CS361 Homework #2 Solutions

1. Prove that the exact solution of
$$f(n) = 4f(n/2) + n^2, \quad f(1) = 0$$
is
$$f(n) = n^2 \log_2 n$$
when n is a power of 2. Instead of just plugging the solution in to the recurrence, use the structure of an inductive proof: show that it's true for the base case, and then show that if it's true for $n/2$, it's also true for n.

   Base case. For $n = 1$, we have $f(1) = 0 = n^2 \log_2 n = 1^2 \cdot 0$.

   Inductive step. Suppose that the statement is true for $n/2$: that is,
$$f(n/2) = (n/2)^2 \log_2(n/2)$$
   Then the recurrence gives
$$\begin{aligned} f(n) &= 4f(n/2) + n^2 \\ &= 4(n/2)^2 \log_2(n/2) + n^2 \text{ using the inductive assumption for } f(n/2) \\ &= n^2(\log_2 n - 1) + n^2 \text{ simplifying} \\ &= n^2 \log_2 n \end{aligned}$$
   proving the statement for n. By induction it follows the statement is true for all powers of 2.

2. Using the fact that we have to handle every possible permutation, give a lower bound on the number of comparisons we need to sort a list of 4 elements. Is there in fact an algorithm that sorts lists of size 4 with this number of comparisons?

   Answer. Using the "twenty questions" argument, we need at least $\log_2(4!) \approx 4.58$ comparisons. But this really means we need 5.

   Indeed, Mergesort sorts lists of length 4 with 5 comparisons: one to sort each of the sub-lists of size 2, and then 3 to merge them together. (In the Mergesort recurrence, we often assume for simplicity that the Merge operation takes n comparisons; but after $n - 1$, one of the two lists is empty, and so we get to merge the last element for free.)

3. Use the exact recurrence for the average number of comparisons done by Quicksort,
$$f(n) = n - 1 + \frac{2}{n} \sum_{\ell=0}^{n-1} f(\ell) \ ,$$
and the base case $f(0) = 0$, to calculate exactly the average number of comparisons Quicksort does for lists of size $n = 2, 3$, and 4. Then explain these numbers in terms of what might happen when the algorithm runs: where the pivot ends up, what size lists it calls itself on recursively, and so on. (Feel free to use symmetry to shorten your answer; for instance, having the pivot at the left end produces the same running time as if it were at the right end.)

Answer. The recurrence gives $f(2) = 1$, $f(3) = 8/3 \approx 2.67$, and $f(4) = 29/6 \approx 4.83$.

In terms of the algorithm, $f(2) = 1$ because we need 1 comparison to partition the list, i.e., to compare the pivot to the other element in the list. Then, to the left or right of the pivot we have two lists of size 0 or 1, and these take no comparisons at all.

For $n = 3$, it first takes 2 comparisons to partition the list, since we have to compare the pivot to the other two elements. Now, with probability 1/3, the pivot was the median of the three and we have two lists of size 1, so no further comparisons are necessary. On the other hand, with probability 2/3 the pivot is the smallest or largest element, in which case we have a list of size 0 on one side and 2 on the other, which take $f(0)+f(2) = 1$ comparison. So, the total average is $2+(1/3)\cdot 0+(2/3)\cdot 1 = 2+2/3 = 8/3$.

Finally, for $n = 4$ we need 3 comparisons to partition the list. With probability 1/2 the pivot is the smallest or largest, giving sublists of size 0 and 3 which require $f(0) + f(3) = 8/3$ comparisons on average; and with probablity 1/2 the pivot is the 2nd or 3rd largest, giving sublists of size 1 and 2 which take $f(1)+f(2) = 1$ comparison. Thus the total is $3+(1/2)\cdot(8/3)+(1/2)\cdot 1 = 3+11/6 = 29/6$.

4. Here is an algorithm for the `partition` step of Quicksort due to Hoare:

```
// partition A[first..last] using A[first] as the partition:
// put the smaller elements on the left, the larger on the right,
// and return the final position of the partition
int partition(A,first,last) {
  p = A[first];
  left = first+1;
  right = last;
  while (left <= right) {
    while (A[left] <= p and left < last) left++;
    while (A[right] > p) right--;
    if (left < right) swap A[left] and A[right];
  }
  // now put pivot in the right place
  if (right > first) swap A[first] and A[right];
  return right;
}
```

Prove that this algorithm works, by defining a loop invariant which makes some guarantee about the state of the world after each run of the outer `while` loop. Prove "initialization" (base case), "maintenance" (inductive step), and "termination" (making sure that when we're done the entire thing works). If you take more than a page or so, you're probably giving too much detail.

