Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

30 October 2023

Algorithms & Data Structures Exercise sheet 6 HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 06 November 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Data structures.

Exercise 6.1 *Finding the i-th smallest key in an AVL tree* **(1 point)**.

Let A be an AVL tree (as described in the lecture) with n nodes. Let $k_1 < k_2 < \ldots < k_n$ be the keys of A, in ascending order. For a given $1 \leq i \leq n$, our goal is to find k_i, the i-th smallest key of A.

(a) Suppose $i = 1$. Describe an algorithm that finds k_1 in $O(\log n)$ time.

Hint: *An AVL tree is a BST (binary search tree).*

(b) Describe an algorithm that finds k_i in $O(i \cdot \log n)$ time.

Hint: *You are allowed to make changes to A while executing your algorithm.*

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of an AVL tree a bit.

(c) Modify the definition of an AVL tree by storing two additional integers $s_l(v), s_r(v) \in \mathbb{N}$ in each node v. Assuming now that A satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. Your modified definition should still allow for the search, insert and remove operations to be performed in $O(\log n)$ time, but you are not required to prove that this is the case.

Exercise 6.2 *Round and square brackets.*

A string of characters on the alphabet $\{$`A`$, \ldots,$ `Z`, `(`, `)`, `[`, `]`$\}$ is called *well-formed* if either

1. It does not contain any round or square brackets, or
2. It can be obtained from an empty string by performing a sequence of the following operations, in any order and with an arbitrary number of repetitions:
   (a) Take two non-empty well-formed strings a and b and concatenate them to obtain ab,
   (b) Take a well-formed string a and add a pair of round brackets around it to obtain `(`a`)`,
   (c) Take a well-formed string a and add a pair of square brackets around it to obtain `[`a`]`.

The above reflects the intuitive definition that all brackets in the string are 'matched' by a bracket of the same type. For example, $s =$ `FOO(BAR[A])`, is well-formed, since it is the concatenation of $s_1 =$ `FOO`, which is well-formed by 1., and $s_2 =$ `(BAR[A])`, which is also well-formed. String s_2 is well-formed because it is obtained by operation 2(b) from $s_3 =$ `BAR[A]`, which is well-formed as the concatenation of well-formed strings $s_4 =$ `BAR` (by 1.) and $s_5 =$ `[A]` (by 2(c) and 1.). String $t =$ `FOO[(BAR])` is not well-formed, since there is no way to obtain it from the above rules. Indeed, to be able to insert the only pair of square brackets according to the rules, its content $t_1 =$ `(BAR` must be well-formed, but this is impossible since t_1 contains only one bracket.

Provide an algorithm that determines whether a string of characters is well-formed. Justify briefly why your algorithm is correct, and provide a precise analysis of its complexity.

Hint: *Use a data structure from the last exercise sheet.*

Dynamic programming.

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$\begin{aligned} A_1 &= 1 \\ A_2 &= 2 \\ A_3 &= 3 \\ A_4 &= 4 \\ A_n &= A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5. \end{aligned}$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. Address the following aspects in your solution:

(1) *Definition of the DP table:* What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

(2) *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

(3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?

(4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?

(5) *Run time*: What is the run time of your solution?

Exercise 6.4 *Jumping game* **(1 point)**.

We consider the jumping game from the lecture for the following array of length $n = 10$:

$$A[1..n] = [2, 4, 2, 2, 1, 1, 1, 1, 5, 2].$$

We start at position 1. From our current position i, we may jump a distance of at most $A[i]$ forwards. Our goal is to reach the end of the array in as few jumps as possible. Recall the dynamic programming solution given for the problem in the lecture, revolving around the numbers:

$$M[k] := \text{'largest position reachable in at most } k \text{ jumps'}.$$

In this exercise, we compare two different methods for computing the $M[k]$.

(a) Consider the recursive relation:

$$\begin{aligned} &M[0] = 1, \\ &M[k] = \text{the maximum element of the array } R_k, \end{aligned} \tag{R}$$

where R_k is the array with indices i in the range $1 \leq i \leq M[k-1]$ and $R_k[i] := A[i] + i$. Compute $M[k]$ for $k = 1, 2, \ldots, K$ using relation (R), where K is the smallest integer for which $M[K] \geq n = 10$. For each $1 \leq k \leq K$, write down the array R_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R_k|$.

