CS504: Analysis of Computations and Systems — Fall 2000

Solution of Midterm test —— 10/30/2000

The test contains three problems. You need to do any two for a perfect score. (I shall only count two for your grade).

1. Here is a C language version of a contender to the title of Least Efficient Procedure. This one searches for the minimum value in an array $A$, which contains values of type $\texttt{ttt}$.

```c
int rmin(A, left, right) ttt *A; int left, right;
{int k; ttt a, b; if (left > right) return (Max);
 if (left == right) return (A[left]);
 k = left + rbd() * (right - left);
 a = rmin(A, left, k);
 b = rmin(A, k+1, right);
 if (a < b) return (a);
 return (b); }
```

where $\texttt{Max}$ is the largest representable value of type $\texttt{ttt}$; the function $\texttt{rbd()}$ returns a random value, uniformly distributed on $[0, 1]$, and the only expensive operation here is comparing two variables of type $\texttt{ttt}$. Therefore the analysis you need to do it to consider the number of comparisons of this type, which occur in one place only in this program. Denote this number by $T(n)$. This is a random variable, for a call of the function to process an array $A$ of size $n$.

Your mission is to calculate $t(n)$, the expected value of $T(n)$.

(a) Find a recurrence for $t(n)$.

Neuroscientists have been working many years, trying to find out how people understand each other. What I need to discover, is how we misunderstand each other. I thought that specifying the two int left, right parameters would be understood as leftmost and rightmost array indices, but quite a number of you found in these two small words a wealth of information that was entirely unrelated to my intention...

I rather expected that many of you will see the similarity to QuickSort, which we analyzed in detail in class and will use this similarity. This seems to have been the case for
some, but not nearly as often as I expected, and at least in one case (this is written before I
graded all the works), led to a student thinking this is exactly as QS...

Let us look at this procedure as if QS did not exist. The basic observation is—and
this is common to all recursive programs—the work the program does consists of two
components. One it does at the current call, and this part is usually easy (here, where we
do not talk about the entire effort, but only count comparisons, the “current” cost is 1, for
the single comparison between a and b). And then there is the cost associated with the
recursive call—two calls here—which can be expressed in terms of the same function we
are out to compute. Let us first write a relation for the random variable $T(n)$. Its value will
depend on the value generated by the call to $rbd()$, which is also a random variable, and
therefore we denote it by a capital letter, $K$. Suppose the integer arguments in the first call
are 1 and $n$—or better, in keeping with the C language conventions, let us say they are 0
and $n-1$, then the calls are to $\text{rmin}(A, 0, K)$ and $\text{rmin}(A, K+1, n-1)$, which have the
costs $T(K+1)$ and $T(n-K-1)$ comparisons, respectively:

\[ T(n) = 1 + T(K+1) + T(n-K-1), \quad n \geq 2. \]

(If you assumed entry with the arguments 1 and $n$, you would have the recurrence
$T(n) = 1 + T(K) + T(n-K)$, which is entirely isomorphic). The range $n \geq 2$ is obtained
from reading the procedure: it has special behavior for $n = 0, 1$, and continues generically
for larger $n$.

Since I asked for a recurrence satisfied by $t = E[T]$, we take the expected value of both
sides in the above equation; this operation simply translates $T(n)$ to $t(n)$ in the left-hand
side, but in the right-hand side we cannot do it directly: the random variable $K$ appears
there. We need to get rid of it first. You get rid of a random variable by randomizing on it,
using the Theorem of Total Probability. This is particularly simple here, since we assumed
that the random generator returns all values (here 0 to $n-1$) in equal probability, which
must be $1/n$. Hence we get for the RHS at first $1 + (1/n)\sum_{k=0}^{n-1}[T(k+1) + T(n-k-1)]$, and
after these $T()$ are replaced by their expected values as well, we get the desired recurrence:

\[ t(n) = 1 + \frac{1}{n} \sum_{k=0}^{n-1}[t(k+1) + t(n-k-1)], \quad n \geq 2. \]  

(1)

