CSCI2100: Regular Exercise Set 1

Prepared by Yufei Tao

Problem 1. Let x be a real value. Define $\lfloor x \rfloor$ to be the largest integer that does not exceed x. For example, $\lfloor 2.5 \rfloor = 2$, whereas $\lfloor 3 \rfloor = 3$.

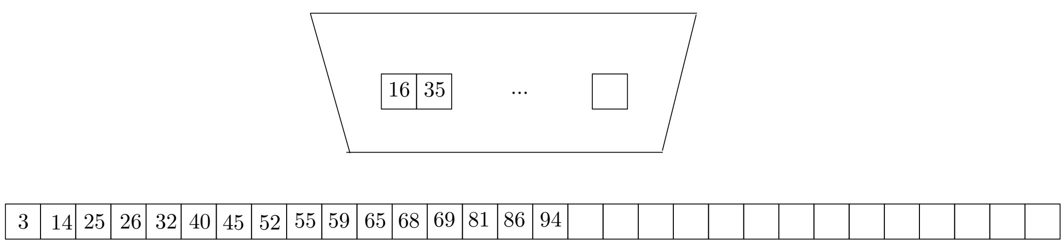
Suppose that you are given an integer $n \geq 2$ in (a register of) the CPU. Write an algorithm to compute the value of $\lfloor \log_2 n \rfloor$ in no more than $100 \log_2 n$ time.

Solution. We will generate $2^1, 2^2, 2^3, ...$ until finding the smallest i such that $2^i > n$. Initially, set $i = 1$ and $p = 2$. Repeat the following until $p > n$:

- Increase i by 1.
- Increase p by a factor of 2.

Finally, return $i - 1$ as the answer. It is clear that i is increased no more than $1 + \log_2 n$ times. For each value of i, performing the aforementioned steps can be easily implemented in less than 10 atomic operations. The total cost is therefore at most $10(1 + \log_2 n)$ which is less than $100 \log_2 n$ for all $n \geq 2$.

Problem 2. The following figure shows an input to the dictionary search problem.



Describe how binary search works using the input.

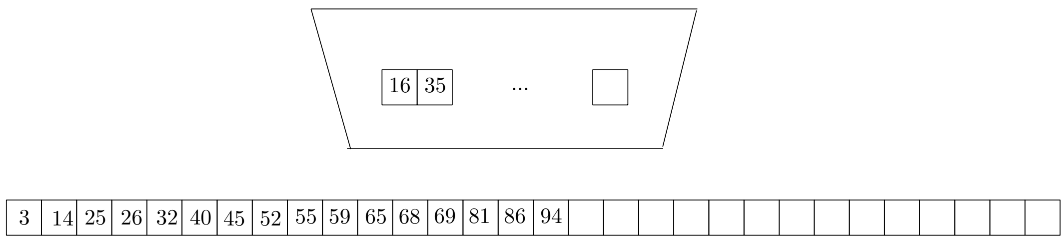
Solution. First, compare 35 to the 8th element 52 in memory. Since $35 < 52$, we can now focus on the first 7 elements. Next, compare 35 to the 4th element 26. Since $35 > 26$, we only need to focus on the 5th and 6th elements. Then, the algorithm compares 35 first to the 5th, and then the 6th, before returning the answer "no".

<u>Remark.</u> Binary search looks for the "middle element" among the subset of elements that are still in consideration. However, what it means by the "middle" can be subjective. In the above, we adopted the following convention. Suppose that t elements still remain. If t is an odd number, then the middle element is the one that stands strictly in the middle (such as 26 in the above example). If t is even, then the middle element is the $(t/2)$-th (such as 52 in the above example). This is—purposely—different from what was shown in the lecture notes. In quizzes/exams, you are free to adopt any conventions.

Problem 3 (Predecessor Search). Let us first define the notion of *predecessor*. Let S be a set of integers. Given an integer v, the *predecessor* of v in S is the largest integer in S that is at most

v. For example, suppose $S = \{3, 14, 15, 26, 32, 40\}$. The predecessor of 25 is 15, while that of 26 is 26.

Consider the following problem. You are given a set S of n integers, which are stored at memory cells 1, 2, ..., n in ascending order. The value of n is given in the CPU, and so is an integer v. The following shows an example with $n = 16$ and $v = 35$.

