Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
AIEP Send-off Python 2 Session 8

PRIORITY QUEUE

Suppose that we have a list of 7 anime titles alongside their ratings on MyAnimeList, and we are interested in knowing the 3rd highest-rated anime.

Chobits	Bakemonogatari	Inuyasha	Clannad	Steins;Gate	Hakozume	Naruto
7.4	8.3	7.8	8.0	9.1	6.9	8.2

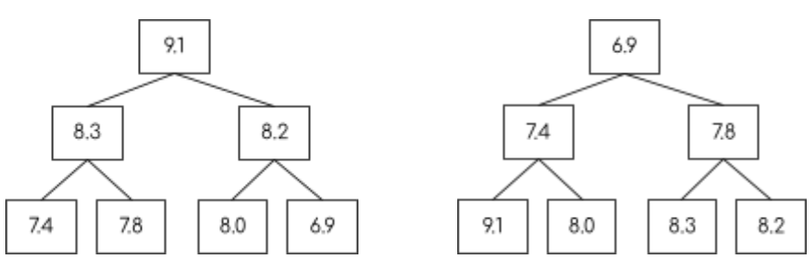
A possible approach would be to sort the titles in order of descending rating then get the 3rd element — however, arranging the *entire* list just to know the 3rd highest-rated anime is a bit of an overkill. Although retrieving the k^{th} item in a list takes $\Theta(1)$ time, sorting n values unfortunately requires $\Theta(n \log n)$ time, resulting in a linearithmic complexity *in toto*. Imagine the inefficiency of performing this on MyAnimeList's full catalog of 19,528[1] anime titles!

This curiosity parallels core concepts in computing. Operating systems (like Windows and macOS) are responsible for scheduling tasks, ensuring that those with higher priority (e.g., instantaneously displaying the characters that the user is typing) take precedence over those with lower priority (e.g., updating an app). Processes finish and new ones pile up in a matter of seconds — ergo, systems cannot afford to squander resources to order *all* the tasks every time, just to figure out which should go first.

These motivate the idea of a priority queue, an abstract data type in which elements with higher priority are "served first." Note that an abstract data type only mandates the description and the operations that the data structure should support; it does not specify anything about the implementation. To put it into context, we can implement a priority queue in however way we want. As long as it is able to accomplish its job of "prioritizing" elements, then it is considered a priority queue.

Needless to say, there is an *efficient* implementation, and this is via a heap[2]. A heap is a tree-based data structure that satisfies the heap property (or heap invariant):

> ***Max-heap:*** If A is the parent of B, the key[3] of A is greater than or equal to the key of B.
> ***Min-heap:*** If A is the parent of B, the key of A is less than or equal to the key of B.



The figure on the left is a max-heap while the one on the right is a min-heap.

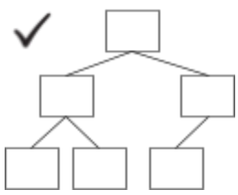
[1] As of March 11, 2022
[2] Heaps are the most popular implementations of priority queues. Some references would even use these two terms interchangeably (although this lack of distinction is technically incorrect).
[3] The *key* is the value stored in a node.

Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

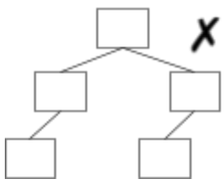
BINARY HEAP

Heaps themselves come in different variants: binary, leftist, Fibonacci, and 2–3 heaps to name a few. In this handout, we are going to focus on the **binary heap**, introduced by the Welsh–Canadian computer scientist J. W. J. Williams in 1964. Aside from the heap property, it also satisfies the **shape property** (or **shape invariant**):

> A binary heap is a **complete binary tree**:
> - All levels, except possibly the lowest level, are completely filled.
> - The nodes in the lowest level are filled from left to right, i.e., they are as far left as possible.



The figure on the left is a complete binary tree while the one on the right is not (since the nodes in the lowest level are not as far left as possible). Moreover, the max- and min-heaps shown on the previous page are binary heaps.



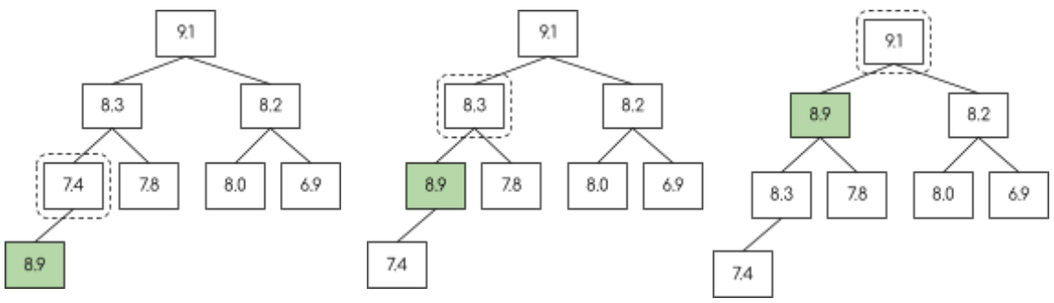
Let us delve into four common operations supported by this data structure. Assume that we are working with max-heaps (operations on min-heaps are analogous).

INSERTING

Suppose we would like to insert the value 8.9 into the max-heap on the previous page.

1. Place the new value in the first available location. The resulting tree should still obey the shape property but not necessarily the heap property. Fittingly, the newly inserted node is described as **out of place** or **offending**.

2. While the value of the offending node is greater than the value of its parent, swap these two nodes. The goal of this iterative procedure is to satisfy the heap property.