Reading some of your works I noticed a benefit of this formulation—working with the
random variables first, and then going for the expectation—that I did not realize before: it
discourages some rash assumptions. What I have in mind are those who said something like "on the average, the call for the random number generator returns a value in the middle of the array" and therefore those recursive calls are to process half the array, and they found themselves with the recurrence
\[ t(n) = 1 + t\left(\left\lceil \frac{n}{2} \right\rceil\right) + t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \]
or something similar. This equation has the merit of being easy to solve, but does not provide the correct answer we need. Now you may ask the natural question – since we are looking for expected values only, why can’t we make this substitution? The somewhat formal answer would be that when you calculate the expected value of a function of some random variable, you can move the expectation operator inside and compute, instead, the function of the expectation of that random variable only if the function is linear. Here the function (of \( K \)) is the random variable \( T(n) \), and there is no reason to assume it is linear in its argument (it is not, although its expected value is, as we show in part (b)). It is interesting, that if you consider this last equation for \( n \) values which are powers of 2, then you find \( t(2^r) = 2^r - 1 \), which—think of this as \( t(n) = n - 1 \)—is obviously the smallest number of comparisons you need to do to find the minimal value—whereas this procedure is presented as Very Inefficient for a good reason (it is the recursive calls to \( \text{rmin} \) with empty subarrays, mostly, which cause additional, useless comparisons). Hence such a result should arouse suspicion.

And then, there were a few who understood my description of \( \text{rbd}() \) to mean it returns 0 or 1, in probability \( \frac{1}{2} \) each. And I think I saw a couple that even got the correct recurrence, under this understanding, though did not continue right:
\[ t(n) = 1 + \frac{1}{2}[t(1)+t(n-1)+t(n)+t(0)], \quad n \geq 2, \]
and since \( t(0) = t(1) = 0 \), it simplifies to
\[ t(n) = 1 + \frac{1}{2}t(n-1) + \frac{1}{2}t(n), \quad n \geq 2, \quad \Rightarrow \quad t(n) = 2 + t(n-1) \]
with the immediate solution \( t(n) = 2(n-1) \). The odd thing about this interpretation is that the procedure appears so dumb: half the time, when \( \text{rbd}() \) returns 1, it makes no progress; the only progress happens when it returns a 0.

(b) Solve the recurrence you found in part (a).
To solve the recurrence (1), we first change the dummy indices. In the first one we write \( i = k + 1 \), and hence we sum on \( t(i) \) in the range \([1..n]\). In the second we write \( j = n - k - 1 \), hence the range of summation on \( j \) is \([n - 0 - 1..n - (n - 1) - 1] = [n - 1..0]\). Substituting these in the last equation, we find, using \( t(0) = 0 \):

\[
t(n) = 1 + \frac{1}{n} \left( \sum_{i=1}^{n} t(i) + \sum_{j=0}^{n-1} t(j) \right) = 1 + \frac{1}{n} \left( t(n) + \sum_{i=1}^{n-1} t(i) + \sum_{j=1}^{n-1} t(j) + t(0) \right)
\]

Multiplying by \( n \) and moving one \( t(n) \) from right to left, we find

\[
(n - 1)t(n) = n + 2 \sum_{j=1}^{n-1} t(j).
\]

We can continue just as we did for the QS equation, but let us do it slightly differently (it really does not change a thing; I saw it in one of the exams and I liked it): Define \( a_n = \sum_{j=1}^{n} t(j) \). Then the last equation becomes

\[
(n - 1)(a_n - a_{n-1}) = n + 2a_{n-1} \quad \Rightarrow \quad a_n = \frac{n + 1}{n - 1}a_{n-1} + \frac{n}{n - 1}, \quad n \geq 2.
\]

This is an first order linear equation, we know how to solve it; it needs one initial value, of \( a_1 = t(1) = 0 \), and then:

\[
a_n = \sum_{j=1}^{n-1} \frac{j + 1}{j} \prod_{i=j+1}^{n-1} \frac{i + 2}{i}.
\]

The product telescopes in a civilized way to produce \( \frac{n(n+1)}{(j+1)(j+2)} \), hence

\[
a_n = n(n + 1) \sum_{j=1}^{n-1} \frac{1}{j(j + 2)}.
\]

Since \( \frac{1}{j(j+2)} = \frac{1/2}{j} - \frac{1/2}{j+2} \), we find

\[
a_n = n(n + 1) \sum_{j=1}^{n-1} \left( \frac{1/2}{j} - \frac{1/2}{j+2} \right) = n(n + 1) \frac{1}{2} \left( H_{n+1} - H_{n+1} + 1 + \frac{1}{2} \right)
\]

\[
= \frac{3}{4} n(n + 1) - \frac{2n + 1}{2}.
\]

