Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer

29. October 2018

Algorithms & Data Structures — Homework 6 — HS 18

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

Points: _____

Exercise 6.1 *iPhone Drop Test* ***(1 Point for Part 4)****.*

You just got a new job at Apple in the department of destructive testing. The first task given is to test the endurance of the new iPhone XR series. Specifically you need to determine the highest floor that the new iPhone can withstand when dropped out of the window.

When the phone is dropped and does not break, it is undamaged and can be dropped again. For simplicity assume that subsequent drops of the phone do not affect its endurance (i.e. if it survives it will have the identical state as if it weren't dropped at all). However, once the iPhone has been broken, you can no longer use it for another test.

If the phone breaks when dropped from floor n, then it would also have broken from any floor above that. If the phone survives a fall, then it will survive any fall below that.

As this is your first responsibility at your new job, you want to impress your new boss, and deliver results as soon as possible. To achieve that, you devise a strategy to minimize the number of drop tests required to find the solution.

1. What strategy would you use if only one phone is given and you perform the drop test on a building with n floors? What are the maximum number of drop tests that you have to perform?

   Solution: If we have one phone on disposal, we really have no other choice but to start at floor 1. If it survives, great, we go up to floor 2 and try again, then floor 3 . . . all the way up the building; one floor at a time. As a result, it will take us n trials in the worst case scenario.

2. What if we are given unlimited amount of identical phones?

   Solution: In that case scenario, we can start at floor $n_1 = \lceil \frac{n}{2} \rceil$ ($\lceil \cdot \rceil$ rounds up). If the phone breaks, we repeat this procedure with the block of floors $1, \ldots, n_1 - 1$, if not, we repeat this procedure with the block of floors $n_1 + 1, \ldots, n$, essentially performing a binary search. Assuming worse case scenario, it will take us $\lfloor \log_2(n) \rfloor + 1$ trials.

3. What if we are given exactly 2 identical phones and the number of floors n is fixed such that $n = 100$?.

   Solution: One option to do this would be to drop the first phone at the 50th floor. If the phone survives, we would need another 50 trials until we reach the top. This is improvement to the *strategy of having one phone*, however, it is not optimal.

Another strategy would be to drop the phone every 10th floor until it breaks. The moment it breaks, we use the *strategy of having one phone* for the 9 floors below the one where the first one did break. Therefore, if the phone could survive a fall on the 99th floor, we would need 19 drops to determine that. Which is also not optimal.

We can do even better. What we need is a solution that minimizes our maximum regret. Imagine we drop our first phone from floor d, if it breaks, we can step through the previous $(d-1)$ floors one-by-one. If it doesn't break, rather than jumping up another d floors, instead we should step up just $(d-1)$ floors (because we have one less drop available if we have to switch to one-by-one floors), so the next floor we should try is floor $d + (d-1)$.

Similarly, if this drop does not break, we next need to jump up to floor $d+(d-1)+(d-2)$, then floor $d + (d-1) + (d-2) + (d-3) \ldots$ We keep reducing the step by one each time we jump up, until that step-up is just one floor, and get the following equation for a 100 floor building:

$$d + (d-1) + (d-2) + (d-3) + (d-4) + ... + 1 = 100,$$

or in other words: $d \cdot (d+1)/2 = 100$. Therefore, the optimal d would be the solution to the equation, i.e. $d = 13.651$, which we round up to 14. Therefore 14 drops:

Drop	1	2	3	4	5	6	7	8	9	10	11	12
Floor	14	27	39	50	60	69	77	84	90	95	99	100

4. Assume that you are given 3 identical phones and a building with n floors. Determine the best search strategy for the floor where the phone breaks and give the number of drops in big-Θ notation. There is no need to prove that is best but only asymptotically optimal algorithms count.

   Solution: We can divide the building into $\Theta(\sqrt[3]{n})$ blocks (sequences of consecutive floors) of size $\mathcal{O}(\sqrt[3]{n^2})$. Starting from the first block we drop the first phone from the last floor of each block until the phone breaks. If the phone didn't break after drop from the n-th floor, the lowest floor where the phone breaks is higher than n (so we cannot determine it using a building with n floors). If the phone broke after drop from the last floor of some block, the correct floor (lowest floor where the phone breaks) is in this block.

   Then we can divide this block of size $b = \mathcal{O}(\sqrt[3]{n^2})$ into $\Theta(\sqrt{b}) = \mathcal{O}(\sqrt[3]{n})$ sub-blocks of size $\mathcal{O}(\sqrt{b}) = \mathcal{O}(\sqrt[3]{n})$. Starting from the first sub-block we drop the second phone from the last floor of each sub-block until the phone breaks. If the phone broke after drop from the last floor of some sub-block, the correct floor is in this sub-block.