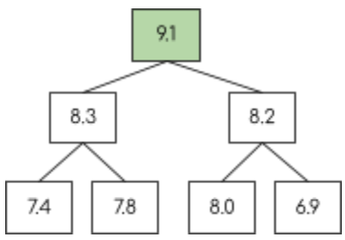


Since the offending node "goes up" the tree, this repeated swapping is also referred to as **bubbling up**, **percolating up**, **trickling up**, **sifting up**, or **swimming**.

The number of operations depends on the number of levels that the offending node has to swim for the heap property to be satisfied. The worst case is that it has to swim all the way to the top of the tree. Since the height of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$, the worst-case time complexity of insertion is $\Theta(\log n)$.

FINDING THE MAXIMUM

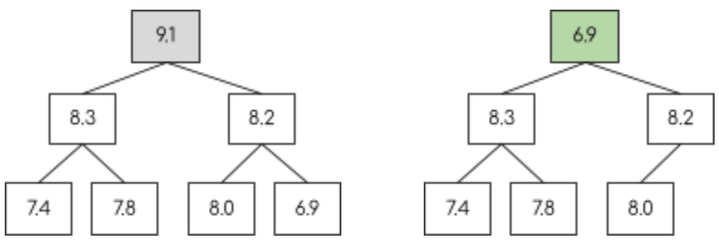
By virtue of the heap property, the maximum value in a max-heap is always stored in the root node. Therefore, the time complexity of finding the maximum is simply $\Theta(1)$.



DELETING THE MAXIMUM

Deleting the maximum is equivalent to removing the root and reorganizing the binary tree so that both the heap and shape properties hold:

1. Replace the root with the last node in the lowest level, in effect deleting the maximum. The resulting tree should still obey the shape property but not necessarily the heap property. Fittingly, this displaced node is described as out of place or offending.



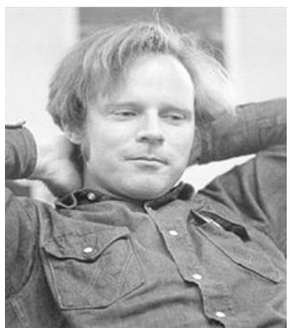
2. While the value of the offending node is less than the value of one of its children, swap it with its *greater-value* child. For example, if the value of the offending node is 6.9 and its two children have values 8.3 and 8.2, then it should be swapped with the child whose value is 8.3.

Analogously, in a min-heap, the offending node is swapped with its *lesser-value* child. Since the offending node "goes down" the tree, this repeated swapping is referred to as bubbling down, percolating down, trickling down, sifting down, or sinking.

The worst case is that the offending node has to sink all the way to the bottom of the tree. Ergo, similar to insertion, the worst-case time complexity of this operation is $\Theta(\log n)$.

BUILDING A HEAP

In J. W. J. Williams' original paper, the method for building a heap from a list of elements is via successive insertions. Consequently, if there are n elements, n insertions would have to be performed, resulting in a worst-case time complexity of $\Theta(n \log n)$.

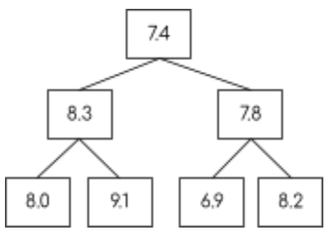


However, this was later found to be suboptimal. The Turing Award laureate[4] Robert W Floyd (pictured on the right) devised a highly ingenious strategy that lowered the time complexity to $\Theta(n)$.

Let us walk through this algorithm using the anime ratings listed on the first page:

Chobits	Bakemonogatari	Inuyasha	Clannad	Steins;Gate	Hakozume	Naruto
7.4	8.3	7.8	8.0	9.1	6.9	8.2

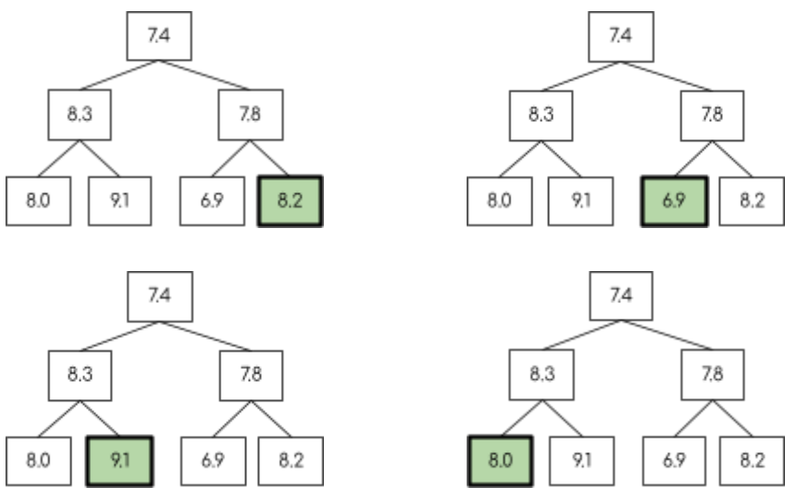
1. Arbitrarily place the elements in a binary tree such that the shape property (but not necessarily the heap property) is respected.



2. Starting at the lowest level, perform sinking on each subtree; the objective is to satisfy the heap property. Continue doing this procedure until the top of the tree is reached. More formally, once the subtrees of height h have been "heapified" (converted to a heap), proceed to heapify the subtrees of height $h + 1$.

   In the visualization below, thicker outlines denote the subtree being heapified. The root of this subtree is colored green to aid us in tracking it as it sinks (or gets displaced).

