Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 4

Greedy Algorithms

Approaches to algorithm design

There are different approaches in designing an algorithm.

So far, we have seen:

- iterative algorithms (e.g. for-loops on arrays)
- divide-and-conquer algorithms (e.g. quicksort, mergesort)
- recursive algorithms (as above)

Today we are going to look at **GREEDY** algorithms

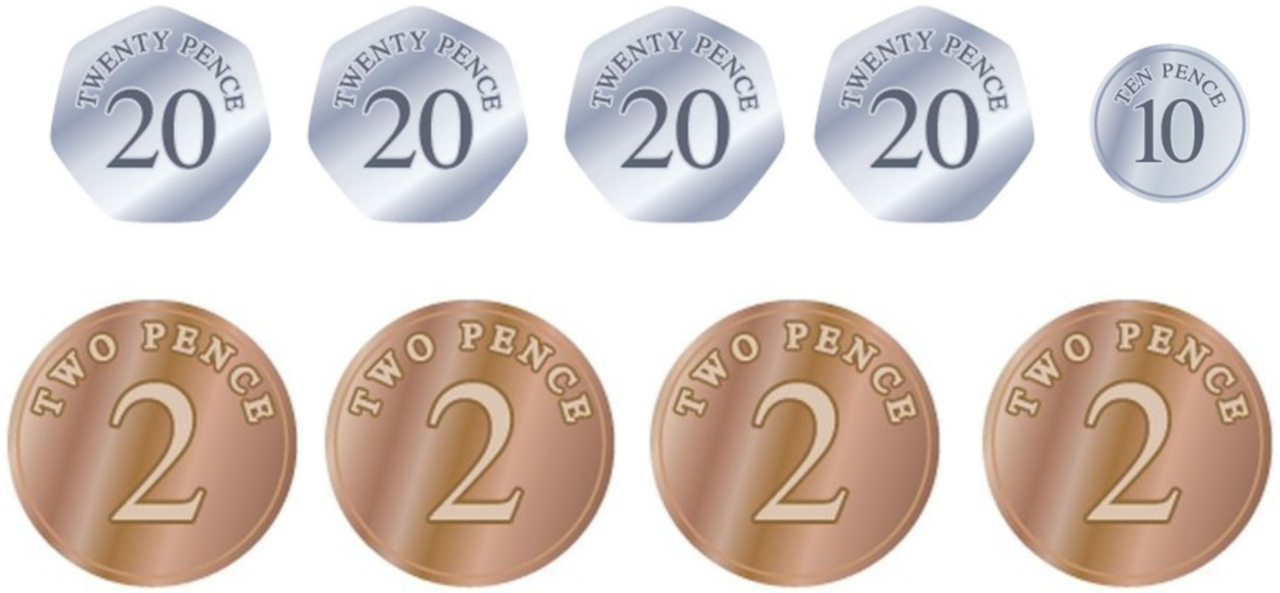
this is a way to build optimal algorithms making what seems the best possible choice at each individual step

How to give change with least possible coins

Least Coin Split: given an amount m of money, find the minimum number of coins whose value adds up to m.

m is an integer, counting the money in pennies.

For example, 98 can be broken down to 9 coins:



... is this optimal ??

Least Coin Split: naive solution

Least Coin Split: given an amount m of money, find the minimum number of coins whose value adds up to m.

Naive solution

To find the minimum number of coins adding up to m:

- pick a coin c whose value is less or equal to m and compute:
    - the min number of coins we can split m into without including coin c in the split
    - the min number of coins we can split m into including coin c in the split
- return the minimum number of coins between these two cases

Least Coin Split: greedy solution

Least Coin Split: given an amount m of money, find the minimum number of coins whose value adds up to m.

Greedy solution

To find the minimum number of coins adding up to m:

- pick the largest coin c whose value is less or equal to m
- find the minimum number of coins adding up to $m - c$
- add 1 to that number and return it

Greedy solution formally

Use an array `coin` to store our coins in descending value:

```
coin = [200, 100, 50, 20, 10, 5, 2, 1]
```

The greedy algorithm looks as follows:

```
def coinSplitGD(m):
    if m == 0:
        return 0
    for i in range(len(coin)):
        if coin[i] <= m:
            return 1 + coinSplitGD(m-coin[i])
```

Greed is fast but can be tricky

The algorithm is simple and fast!

