CSE102-01: Intro to Algos[*], PSET-3 Solutions
4th Feb. 2024

Instructions: Deadline in a week, so submit by **Sat. Feb. 4th midnight PST**. Submit to Canvas a pdf of your solutions, with your name and ucsc id number. Feel free to discuss solutions with your colleagues (other students), explore the internet, read and be inspired from the textbooks etc. But what you type should be typed by you and you only in latex. In algorithms, we appreciate brevity and accuracy: your solutions should be to the point, often one or two sentences suffice to provide all necessary details and descriptions.

Ex.1 (Bogosort) Bogosort (also called Stupidsort) is an algorithm for sorting, and is perhaps the worst algorithm you have heard of. In order to sort n numbers given as an array A, Bogosort uniformly at random generates a permutation of the n numbers in A, and then checks if they are actually sorted in increasing order.

Let X be a random variable that denotes the number of iterations of Bogosort until it finds the sorted version of A in increasing order. What is the expected value of X? If we were happy with either increasing or decreasing order, what would be the expected number of iterations then? (Hint: if you have a fair coin, how many times in expectation should you toss it, till you see the first "Tails?")

Solution The answer is $n!$, that is n factorial if we wanted to find the increasing sorted order. If we were happy either with increasing or decreasing, then the answer is $n!/2$, so half the previous answer.

Regarding the hint, the expected number of times we need to toss a coin until we see "Tails" is 2. To see this:

$$E[\#tosses\ till\ tails] = \frac{1}{2} + \frac{1}{2}(1 + E[\#tosses\ till\ tails])$$

Solving for the expected value we see the answer is 2. If the coin was unfair, for example, if the success ("Tails") probability was 1/3, it would take on average 3 trials to get a success (can you see why? Convince yourself.).

To see this more generally, the random variable X as defined in the exercise has the following distribution:

- $X = 1$: In this case, Bogosort outputs the correct increasing sorted order with the first try. This happens by definition of Bogosort with probability $p = \frac{1}{n!}$.

[*]PI: Vaggos Chatziafratis

- $X = k$: In this case, Bogosort outputs the correct increasing sorted order with the k^{th} try. This means there were $k-1$ failed attempts to sort the array, followed with a successful output that was the sorted array in increasing order. This happens with probability $(1-p)^{k-1}p$, where $p = \frac{1}{n!}$ was defined in the previous bullet point.

- By definition of expected values: $E(X) = \sum_{k=0}^{\infty} k \cdot (1-p)^{k-1}p = 1 \cdot p + 2 \cdot (1-p)p + 3 \cdot (1-p)^2 p + \ldots + k(1-p)^{k-1}p + \ldots$ where the sum continues for all k up to infinity. More compactly the sum is written as:

  $$E(X) = p \cdot 1 + (1-p)(1 + E(X))$$

  where we interpret this relation as follows: $E(X)$ is the expected number of trials till we see the first success; X gets the value 1 (success) with probability p, but with probability $(1-p)$ we failed in the first try, so the trials till success start all over, but with an additional $+1$, that's why we have the term $(1 + E(X))$. Solving for $E(X)$ we find that the answer is $E(X) = \frac{1}{p}$.

Since in Bogosort for increasing order $p = 1/n!$, we get $E(X) = n!$ and since in Bogosort for either increasing or decreasing order $p = 2/n!$ (there are exactly two orderings out of $n!$ that we are happy with) we have for the second part of the exercise that $E(X) = n!/2$.

Ex.2 (a^n, but done fast) Suppose you want to calculate the n^{th} power of a given number a. Suppose you know a (fix it say to 42), but n is an input.

- Perhaps the first thing you would do is to compute a times a times a etc. for n times. How much time would you spend in this case? (we assume multiplying and addition takes constant time)

- Can we do better? If yes, how? If not, why not? (hint: divide and conquer)

Solution The first suggestion takes $n-1$ multiplications of a with itself so it takes $\Theta(n)$ time.

The following counterintuitive observation as we discussed in class is what will allow us to prove that we can actually do better. We will in fact compute a^n in time $\Theta(\log n)$ which is **exponentially faster** than the previous suggestion.

The idea is to do **repeated squaring**. So we will compute $a, a^2, a^4, a^8, \ldots, a^{2^k}$. Observe that every time we square the previous term on the sequence. As you know, there are exactly $k+1$ terms in this sequence. If $n = 2^k \iff k = \log n$ then we are done with only $k = \log n$ multiplications.