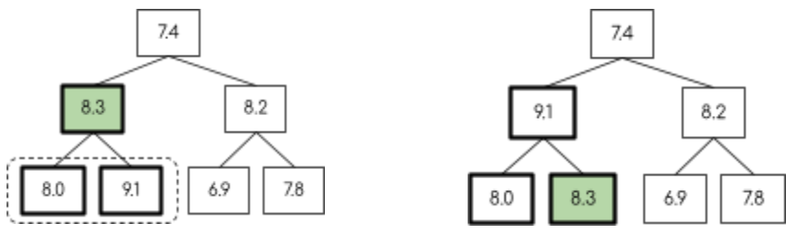
   - Heapifying subtrees of height 0



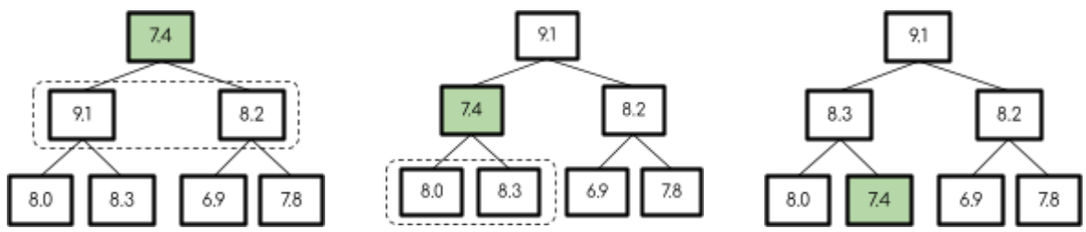
[4] Awarded by the Association for Computing Machinery (ACM), the largest society of computing professionals and students, the Turing Award (named after Alan Turing) is the Nobel Prize in computer science.

- Heapifying right subtree of height 1

- Heapifying left subtree of height 1



- Heapifying subtree of height 2



It can be verified that the resulting tree is a binary max-heap.

Time Complexity Analysis

Since the analysis of the worst-case time complexity of Floyd's algorithm is quite involved compared to the other operations, we dedicate a separate section for it. As we have seen in the walkthrough, we decomposed the task of heapifying a binary tree with 7 nodes and height 2 into the following subtasks:

- There are 4 subtrees of height 0. Since they consist only of leaf nodes, heapifying each of these subtrees does not involve any swaps, totaling $4 \times 0 = 0$ operations.
- There are 2 subtrees of height 1, and heapifying each of these subtrees involves at most 1 swap, totaling $2 \times 1 = 2$ operations.
- There is 1 subtree of height 2, and heapifying it involves at most 2 swaps, totaling $1 \times 2 = 2$ operations.

Big O. To generalize, heapifying a binary tree with n nodes requires roughly

$$\frac{n}{2^1} \cdot 0 + \frac{n}{2^2} \cdot 1 + \frac{n}{2^3} \cdot 2 + \ \ldots$$

operations, with the number of terms equal to the height of a complete binary tree: $\lfloor \log_2 n \rfloor$. The expression can, thus, be rewritten as

$$\sum_{k=1}^{\lfloor \log_2 n \rfloor} \left[\frac{n}{2^k} \cdot (k-1) \right] = n \sum_{k=1}^{\lfloor \log_2 n \rfloor} \frac{k-1}{2^k}.$$

Since the big O gives only an upper bound, we have the liberty to "overshoot" in order to simplify our analysis. We establish a chain of inequalities, culminating in an infinite series:

$$n \sum_{k=1}^{\lfloor \log_2 n \rfloor} \frac{k-1}{2^k} < n \sum_{k=1}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} < n \sum_{k=1}^{\infty} \frac{k}{2^k}.$$

Using the identity[5]

$$\sum_{k=1}^{\infty} \frac{k}{2^k} = 2,$$

we obtain the desired upper bound:

$$n \sum_{k=1}^{\infty} \frac{k}{2^k} = 2n = O(n).$$

Big Omega. Our walkthrough of Floyd's algorithm shows that we are trying to sink each of the n nodes; technically, since the subtrees at the lowest level (i.e., the subtrees of height 0) do not require any swapping, we only need to sink at most half of the nodes. Regardless, the asymptotic lower bound is $\Omega(n)$.

Big Theta. Since both the big O (upper bound) and the big omega (lower bound) are linear, the big theta (tight bound) is $\Theta(n)$, completing our proof.

ARRAY REPRESENTATION

The standard implementation of a binary heap is via an array that reflects a breadth-first traversal of the tree. To illustrate, the max-heap on the previous page can be represented as follows (to highlight parent-child relationships, the values in the left subtree of height 1 are colored yellow while those in the right subtree are colored blue):

0	1	2	3	4	5	6	7
-	9.1	8.3	8.2	8.0	7.4	6.9	7.8

Here are some salient observations:

- The root (which contains the maximum value) is at index 1.
- The parent of the node at index i is at $\left\lfloor \frac{i}{2} \right\rfloor$.
- The children of the node at index i are at $2i$ and $2i + 1$.

Hence, implementing a heap in this fashion allows us to track parent-child relationships using only array indices, discarding the need for pointers. The shape property ensures that the tree's height is minimal and the array representation is as "compact" as possible.

In our analyses of the runtime of insertion and deletion, notice that we ignored their first steps, namely adding the new value in the first available location and replacing the root, respectively. Fortunately, these are not points of concern, as the shape property and the array representation guarantee that these steps finish in $\Theta(1)$ time.