- try to compare it e.g. to trying out all possible ways to break m into coins and picking up the minimum value...

How do we know this is correct?

The reason is delicate. The coins we use have the following property:

- if smaller coins add up to a sum that reaches a bigger coin, it is optimal to pick the bigger coin instead

We can see that that would not be the case if e.g. our coins were:

`coin = [200, 100, 50, 25, 20, 10, 5, 2, 1]` (see Exercises)

Another example: Event scheduling problem

Suppose we have a venue and want to schedule events in it:

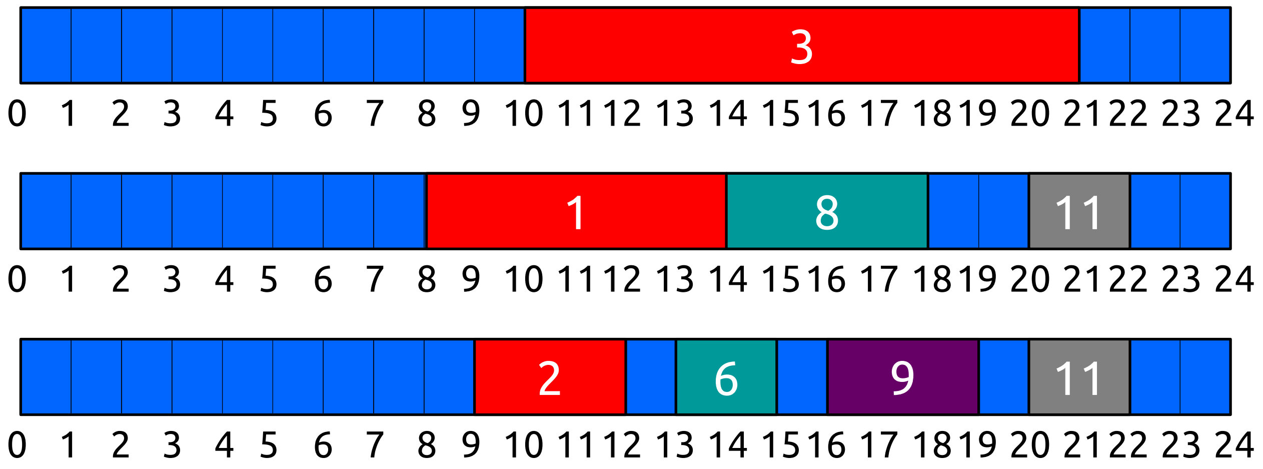
- at each time, there can be at most one event scheduled
- we want to schedule **AS MANY EVENTS AS POSSIBLE** (i.e. maximise the number of scheduled events)
- for simplicity, suppose we want a function that returns the maximum number of events we can schedule

Again, this is an *optimisation* problem.

I.e. there are many schedulings of events that are valid (i.e. no two events overlap), but we want to figure out the maximum number of events that we can schedule in a valid manner.

Example

Possible valid schedulings:



event	times
1	8:00 – 14:00
2	9:00 – 12:00
3	10:00 – 21:00
4	11:00 – 13:00
5	11:00 – 16:00
6	13:00 – 15:00
7	13:00 – 17:00
8	14:00 – 18:00
9	16:00 – 19:00
10	16:00 – 20:00
11	20:00 – 22:00

and many many more ...

The number of schedulings is actually exponential

(how many sets can we make out of 11 elements?)

Though most of them are not valid, still there are many valid schedulings to consider!

Example – recursive solution

We represent events using a simple `Event` class:

```
class Event:
    def __init__(self, st, et):
        self.startTime = st
        self.endTime = et
```

We put all events in an array `E` of events. We assume that `E` is sorted with respect to event start time.

