Pseudocode

(1) defining a class named "product" (index, value, fraction, weight as variables and a function named frac.)

(2) result ← 0

(3) Taking n(no. of objects) and weight_cap (capacity of weight) as input

(4) product object[n] // declaration of array of product type object

(5) Taking input of all n pairs

(6) from i = object.length/2 downto 1 do // building heap

(7) max_heapify (object, object.length)

```
/* pseudocode of max-heapify (product *A, int i,
                              int size)
{   l ← 2*i
    r ← 2*i+1
if (l<=size and A[l].fraction > A[i].fraction
        largest ← l  //taken as index
else
    largest ← i
if (r<=size and A[r].fraction > A[largest].fraction)
    largest ← r
if (largest != i)
{   exchange A[largest] and A[i]
    max-heapify (A, largest, size)
}
*/
```

(8) from i = 1 to n do

{

(9) maximum ← object[1]

(10) object[1] ← object[length]

(11) length--;

(12) max-heapify (object, 1, length)

(13) if (maximum.weight <= weight_cap)

{

(14) weight_cap ← weight_cap - maximum.weight

(15) result ← result + maximum.value

}

(16) else if (maximum.weight > weight_cap and
weight_cap != 0)

{

(17) result ← result + (maximum.value / maximum.weight)
* weight_cap

~~(18) ∞ ∞ ← (maximum.value / maximum weight~~

(18) weight_cap ← 0

}

(19) else if (weight_cap == 0)

break;

}

← Time analysis

~~# from~~ line (1) to (4) ⇒ c_1

~~# from~~ line (5) + (6) ⇒ $c_2 n$

~~line 6~~

line (7)

Analysis

In max-heapify

$T(n) \leq T(2n/3) + \Theta(1)$

$n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

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

$\therefore \Theta(\log n)$

This function is called $\frac{n}{2}$ times

$\therefore \Theta(\frac{n}{2} \log n)$

But there are some other things to consider. Not all the nodes have height $\lfloor \log n \rfloor$

An n element heap has height $\lfloor \log n \rfloor$

and at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h

$$\therefore \sum_{k=0}^{\log n} \left(\frac{n}{2^{k+1}}\right) o(k) = O\left(n \sum_{k=0}^{\log n} \frac{k}{2^k}\right)$$

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

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

$\therefore O(n)$

line (7) takes $c_3 n$ time

line (8) to (11) $\Rightarrow c_4 n$ time

line (12)

Analysis

Time = $\log n + \log(n-1) + \log(n-2) + \cdots + \log 1$

(1st extraction, 2nd extraction, nth extraction)

$= \log n!$

$= O(n \log n)$

$\therefore$ line (12) $\Rightarrow c_5 n \log n$ time

line (13) to (19) $\Rightarrow c_6 n$

best case => only 1st weight taken. only $c_5 \log n$ in line (12).

$\therefore$ Total $= c_1 + c_2 n + c_3 n + c_4 n + c_5 n \log n + c_6 n$

$= c_1 + n(c_2 + c_3 + c_4 + c_6) + c_5 n \log n$

$= c_1 + cn + c_5 n \log n$

(worst case; all the weights taken)

$\therefore$ ~~It is~~

✱ Data structure implemented

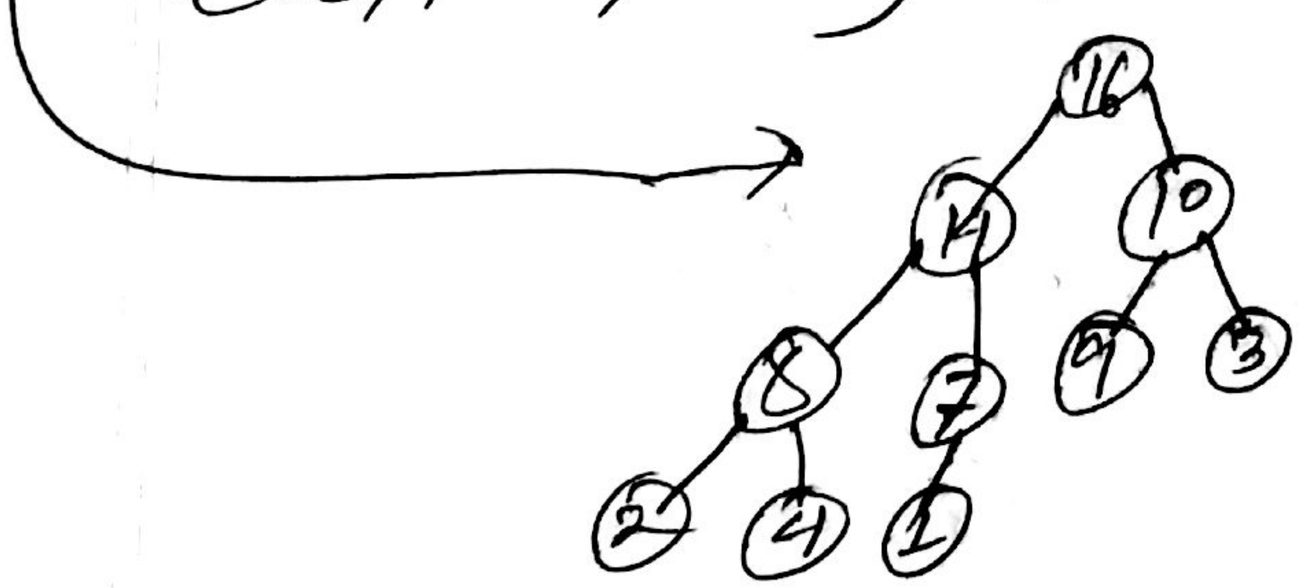
Heap is an array object that can be viewed as a complete binary tree. The tree is filled from left to right.

Each node of the tree corresponds to an element of the array that stores the value in the node.

$A =$

16	14	10	8	7	9	3	2	4	1

The tree is completely filled on all levels except possibly the lowest.



* The root node is A[1]
* The parent node of node i is A[i/2]
* The left child of node i is A[2i]
  and the right child is A[2i+1]

There are 2 types of heaps.

(1) Min-heap

* The element in the root is less than or equal to all elements in both of its subtrees
* Both of its subtrees are min heaps

(2) Max-heap ⟹

* " " " " " " greater " "

A[parent(i)] ≥ A[i] for all nodes i > 1
The largest element in a max heap is stored at the root

$\forall i \Rightarrow A[Parent(i)] \geq A[i]$ for all nodes $i > 1$

The smallest ... root.

Correctness

Here, using max-heapify and building max-heap functions are the basis and complex parts of the whole algorithm.

Build ~~heap~~ heap (max)

Initialization:

prior to 1st loop, $i = n/2$; Each node $\frac{n}{2}+1, \frac{n}{2}+2, \ldots, n$ is a leaf is thus the root of a trivial max-heap.

Maintenance:

Let any node be node i. Its left child is $2i$ and right child is $2i+1$. We assume $i+1, i+2 \ldots n$ to be roots of subtree of max-heap. In that case $2i$ and $2i+1$ are also roots of max-heaps. This is precisely the condition for

the call max-heapify (A, i) to make node i a max-heap root. decrementing i in the for loop ~~and~~ update reestablishes the loop invariant for the next ~~iter~~ iteration.

Termination:

At termination, max-heapify will be called with A and 1 and ~~the~~ each node 1, 2, ---, n is the root of a max-heap.

~~Max-heapify~~