

# 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
8:    $j \leftarrow \text{Find-Clash}(\text{Ret\_list}, e)$ 
9:   if  $\text{Part-sums}[j-1] + e[3] > \text{Part-sums}[-1]$  then
10:     $\text{Ret\_list} \leftarrow \text{Ret\_list}[0:j-1].\text{append}(e)$ 
11:     $\text{Part-sums} \leftarrow \text{Part-sums}[0:j-1].\text{append}(\text{Part-sums}[j-1] + e[3])$ 
12:   end if
13: end for
14: Output  $\text{Ret\_list}$ 

```

---

17th Dec 2023

---

## Coins problem and Dynamic programming

**Question** Say there are certain denominations of coins, and you have to give a certain total amount of money to someone. What is the least amount of coins needed?

Ofcourse this is a problem of high interest. Given a set of denominations  $\mathcal{D}$  and a target money  $T$ , let  $M(n, \mathcal{D})$  denote our answer, A smallest list of coins of total  $T$ . Then our solution is as follows.

$$M(n, \mathcal{D}) = \operatorname{argmin} \operatorname{len}\{(M(n - d, \mathcal{D})).\text{append}(d) | d \in \mathcal{D}\}$$

Now this basically reduces the problem of solving a bigger problem into solving a lot of subproblems. So we essentially do not want to solve the subproblems repeatedly. Hence we need a register of the subproblems that have been solved. Such an approach to breaking down problems is called as dynamic programming. First we shall write the brute force algorithm.

---

Brute-force( $T$ :total, $\mathcal{D}$ :List of denominations)

---

```
1: if  $T \leq 0$  then
2:   Output []
3: end if
4: Ret_list  $\leftarrow$  []
5: procedure CONVINIENT-MIN( $a, b$ )
6:   if  $a$  isempty then
7:     Output  $b$ 
8:   else if  $b$  isempty then
9:     Output  $a$ 
10:  else
11:    Output lesser length list of the two.
12:  end if
13: end procedure
14: for  $d \in \mathcal{D}$  do
15:   temp  $\leftarrow$  Brute-force( $T - d, \mathcal{D}$ )
16:   Ret_list  $\leftarrow$  Convinient-min(temp, Ret_list)
17: end for
18: Output Ret_list
```

---

Here it is easily seen that the same sub-problem is solved many times in different calls of the function. We can greatly save this time by just using memory. Consider the modified code below. Here the curly brace  $\{ \}$  denotes an empty dictionary. (The code in the next page) This code is many times faster because it only calls the function recursively  $\mathcal{O}(n)$  time, which is much better, than the exponentially large number of recursive calls made otherwise.

We can recast the same idea in a bottom up manner. We see that we reduce the problem to that sub-problem that has the most optimal output. We shall call that sub-problem its predecessor. Here is the implementation.

---

Bottom-up-formulation( $T$ :total, $\mathcal{D}$ :List of denominations)

---

```
1: Pre  $\leftarrow$  arr[ $T + 1$ ] \\array of length  $T+1$ 
2: Number  $\leftarrow$  arr[ $T + 1$ ] \\array of length  $T+1$ 
3: Pre[0], Number[0], i  $\leftarrow$  0, 0, 1
4: while  $i \leq T$  do
5:   Temp  $\leftarrow \infty$ 
6:   for  $d \in \mathcal{D}$  and  $d \leq i$  do
7:     if Temp  $\geq$  Number[ $i-d$ ] then
8:       Pre[ $i$ ]  $\leftarrow i-d$ 
9:       Number[ $i$ ]  $\leftarrow$  Number[ $i-d$ ]+1
10:    end if
11:  end for
12:  i  $\leftarrow i+1$ 
13: end while
14: Output Number[ $T+1$ ]
```

---

The optimal sequence of coins can be inferred from the predecessor list. Follow the predecessor of  $T$  until you hit zero. The successive differences of the generated sequence is the list of coins.

---

Brute-force-withmemory( $T$ :total, $\mathcal{D}$ :List of denominations)

---

```
1: if  $T \leq 0$  then
2:   Output []
3: end if
4: memo  $\leftarrow \{ \} \setminus \setminus$  Entries stored as  $T : Min(T, \mathcal{D})$ 
5: if  $T$  in memo then
6:   Output memo[ $T$ ]
7: end if
8: Ret_list  $\leftarrow []$ 
9: procedure CONVINIENT-MIN( $a, b$ )
10:  if  $a$  isempty then
11:    Output  $b$ 
12:  else if  $b$  isempty then
13:    Output  $a$ 
14:  else
15:    Output lesser length list of the two.
16:  end if
17: end procedure
18: for  $d \in \mathcal{D}$  do
19:   temp  $\leftarrow$  Brute-force( $T - d, \mathcal{D}$ )
20:   Ret_list  $\leftarrow$  Convinient-min(temp, Ret_list)
21: end for
22: memo.add( $T$ :Ret_list)
```

---

The correctness of these procedures is guarenteed by the recursive relation that they solve.

## Dictionary DS

Here we shall see an implementation of the **Dictionary** data structure. It supports the following operations.

- **INSERT(val,key)** procedure puts a certain value with key into the dictionary.
- **SEARCH(key)** procedure outputs the value of the given key.
- **DELETE(key)** procedure removes the value associated with a certain key.

An implementation of it using binary search trees. Checking if things are still fine.