1st tutorial

Jamie Fravel
jfravel@iuuk.mff.cuni.cz

1. Missing Number We are given an unsorted array A which contains n pairwise different numbers from set $\{0, \ldots, n\}$. One number from this set is missing in A. Describe an efficient algorithm that finds the missing number. Try to describe an algorithm that uses only a constant amount of additional memory.

Proposition 1.1 (Gauss's sum)**.** $\sum_{i=1}^{n} i = \frac{n(n-1)}{2}$

The missing number m can be derived from Gauss's sum. In particular,

$$m = \frac{n(n-1)}{2} - \sum_{i=1}^{n} A[i].$$

The sum requires n operations but, by doing it in-place, takes only one variable (constant memory).

2. Sorted Array Fixed Point We are a given an array A of n integers that are sorted in increasing order. The range of the integers is not bounded in this case, in particular, numbers can be negative. Describe an efficient algorithm which finds an index i such that $A[i] = i$.

Bonus. Does your approach need to change if A contains some duplicate entries?

Consider the array $B = \big[A[i] - i\big]_{i=1}^{n}$. What are we looking for in this array?

Bonus. Notice that standard binary search fails on $A = [1, 1, 4, 6, 6]$. Then $B = [0, -1, 1, 2, 1]$.
Proposition 1.2. *For an index j, let $m = A[j]$. If $j < m$, then $i \notin (m, j]$.*

Without duplicate entries, A is strictly increasing and standard binary search works just fine.

Bonus. With duplicate entries, we can only prune the entries between i and $A[i]$, not the entire half-array. This ends up giving us a worst case complexity $\mathcal{O}(n)$.

3. Egg testing The Empire State Building is a 102-story skyscraper, and we know that if we throw an egg from the K-th floor or higher, it will break. Unless the egg breaks, it can be collected and reused. We want to determine K, but use as few attempts (throws) as possible. What's the best strategy (minimizing the number of throws) if we have

a) one egg,

b) unlimited eggs,

c) two eggs,

d) **Bonus.** three eggs, or in general $e \in \mathbb{N}$ eggs?

(The eggs can be special, so nothing can be *a priori* assumed about K.)

Binary search is efficient because it rules out large chunks of the array at every step. But this can't work with only two eggs... right?

Suppose that we throw the first egg from floors m, $2m$, $3m$ etc. How many throws do we use in the worst case?

a) With only one egg, we are forced to perform a linear search. We can throw the egg from each floor in increasing order. This requires $K \leq N$ attempts.

b) With an unlimited number of eggs, we can perform a binary search! Throw from the middle floor, disregard half of the floors and repeat until k is found.

c) With two eggs, we cannot complete a binary search, but the concept is still useful. If the first egg is used to rule out some set of the floors, then the problem reduces to the one egg case on a smaller

building.

Binary search searches the midpoint because it never needs to reduce to a linear search. In this two-stage search we may be better served by a small second stage (by skipping a smaller number of floors). Suppose we skip m floors at a time; in the worst case we will need $\frac{102}{m} + m$ throws. The minimum of this function occurs at $m = \sqrt{102}$ giving $2\sqrt{102}$ throws. If we had n floors, then m would need to be $\sqrt{n}$.

d) **Bonus.** With three eggs, we can do add a second skip-stage to the above scheme. After the first egg, we will have the two-egg case on a smaller building! Say we skip m_1 then m_2 floors with the first then second egg respectively; the worst case is now $\frac{102}{m_1} + \frac{m_1}{m_2} + m_2$ which is minimized at $m_1 = n^{\frac{2}{3}}$ and $m_2 = n^{\frac{1}{3}}$. Notice that this lines up with the two egg case ($m_2 = \sqrt{m_1}$) and uses at most $3n^{\frac{1}{3}}$ throws. Following this formula up to e eggs gives us $\Theta(n^{\frac{1}{e}})$ many throws.

4. Laser There is a row of N buildings with $h_1, \ldots, h_n$ floors, and we need to demolish all of them. To that end, you found at home a demolition laser which is capable of firing *vertically* to destroy an arbitrary building or *horizontally* to destroy a given floor in all buildings (i.e., if you choose to destroy floor L, then the number of floors decreases by 1 for all buildings whose number of floors is $\geq L$). Develop an algorithm to determine minimum number of firings necessary to eliminate all buildings? (Beware that the maximum number of floors can be much more than N.)

Assume that every vertical firing targets the tallest remaining building and that every horizontal firing targets the ground floor. Clearly, there exists an optimal firing strategy which follows these rules. Now, let u and v represent the number of horizontal and vertical firings respectively. Define $f(v)$ to be the number of floors in the v^{th} tallest building. If h is sorted, then $f(v) = h_{n-v}$.

