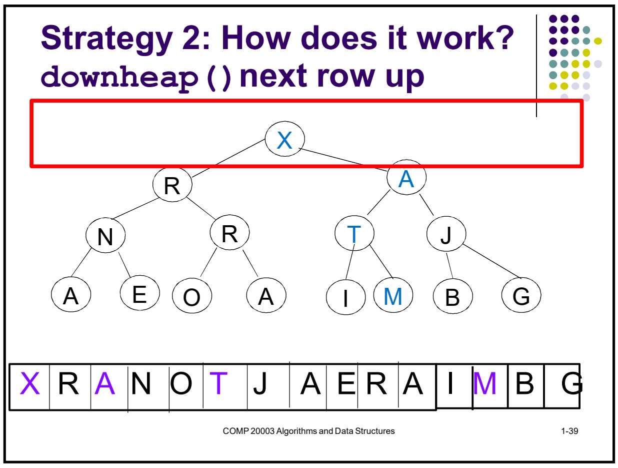

Strategy 2: How does it work?
downheap() next row up
A O T N R X J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-34

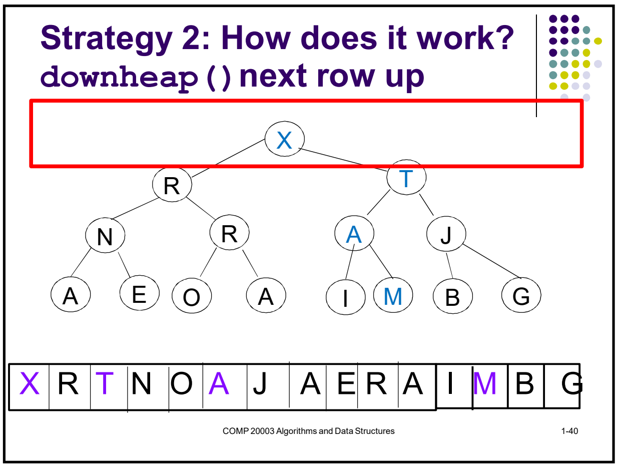

Strategy 2: How does it work?
downheap() next row up
A O T N R X J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-35


Strategy 2: How does it work?
downheap() next row up
A R X N O T J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-36

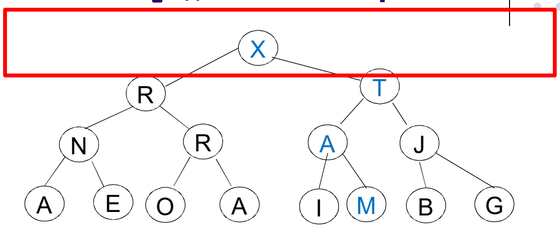

Strategy 2: How does it work?
downheap () next row up
A R X N R T J A E O A I M B G
COMP 20003 Algorithms and Data Structures
1-37


Strategy 2: How does it work?
downheap () next row up
A R X N O T J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-38


Strategy 2: How does it work?
downheap () next row up
X R A N O T J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-39

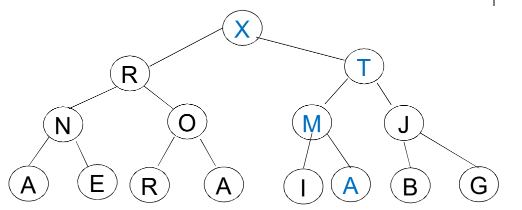

Strategy 2: How does it work?
downheap () next row up
X R T N O A J A E R A I M B G
COMP 20003 Algorithms and Data Structures
1-40