The space complexity is also minimal: $\Theta(n)$, where n is the number of values to be stored. Finally, note that placing the root at index 1 is solely for arithmetic convenience.

[5] The derivation of this identity is left as an exercise for the reader.

💭 Breakpoint A

1. Simulate Floyd's algorithm on larger lists. Where do most of the swapping operations occur: near the bottom of the tree or near the top? Technically, both sinking and swimming are valid procedures for heapifying lists, but why do you think does Floyd's algorithm specifically employ sinking? How would the time complexity be affected if swimming were to be used?
2. Try to write your own implementation of a binary heap data structure.
3. In J. W. J. Williams' paper, he also introduced an efficient $\Theta(n \log n)$ sorting algorithm known as **heapsort**. For a discussion of this algorithm, you may refer to pages 28 to 32 of these lecture notes from the National Taiwan Ocean University.

PYTHON'S `heapq`

The Python Standard Library provides a module that implements a priority queue: **`heapq`**[6]. Unlike the textbook implementation presented in the previous section, Python places the root at index 0 in sync with its zero-based indexing. A heap can be created from an empty list, but an already-populated list can also be converted to a heap using `heapify()`.

Unlike C++'s `priority_queue`, `heapq` implements a **min-heap**; hence, `heap[0]` pertains to the smallest value. A workaround to emulate a max-heap would be to multiply the values by –1 before storing them. In this way, `-1 * heap[0]` would correspond to the highest value in the heap. We shall demonstrate this technique in the first illustrative example.

The table below enumerates some of the core functions offered by this module:

Function	Description
`heapify(x)`	Transforms the list `x` into a min-heap.
`heappush(heap, item)`	Inserts `item` into `heap`.
`heappop(heap)`	Deletes the smallest value in `heap` and returns this (deleted) value.
`heappushpop(heap, item)`	Inserts `item` into `heap`, deletes the smallest value in `heap`, and returns this (deleted) value. Calling this method is more efficient than performing these actions individually.
`heapreplace(heap, item)`	Deletes the smallest value in `heap`, returns this (deleted) value, and inserts `item` into `heap`. Calling this method is more efficient than performing these actions individually.

A complete discussion can be found in Python's official documentation. Lastly, whenever you intend to use this module, remember to add `import heapq` at the start of your code.

[6] Nice pun!

💭 Breakpoint B

1. Examine the source code of `heapq`. How do its implementations of the methods for inserting, deleting the minimum, and heapifying differ from yours?
2. What specific optimization strategies does Python employ in its `heappushpop()` and `heapreplace()` methods? How do they improve the runtime compared to performing the bundled operations individually?

ILLUSTRATIVE EXAMPLES

Priority queues are at the heart of efficient implementations of **greedy algorithms** where it suffices to settle for a **partial ordering**, especially if the list of values is frequently altered. Here are some examples to concretize the utility of our newly learned data structure:

Classic k^{th} Highest Element

Let us generalize our opening problem: Suppose we have a list of n anime titles alongside their ratings at MyAnimeList. Which anime title has the k^{th} highest rating?

📝 **Discussion**
We can avoid sorting the entire list by turning it instead into a max-heap and proceeding to delete and return the highest element in the heap k times. It can be verified that the worst-case time complexity of this approach is $\Theta(n + k \log n)$.

A possible implementation is given below. `anime` is a list of tuples, with the first element of each tuple corresponding to the rating and the second element, to the title of the anime. The rating was selected to be the first element since `heapq` partially orders tuples based on the first element.

```
RATING = 0
TITLE = 1

def kth_highest(anime, k):
    anime_pq = [(-1 * show[RATING], show[TITLE]) for show in anime]
    heapq.heapify(anime_pq)
    kth_highest_rated_title = ""

    for _ in range(k):
        kth_highest_rated_title = heapq.heappop(anime_pq)[TITLE]

    return kth_highest_rated_title
```

The highlighted code demonstrates the "max-heap" workaround described in the previous section. To illustrate, the contents of `anime_pq` after heapification are as follows:

0	1	2	3	4	5	6
-9.1	-8.3	-8.2	-8.0	-7.4	-6.9	-7.8
Steins;Gate	Bakemonogatari	Naruto	Clannad	Chobits	Hakozume	Inuyasha

> 💭 **Breakpoint C**
>
> There is an even more efficient technique for getting the k^{th} largest element: **quickselect**, a divide-and-conquer algorithm with an average-case time complexity of $\Theta(n)$. You may refer to this slide deck from Princeton University for a walkthrough.

Classic Huffman Coding

One of the most important applications of priority queues is in the implementation of Huffman coding. To preface our discussion, consider the task of encoding the name `MAYOIHACHIKUJI` in binary (i.e., using only `1`s and `0`s). A possible solution would be to use the UTF-32 (Unicode Transformation Format) encoding:

Letter	Codeword	Letter	Codeword
M	00000000000000000000000001001101	H	00000000000000000000000001001000
A	00000000000000000000000001000001	C	00000000000000000000000001000011
Y	00000000000000000000000001011001	K	00000000000000000000000001001011
O	00000000000000000000000001001111	U	00000000000000000000000001010101
I	00000000000000000000000001001001	J	00000000000000000000000001001010

`MAYOIHACHIKUJI` would then be encoded as

```
00000000000000000000000001001101000000000000000000000000010000010000000000000000000000000010
11001000000000000000000000000100111100000000000000000000000010010010000000000000000000000000
00010010000000000000000000000000010000010000000000000000000000010000110000000000000000000000
00000001001000000000000000000000000000010010010000000000000000000000010010110000000000000000
000000000000010101010000000000000000000000000100101000000000000000000000000001001001.
```

Notice that, in UTF-32, the length of each encoding is fixed at 32 bits, thereby making it a **fixed-length coding scheme**. Although it is fast to parse (decoding can be done by simply reading and translating contiguous sets of 32 bits), it is quite wasteful; we spent 448 bits or 56 bytes (or 5 lines of this handout) to encode a single Japanese name.