For the implementation details we have to be slightly more careful for how to deal with odd numbers n. The following fast divide-and-conquer method, originally proposed by the Indian prosodist Pingala in the 2nd century BCE, which uses the following simple recursive formula:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \\ (a^{\lfloor n/2 \rfloor})^2 \cdot a & \text{otherwise} \end{cases}$$

```
PiṅgalaPower(a, n):
  if n = 1
      return a
  else
      x ← PiṅgalaPower(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

Figure 1: Repeated Squaring Implementation.

The total number of multiplications performed by this algorithm satisfies the recurrence $T(n) \leq T(n/2) + 2$. The recursion-tree method or the master theorem immediately give us the solution $T(n) = O(\log n)$ which is optimal as we need $\Omega(\log n)$ only to represent the number n (Convince yourself that this is indeed true).

Ex.3 (Tiling) In front of you, there is a 2-by-n chessboard, that you want to completely cover with tiles that have size 1-by-2 (A *tiling* is a placement of dominoes that covers all the squares of the board perfectly i.e. no overlaps, no diagonal placements, no protrusions off the board, and so on). Each tile can be placed either vertically or horizontally. Let $T(n)$ denote the number of ways that you can tile the chessboard using the tiles. Write down a recursive relation for $T(n)$ and then using it, compute the total number of possible tilings of the chessboard.

Solution The answer is that there are $\Theta(\phi^n)$ possible ways of tiling the 2-by-n chessboard, where $\phi = 1.618$ is the golden ratio. In fact, the answer is exactly given by the n^{th} Fibonacci number F_n. Recall, $F_n = F_{n-1} + F_{n-2}$ for $n > 2$, and the base cases are $F_1 = F_2 = 1$. To prove this look at the following picture:

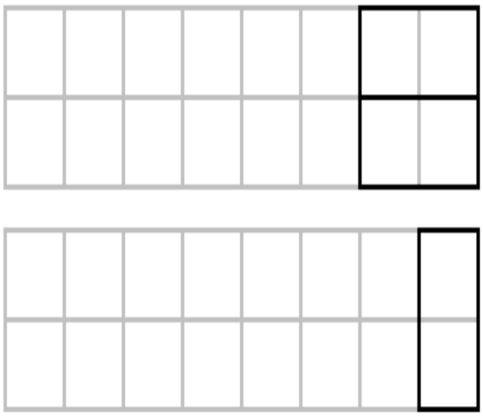

Figure 2: We can start the tiling either with one vertical tile, or with two horizontal tiles. In the first case, we end up with a 2-by-$(n-1)$ chessboard remaining-to-be-tiled, whereas in the second case we end up with a 2-by-$(n-2)$ chessboard remaining-to-be-tiled.

We see that there is a recursive structure in the computation of the tilings that allow us to write down the recursive relation:

$$T(n) = T(n-1) + T(n-2)$$

which is exactly the Fibonacci recursion. Note that the base cases for the chessboard for $n = 1$ is that there is 1 way of tiling, and for $n = 2$ that there are 2 ways of tiling. So the sequence is after all:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

Ex.4 (More Recursions) Like in many previous exercises and homeworks, find tight asymptotic bounds (big-Theta) for $T(n)$ is each of the cases.

1. $T(n) = 2T(n/4) + n^2\sqrt{n}$
2. $T(n) = T(n-1) + \frac{1}{n}$
3. $T(n) = 1600T(n/4) + n!$ (hint: answering this shouldn't require too many, if any, difficult calculations)
4. $T(n) = 6T(n/3) + n^4/\log^{25} n$ (hint: answering this shouldn't require too many, if any, difficult calculations)
5. $T(n) = \sqrt{n}T(\sqrt{n}) + n$ (hint: when everything fails, you guess and check)
6. $T(n) = T(n/2) + n(5 - \cos^2 n \sin^{20} n)$ (hint: answering this shouldn't require too many, if any, difficult calculations, just think the most basic trigonometric inequality)

7. $T(n) = \alpha T(n/4) + n^2$ (hint: your answer should depend on the α parameter)

8. $T(n) = 5T(n/5) + \frac{n}{\log_5 n}$ (hint: think of $n = 5^m$. Also the recursion $T(n) = T(n-1) + \frac{1}{n}$ above may come in handy.)

Solution

1. Answer is $T(n) = \Theta(n^2\sqrt{n})$. We can apply master theorem.

2. Answer is $T(n) = \log n$. We can see by unpacking that $T(n)$ is the n^{th} harmonic number which as we saw in class is very close to $\log n$. Perhaps again one way to see why this is true is by using the integral of the function $1/x$ for x ranging from 1 up to n to approximate $T(n)$. Solving the integral directly yields $\ln n$ (natural logarithm) which is of course $\Omega(\log n)$ under any base for the logarithm (they only differ by a constant).