(b) Now consider the recursive relation:

$$\begin{aligned} &M'[0] = 1, \\ &M'[1] = 1 + A[1], \\ &M'[k] = \text{the maximum element of the array } R'_k, \end{aligned} \tag{R'}$$

were R'_k is the array with indices i in the range $M'[k-2] < i \leq M'[k-1]$ and $R'_k[i] := A[i] + i$. Compute $M'[k]$ for $k = 1, 2, \ldots, K$ using relation (R'), where K is the smallest integer for which $M'[K] \geq n = 10$. For each $2 \leq k \leq K$, write down the array R'_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R'_k|$.

(c*) Now let A be an arbitrary array of size $n \geq 2$ containing positive, non-repeating[1] integers.
Let $M[k], M'[k]$ be the numbers computed using relations (R) and (R'), respectively. Prove that $M[k] = M'[k]$ for all $k \geq 0$.

Hint: *Use induction. First show that $M[0] = M'[0]$ and that $M[1] = M'[1]$. Then, use the induction hypothesis '$M[k-2] = M'[k-2]$ and $M[k-1] = M'[k-1]$' to show that $\max R_k = \max R'_k$ for all $k \geq 2$.*

Exercise 6.5 *Longest common subsequence and edit distance.*

In this exercise, we are going to consider two examples of problems that have been discussed in the lecture.
For part (a), we are going to look at the problem of finding the longest common subsequence in two arrays. So, we are given two arrays, A of length n, and B of length m, and we want to find their longest common subsequence and its length. The subsequence does not have to be contiguous. For example, if $A = [1, 8, 5, 2, 3, 4]$ and $B = [8, 2, 5, 1, 9, 3]$, a longest common subsequence is $8, 5, 3$ and its length is 3. Notice that $8, 2, 3$ is another longest common subsequence.
For part (b), we are looking at the problem of determining the edit distance between two sequences. We

[1]This assumption is only for convenience in writing the proof.

are again given two arrays, A of length n, and B of length m. We want to find the smallest number of operations in “change”, “insert” and “remove” that are needed to transform one array into the other. If for example $A =$ [“A”, “N”, “D”] and $B =$ [“A”, “R”, “E”], then the edit distance is 2 since we can perform 2 “change” operations to transform A to B but no less than 2 operations work for transforming A into B.

(a) Given are the two arrays

$$A = [7, 6, 3, 2, 8, 4, 5, 1]$$

and

$$B = [3, 9, 10, 8, 7, 1, 2, 6, 4, 5].$$

Use the dynamic programming algorithm from the lecture to find the length of a longest common subsequence and the subsequence itself. Show all necessary tables and information you used to obtain the solution.

(b) Define the arrays

$$A = [“S”, “O”, “R”, “T”]$$

and

$$B = [“S”, “E”, “A”, “R”, “C”, “H”].$$

Use the dynamic programming algorithm from the lecture to find the edit distance between these arrays. Also determine which operations one needs to achieve this number of operations. Show all necessary tables and information you used to obtain the solution.

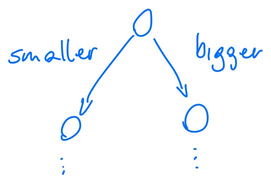
Exercise 6.1 *Finding the i-th smallest key in an AVL tree* **(1 point)**.

Let A be an AVL tree (as described in the lecture) with n nodes. Let $k_1 < k_2 < \ldots < k_n$ be the keys of A, in ascending order. For a given $1 \leq i \leq n$, our goal is to find k_i, the i-th smallest key of A.

(a) Suppose $i = 1$. Describe an algorithm that finds k_1 in $O(\log n)$ time.

Hint: *An AVL tree is a BST (binary search tree).*

Intuition :



we move left as long as possible ...

Since A is a BST, we know that for each node v with key $\text{key}(v)$, the keys of its left subtree are all smaller than $\text{key}(v)$, while the keys of its right subtree are all greater than $\text{key}(v)$. It follows that k_1 can be found by starting at the root node, and then repeatedly moving to the left child, until we arrive at a node without a left child. As A is an AVL tree, it has depth $O(\log n)$, and so this procedure takes time $O(\log n)$.

