Have a better solution? Found a mistake? Please let us know

Award Budget Cuts

The awards committee had planned to give `n` research grants this year, out of a its total yearly budget.
However, the budget was reduced to `b` dollars. The committee members has decided to affect the minimal number of highest grants, by applying a maximum cap `c` on all grants: every grant that was planned to be higher than `c` will now be `c` dollars.
Help the committee to choose the right value of `c` that would make the total sum of grants equal to the new budget.

Given an array of grants `g` and a new budget `b`, explain and code an efficient method to find the cap `c`.
Analyze the time and space complexity of your solution.

Hints & Tips

- There can be only one possible value of `c`. Some people don't pay attention to the cap definition and use a lower cap that affect more grants. Make sure your peer is solving for the right cap as defined - one the affect the minimal number of grants.
- If the array is sorted, we can get linear complexity. To achieve a full solution, complexity of all steps besides sorting can not exceed linear runtime complexity.
- As in any array iteration, watch for wrong / negative / missed indices
- Possible hints: If you peer is stuck, offer them to form their observations about the new budget into an equation. If still stuck, ask about the variables and what they know about it and how can that be used to find the cap.
- Ask you peer about the time/space trade off regarding performing pre-computations or not, and verify your peer understands it

Solution

Assuming `g` isn't empty and `b` is indeed smaller than the sum of all grants, we need to find the cap `c`.
To compute effectively, we sort the array first. Let `r` be the index of the lowest grant to be capped, hence: `g[r-1]` ≤ `c` ≤ `g[r]`. The capped sum is: `capped`(`cap`) = `sum`(`g[0]`, `g[1]`, ..., `g[r-1]`) + (`n` - `r`) · `cap`, for such `r`.
Clearly `capped`(`c`) = `b`, and by our definition of `r`: `capped`(`g[r-1]`) ≤ `b` ≤ `capped`(`g[r]`). Once we find `r`, we can calculate `c` easily from the capped sum equation.

Linear scan for `r` is possible but not optimal. A better approach to find `r` is by a something similar to binary search. This can work because the array is sorted. We can apply the inequality `capped`(`g[r-1]`) ≤ `b` ≤ `capped`(`g[r]`) as our stopping condition for the search. For each middle index on the search we compute `capped`(`g[m]`) and `capped`(`g[m-1]`) and direct the search accordingly.

```
function findGrantsCap(g, b):
   if (g == null OR length(g) == 0):
      return 0
   n = length(g)
   partialSums = []
   tempSum = 0
   for (i from 0 to n-1):
      tempSum = tempSum + g[i]
      partialSums[i] = tempSum
   if (partialSums[n-1] <= b):
      return 0

   function cappedSum(i):
      return partialSums[i-1] + g[i]*(n-i)

   start = 0
   end   = n-1
   while (end > start):
      r = floor((end+start)/2)
      if (r > 0):
         if (cappedSum(r) > b):
            if (cappedSum(r-1) < b):
               break
            else:
               end = r - 1
         else:
            start = r + 1

   c = (b - partialSums[r-1]) / (n-r)
   return c
```

Runtime Complexity: Since we know nothing about the grants, sorting the grants takes **O(n · log n)**. Pre-computation over the grants is a linear **O(n)**. Binary search is **O(log n)** because the reminder to search in is divided by 2 at each iteration. Total complexity is a linear **O(n)** for sorted grants or **O(n · log n)** for non-sorted.

Space Complexity: **O(n)** for using another array for pre-computations. If short in memory space to store this, we avoid the pre-computation and do a linear iteration for each cappedSum calculation. This would make the space complexity **O(1)** and the runtime complexity **O(n · log n)**.