To minimize space consumption, characters that appear more frequently should ideally be encoded with fewer bits. This is the motivation for **variable-length coding**. However, there is a catch: ambiguity can ensue if the code is not uniquely decodable. Consider the following message in Morse code (sans spaces):

```
-....-....-.-.------.
```

There are at least two ways to decode it:

```
-... | . | -.. | .. | -. | -.. | --- | --.   BED IN DOG

-... | .-.. | ..- | . | -.. | --- | --.   BLUE DOG
```

While putting delimiters is an option, they also consume space. A more efficient approach would be to develop a **prefix code**. In this type of coding system, no codeword is a prefix (initial segment) of another codeword.

Letter	Prefix Codeword	Non-Prefix Codeword
M	0110	00
A	00	001
Y	0111	110
O	1101	1101

The table on the right may help in illustrating the distinction between prefix and non-prefix codes. The last column is not a prefix code since the codewords for `M` (`00`) and `Y` (`110`) are prefixes of those for `A` (`001`) and `O` (`1101`), respectively.

📝 Concept

The Huffman coding scheme is an optimal prefix coding scheme[7]: frequently occurring characters are encoded with as few bits as possible.

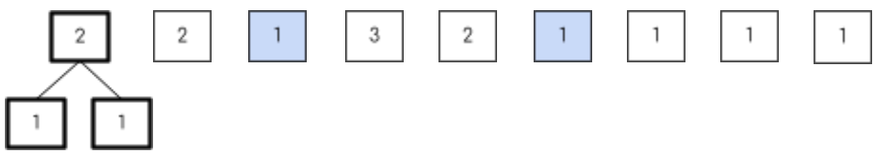
The search for a procedure to generate an optimal prefix coding system has been a central problem in data compression until its solution was given in 1951 (and published a year after) by David A. Huffman in his attempt to accomplish a term paper assigned by his professor Robert Fano. Fano and Claude Shannon (the Father of Information Theory) tried to solve this problem in the late 40s, but the methods they conceived turned out to be suboptimal.

Returning to our task of encoding `MAYOIHACHIKUJI`, the algorithm starts with counting the frequency of each character.

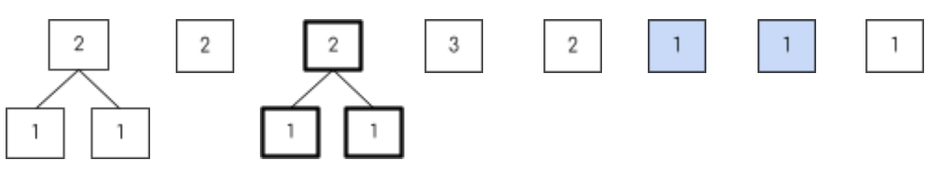
Letter	M	A	Y	O	I	H	C	K	U	J
Frequency	1	2	1	1	3	2	1	1	1	1

The frequencies are then treated as single-node trees inside a data structure.

The two root nodes with the lowest frequencies (blue) are removed from the data structure and become the children of a new node whose value is the sum of these frequencies. The resulting tree (outlined more thickly) is then stored in the data structure.



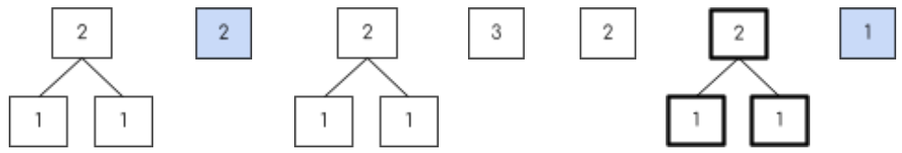
This greedy process of "merging" the two lowest frequencies is repeated until only a single root node remains.

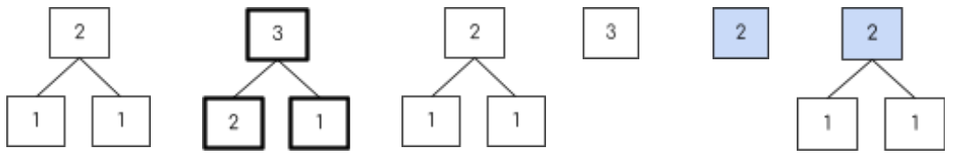


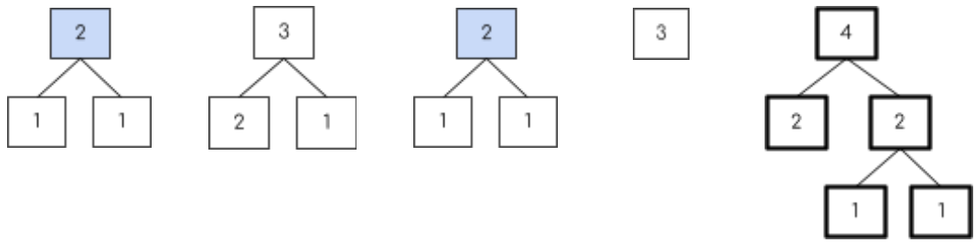
[7] For a formal definition of "optimality" and a rigorous proof of the optimality of the Huffman coding scheme, you may refer to this document from the University of Toronto.