(b) Describe an algorithm that finds k_i in $O(i \cdot \log n)$ time.

Hint: *You are allowed to make changes to A while executing your algorithm.*

Using the algorithm of part (a), we can find the smallest key k_1 of A in time $O(\log n)$. Then we can remove the node with key k_1 from A in time $O(\log n)$, yielding a new AVL tree A' whose smallest key is k_2. Repeating this procedure i times yields k_i, using total time $O(i \cdot \log n)$.

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of an AVL tree a bit.

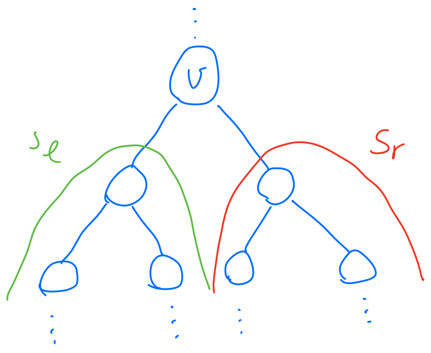
(c) Modify the definition of an AVL tree by storing two additional integers $s_l(v), s_r(v) \in \mathbb{N}$ in each node v. Assuming now that A satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. Your modified definition should still allow for the search, insert and remove operations to be performed in $O(\log n)$ time, but you are not required to prove that this is the case.

Intution: we store s_ℓ and s_r in v where

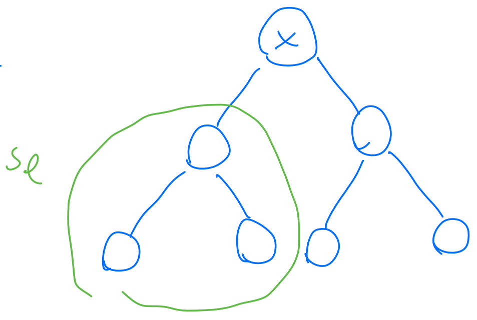
s_ℓ := # nodes in left subtree rooted at left child of v

s_ℓ := # nodes in right subtree rooted at right child of v



How does that help?

Simple example



4th smallest node?

$s_\ell(x) = 3 = 4-1$

$\Longrightarrow$ there are 3 smaller nodes than $x \Longrightarrow x$ is 4th smallest!

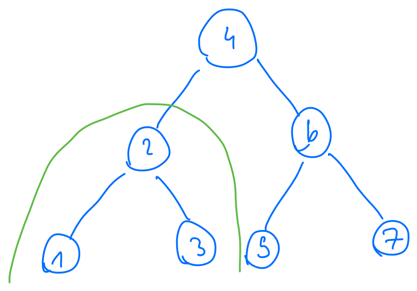
rooted at left/right child of v.

The additional integers we store are the sizes $0 \leq s_l(v), s_r(v) \leq n$ of the left and right subtree ~~rooted at v~~, respectively. (This information can be updated in time $O(1)$ during the rebalancing rotations performed during the insert and remove operations). Assuming A is modified so that each node contains these integers, our algorithm to find k_i proceeds as follows.

Let v_0 be the root node of A. Set $i_0 = i$. We want to find the i_0-th smallest key in the subtree rooted at v_0 (which is just A). We consider the following three cases:

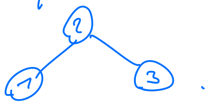
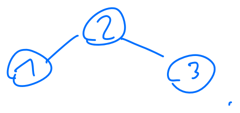
(i) If $s_l(v_0) = i_0 - 1$, we know that there are precisely $i_0 - 1$ keys in the subtree rooted in v_0 that are smaller than $\text{key}(v_0)$; namely the keys in the subtree rooted in the *left child* of v_0. But that means that $\text{key}(v_0)$ is the i_0-th smallest key in the subtree rooted in v_0, and so we output $\text{key}(v_0)$.

(ii) If $s_l(v_0) > i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *left* child $\text{lc}(v_0)$ of v_0.