So, in this example `E` looks roughly like this:

```
[(9, 12), (11, 13), (8, 14), (13, 15), (11, 16),
 (13, 17), (14, 18), (16, 19), (16, 20), (10, 21),
 (20, 22)]
```

event	times
1	8:00 – 14:00
2	9:00 – 12:00
3	10:00 – 21:00
4	11:00 – 13:00
5	11:00 – 16:00
6	13:00 – 15:00
7	13:00 – 17:00
8	14:00 – 18:00
9	16:00 – 19:00
10	16:00 – 20:00
11	20:00 – 22:00

Event scheduling: recursive solution

Here is a recursive algorithm for this problem:

- start with an empty schedule
- pick the next event ***E*** that can be scheduled and compute:
    - the max number of events we can schedule if we schedule ***E***
    - the max number if we do not schedule ***E***
- return the maximum number of events between these two cases

Recursive solution code

```
def schedule(E):
    return scheduleRec(E,0,0)

def scheduleRec(E, eventPos, startTime):
    while eventPos < len(E) and E[eventPos].startTime < startTime:
        eventPos += 1
    if eventPos == len(E):
        return 0
    withoutIt = scheduleRec(E, eventPos+1, startTime)
    withIt = 1 + scheduleRec(E, eventPos+1, E[eventPos].endTime)
    if withIt < withoutIt:
        return withoutIt
    return withIt
```

Recursive solution code explanation

```
def schedule(E):
    return scheduleRec(E,0,0)

def scheduleRec(E, eventPos, startTime):
    while eventPos < len(E) and E[eventPos].startTime < startTime:
        eventPos += 1
    if eventPos == len(E):
        return 0
    withoutIt = scheduleRec(E, eventPos+1, startTime)
    withIt = 1 + scheduleRec(E, eventPos+1, E[eventPos].endTime)
    if withIt < withoutIt:
        return withoutIt
    return withIt
```

The auxiliary method `scheduleRec(E,eventPos,startTime)` finds the maximum number of events that can be scheduled:

- starting from `startTime` and
- with `eventPos` being the position of next possible event

Increase `eventPos` by one until we find an event that can be scheduled, or we reach the end of E

If `eventPos` has reached the end of E then we are done

Return the maximum of `withIt` and `withoutIt`

Recursively compute the maximum number of events that can be scheduled:

- if we leave `E[eventPos]` out (store result in `withoutIt`)
- if we schedule `E[eventPos]` (store result in `withIt`)

Event scheduling: greedy solution

While the recursive algorithm works, it is not efficient as it can make exponentially many recursive calls.

We can do much better than that, by being **greedy**.

Idea:

at each recursive step, select the event with the earliest end time possible

That is:

- start with an empty schedule
- pick the event ***E*** that has the earliest end time of all events that can be scheduled
- return the max number of events we can schedule if we schedule ***E***

Greedy solution code

```
def scheduleGD(E):
    return scheduleRecGD(E,0,0)

def scheduleRecGD(E, eventPos, startTime):
    while eventPos < len(E) and E[eventPos].startTime < startTime:
        eventPos += 1
    if eventPos == len(E):
        return 0
    minEndPos = eventPos
    for i in range(eventPos+1, len(E)):
        if E[i].endTime < E[minEndPos].endTime:
            minEndPos = i
    return 1 + scheduleRecGD(E, minEndPos+1, E[minEndPos].endTime)
```

Greedy solution code explanation

```
def scheduleGD(E):
    return scheduleRecGD(E,0,0)

def scheduleRecGD(E, eventPos, startTime):
    while eventPos < len(E) and E[eventPos].startTime < startTime:
        eventPos += 1
    if eventPos == len(E):
        return 0
    minEndPos = eventPos
    for i in range(eventPos+1, len(E)):
        if E[i].endTime < E[minEndPos].endTime:
            minEndPos = i
    return 1 + scheduleRecGD(E, minEndPos+1, E[minEndPos].endTime)
```

Auxiliary method `scheduleRecGD(E,eventPos,startTime)` **greedily** finds max number of events that can be scheduled:
- starting from `startTime` and
- with `eventPos` being the position of next possible event

Increase `eventPos` by one until we find an event that can be scheduled, or we reach the end of E