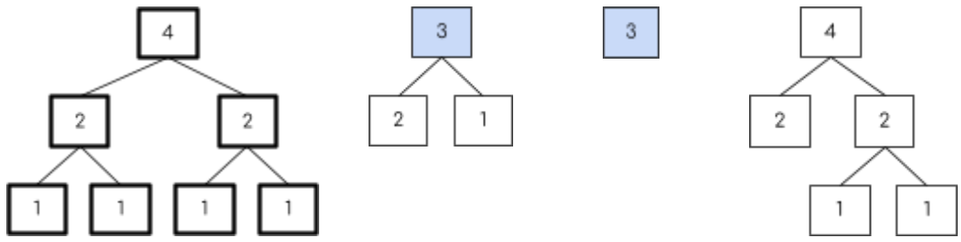
Asian MathSci League, Inc (AMSLI)

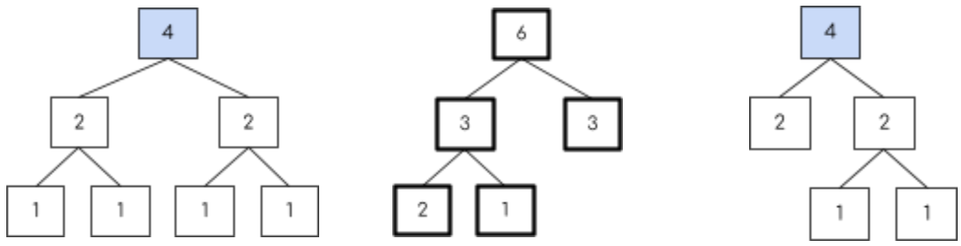
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

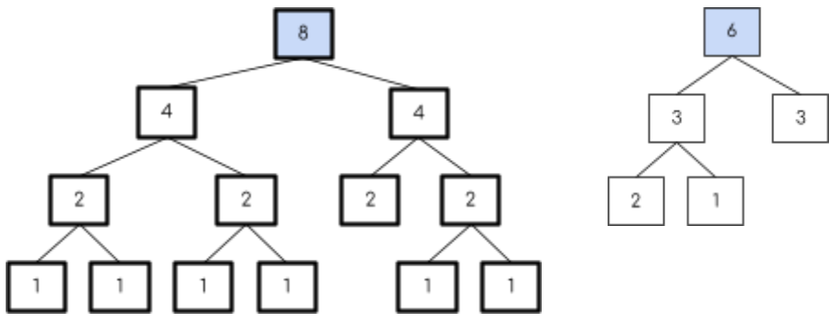


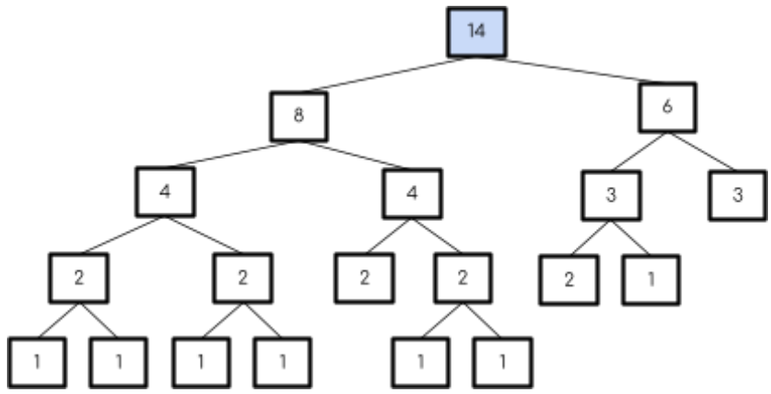






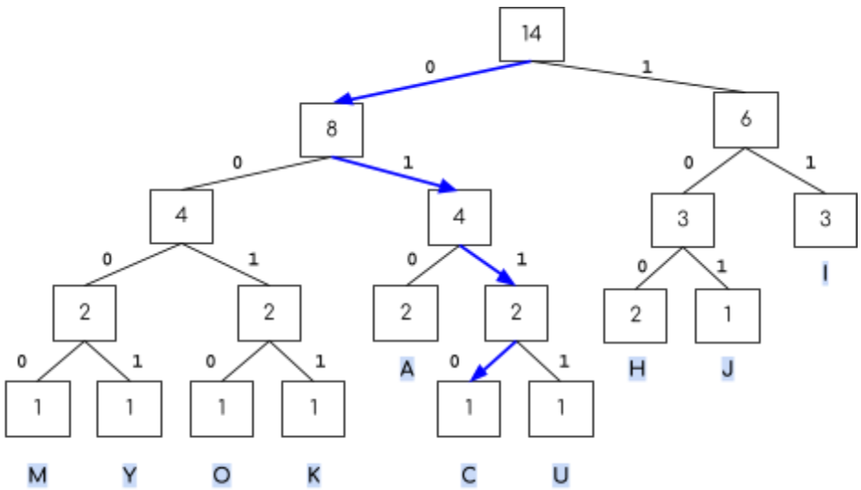






Prepared by ***Mark Edward M. Gonzales***

To visualize how codewords are yielded, we annotate the optimal prefix tree[8]:



The codeword for a character is dictated by the path from the root to the pertinent leaf node; going to the left and to the right (arbitrarily) map to `0` and `1`, respectively. The figure above shows the path corresponding to the codeword for `C`.