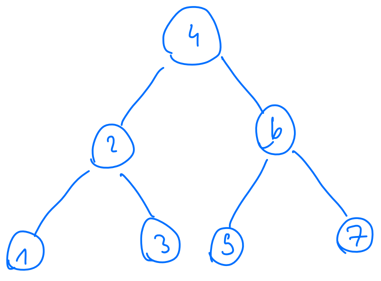
3rd smallest ? $\underline{s_l(4)} = 3 > 3-1=2.$

$\Longrightarrow$ 3rd smallest must be in the left subtree rooted at left child of 4.

we continue looking at  . Notice that the 3rd smallest in the entire tree is still 3rd smallest in  .

In case (ii), note that the i_0-th smallest key of the subtree rooted in $\text{lc}(v_0)$ is equal to the i_0-th smallest key of A (since all keys in the right subtree are too large). In this case, we can thus set $v_1 = \text{lc}(v_0)$ and $i_1 = i_0$, and apply the procedure above.

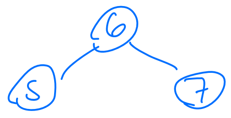
(iii) If $s_l(v_0) < i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *right* child $\mathrm{rc}(v_0)$ of v_0.



5th smallest ? $s_l(4) = 3 < 5-1$.

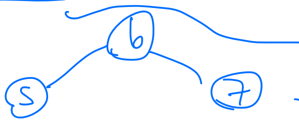
$\Longrightarrow$ 5th smallest must be in right subtree rooted at right child of 4.

we continue looking at



.

In this subtree we look for the $5-(3+1)^{(*)} = 1$ smallest, since all nodes of the $\underline{\text{left subtree}}$ and $\underline{4 \text{ itself}}$ are smaller than



. $3+1$ (*)

In case (iii), things are slightly more complicated. Note that all $s_l(v_0)$ keys in the subtree rooted at $\mathrm{lc}(v_0)$ are smaller than $\mathrm{key}(v_0)$, and that $\mathrm{key}(\mathrm{rc}(v_0)) > \mathrm{key}(v_0)$. That is to say, if we set

$$i_1 = i_0 - (s_l(v_0) + 1),$$

then the i_1-th smallest key in the subtree rooted in $v_1 = \mathrm{rc}(v_0)$ is precisely the i_0-th smallest key in the subtree rooted in v_0 (which is what we are after).

We now repeat this procedure until we reach case (i). Each repetition takes $O(1)$ time. We move one layer down in each repetition. If we ever reach a leaf, we are certainly in case (i). Therefore, the whole algorithm takes at most $O(\log n)$ time (recall that A has depth $O(\log n)$).

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$\begin{aligned} A_1 &= 1 \\ A_2 &= 2 \\ A_3 &= 3 \\ A_4 &= 4 \\ A_n &= A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5. \end{aligned}$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

Algorithm 2 $A(n)$

```
if $n \leq 4$ then
    return $n$
else
    return $A(n-1) + A(n-3) + 2A(n-4)$
```

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

Intuition:

Runtime of recursive algorithm like in previous exercises:

$$T(n) = \begin{cases} T(n-1) + T(n-3) + T(n-4), & n \geq 5 \\ 1, & \text{else} \end{cases}$$

To get to the lower bound, we "estimate downward" (nach unten abschätzen) such that $T(n) \geq \Omega(C^n)$.

$$T(n) = T(n-1) + T(n-3) + T(n-4)$$

Assume $T(n)$ is mon. increasing, then

$$T(n) \geq 3T(n-4) \geq 3^2 T(n-2\cdot 4) \geq \dots$$
$$\geq 3^k T(n-4\cdot k)$$

we choose $k \approx \frac{n}{4}$, thus

$$T(n) \geq 3^{n/4}, \quad n \geq 4$$

For $n \in \{1, \dots, 4\}$ we have $3^{n/4} > 1$

but $T(n) = 1$ thus we have to add

a constant to make it work for $n \geq 1$

$$T(n) \geq \frac{1}{3} 3^{n/4}, \quad n \geq 1$$

Notice that so far this is just an assumption, we have to prove it!