If `eventPos` has reached the end of E then we are done

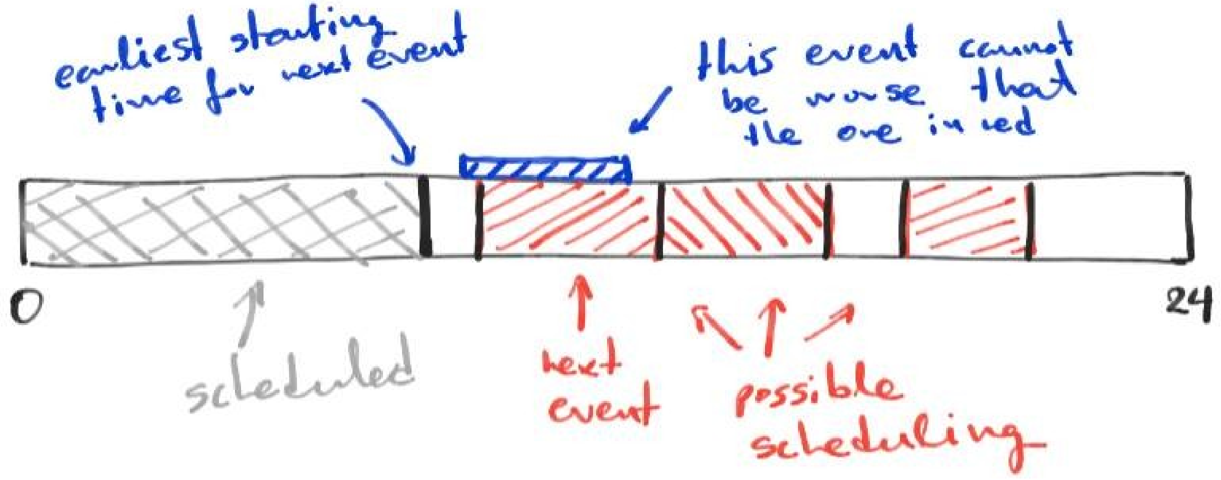
Finally, recursively compute, and return, the maximum number of events that can be scheduled if we schedule the event in `minEndPos`

Find schedulable event with **earliest end time** and store its position in `minEndPos`

Warning: greed is not always good

Our greedy solution works because:

- however we schedule our event from a given starting time,
- it is always optimal to select the event with the earliest end time possible


earliest starting time for next event
this event cannot be worse that the one in red
0
24
scheduled
next event
possible scheduling

But: greedy solutions do not work for every problem.

- we need to be cautious on what problems to use them
- when devising a greedy solution to a problem we also need to make sure (i.e. prove) that the solution is correct

Strength of greed

The strength of the algorithm is in that we do not need to search all schedulings to find an optimal one:

- rather, we make greedy choices that cut down (or ***prune***) the number of schedulings we need to look at

Pruning is a standard technique when looking at all solutions is not feasible:

- e.g. think of algorithms for playing games like chess or go
- greediness is one of the most primitive pruning techniques

Summary

The greedy approach is great for writing efficient algorithms for problems that otherwise have too many candidate solutions to check.

When applicable, the greedy approach can lead to exponential speedup.

It is applicable when there is a guarantee that being greedy pays off (i.e. leads to an optimal solution).

We therefore need to make extra effort to convince ourselves that our algorithm works.

Exercises

1. Suppose our available coins are:

   ```
   coin = [200, 100, 50, 25, 20, 10, 5, 2, 1]
   ```

   Find an amount m of money for which the greedy algorithm is not optimal, i.e. it splits m in more coins than is the least possible.

   Then, do the same using as coins:

   ```
   coin = [200, 100, 42, 20, 10, 5, 1]
   ```

2. Recall the greedy algorithm for coin splitting:

   ```
   def coinSplitGD(m):
     if m == 0:
       return 0
     for i in range(len(coin)):
       if coin[i] <= m:
         return 1 + coinSplitGD(m-coin[i])
   ```

   Modify the algorithm so that, instead of returning the number of coins in the minimal coin split, it returns an array of indices representing the coins in the split. E.g. on input `125` it should return `[1,3,5]`.

   You can use the function `append(A,k)` that returns a new array which contains the same elements as `A` but with `k` appended at the end.

   Use `coin = [200, 100, 50, 20, 10, 5, 2, 1]`.

3. Compute the time complexity, in terms of big-O notation, of the algorithms for recursive scheduling and greedy scheduling.