   Then, starting from the first floor of this sub-block we drop the third phone from each floor of this sub-block until the phone breaks, and correctly determine the floor. Since we have $\Theta(\sqrt[3]{n})$ blocks, $\mathcal{O}(\sqrt[3]{n})$ sub-blocks in each block and $\mathcal{O}(\sqrt[3]{n})$ floors in each sub-block, the number of drops is $\Theta(\sqrt[3]{n}) + \mathcal{O}(\sqrt[3]{n}) + \mathcal{O}(\sqrt[3]{n}) = \Theta(\sqrt[3]{n})$.

   The precise number of drops can be obtained as follows:

   Let's assume that we would need d drops to cover the whole building. And then let's devise a function $f_p(d)$ that calculates how many floors we can cover, where p represents the number of available phones, and d represents the number of drops.

   Let's start by dropping one phone. If it breaks, we will explore the lower $f_2(d-1)$ floors; if it survives, we will explore the upper $f_3(d-1)$ floors. Consequently we have:

$$f_3(d) = 1 + f_2(d-1) + f_3(d-1),$$

and from the previous part, we know that $f_2(d) = \frac{d\cdot(d+1)}{2}$, thus:

$$f_3(d) = 1 + \frac{d\cdot(d-1)}{2} + f_3(d-1) = \sum_{i=3}^{d} 1 + \frac{1}{2}\sum_{i=3}^{d} i^2 - \frac{1}{2}\sum_{i=3}^{d} i + f_2(2) = \frac{d\cdot(d^2+5)}{6}.$$

Assuming n floors, the number of floors covered must be greater or equal than n:

$$\frac{d\cdot(d^2+5)}{6} \geq n.$$

As a result, the number of drops will be the solution of the cubic equation defined above, creating asymptotic number of drops for the search is $\Theta(\sqrt[3]{n})$.

Exercise 6.2 *Simple sorting.*

1. Perform two iterations of Bubble Sort on the following array. The array has already been partially sorted by previous iterations (after the double bar). By iterations we mean iterations of outer loop. You should only write two arrays corresponding to the end of first and second iterations.

9	5	8	13	15	10	11	7	6 ‖	20	21	35
1	2	3	4	5	6	7	8	9	10	11	12

2. Perform two iterations of Selection Sort on the following array. The array has already been partially sorted by previous iterations (up to the double bar). By iterations we mean iterations of outer loop. You should only write two arrays corresponding to the end of first and second iterations.

2	3	5	6 ‖	15	17	22	8	16	12	13	10
1	2	3	4	5	6	7	8	9	10	11	12

Solution:

1. First itetation:

5	8	9	13	10	11	7	6 ‖	15	20	21	35
1	2	3	4	5	6	7	8	9	10	11	12

   Second iteration:

5	8	9	10	11	7	6 ‖	13	15	20	21	35
1	2	3	4	5	6	7	8	9	10	11	12

2. First iteration:

2	3	5	6	8 ‖	17	22	15	16	12	13	10
1	2	3	4	5	6	7	8	9	10	11	12

   Second iteration:

2	3	5	6	8	10 ‖	22	15	16	12	13	17
1	2	3	4	5	6	7	8	9	10	11	12

Exercise 6.3 *Inverse questions.*

1. Give a sequence of 5 numbers for which Bubble Sort performs exactly 10 swaps of keys in order to sort the sequence.
2. For all $n > 1$ give a sequence of n numbers for which Bubble Sort performs $\Theta(n\sqrt{n})$ swaps of keys in order to sort the sequence.
3. Assume that Selection Sort does not swap elements with the same index. For all $n > 1$ give a sequence of n numbers for which Selection Sort performs exactly 1 swap of keys in order to sort the sequence, but Bubble Sort and Insertion Sort perform at least $\Omega(n)$ swaps of keys.
4. For all $n > 1$ give a sequence of n numbers for which Bubble Sort, Selection Sort and Insertion Sort perform $\Theta(n)$ swaps of keys in order to sort the sequence.

Solution:

1. For example, a sequence $5, 4, 3, 2, 1$.
2. For example, a sequence

$$m, m-1, m-2, \ldots, 3, 2, 1, m+1, m+2 \ldots, n-1, n,$$

   where $m = \lfloor \sqrt{n\sqrt{n}} \rfloor$.
3. For example, a sequence

$$n, 2, 3, 4, \ldots, n-1, 1.$$

4. For example, if n is even, a sequence

$$2, 1, 4, 3, 6, 5, \ldots, n, n-1,$$

   and if n is odd, a sequence

$$2, 1, 4, 3, 6, 5, \ldots, n-1, n-2, n.$$

Exercise 6.4 *Loop invariant **(1 Point)**.*

Consider the pseudocode of the MaxSubarraySum algorithm on an integer array $a[0, \ldots, n-1]$, $n \geq 1$.

```
procedure MaxSubarraySum(a)
    randmax ← 0
    max ← 0
    for 0 ≤ i < n do
        randmax ← randmax + a[i]
        if randmax > max then
            max ← randmax
        if randmax < 0 then
            randmax ← 0
    return max
```

Find a loop invariant INV such that:

1. INV(0) holds before the execution of the loop.
2. If INV(i) holds at the beginning of a loop iteration, then INV$(i+1)$ holds at the end of the loop iteration. Prove this.
3. INV(n) implies the correct solution.