- **Base Case.**
  For $n \in \mathbb{N}$ with $n \leq 4$, we have $T(n) = 1 \geq \frac{1}{3} \cdot 3^{n/4}$ since $n/4 \leq 1$ implies $3^{n/4} \leq 3$.
- **Induction Hypothesis.**
  Assume that for some integer $k \geq 5$ the statement holds for all $k' < k$.

Notice how we formulated our IH.
We showed that for $k = 5$ all strictly positive strictly smaller integers $k' \in \{1,2,3,4\}$ we have $T(n) \geq \frac{1}{3} 3^{n/4}$.
If we now show $T(k)$ for some k, we extend the range of k' by 1, which is essentially the same as proving $k \rightsquigarrow k+1$.

- **Induction Step.**
  We compute

$$\begin{aligned}
T(k) &= T(k-1) + T(k-3) + T(k-4) + d \\
&\geq \frac{1}{3} \cdot 3^{(k-1)/4} + \frac{1}{3} \cdot 3^{(k-3)/4} + \frac{1}{3} \cdot 3^{(k-4)/4} \\
&\geq \frac{1}{3} \cdot \left(3^{(k-4)/4} + 3^{(k-4)/4} + 3^{(k-4)/4}\right) \\
&= \frac{1}{3} \cdot 3 \cdot 3^{k/4-1} \\
&= \frac{1}{3} \cdot 3^{k/4}.
\end{aligned}$$

  Thus, the statement also holds for k.
  By the principle of mathematical induction, $T(n) \geq \frac{1}{3} \cdot 3^{n/4}$ holds for every $n \in \mathbb{N}$.

Hence, the run time of the algorithm in (a) is $T(n) \geq \frac{1}{3} \cdot 3^{n/4} \geq \Omega(C^n)$ for $C = 3^{1/4} > 1$.
Remark: With a bit more care, it can be shown by induction that $T(n) = \Theta(\phi^n)$, where $\phi \approx 1.618$ is the unique positive solution of $x^4 = x^3 + x + 1$.

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

Algorithm 3 Compute A_n using memoization

```
memory ← $n$-dimensional array filled with $(-1)$s
function A_MEM(n)
    if memory[n] $\neq -1$ then                         ▷ If $A_n$ is already computed.
        return memory[n]
    if $n \leq 4$ then
        memory[n] ← $n$
        return $n$
    else
        $A_n$ ← A_Mem$(n-1)$ + A_Mem$(n-3)$ + 2A_Mem$(n-4)$
        memory[n] ← $A_n$
        return $A_n$
```

When calling A_Mem(n), each A_k for $1 \leq k \leq n$ is computed exactly once and then stored in memory. Thus the run time of A_Mem(n) is $\Theta(n)$.

Very similar to fibonacci which I showed you last week (week 6 slides).

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. Address the following aspects in your solution:

(1) *Definition of the DP table:* What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

(2) *Computation of an entry:* How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

(3) *Calculation order:* In which order can entries be computed so that values needed for each entry have been determined in previous steps?

(4) *Extracting the solution:* How can the final solution be extracted once the table has been filled?

(5) *Run time:* What is the run time of your solution?

This is mostly reading off of the pseudocode for this exercise.

(1) DP[k] stores A_k for $1 \leq k \leq n$, size is n.

(2)

$$DP[i] = \begin{cases} i, & 1 \leq i \leq 4 \\ DP[i-1] + DP[i-3] + 2\,DP[i-4], & 5 \leq i \leq n \end{cases}$$

(3) we go from smallest to largest DP entry.

(4) solution is in DP[n]

(5) all entries are computed in $\Theta(1)$ thus $\Theta(n)$.

Exercise 6.4 *Jumping game* **(1 point).**

We consider the jumping game from the lecture for the following array of length $n = 10$:

$$A[1..n] = [2, 4, 2, 2, 1, 1, 1, 1, 5, 2].$$

We start at position 1. From our current position i, we may jump a distance of at most $A[i]$ forwards. Our goal is to reach the end of the array in as few jumps as possible. Recall the dynamic programming solution given for the problem in the lecture, revolving around the numbers:

$$M[k] := \text{'largest position reachable in at most } k \text{ jumps'}.$$

In this exercise, we compare two different methods for computing the $M[k]$.

