INTRO TO ALGORITHMS, COMP 160, Fall 2015, HOMEWORK 4

- Deadline: Thursday, Oct. 22
- Worth 2% of the final grade.
- Each problem will be graded out of 100 points.

1. We have seen two heap-build methods in class.

   (a) For each method, give one reason why we would prefer it instead of the other.

   (b) Given an array with distinct elements, will both methods give the same heap, not
give the same heap, or does it depend on the input? Justify your answer.

   (c) We say an algorithm is stable if any pair of elements with equal value appear in the
same order in both the input and output. Assume that both heap-building meth-
ods will not swap elements that have equal value. Which heap-building method
is stable, if any? What about the extraction phase of heapsort that follows heap-
building?

Answer:

(a) The reverse method is faster. On the other hand, the forward method is naturally
designed for a data set of unknown size (streaming data). It just spends logarithmic
time per new element. The reverse method requires knowledge of the data set size. It
could be modified to allow a new element and iteratively make heaps, but that would
cost linear time, just to check each element in reverse order. One could notice that if
all nodes were heaps to begin with, only $O(\log n)$ nodes could be affected by a new
insertion, but then that just leads to the forward method.

(b) It’s possible that the same heap will be produced, for instance if the input already
represents a heap, like [3, 2, 1], in which both methods don’t change the input. On the
other hand, try array [1, 2, 3]. The forward scan will start by building 1, then add 2 as
1’s left child and swap them, then add 3 as 2’s right child, and swap them. The heap
will be [3, 1, 2]. The reverse scan will start with 3 and 2 as mini-heaps because they are
leaves. Then we will heapify the root, 1. It will swap with 3, so we will get [3, 2, 1].

(c) Consider the forward method: suppose we have an array with $n$ elements, for which
we have made a heap on all but the last element, $A[n]$, which has value $x$. Its parent
in the heap is at $A[n/2]$. Let the parent have value smaller $x$, but all other elements
have value $x$. The forward heap-build will swap $A[n]$ with $A[n/2]$ and then stop.
Now $A[n/2]$ holds the last element in input order, but roughly half of the elements
also have value $x$ and are located after it.

The reverse method would suffer from the same problem. It would ignore the last half
of the input array, because all those elements correspond to leaves in the heap. The
first and only swap to be made would be between the single element in the array that
is smaller than $x$, with its child, the last leaf. Again that places the (former) leaf with
value $x$ at a position in the array where it is before several other nodes of equal value.
The extraction phase is also not stable. For instance consider a set of equal values.
The first element ends up last.
2. You are given $k$ coins, arranged in various stacks. In one “move”, you may choose any stack and redistribute it into other stacks, but in doing so you can place at most one coin in each other stack. During the move, you are allowed to make as many new stacks as you like, but only one coin can go in each. The reward of one move is precisely the number of coins in the stack that you choose to redistribute.

(a) Suppose that you get to make some huge number of moves (say, $n$). Develop a strategy to maximize your total reward. Notice that any strategy provides a lower bound on the maximum possible reward. The better the strategy, the better (higher) the lower bound.

(b) Use amortization to obtain an upper bound on the maximum possible reward. If it helps to think in terms of cost, pretend that your friend is playing this game and you must pay the reward. In this context, you want to find an upper bound on the worst-case cost. For amortization, as usual, you want to show that “expensive” moves do not happen that often, or in other words you want a function that offsets expensive moves, making their amortized cost smaller. So, think about defining what an expensive move is, and what a non-expensive move is. It will help to have a good strategy in part (a) for this.

Answer:
Here is a strategy that provides a reward of $\Theta(\sqrt{k})$ per move. First, take as much (finite) time as you like, to build $\sqrt{k}$ stacks, of height $\sqrt{k}, \sqrt{k}-1, \sqrt{k}-2, \ldots, 1$. Then for all subsequent moves, distribute the stack of height $\sqrt{k}$ to all the other ones, and to one empty position. This simply restores the initial configuration. The reward is $\sqrt{k}$ per move, after the initial phase. Notice that this uses only half of the coins, but a similar arrangement using all of the coins would not change the reward asymptotically.