4. Consider the following problem. We have a bag and we want to fill it with books. The bag can take at most `w` kilos of weight, while the weights of our books are given by an array `bkWeight` (e.g. `bkWeight[0]` is the weight of the first book, etc.). Write a greedy function

   ```
   def maxBooks(w, bkWeight)
   ```

   that returns the maximum number of books that we can fill our bag with. You can assume that `bkWeight` is ordered in increasing order.

5. Think how you could solve the previous problem if `bkWeight` were not sorted.

6. Consider a variation of the previous problem where each book has a value, given by an array `bkVal` (e.g. `bkVal[0]` is the value of the first book, etc.). Write a function

   ```
   def maxBooks2(w, bkWeight, bkVal)
   ```

   which returns the maximum value of books that we can fill our bag with.

Exercises – code

```
def scheduleGD(E):
    return scheduleRecGD(E,0,0)

def scheduleRecGD(E, eventPos, startTime):
    while (eventPos < len(E)
    and E[eventPos].startTime < startTime):
        eventPos += 1
    if eventPos == len(E):
        return 0
    minEndPos = eventPos
    for i in range(eventPos+1, len(E)):
        if E[i].endTime < E[minEndPos].endTime:
            minEndPos = i
    return 1 + scheduleRecGD(E, minEndPos+1,
               E[minEndPos].endTime)
```

```
def schedule(E):
    return scheduleRec(E,0,0)


def scheduleRec(E, eventPos, startTime):
    while (eventPos < len(E)
    and E[eventPos].startTime < startTime):
        eventPos += 1
    if eventPos == len(E):
        return 0
    withoutIt = scheduleRec(E, eventPos+1,
                startTime)
    withIt = 1 + scheduleRec(E, eventPos+1,
                 E[eventPos].endTime)
    if withIt < withoutIt:
        return withoutIt
    return withIt
```

Exercises – solutions

1. For `coin = [200, 100, 50, 25, 20, 10, 5, 2, 1]` if we split 40 greedily then we need 3 coins instead of 2 that is the minimum.
   For `coin = [200, 100, 42, 20, 10, 5, 1]` if we split 45 greedily then we need 4 coins instead of 3.
2. Here is the modified algorithm:
```
def coinSplitGD2(m):
  if m == 0:
    return []
  for i in range(len(coin)):
    if coin[i] <= m:
      return append(coinSplitGD2(m-coin[i]),i)
```
3. We suppose `E` has length n and compute the running time of each algorithm. Both algorithms are based on going through `E` using `eventPos` for tracking our position.

   In the recursive algorithm, each call of `scheduleRec` makes (in the worst case) two recursive calls to itself increasing `eventPos` by 1. So:
   - with `eventPos` =0 we make 2 calls with `eventPos` =1
   - with `eventPos` =1 we make 2 calls with `eventPos` =2

   and so on. So, we have 2 calls with `eventPos` =1, and each of them makes 2 calls with `eventPos` =2. And each of them makes 2 calls with `eventPos` =3. I.e. we have exponential growth. Summing up the calls to `scheduleRec` the are:
   $$1 + 2 + 4 + 8 + 16 + \dots + 2^{n-1} = 2^n - 1.$$
   So, the recursive algorithm is $O(2^n)$.

   In the greedy algorithm, each call of `scheduleRec` makes one recursive call to itself increasing `eventPos` by 1 (in the worst case). So, we have one call to for each value of `eventPos`, which sums up to n calls overall. So, the greedy algorithm is $O(n)$.
4. We use the same idea as when scheduling, only that in this case greediness means that in each step we choose the lighter book possible. Thus:
```
def maxBooks(w, bkWeight):
  return maxBooksRec(w, bkWeight, 0)

def maxBooksRec(w, bkWeight, pos):
  if (pos == len(bkWeight)
      or w < bkWeight[pos]):
    return 0
  return 1+maxBooksRec(w-bkWeight[pos],
                       bkWeight,pos+1)
```
5. An easy solution would be to first sort `bkWeight`, e.g. using merge sort or quicksort, and then apply the algorithm above. This would be faster in general than e.g. searching each time the least element in `bkWeight` that is not in our bag yet.
6. This problem cannot be solved greedily in general as there are two conflicting factors we can be greedy with: select the lighter books or the most valuable ones. In fact, we need to check all possible ways to fill our bag.
   This is also known as the “knapsack problem”.