(a) Consider the recursive relation:

$$\begin{aligned} M[0] &= 1, \\ M[k] &= \text{the maximum element of the array } R_k, \end{aligned} \tag{R}$$

where R_k is the array with indices i in the range $1 \leq i \leq M[k-1]$ and $R_k[i] := A[i] + i$. Compute $M[k]$ for $k = 1, 2, \ldots, K$ using relation (R), where K is the smallest integer for which $M[K] \geq n = 10$. For each $1 \leq k \leq K$, write down the array R_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R_k|$.

$A = [2, 4, 2, 2, 1, 1, 1, 1, 5, 2]$, Ⓟ := player

$M[0]$:= "largest position reachable in ≤ 0 jumps"

$\Longrightarrow M[0] = 1$

How far can we jump from $i = 1$?

At most to $i + A[i]$.

$\Longrightarrow R[i] := A[i] + i$.

Solution:

k	$M[k]$	R_k
0	1	-
1	3	$[2+1]$
2	6	$[2+1,\ 4+2,\ 2+3]$
3	7	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6]$
4	8	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7]$
5	9	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7,\ 1+8]$
$K=6$	14	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7,\ 1+8,\ 5+9]$

So, $\sum_{k=1}^{K} |R_k| = 34$.

(b) Now consider the recursive relation:

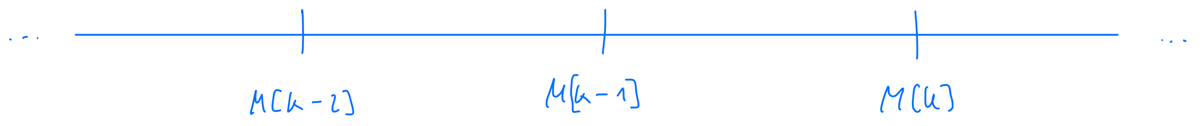
$$\begin{aligned} M'[0] &= 1, \\ M'[1] &= 1 + A[1], \\ M'[k] &= \text{the maximum element of the array } R'_k, \end{aligned} \tag{R'}$$

were R'_k is the array with indices i in the range $M'[k-2] < i \le M'[k-1]$ and $R'_k[i] := A[i] + i$. Compute $M'[k]$ for $k = 1, 2, \ldots, K$ using relation (R'), where K is the smallest integer for which $M'[K] \ge n = 10$. For each $2 \le k \le K$, write down the array R'_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R'_k|$.

At this point you should've asked yourself "Why $M'[k-2] < i \le M'[k-1]$?" and if it is still correct.

Visual representation of idea:

remember: $M[k] :=$ "largest position reachable in k jumps"



For position i we jump from to land on $M[k]$ we must have $M[k-2] < i \leq M[k-1]$.

"Proof sketch" of $M[k-2] < i \leq M[k-1]$:

If we assume A consists only of integers ≥ 1 then surely we must have $M[i-1] < M[i]$ for all $1 \leq i \leq K$.

Assume $i + A[i] = M[k]$ for some $i \leq M[k-2]$. Since $i \leq M[k-2]$ and we reach $M[k]$ in only a single jump using $A[i]$ we must have $M[k] = M[k-1]$ $\lightning$ contradiction to $M[k-1] < M[k]$.

Solution:

k	$M'[k]$	R'_k
0	1	-
1	3	-
2	6	$[4+2,\ 2+3]$
3	7	$[2+4,\ 1+5,\ 1+6]$
4	8	$[1+7]$
5	9	$[1+8]$
$K=6$	14	$[5+9]$

So, $\sum_{k=1}^{K} |R'_k| = 8$.

(c*) Now let A be an arbitrary array of size $n \geq 2$ containing positive, non-repeating[1] integers. Let $M[k], M'[k]$ be the numbers computed using relations (R) and (R'), respectively. Prove that $M[k] = M'[k]$ for all $k \geq 0$.

Hint: *Use induction. First show that* $M[0] = M'[0]$ *and that* $M[1] = M'[1]$. *Then, use the induction hypothesis '*$M[k-2] = M'[k-2]$ *and* $M[k-1] = M'[k-1]$*' to show that* $\max R_k = \max R'_k$ *for all* $k \geq 2$.

