

# DAA, DSA-Attempted self study

---

14th Dec 2023

---

## Egyptian fractions

**Definition.** A rational number which in reduced form has a numerator of 1 is called as an Egyptian fraction.

**FACT** Ancient Egyptians who also worked with base 10 number systems, actually had in their syntax only these type of fractions and all others were built from these basic fractions.

**Question:** Can Egyptians actually write all fractions less than one in this way?

**Answer:** Yes as the following algorithm to obtain such an expansion shows.

---

Egyptifier( $p/q \in \mathbb{Q}$ )

---

**Ensure:**  $p < q$  and  $q \neq 0$

```

1: nume  $\leftarrow p$ 
2: ret_list  $\leftarrow []$     \\ This list will have all the denominators of the component Egyptian fractions.
3: while nume  $\neq 0$  do
4:    $n \leftarrow \lceil q/p \rceil$ 
5:   ret_list.append(n)
6:   nume  $\leftarrow pn - q$ 
7: end while
8: Output ret_list

```

---

**Claim.** This algorithm works!

**Proof.** The idea is to use the monovariant *nume*. On each iteration, by the choice of  $n$ , nume monotonously decreases while being non-negative. This is so because,

$$n = \lceil q/p \rceil \iff n - 1 < q/p \leq n \iff 1/n \leq p/q < 1/(n - 1) \iff 0 \leq (pn - q)/q \text{ and } pn - q < p$$

■

---

15th Dec 2023

---

## Interval Scheduling

**Input** Certain events given as a list of 2-tuples  $(s, f)$  denoting the start and finish timings of the event.

**Output** A sub-list of events from the original list, that are non overlapping and have the maximum possible number of events.

This can naturally be useful in several settings of practical interests like picking the maximum number of rides in a theme park.

The idea is the following. Keep picking the event that ends at the earliest, and does not clash with the ones already chosen. This greedy approach is implemented in the following way. We use the heap data structure with the following terminology.

1. **MinHeapify** with some order function does heapify where the comparison is made with the given function.
2. **Pop** removes that element from the heap.
3. **Peek** Returns the first element from the heap.

---

Scheduler(L: list of event tuples)

---

```

1: S ← MinHeapify(L) with start time ordering.
2: F ← MinHeapify(L) with end time ordering.
3: Ret_list ← [ ]
4: while F not empty do
5:   e ← Peek(F)
6:   s ← Peek(S)
7:   while s[0] ≤ e[1] do
8:     S.Pop(s)
9:     F.Pop(s)
10:    s ← Peek(S)
11:   end while
12:   Ret_list.append(e)
13: end while
14: Output Ret_list

```

---

**Claim.** This algorithm does return a list with maximum number of non-clashing (here after referred as compatible) events.

**Proof.** Say  $\mathcal{O}$  is a list of events with  $m$  elements such that it has the maximal number of compatible events. Let  $R$  be the returned list. We shall prove that  $|R| = |\mathcal{O}|$ .  
The idea is that the greedy algorithm always stays ahead of this optimal list.

**Claim.**  $R[i][1] \leq \mathcal{O}[i][1]$

**proof.** We prove by induction. For the first step it's clear by design. We assume the statement holds for  $j - 1$ . But if so, then in the  $j$ th step,  $\mathcal{O}[j]$  begins after  $R[j - 1]$  ends. Since the algorithm chooses the event with least finish time that does not clash with the existing ones, the above statement follows. ■

Consequently, If the list  $\mathcal{O}$  has got some  $m$  events, then  $R$  has at least those many events. ■

---

16th Dec 2023

---

## Weighted Interval Scheduling

Not all rides in a theme park are equally liked. Some are better than others. Say this is denoted by their weight. How do we pick a compatible set of rides with the largest weight?

**Input** List of tuples of events and corresponding weights.

**Output** Sub-list of the input such that the events are all compatible and add to maximal weight.

The idea is again simple. We shall build the set slowly such that at all points the list has the heaviest-compatible list up to a certain finish time. When the finish time of a new event is encountered, the weight of having it compared with the weight of not having it. Correspondingly the list is updated.

The procedure can be described as follows.

1. Sort the events with start times and finish times.
2. Pick the event that ends the earliest.
3. Move to the event that ends second earliest.
4. If no clash, add it. If clashed, compare and choose.
5. At an arbitrary state, say the max finish time of chosen events is  $f$ . Move to the event that ends after but closest to  $f$ .
6. If no clash add it. If clashed, compare having it and not having it, based on which whether to choose it.
7. Go through all events.

Here is a pseudocode.

---

Weighted-scheduler( $E$ :list of 3-tuples)

---

```

1:  $F \leftarrow \text{Sort}(E)$  with finish key.
2:  $\text{Ret\_list} \leftarrow []$ 
3:  $\text{Part-sums} \leftarrow []$ 
4: procedure FIND-CLASH( $L$ :list of compatible events, $e$ :some event)
5:   Binary-Search to return the index of the least element that clashes with  $e$  in  $L$ 
6: end procedure
7: for  $e$  in  $F$  do  $j \leftarrow \text{Find-Clash}(\text{Ret\_list}, e)$ 
8:   if  $\text{Part-sums}[j-1] + e[3] > \text{Part-sums}[-1]$  then
9:      $\text{Ret\_list} \leftarrow \text{Ret\_list}[0:j-1].\text{append}(e)$ 
10:     $\text{Part-sums} \leftarrow \text{Part-sums}[0:j-1].\text{append}(\text{Part-sums}[j-1] + e[3])$ 
11:   end if
12: end for Output  $\text{Ret\_list}$ 

```

---