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 thier 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 algortithm to obtain such an expansion shows.

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

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

```
nume ←p
ret_list ←[ ]    \\This list will have all the denominators of the component egyptian fractions.
while nume ≠ 0 do
    n ←⌈q/p⌉
    ret_list.append(n)
    nume ←pn-q
end while
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 usefull 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 implemeted in the following way. We use the heap data structure with the following terminology.

1. **MinHeapify** with some order function does heapify where the comaprision 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)

```
S ←MinHeapify(L) with start time ordering.
F ←MinHeapify(L) with end time ordering.
Ret_list ←[ ]
while F not empty do
    e ←Peek(F)
    s ←Peek(S)
    while s[0] ≤ e[1] do
        S.Pop(s)
        F.Pop(s)
        s ←Peek(S)
    end while
    Ret_list.append(e)
end while
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 ahed of this optimal list.
>
> **Claim.** $R[i][1] \leq \mathcal{O}[i][1]$
>
> > **proof.** We prove by induction. For the first step its clear by design. We assume the statement holds for $j-1$. But if so, then in the jth step, $\mathcal{O}[j]$ begins after $\mathcal{R}[j-1]$ 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 atleast 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 wieght?

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 upto 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 evests 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 wether to choose it.
7. Go through all events.

Here is a pseudocode.

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

```
F ←Sort(E) with finish key.
Ret_list ←[ ]
Part-sums ←[ ]
procedure FIND-CLASH(L:list of compatible events,e:some event)
    Binary-Search to return the index of the least element that clashes with e in L
end procedure
for e in F do
    j ←Find-Clash(Ret_list,e)
    if Part-sums[j-1]+e[3]>Part-sums[-1] then
        Ret_list ←Ret_list[0:j-1].append(e)
        Part-sums ←Part-sums[0:j-1].append(Part-sums[j-1]+e[3])
    end if
end for
Output 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 $\mathcal{D}$ ans 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.

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

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 subroblems 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 algortihm.

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

```
if T ≤ 0 then
    Output []
end if
Ret_list ←[]
procedure Convinient-min(a, b)
    if a isempty then
        Output b
    else if b isempty then
        Output a
    else
        Output lesser length list of the two.
    end if
end procedure
for d ∈ D do
    temp ←Brute-force(T − d, D)
    Ret_list ←Convinient-min(temp,Ret_list)
end for
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 in the next page) This is code is many times fater because it only calls the function reursively $\mathcal{O}(n)$ 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 prdecessor. Here is the implementation.

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

```
Pre ←arr[T + 1] \\array of length T+1
Number ←arr[T + 1] \\array of length T+1
Pre[0],Number[0],i ←0,0,1
while i ≤ T do
    Temp ←∞
    for d ∈ D and d ≤ i do
        if Temp¿Number[i-d] then
            Pre[i] ←i-d
            Number[i] ←Number[i-d]+1
        end if
    end for
    i ←i+1
end while
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 succesive differences of the generated sequence is the list of coins.

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

```
if T ≤ 0 then
    Output []
end if
memo ←{ } \\Entries stored as T : Min(T, D)
if T in memo then
    Output memo[T]
end if
Ret_list ←[]
procedure CONVINIENT-MIN(a, b)
    if a isempty then
        Output b
    else if b isempty then
        Output a
    else
        Output lesser length list of the two.
    end if
end procedure
for d ∈ D do
    temp ←Brute-force(T − d, D)
    Ret_list ←Convinient-min(temp,Ret_list)
end for
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 is possible and will be discussed later.

18th Dec 2023

Knap-sack problem an interlude

Question Say there are certain items with certain weights and values. What is the maximum value of items that can be put into a knapsack of a certain capacity?

Say a set if items($\mathcal{I}$) is give with values given by the $val : \mathcal{I} \to \mathbb{R}$ function, and weights given by the $wt : \mathcal{I} \to \mathbb{R}$ function, we need to find $S \subseteq I$ such that,

$$\sum_{s\in S} val(s) \text{ is maximized under the constraint } \sum_{s\in S} wt(s) \leq C$$

where C is the capacity of the knapsack.

Turns out this is quite a challenging problem. A polynomial time algorithm for the general case is not known for this problem. However, there is a polynomial time algorithm for the special case where the weights are integers.

We shall see that algorithm here.

The idea is to use dynamic programming. We shall define a function $M(i, c)$ that gives the maximum value of items that can be put into a knapsack of capacity c, using only the first i items. We shall see that this function can be computed in a bottom up manner.

The recursive relation is as follows.

$$M(i, c) = \max\{M(i-1, c), M(i-1, c - wt(i)) + val(i)\}$$

The first term in the max is the case where the ith item is not taken, and the second term is the case where it is taken. Here is the pseudocode.

Knap-sack($\mathcal{I}$:set of items,$wt : \mathcal{I} \to \mathbb{R}$:weight function,$val : \mathcal{I} \to \mathbb{R}$:value function,$C$:capacity)

$M \leftarrow arr[|\mathcal{I}|+1][C+1]$
for i **in** $[0, |\mathcal{I}|]$ **do**
    $M[i][0] \leftarrow 0$
end for
for c **in** $[0, C]$ **do**
    $M[0][c] \leftarrow 0$
end for
for i **in** $[1, |\mathcal{I}|]$ **do**
    for c **in** $[1, C]$ **do**
        if $wt(i) \leq c$ **then**
            $M[i][c] \leftarrow \max\{M[i-1][c], M[i-1][c - wt(i)] + val(i)\}$
        else
            $M[i][c] \leftarrow M[i-1][c]$
        end if
    end for
end for
Output $M[|\mathcal{I}|][C]$

even when the weights are not integers we can use this algortihm by scaling by a factor or reducing the step size.

19th Dec 2023

Dictionary DS continued...

We shall treat nodes as objects with the following attributes,

1. **key** a number serving as its name.
2. **val** the data stored in the node. If homogenity is needed then pointers can be stored instead of values.
3. **left** is a pointer to the left node.
4. **right** is a pointer to the right node.

Having these consider the following operations.

For searching basically go through the same process. Except in the last If block instead assigning the key and val, you can check the key and read the val. Deletion is more complicated.

INSERT(val,key,root)

```
ptr ←root
procedure NAVIGATE(ptr,key)
    if *ptr.key > key then
        Output *ptr.left
    else
        Output *ptr.right
    end if
end procedure
temp ←Navigate(ptr,key)
while temp != NULL do
    ptr ←temp
    temp ←Navigate(ptr,key)
end while
if *ptr.key > key then
    (*ptr.left).key ←key
    (*ptr.left).val ←val
else
    (*ptr.right).key ←key
    (*ptr.right).val ←val
end if
```

DELETE(key,root)

```
ptr ←root
procedure FIND(ptr,key)
    if *ptr.key > key then
        Output *ptr.left
    else if *ptr.key < key then
        Output *ptr.right
    else
        Output FOUND
    end if
end procedure
temp ←ptr
while temp != FOUND do
    ptr ←temp
    temp ←Navigate(temp,key)
end while
if *temp.left != NULL then
    temp_1 ←*ptr.left
    while *(*temp_1.right).right != NULL do
        temp_1 = *temp_1.right
    end while
    ins ←*temp_1.right
    *temp_1.right ←*(*temp_1.right).left
    *ptr.
end if
```