IB:

$M[0] = 1 = M'[0]$ and

$M[1] = \max \{ i + A[i] \mid 1 \leq i \leq M[0] \}$

$= \max \{ i + A[i] \mid i = 1 \} = 1 + A[1] = M'[1].$

IH: Assume $M[i] = M'[i]$ for $0 \leq i \leq k$, for some $k \in \mathbb{N}$.

IS: $k \rightsquigarrow k+1$

We show $\max R_{k+1} = \max R'_{k+1}$.

Since $R_{k+1} = \{ i + A[i] \mid 1 \leq i \leq M[k] \}$

and $R'_{k+1} = \{ i + A[i] \mid M'[k-1] < i \leq M'[k] \}$

we clearly have $R'_{k+1} \subseteq R_{k+1}$ and

therefore $\max R'_{k+1} \leq \max R_{k+1}$

It remains to show that

$\max R'_{k+1} \not< \max R_{k+1}$

(c*) Now let A be an arbitrary array of size $n \geq 2$ containing positive, non-repeating[1] integers. Let $M[k], M'[k]$ be the numbers computed using relations (R) and (R'), respectively. Prove that $M[k] = M'[k]$ for all $k \geq 0$.

Hint: *Use induction. First show that $M[0] = M'[0]$ and that $M[1] = M'[1]$. Then, use the induction hypothesis '$M[k-2] = M'[k-2]$ and $M[k-1] = M'[k-1]$' to show that* $\max R_k = \max R'_k$ *for all $k \geq 2$.*

IB:

$M[0] = 1 = M'[0]$ and

$M[1] = \max \{ i + A[i] \mid 1 \leq i \leq M[0] \}$

$= \max \{ i + A[i] \mid i = 1 \} = 1 + A[1] = M'[1]$.

IH: Assume $M[i] = M'[i]$ for $0 \leq i \leq k$, for some $k \in \mathbb{N}$.

IS: $k \rightsquigarrow k+1$

We show $\max R_{k+1} = \max R'_{k+1}$.

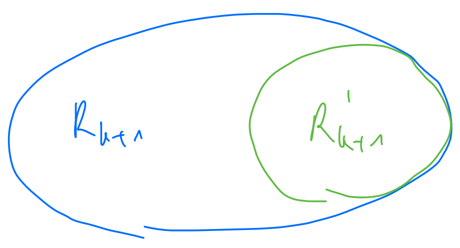
Since $R_{k+1} = \{ i + A[i] \mid 1 \leq i \leq M[k] \}$

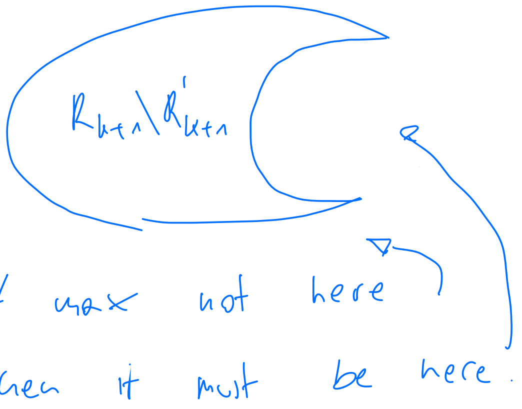
and $R'_{k+1} = \{ i + A[i] \mid M'[k-1] < i \leq M'[k] \}$

Clearly, we have $R'_{k+1} \subseteq R_{k+1}$.

If we have that $\max R_{k+1} \notin R_{k+1} \setminus R'_{k+1}$

then $\max R_{k+1} = \max R'_{k+1}$ must hold





Now notice that

$$R_{k+1} \setminus R'_{k+1} = \{A[1]+i \mid 1 \le i \le M[k-1]\} = R_k$$

but then

$$\max R_{k+1} \setminus R'_{k+1} = \max R_k = M[k] \overset{\circledast}{<} M[k+1]$$

where we used the fact that A only contains numbers > 0.

Therefore

$$\begin{aligned} M[k+1] = \max R_{k+1} &= \max R_{k+1} \cap R'_{k+1} \\ &= \max R'_{k+1} \\ &= M'[k+1] . \end{aligned}$$