3. The answer is $T(n) = \Theta(n!)$. To see this observe that $n! \geq n^6$ for sufficiently large n. On the other hand, $\log_4 1600 = 5.32 < 6$. We conclude that even if in the recursive relationship we had n^6 instead of the much faster-growing $n!$, the n^6 term would be the dominant term yielding a solution of $\Theta(n^6)$. This is also true for any growth rate which grows faster than n^6, and in particular it is true for the $n!$.

4. Same logic as the previous, the answer is $T(n) = \Theta(n^4/\log^{25} n)$. To see this, we need to compare $\log_3 6 = 1.63$ against n^3 and we see already that n^3 is the dominant term. Since $n^4/\log^{25} n$ is much larger than n^3 (in asymptotic notation we write $n^3 = o(n^4/\log^{25} n)$ we conclude that the dominant term in the recursion is indeed $n^4/\log^{25} n$.

5. The answer is $T(n) = \Theta(n \log\log n)$. The way to think for how to find the answer goes as follows. We can try to make some simple guesses for $T(n)$ that we shall check: for example, the first guess might be perhaps that $T(n) \leq c \cdot n$ for some constant c. But if we try to verify we quickly see that this cannot be correct, no matter what constant c we pick. Then we need to make another guess. For a second guess, we might try $T(n) \leq c \cdot n \log n$ for some constant c again. Doing the substitution as before, now yields a correct inequality. But we are not done. At this point, we know $T(n) = O(n \log n)$ but the exercise asks to find **tight** asymptotic notation. Finally, we now know the answer is between n and $n \log n$, so what function growth rate falls in that interval? Here, we need to consider the guess $T(n) \leq c \cdot n \log\log n$. Indeed, using the substitution method (and induction) we can verify $T(n) \leq c \cdot n \log\log n$. Are we done? Not yet, as we still have to prove that $T(n)$ also grows as fast as $\Omega(n \log\log n)$. To do so we again use the substitution method (and induction) to show that $T(n) \geq c' \cdot n$ for some other constant c'. Combining both we get that $T(n) = \Theta(n \log\log n)$ and this is the tight asymptotic growth rate and we are done.

Another solution was proposed by a student in the class (thanks Jose!): Simply do the substitution $n = 2^m$ then:

$$T(2^m) = 2^{m/2}T(2^{m/2}) + 2^m$$

Then divide by 2^m both sides and solve for $S(m) = T(2^m)/2^m$. This will yield using the Master Theorem, $S(m) = S(m/2) + 1$ with the solution $S(m) = \log m$. Replacing back to get $T(2^m) = 2^m S(m) = 2^m \cdot \log m$ implies that $T(n) = n \cdot \log\log n$ which agrees with our previous solution.

6. The answer is $T(n) = \Theta(n)$. To see this we will "sandwich" $T(n)$ between two other functions $Q(n), H(n)$ which are both growing linearly in n. We can select $Q(n) = Q(n/2) + n$ and $H(n) = H(n/2) + 6n$. Observe that $Q(n) \leq T(n) \leq H(n)$ as the trigonometric functions are always some number between -1 and 1. Solving for $Q(n), H(n)$ using the Master theorem we conclude they are both $Q(n) = \Theta(n), H(n) = \Theta(n)$, hence $T(n)$ is also $\Theta(n)$.

7. The answer is given by a simple application of the Master Theorem. If $\alpha > 16$ then the subproblems part of the equation will dominate the runtime because $\log_4 \alpha > \log_4 16 = 2$, hence $\log_4 \alpha > d$ where $d = 2$ from the master theorem coefficients. In particular the term n^2 (so $d = 2$ for the master theorem) is not sufficient to affect the runtime. The answer is $T(n) = \Theta(n^{\log_4 \alpha})$. If $\alpha < 16$, the term n^2 will dominate, so $T(n) = \Theta(n^2)$. Finally, if $\alpha = 16$ then we are in the case of Master theorem where an extra logarithm appears and the answer is $T(n) = \Theta(n^2 \log n)$.

8. The answer is $T(n) = \Theta(n \log\log n)$. We observe that we cannot use the Master theorem. But we can use the hint and set $n = 5^m \iff m = \log_5 n$. The recursion becomes:

$$T(5^m) = 5T(5^{m-1}) + \frac{5^m}{m}$$

We rename now $S(m) = T(5^m)$ as we have done in previous homeworks:

$$S(m) = 5S(m-1) + \frac{5^m}{m}$$

For this recursion, we have to do "unpacking" and we end up with:

$$S(m) = \frac{5^m}{m} + \frac{5^m}{m-1} + \frac{5^m}{m-2} + \frac{5^m}{m-3} + \ldots + \frac{5^m}{1} = 5^m \sum_{k=1}^{m} \frac{1}{k} = 5^m H_m$$

where H_m is the m^{th} harmonic number we saw previously. We know $H_m = \Theta(\log m)$ so we conclude that $S(m) = \Theta(5^m \log m)$. Substituting $m = \log n$ we conclude that $T(n) = \Theta(n \log\log n)$.

Ex.5 (Binary Search Lower Bound) In class we saw how sorting n numbers using only comparisons, requires $\Omega(n \log n)$ time, that is **no** algorithm (no matter how clever it is) can sort every input array in significantly less time than $n \log n$. Consider now the problem of searching a number x in a sorted array A (that may or may not contain x). We know binary search does this in $O(\log n)$ time.

Can we do better? In other words, is there a searching algorithm (that uses comparisons) that runs in time $o(\log n)$ and correctly determines whether or not an element x appears in A? (hint: think of a tree representation for the execution of any algorithm, as we did in class for sorting. How many leaves are there in the tree? What should the depth of this tree be?)

Solution We can't do better, that is $\Omega(\log n)$ steps are required by any algorithm in order to do search over a sorted array of n elements.

Any algorithm that uses comparisons has to distinguish among the n possible choices for the answer. That is there are n candidate answers for the sought after number x. Since every algorithm that is correct needs to be able to distinguish these among the n different cases, we can represent its execution by a binary tree that has n leaves. The internal nodes in this tree correspond to the outcomes of comparisons that the algorithm makes. (Recall, we did something analogous for the more challenging problem of sorting n numbers where we showed a lower bound of $\Omega(\log(n!)) = \Omega(n \log n)$.)

The (binary) tree in the case of searching only has n leaves which means its height needs to be at least $\Omega(\log n)$. To see this, if h is the height, then $2^h \geq n$ in order to have n leaves which means $h \geq \log n$ and we are done. We proved that any algorithm based on comparisons requires $\Omega(\log n)$ comparisons in order to figure out whether x is in A or not.

Ex.6 (Inversions) An **inversion** in an array A of n numbers is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2} = n(n-1)/2$ (if the array is sorted backwards). Convince yourselves about the last statement.

Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. (Hint: Modify Mergesort.)

Solution The idea is very similar to mergesort. Suppose we have an array with n numbers. We do the following:

- (Divide) Split A into two subarrays say B, C of half the size (roughly $n/2$) like we did for Mergesort. Say B contains the first half of A whereas C contains the last half of A.
- (Recurse) Recursively solve the problem for B and C that have smaller size. As we do this, we can also sort the subarray B and subarray C. By this recursive calls, we now know the number of inversions r_B and r_C for the subarrays B, C respectively.

- (Conquer) How do we combine the information we have so far? It's simple: the total number of inversions for A is the sum of $r_B + r_C + r_{BC}$ where the symbol r_{BC} denotes the number of inversions that involve an element from B and an element from C. Notice that whenever an element b from B is larger than an element c from C then this pair (b, c) of elements constitutes an inversion. We can easily count such pairs by a simple modification of the "Merge" procedure in Mergesort. Simply, whenever we merge an element from C to the final array we need increment r_{BC} by the number of elements in B that are not merged yet. Convince yourselves why this is indeed the case.

The final runtime is exactly the same as mergesort given by the formula:

$$T(n) = 2T(n/2) + n$$

which gives the $O(n \log n)$ solution. Notice that again here the modified "Merge" step takes $O(n)$ time, because during the recursive steps we were also sorting the subarrays. Run this algorithm on a simple example to verify that it finds the correct number of inversions.