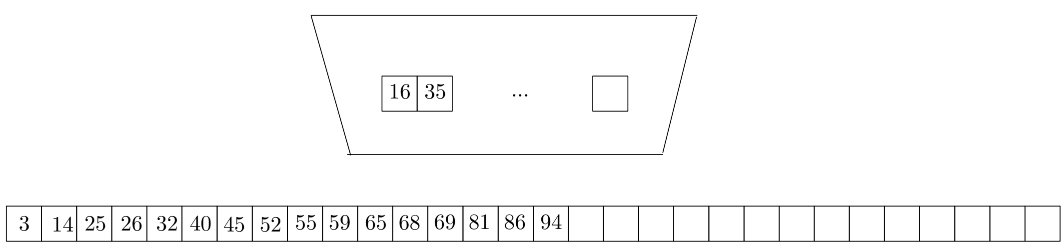


Describe an algorithm to find the predecessor of v. Your algorithm should have running time at most $100 + 100 \log_2 n$.

Solution. First perform binary search on S using v. Let x be the last element that the algorithm compared v to. If $x = v$, then x is the predecessor of v. If $x < v$, then x is the predecessor of v. Finally, if $x > v$, then the predecessor of v is the element immediately before x—in the special case where x is already the smallest element in S, then v has no predecessor in S.

Binary search takes no more than $7 \log_2 n$ atomic operations. Clearly, the algorithm finishes in less than 10 atomic operations after binary search. The overall cost is therefore less than $10 + 7 \log_2 n < 100 \log_2 n$.

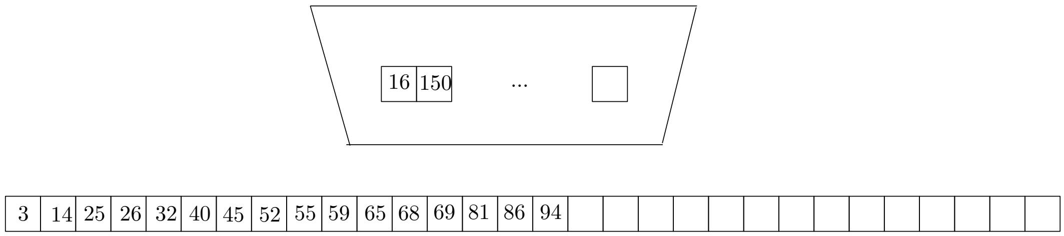
Problem 4 (Prefix Counting). Consider the following problem. You are given a set S of n integers, which are stored at memory cells 1, 2, ..., n in ascending order. The value of n is given in the CPU, and so is an integer v. The following shows an example with $n = 16$ and $v = 35$.



Describe an algorithm to find the number of integers in S that are at most v. In the above example, for instance, you should return 5. Your algorithm should have running time at most $100 + 100 \log_2 n$.

Solution. Notice that our predecessor search algorithm in Problem 4 not only finds the predecessor x of v, but also the address a of the memory cell where x is stored. To solve the prefix counting problem, first find x. If x does not exist, return 0. Otherwise, return a. The cost is clearly no more than $100 \log_2 n$.

Problem 5 (The 3-Sum Problem). Consider the following problem. The input S consists of n integers, which are given at memory cells 1, 2, ..., n, arranged in ascending order. The value of n is given in the CPU. So is a value v. The following shows an example with $n = 16$ and $v = 150$.



Describe an algorithm to determine whether S has 3 numbers that sum up to v. In the above example, the answer is "yes" because $150 = 40 + 45 + 65$. Your algorithm should have running time at most $100 + 100 \cdot n^2 \log_2 n$.

Solution. The tutorial slides described how to solve the following "2-sum" problem. There, we are given a set S of n integers as in the 3-Sum problem, the value of n, and also an integer v. The goal is to determine whether S contains two numbers that add up to v. The tutorial slides presented two algorithms for solving this problem. The first one has running time less than $7n \log_2 n$.

Observe that we can settle the 3-sum problem by solving n instances of the 2-sum problem. Specifically, for each $i \in [1, n]$, let x be the i-th integer from S. We invoke an algorithm for solving the 2-sum problem where we ask whether S contains 2 elements (other than x) that sum up to $v - x$. If so, then we return "yes" immediately to the 3-sum problem. Otherwise, we attempt the next i. After all the values of i have been attempted, we return "no".

It is easy to implement the above strategy in cost less than $100 \cdot n^2 \log_2 n$.