For completeness, the table below enumerates the codewords after applying the Huffman coding scheme:

Letter	Codeword	Letter	Codeword	Letter	Codeword
M	0000	I	11	U	0111
A	010	H	100	J	101
Y	0001	C	0110		
O	0010	K	0011		

Only 45 bits are required to encode `MAYOIHACHIKUJI`:

```
000001000010010111000100110100110011011110111
```

The wisdom for "merging" the two lowest frequencies at each iteration becomes clearer in hindsight: it is to ensure that they will be at the lower levels of the prefix tree. Those at the lower levels have longer paths from the root and, therefore, map to longer codewords.

💻 Implementation

Since only a partial ordering is needed to obtain the two lowest frequencies, an efficient implementation of Huffman coding utilizes a min-heap as the primary data structure.

Let us first create a method to count the frequency of each character; the optimal way to accomplish this is via a dictionary. However, since it starts off as empty, we have to use a `defaultdict` instead of a vanilla dictionary so that we can set the value of nonexistent keys to 0 (refer to the highlighted code on the next page). This translates to setting the initial count of the characters to 0.

[8] The Huffman coding scheme can result in multiple but equally optimal prefix trees.

Assuming that the `collections` module has already been imported, the said function can be implemented as follows:

```
def get_frequencies(message):
    frequencies = collections.defaultdict(int)
    for character in message:
        frequencies[character] += 1

    return frequencies
```

In the interest of avoiding the complications of populating the priority queue with the usual tree nodes that require pointers to their children, it would be simpler to store tuples instead. The first element of each tuple is the frequency (the value stored in the root) whereas the second element onwards are the character-codeword pairings in the tree.

To illustrate, the priority queue will contain these tuples after the first "merging" (blue):

```
(1, ['K', '']), (1, ['O', '']), (1, ['M', '']), (1, ['U', '']), (2, ['A', '']),
  (2, ['H', '']), (1, ['Y', '']), (3, ['I', '']), (2, ['C', '1'], ['J', '0'])
```

Below is an implementation of Huffman coding using the approach described:

```
FREQUENCY = 0
CHILDREN = 1
CODEWORD = 1

def huffman_coding(frequencies):
    huffman_pq = [(frequency, [character, ''])
         for character, frequency in frequencies.items()]
    heapq.heapify(huffman_pq)

    while len(huffman_pq) > 1:
        lowest_freq_tree1 = heapq.heappop(huffman_pq)
        lowest_freq_tree2 = heapq.heappop(huffman_pq)

        for node in lowest_freq_tree1[CHILDREN:]:
            node[CODEWORD] = '1' + node[CODEWORD]
        for node in lowest_freq_tree2[CHILDREN:]:
            node[CODEWORD] = '0' + node[CODEWORD]

        sum_lowest_freq = (lowest_freq_tree1[FREQUENCY] +
                lowest_freq_tree2[FREQUENCY])
        sum_lowest_freq_node = ((sum_lowest_freq,) +
                lowest_freq_tree1[CHILDREN:] + lowest_freq_tree2[CHILDREN:])
        heapq.heappush(huffman_pq, sum_lowest_freq_node)

    return huffman_pq
```

Running this code will result in a different encoding for `MAYOIHACHIKUJI` from the one that we manually derived. Nonetheless, it still consists of 45 bits and is, thus, equally optimal.

💭 Breakpoint D

1. Write a method for encoding a message using the Huffman coding scheme.
2. Write a method for decoding a message given the character-to-codeword mapping.
3. One of the most widely used algorithms for creating compressed (zipped) folders, DEFLATE, combines Huffman coding and a lossless data compression algorithm known as LZ77. Its formal specifications are laid down in this document.

Contest Assigning Workstations

2015 ACM-ICPC Northwestern Europe Regional Contest
(Association for Computing Machinery – International Collegiate Programming Contest)

Penelope is part of the admin team of the newly built supercomputer. Her job is to assign workstations to the researchers who come here to run their computations at the supercomputer.

Penelope is very lazy and hates unlocking machines for the arriving researchers. She can unlock the machines remotely from her desk, but does not feel that this menial task matches her qualifications. Should she decide to ignore the security guidelines she could simply ask the researchers not to lock their workstations when they leave, and then assign new researchers to workstations that are not used any more but that are still unlocked. That way, she only needs to unlock each workstation for the first researcher using it, which would be a huge improvement for Penelope.

Unfortunately, unused workstations lock themselves automatically if they are unused for more than m minutes. After a workstation has locked itself, Penelope has to unlock it again for the next researcher using it. Given the exact schedule of arriving and leaving researchers, can you tell Penelope how many unlockings she may save by asking the researchers not to lock their workstations when they leave and assigning arriving researchers to workstations in an optimal way? You may assume that there are always enough workstations available.

Input
The input consists of:

- One line with two integers n ($1 \leq n \leq 300\,000$), the number of researchers, and m ($1 \leq m \leq 10^8$), the number of minutes of inactivity after which a workstation locks itself;
- n lines each with two integers a and s ($1 \leq a, s \leq 10^8$), representing a researcher that arrives after a minutes and stays for exactly s minutes.

Output
Output the maximum number of unlockings Penelope may save herself.