Answer. The key thing to notice is that `left` and `right` move in from the ends of the list until they reach a pair of elements which have the "wrong" relationship to the pivot, and then they switch them. Therefore, the loop invariant is that 1) all the elements to the left of `left` are less than (or equal to) the pivot, and 2 all the elements to the right of `right` are greater than the pivot. To be more precise, we claim that after each trip through the `while` loop, we have

$$A[i] \leq p \text{ for all } i \text{ such that } \texttt{first} < i < \texttt{left}$$

and

$$A[i] > p \text{ for all } i \text{ such that } \texttt{right} < i \leq \texttt{last} \ .$$

(Actually, we could strengthen these claims slightly, to $i \leq \texttt{left}$ and $i \geq \texttt{right}$, but this isn't necessary.)

For the base case, note that when $\texttt{left} = \texttt{first} + 1$ and $\texttt{right} = \texttt{last}$, this invariant says nothing at all, since there are no values of i in these ranges. Since it make no claims, it is automatically true.

For the inductive step, the `while` loop increases `left` and decreases `right` as long as $A[\texttt{left}] \leq p$ and $A[\texttt{right}] > p$, so whatever value of these we end with satisfy both parts of the invariant.

For termination, there are two things we need to say. First, since the `while` loop continues until `left` and `right` cross, when we leave the `while` loop we have $\texttt{left} = \texttt{right} + 1$. Since by induction the invariant is true at this point, we have $A[i] \leq p$ for all i ranging from $\texttt{first} + 1$ to `right`, and $A[i] < p$ for all i ranging from $\texttt{left} = \texttt{right} + 1$ to `last`. Then switching $p = A[\texttt{first}]$ with $A[\texttt{right}]$ puts all the elements less than or equal to p to the left of p, and all the elements greater than p to its right.

However, since this is a `while` loop rather than a `for` loop, we have to prove that it terminates after a finite number of iterations. In other words, we have to make sure that it doesn't run for ever. To see this, we have to prove that each run of the `while` loop increases `left` by at least one, and decreases `right` by at least one, so that after $O(n)$ repetitions `left` crosses `right` and we exit the loop. But this is true—except, perhaps, on the first run through—because the `swap` step at the end of the each loop ensures that $A[\texttt{left}] \leq p$ and $A[\texttt{right}] > p$ at the beginning of the next loop. Therefore, each of the little `while` loops will run at least once, incrementing `left` and decrementing `right`.

Finally, we note that while we can end with `left` falling off the right end of the list—that is, with $\texttt{left} = \texttt{last} + 1$—the invariant, and this inductive proof, still works.

[Note to nervous readers: it's ok if your proof makes these points at somewhat different levels of detail, as long as all the ideas are there.]

5. In Quicksort, suppose I could guarantee that the pivot's position is a fraction a through the list for some constant $0 < a < 1$. The recurrence for the number of comparisons would then be roughly

$$f(n) = f(an) + f((1-a)n) + n$$

(setting $a = 1/2$ gives the recurrence for Mergesort). Solve this recurrence by plugging in a solution of the form $f(n) = An \log n$. Discuss what happens to the constant A in the limit $a \to 0$ or $a \to 1$.

Plugging $f(n) = An \log n$ into the recurrence gives

$$\begin{aligned} An \log n &= Aan \log(an) + A(1-a)n \log((1-a)n) + n \\ &= Aan(\log n + \log a) + A(1-a)n(\log n + \log(1-a)) + n \\ &= An \log n + An\big(a \log a + (1-a)\log(1-a)\big) + n \\ \Rightarrow 0 &= A(a \log a + (1-a)\log(1-a)) + 1 \end{aligned}$$

so

$$A = \frac{1}{-a \log a - (1-a)\log(1-a)}$$

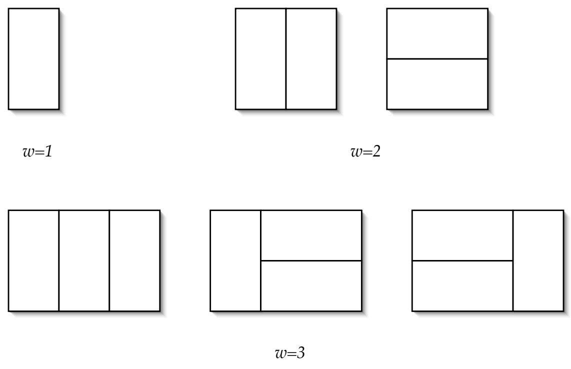
Note that this expression for A is positive, since a and $1-a$ are both less than 1 and their logarithms are negative. When $a = 1/2$, this gives $A = 1/\log 2$, which is $A = 1$ if we use log base 2, and this fits exactly with Mergesort. When $a = 1/3$, say, we get $A \approx 1.09$, taking about 10% longer than Mergesort. However, in the limit $a \to 0$ or $a \to 1$, i.e., if the pivot is close to one end of the list or the other, A goes to infinity and the algorithm no longer takes $\Theta(n \log n)$ time.