Instead, we could use the following strategy. Take some time to create $x$ stacks, each of size $x$. Make sure that $x = \Theta(\sqrt{k})$, which is easy to do. We can ignore the remaining $k - x^2$ coins. Suppose that the $x$ stacks are in “positions” $1, \ldots, x$. Then perform $x$ moves, each of which takes one of the stacks and spreads it out into positions $x+1, \ldots, 2x$. This gives us a profit of $x^2 = \Theta(k)$, and recreates a block of $x$ stacks of size $x$. So our exact profit per move is $x$.

So, now we know that the best possible reward is $\Omega(\sqrt{k})$.

It is more interesting to figure out why any strategy will yield a reward of $O(\sqrt{k})$. Rephrasing reward as cost, we want to show that any sequence of $n$ moves will cost at most $O(n\sqrt{k})$, whereas we know that in the “worst” case a single move could cost $k$. In other words, a naive worst-case analysis would say that $n$ moves will cost $O(nk)$, but an amortized analysis can do better. Consider a potential function $\Phi$ that counts the number of coins above height $\sqrt{k}$, in all stacks. Notice that at most $\sqrt{k}$ stacks can contribute to this count. Call these stacks “large”. When we make a move, we can increase the potential by at most $\sqrt{k}$, because we can place at most one coin in each large stack. In particular, whenever we move a non-large stack, the true cost $c$ is less than $\sqrt{k}$, and the potential can increase by at most $\sqrt{k}$. Thus, by definition the amortized cost $\hat{c}$ is $c + \Delta \Phi = O(\sqrt{k})$. Also, if we ever move a large stack, the true
cost will be $x + \sqrt{k}$, where $x$ is the excess above height $\sqrt{k}$. However the amortized cost will still be at most $\sqrt{k}$, because at least $x$ coins will have to move to non-large stacks. In other words, there is definitely a contribution of $-x$ to $\Delta \Phi$, and there might be another contribution of up to $\sqrt{k}$ to $\Delta \Phi$, so $\hat{c} = c + \Delta \Phi = (x + \sqrt{k}) + \Delta \Phi \leq (x + \sqrt{k}) - x + \sqrt{k} = O(\sqrt{k})$.

The above arguments are enough for full points. Technically, we need to claim that the sum of amortized costs is no smaller than the sum of true costs. This is normally done by ensuring that $\Phi_n - \Phi_0 \geq 0$, typically by ensuring that $\Phi_n \geq 0$ and $\Phi_0 = 0$. We know that $\Phi_n \geq 0$, because we can’t have a nonzero number of coins above height $\sqrt{k}$. But if we are allowed to start from any configuration, then $\Phi_0$ might be positive. If the problem statement specified that we start from some configuration where all stacks have height at most $\sqrt{k}$ (e.g., they all start at height 1), we’d be fine. But our analysis is not doomed. Consider an initial condition where we have only one stack of size $k$. This actually gives the worst possible relationship between $\Phi_n$ and $\Phi_0$, because the latter is as large as $\Phi$ can be. However, in the very next step, after a profit of $k$, we have only stacks of height 1 so we can begin our amortized analysis from there on. We just need to concede that an algorithm that competes with the solution from part (a) is given a head start of $k$. But this is a drop in the ocean compared to the $n$ moves that will follow.

On the other hand, it’s easy to come up with a starting configuration $C$ and a strategy $S$ where $\Phi$ is always positive. How do we prove that our choice of $\Phi$ still works, i.e., that $S$ fails to beat $\sqrt{k}$? Given that we know that any strategy must fail to beat $\sqrt{k}$ if it starts from a configuration $C_0$ where $\Phi_0 = 0$, then we can consider one such strategy, $S^*$, that builds $C$ from $C_0$ as a first phase, and then continues as $S$ does. The first phase would cost at most $2k$ moves (from any starting configuration, spread every stack out, then build $C$ one coin at a time). Each such move has a profit between 1 and $k$. So $S^*$ differs from $S$ by some amount between 1 and $2k^2$, but this is insignificant over $n$ moves. In other words, if $S^*$ fails (which it does), it must be because $S$ failed.