Sample Input	Sample Output
3 5 1 5 6 3 14 6	2
5 10 2 6 1 2 17 7 3 9 15 6	3

📝 Discussion

Researcher	Arrival Time	Ending Time
A	1	1 + 2 = 3
B	2	2 + 6 = 8
C	3	3 + 9 = 12
D	15	15 + 6 = 21
E	17	17 + 7 = 24
F	37	37 + 4 = 41

To get a feel for the problem, it may be helpful to run a simulation using an expanded version of the second set of sample input.

For clarity and convenience, we begin by ordering the input based on the arrival time and assigning letters to the researchers.

We proceed to the simulation per se (the workstations are numbered from 1 onwards):

Time *(t)*	Arriving Researcher	Unlocked Workstations	Assigned Workstation	Remarks
1	A	-	1	Penelope has to unlock a workstation for the first researcher. Workstation 1 is now in use until $t = 3$ and unlocked until $t = 13$.
2	B	1 In use until $t = 3$ Unlocked until $t = 13$	2	Since B's arrival time is before A's ending time, B is blocked from using Workstation 1, and Penelope has to unlock another workstation. Workstation 2 is now in use until $t = 8$ and unlocked until $t = 18$.
3	C	1 Free Unlocked until $t = 13$ 2 In use until $t = 8$ Unlocked until $t = 18$	1	C can be assigned to Workstation 1 since A has just vacated it and it still has not auto-locked. Workstation 1 is now in use until $t = 12$ and unlocked until $t = 22$.

Time *(t)*	Arriving Researcher	Unlocked Workstations	Assigned Workstation	Remarks
15	D	1 Free Unlocked until t = 22 2 Free Unlocked until t = 18	2	Penelope has the choice to assign D to either Workstation 1 or 2. But the greedy strategy favors assigning D to Workstation 2 since it will lock first. Workstation 2 is now in use until t = 21 and unlocked until t = 31.
17	E	1 Free Unlocked until t = 22 2 In use until t = 21 Unlocked until t = 31	1	E can be assigned to Workstation 1 since it still has not auto-locked. Workstation 1 is now in use until t = 24 and unlocked until t = 34.
37	F	-	1 or 2	At t = 37, both Workstations 1 and 2 have auto-locked, forcing Penelope to unlock any one of them.

Our simulation suggests that the crux of the greedy exploit is assigning the arriving researcher to the free and unlocked workstation that will lock first. We expound a bit more on the cases to be considered, with an emphasis on the workstation that will lock first:

- If this workstation is not free (the incoming researcher's arrival time is before the current user's ending time), Penelope is forced to unlock another workstation.

  To lend some justification, suppose the two workstations with the earliest lock times are W_1 and W_2, and their lock times are ℓ_1 and ℓ_2, respectively, where $\ell_1 \leq \ell_2$. The ending times of their current users are $\ell_1 - m$ and $\ell_2 - m$, respectively.

  Since $\ell_1 \leq \ell_2$, it follows that $\ell_1 - m \leq \ell_2 - m$. This means that the ending time of the user of W_2 is the same as the ending time of the user of W_1 or even past it.

- If this workstation is free and has not yet auto-locked (the incoming researcher's arrival time is not past the workstation's lock time), Penelope can just assign the researcher to this workstation and, therefore, save an unlocking.

- If this workstation has already auto-locked, Penelope has to move on and check the next workstation.

The sketch of our algorithm underscores the need for a data structure that can store the lock times of the workstations and allow us to identify which will lock first. However, since lock times are updated with every arriving researcher, it is inefficient to have to sort at each iteration. Fortunately, since we are only concerned with the workstation that will lock first, we can take advantage of the partial ordering offered by a priority queue (min-heap).

An implementation of this idea is provided on the next page.

```
ARRIVAL = 0
END = 1

def assigning_workstations(arrival_end_times, lock_wait_time):
    arrival_end_times.sort()
    lock_times_pq = []
    saved_unlockings = 0

    for researcher_time in arrival_end_times:
        assigned = False
        while lock_times_pq and not assigned:
            end_time = lock_times_pq[0] - lock_wait_time
            if researcher_time[ARRIVAL] < end_time:
                assigned = True
            elif researcher_time[ARRIVAL] <= lock_times_pq[0]:
                heapq.heappop(lock_times_pq)
                assigned = True
                saved_unlockings += 1
            else:
                heapq.heappop(lock_times_pq)

        heapq.heappush(lock_times_pq, researcher_time[END] + lock_wait_time)

    return saved_unlockings
```

The parameter `arrival_end_times` is a list of tuples, with the arrival and ending times as the first and second elements of each tuple, respectively. Remember to process the input, as it gives the length of the researcher's stay rather than the ending time.

👩‍💻 EXERCISES

1. Prim's Algorithm. Proposed by Czech mathematician Vojtěch Jarník and popularized by computer scientists Robert C. Prim and Edsger W. Dijkstra, this algorithm is used to find the minimum spanning tree of a weighted undirected graph. Write an efficient implementation of Prim's algorithm using a priority queue.

   For a high-level walkthrough (as well as some hints), you may refer to pages 14 to 25 of these lecture notes from the Hong Kong University of Science and Technology.

2. Add All (University of Valladolid [UVa] Online Judge 10954)
3. Keep the Customer Satisfied (UVa Online Judge 1153)
4. Jane Eyre (Bergen Open 2019)
5. I Can Guess the Data Structure! (Rujia Liu's Present 3: A Data Structure Contest Celebrating the 100th Anniversary of Tsinghua University)

Exercises 2 to 5 were selected from the list of practice problems suggested by S. Halim and F. Halim in *Competitive Programming 3: The New Lower Bound of Programming Contests*.

-- return 0; --