6. Let $f(w)$ be the number of ways to tile a rectangle of height 2 and width w with horizontal and vertical dominoes. For instance, $f(1) = 1$, $f(2) = 2$, and $f(3) = 3$ as the following picture shows:

What is the recurrence for $f(w)$? What is $f(10)$, for instance? Working backwards, what is $f(0)$?

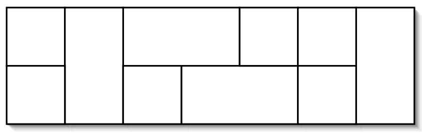
Answer: Either the first column contains a vertical domino, or the last two contain two horizontal dominoes. The number of ways to tile the rest of the rectangle is then either $f(w-1)$ or $f(w-2)$ respectively, so $f(w)$ obeys the Fibonacci recurrence

$$f(w) = f(w-1) + f(w-2) \ .$$



Working upwards we get $f(10) = 89$. Also, $f(0) = 1$: there is exactly one way to tile a rectangle of width 0, which is to do nothing! However, there are no ways to tile a rectangle of width -1, so $f(-1) = 0$.

Extra Credit: What happens if we allow 1×1 blocks as well as dominoes, such as



Hint: it might help to write a recurrence for a *pair* of functions.

Answer: Indeed it might. Now the first column can be covered by a vertical domino, two 1×1 blocks, or two horizontal ones, giving two ways to leave a rectangle of width $n - 1$ and one way to leave one of width $n - 2$. This contributes $2f(n-1) + f(n-2)$ to the recurrence for $f(n)$.

However, the first column can also be covered by a horizontal domino and a 1×1 block. There are 2 ways for this to happen (with the domino on the top or bottom) and in each case, the rest of the rectangle is a $2 \times (n-1)$ rectangle with one corner removed. Let's call this a "notched rectangle of width $n - 1$" and define the number of ways to tile a notched rectangle of width n as $g(n)$. Then

$$f(n) = 2f(n-1) + f(n-2) + 2g(n-1)$$

Now we need a recurrence for $g(n)$. If we have a notched rectangle, the single square left by the notch is either covered by a 1×1 block or a horizontal domino. The remaining shape is then either a rectangle or a notched rectangle respectively, of width $n - 1$. This gives the recurrence

$$g(n) = f(n-1) + g(n-1)$$

The first few values of this double recurrence are

n	0	1	2	3	4	5
$f(n)$	1	2	7	22	71	228
$g(n)$	0	1	3	10	32	103

If you actually solve this recurrence I will be *very* impressed. Here's how I did it:

Since these recurrences look like the Fibonacci series, it's reasonable to think that $f(n)$ and $g(n)$ grow exponentially. It's also reasonable that they grow at the same rate, and have a constant ratio. So, let's try a solution of the form

$$f(n) = r^n, \quad g(n) = Ar^n$$

In fact, A is the fraction of tilings of a $2 \times n$ rectangle that have a 1×1 block in, say, the upper-left corner. Our guess amounts to the belief that this fraction converges to a constant when n is large.

Plugging this into the recurrences gives a pair of equations for r and A,

$$\begin{aligned} r^2 &= 2(1+A)r+1 \\ Ar &= 1+A \end{aligned}$$

The second equation is linear, and gives

$$A = 1/(r-1)$$

Substituting this into the first equation and simplifying gives a *cubic* equation for r,

$$r^3 - 3r^2 - r + 1 = 0$$

Mathematica tells me that this has two complex roots and one real one. The real one is a complicated expression, but numerically it is around 3.214. (It is also the one with the largest absolute value, so it dominates when n is large.) So we have

$$f(n), g(n) = \Theta(r^n) \text{ where } r \approx 3.214$$

and

$$g(n)/f(n) \to A = 1/(r-1) \approx 0.452$$

so about 45% of tilings of $2 \times n$ rectangles have a 1×1 block in, say, the upper-left corner when n is large.