Strategy 2: How does it work?
`downheap()` next row up



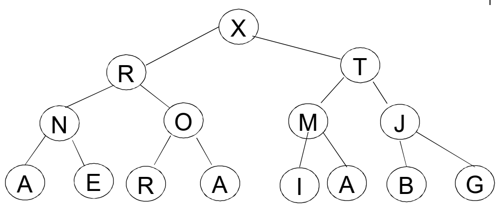
X	R	T	N	O	A	J	A	E	R	A	I	M	B	G

Strategy 2: How does it work?
`downheap()` next row up



X	R	T	N	O	M	J	A	E	R	A	I	A	B	G

Strategy 2: Finished heap after bottom-up heap construction



X	R	T	N	O	M	J	A	E	R	A	I	A	B	G

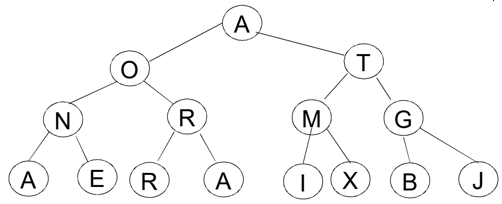
Strategy 2: Analysis

Strategy 2:
- Insert items into unordered array
- Once all items are in, **`downheap()`** for each subheap with roots from **`A[n/2]`** to **`A[1]`**

Insert *n* items into heap of size *n*:
- Start with Insert into unordered array: *O( )*
- Then **`downheap( )`** subheaps from **`A[n/2]`** to **`A[1]`**

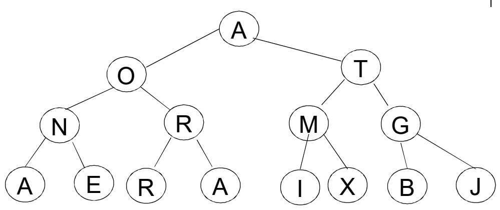
- **`downheap( )`** subheaps from **`A[n/2]`** to **`A[1]`**
- Bottom n/2 nodes are already heaps
  - Cost to fix: ?
- Next level up nodes:
  - n/4 nodes, max cost each = 2 (cmp both children)
  - n/8 nodes, max cost each = 2 levels * 2 cmps
  - n/16, 3 levels*2 cmps
  - At the root, may need up to log n cmps to fix up
    - but there is only one node at root level

Strategy 2: Analysis

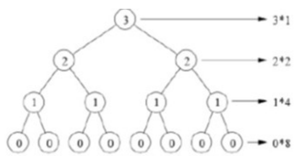


A	O	T	N	R	M	G	A	E	R	A	I	X	B	J

Strategy 2: Analysis



A	O	T	N	R	M	G	A	E	R	A	I	X	B	J



Analysis of buildheap()

Loose bound:

- **`downheap()`** *O(logn)*
- *n* operations
- On first glance: *O(n log n)*

BUT: observe

- only the root ever goes has a *log n* **`downheap()`**
- The n/2 leaves have 0 work for **`downheap()`**
- n/4 leaves at level h-1 have max 1 **`downheap()`**

Analysis of buildheap()

- Overall:
  - at most *ceil*(n/$2^{(h+1)}$) nodes exist at height h
  - When *h = 0*, n/2 nodes
  - When *h = 1*, n/4 nodes
  - When *h = floor(log n),* 1 node
- Total cost =
  - $\sum_{(h=0 \rightarrow \text{floor}(\log n))} \text{ceil}(n/2^{(h+1)}) * O(h)$

number of nodes at this level

work for each node at this level to restore heap

Analysis of `buildheap()`

$\sum_{(h=0 \rightarrow \text{floor}(\log n))} \text{ceil}(n/2^{(h+1)}) * O(h)$

$= O(n \sum h/2^h)$

(converging geometric series)

$= O(n)$

See Cormen, Leiserson, and Rivest for more detail

Heapsort

We will be using Priority Queues in the context of graph algorithms, a lot!

But note that the Priority Queue suggests an efficient sorting algorithm:

- Heapsort

Applications

- Bandwidth Management:
  - VoIP, IPTV
- Shortest Path Algorithms:
  - Pathfinding, navigation, games
- Job Scheduling:
  - OS, Clusters
- Minimum Spanning Tree algorithm:
  - network design
- Huffman Code:
  - Entropy encoding, compression jpeg, mp3

1-53