Proposition 1.3. *An optimal firing strategy will minimize $f(v) + v$.*

Algorithm `vParamLaser`(h)

$h \leftarrow \texttt{sort}(h)$
$v^* \leftarrow 0, \quad f^* \leftarrow 0$
for $v \in [1 : n]$ **do**
  if $h_{n-v} + v < f^* + v^*$ **then**
    $v^* \leftarrow v, \quad f^* \leftarrow h_{n-v}$
  end if
end for
return $\big(f(v), v\big)$

Algorithm `uParamLaser`(h)

$h \leftarrow \texttt{sort}(h)$
$i \leftarrow 1, \quad g \leftarrow 0$
$u^* \leftarrow 0, \quad g^* \leftarrow 0$
for $u \in [1 : h_n]$ **do**
  while $h_i \leq u$ **do**
    $i \leftarrow i + 1, \quad g \leftarrow g + 1$
  end while
  if $u + g < u^* + g^*$ **then**
    $v^* \leftarrow v, \quad f^* \leftarrow h_{n-v}$
  end if
end for
return $\big(u, g(u)\big)$

Proposition 1.4. *The algorithm* `vParamLaser` *returns a minimal firing pattern in $\mathcal{O}(n)$ time.*

Proof. The algorithm has a sort call (counting sort is linear time) and one loop containing only constant time operations. Said loop will run n times because there are only n buildings to destroy. Thus, $\mathcal{O}(n)$. Knowing that $v = n$ destroys all buildings, we loop over every relevant instance, saving the minimum-known strategy at each step. □

Interestingly, following similar logic while parameterized on u rather than v results in a polynomial algorithm. This is because, unlike v, we don't know exactly how many buildings get destroyed as we increase h. It could be none. It could be all n. We end up spending the extra time computing $g(u)$ even if h is sorted. See `uParamLaser`.

5. Submatrix search We are given an integer matrix A with n columns and m rows. Describe an efficient algorithm that finds a maximum submatrix of A consisting only of values 0 (i.e. the submatrix with largest area). Can you achieve complexity $O(nm)$?

Each row j has at most n intervals (i_1, i_2) of zero.

The first idea is to check all top left corners (i_1, j_1) and all bottom right corners (i_2, j_2) and check if a submatrix given by these corners contains all zeros, then pick the largest such submatrix. This would lead to an algorithm of complexity $\Theta(n^3m^3)$.

A better idea is to consider, for every $j = 1, \ldots, m$, the intervals (i_1, i_2) of zeroes on row j. There is only at most n of these intervals. Then, we try to extend this interval down to find a maximal $j_2 > j_1$ such that submatrix with top left corner (i_1, j_1) and bottom right corner (i_2, j_2) is full of zeros. This takes only time $O(n^2m^2)$ altogether.

The previous idea can be improved. For every cell of the matrix at coordinates (i, j), we consider number $c(i, j)$ that denotes the number of zeros to the right of $A[i, j]$ before some nonzero value (i.e. max k such that $A[i + t, j] = 0$ for $t = 0, \ldots, k$). Then in the above approach, j_2 can be found in linear time. By symmetry, this gives an algorithm of overall complexity $O(nm \min m, n)$.

If we also precompute the number of zeros above, below and to the left of each cell, we can further optimize the above approach to get an $O(nm)$ time algorithm.

6. Fibonacci sequence Fibonacci sequence is defined using $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

a) What is the value of F_n for a given n?

b) Describe an algorithm for computing F_n using only $\Theta(\log_2 n)$ arithmetic operations.

a) Suppose that F_n takes the form r^n. What does r need to be? Does this work for $n = 0$ or $n = 1$?

b) Try representing the recursion in the form of matrix multiplication.

a) For $n \geq 0$ we have that

$$F_n = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right).$$

Asymptotically, $F_n = \Theta(\varphi^n)$ where $\varphi = \frac{1+\sqrt{5}}{2}$ is the Golden ratio.

Unfortunately $\sqrt{5}$ is irrational, so actually computing this value is not possible on any modern computer. Hence part b.

b) The Fibonacci recursion can be thought of a shifting window of three values, F_n, F_{n-1}, F_{n-2}. Shifting the window can be interpreted as a linear transformation and thus as a matrix multiplication:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

Since matrix multiplication is an associative operation, we have

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

We can compute exponents (including matrix exponents) in logarithmic time by repeated squaring according to the binary representation of n.