Solution: INV(i) is a following statement: At the beginning of the i-th loop iteration:

- max is a maximal subarray sum of the array $a[0,\dots,i-1]$, if $i=0$, max is 0.
- randmax is a maximal nonnegative sum of subarrays with end index $i-1$, that is,

$$\text{randmax} = \max\{\max_{0\le j<i}\sum_{k=j}^{i-1} a[k], 0\},$$

  if $i=0$, randmax is 0.

1. INV(0) holds before the execution of the loop, since max $=0$ and randmax $=0$ in this case.
2. Assume that INV(i) holds at the beginning of the loop. At first, consider the case $i>0$.
   - At the end of the loop randmax is

$$\max\{\max\{\max_{0\le j<i}\sum_{k=j}^{i-1} a[k],0\}+a[i],0\} = \max\{\max_{0\le j<i}\sum_{k=j}^{i-1} a[k]+a[i], a[i], 0\} =$$
$$= \max\{\max_{0\le j\le i}\sum_{k=j}^{i} a[k], 0\}.$$

   - At the end of the loop max is

$$\max\{\text{max subarray sum of } a[0,\dots,i-1], \max\{\max_{0\le j<i}\sum_{k=j}^{i-1} a[k],0\}+a[i]\} =$$
$$= \max\{\text{max subarray sum of } a[0,\dots,i-1], \max_{0\le j<i}\sum_{k=j}^{i} a[k], a[i]\} =$$
$$= \max\{\text{max subarray sum of } a[0,\dots,i-1], \max_{0\le j\le i}\sum_{k=j}^{i} a[k]\}.$$

     Since any subarray of $a[0,\dots,i]$ either is a subarray of $a[0,\dots,i-1]$ or has end index i, this value is equal to a maximal subarray sum of the array $a[0,\dots,i]$.

   Hence INV($i+1$) holds at the end of the loop.

   If $i=0$, then at the end of the loop max and randmax are both $\max\{0,a[0]\}$, hence in this case INV($i+1$) also holds at the end of the loop.

3. INV(n) implies that at the end of the last loop iteration max is a maximal subarray sum of the array $a[0,\dots,n-1]=a$, which is the correct solution.

Exercise 6.5 *TCP: Determine the maximum bandwidth* ***(1 Point)****.*

When transferring a large file over the internet, you want the file to arrive as fast as possible at the receiver. For this, the TCP protocol must determine the maximum bandwidth (e.g., measured in number of characters per second) which is available between sender and receiver. The available bandwidth is in general unlimited, time-dependent, and different for each transmitter-receiver pair. In this exercise, the task is to design a procedure (an algorithm) that is as efficient as possible to determine the available bandwidth.

For simplicity, we assume that during a connection the available bandwidth remains constant. The TCP protocol sends the data in each time unit with a bandwidth selected by the server. If the actual available bandwidth is sufficient (i.e., higher than the selected one), then the data arrives at the receiver. The receiver sends in this case indirectly before the end of the unit of time a confirmation back (the so-called Acknowledgement). If the bandwidth selected by the server has exceeded the available bandwidth, then the data sent in this time unit will be lost. The server detects this case by not receiving an acknowledgment from the receiver at the end of the time unit. So in each time unit one bandwidth can be tested by the server.

Design a procedure whereby the TCP protocol at the server determines the available bandwidth in as few time units as possible. What is the asymptotic number $O(f(b))$ of time units needed to compute the bandwidth b? (Assume that $b > 0$ is an integer.) Is your algorithm asymptotically optimal?

Solution: The idea of the procedure is as follows: the server starts with bandwidth 1 at time 1 (which we assume to always be available). After that TCP doubles the bandwidth in each time step until no Acknowledgment is received anymore. If this happens at timestep k, this means that the available bandwidth b is between 2^{k-2} and 2^{k-1} (since the server recieved an acknowledment at time step $k-1$ for bandwidth 2^{k-2}).

In this interval, we then use binary search: in the first step, the protocol tries the bandwidth $m = \frac{2^{k-2}+2^{k-1}}{2}$. If the bandwidth is sufficient, the binary search continues in the interval $[m, 2^{k-1}]$. If the server does not get an Ackonwledgement for bandwidth m, then the search continues in the interval $[2^{k-2}, m]$. The procedure terminates if the interval contains exactly one element.

Let b be the bandwidth we are looking for. Then the doubling phase takes maximum $\log(b) + 1$ steps. The binary search is then, in the worst case, on the interval $[b, 2b]$, and thus contains b elements. Again, a maximum of $\log(b)$ steps are needed until we find b. It follows that this procedure as a whole requires $\mathcal{O}(\log(b))$ time units to determine b.

To see why this is optimal, think of the following: our taks is more difficult than the task to search for b in a sorted range containing b elements (which is exactly the second subtask we solve). We know from the lecture, that already this search has a lower bound of $\log(b)$ time steps.

Submission: On Monday, 06.11.2018, hand in your solution to your TA *before* the exercise class starts.