Hence \( t(n) = a_n - a_{n-1} = (3/2)n - 1 \), after some cancellations. Asking MAPLE to check this out, by substituting in the original equation provides confirmation.
2. (a) We consider a sequence of real numbers \( \{a_i\}_{i \geq 0} \), given by \( a_i = \frac{u^i}{(2i)!} \), where \( u \) is a real number \( (u \in \mathbb{R}) \). Find its ordinary generating function \( a(z) = \sum_{i \geq 0} a_i z^i \).

This produced major confusion... I ask to calculate a GF, which is simply as given above, \( a(z) = \sum_{i \geq 0} a_i z^i \), and this is a multisection – because \( a_i \) has this pesky \((2i)!\) in the denominator – and a multisection you were told to resolve by taking its own GF!

The idea is to write

\[
a(z) = \sum_{i \geq 0} a_i z^i = \sum_{i \geq 0} \frac{u^i}{(2i)!} z^i = \sum_{i \geq 0} \frac{(uz)^i}{(2i)!} = \sum_{i \geq 0} b_{2i},
\]

where \( b_i = (\sqrt{uz})^i/i! \). Now this is exactly the multisection with which we opened the discussion of multisections in class... we said that if \( b(x) = \sum_{i \geq 0} b_i x^i \), then \( \sum_{i \geq 0} b_{2i} = (1/2) (b(1) + b(-1)) \). Now \( b(x) = \sum_{i \geq 0} (\sqrt{uz})^i/i!x^i = \sum_{i \geq 0} (\sqrt{uz}x)^i/i! = \exp(\sqrt{uz}x) \). Hence the desired sum is \( a(z) = (1/2)(\exp(\sqrt{uz}) + \exp(-\sqrt{uz})) = \cosh(\sqrt{uz}) \).

One of the particularly jarring errors, that was repeated in various forms and flavors several times was as follows (I am copying a particular one)

\[
a(z) = \sum_{i \geq 0} \frac{u^i}{(2i)!} z^i = \sum_{n \geq 0} (uz)^n (2n)! = \sum_{n \geq 0} (uz)^n \sum_{n \geq 0} \frac{1}{(2n)!}.
\]

There are of course several problems with this calculation, and the infuriating thing is that there is an example in handout 2 where it is shown how duplicating an index in the same equation leads to trouble, and I thought I should drop it — using the argument that nobody in graduate school is likely to do such a boo-boo, and if I keep it, I am just insulting the reader — and here it comes, and appears, again, and again, and again....

(b) Compute the infinite sum

\[
\sum_{k \geq 0} \frac{3k + 1}{3^{k+1}}.
\]

This proved to be the “here is an easy one for you,” for most people (except the unfortunate few who simply did not believe I would give them such a present, and looked for traps and hidden meanings...)

\[
\sum_{k \geq 0} \frac{3k + 1}{3^{k+1}} = \sum_{k \geq 0} k3^{-k} + \frac{1}{3} \sum_{k \geq 0} 3^{-k} = \frac{1/3}{1 - 1/3} + \frac{1/3}{2/3} = \frac{1}{3} \left( \frac{1}{2} + 1 \right) = \frac{1}{2} \cdot \frac{5}{2} = \frac{5}{4}.
\]
3. The following is a C version of a procedure that performs the usual insertion sort of an array, of double-precision numbers, called $a$ of size $n$.

```c
insort(a, n) int n; double *a;
{ int i,j; double t;
    if (n <= 1) return;
    for (j=1; j<n; j++) { t = a[j];
        for (i=j-1; i>=0; i--)
            if (t < a[i]) a[i+1] = a[i]; else break;
        a[i+1] = t;
    }
}
```

(a) Prove by mathematical induction that the procedure does as claimed: given an array where the entries are in any order, then on exit all the original numbers are there, not one number has been changed or omitted, and they are sorted in increasing order (non-decreasing is the term that covers the case when values may be repeated).

Not all of you tried this problem, which is a shame, since if you look at it the right way, it is the easiest of the lot.

Let $P(n)$ be the property that the procedure operates correctly on a list of size $n$.

Of course, `insort` only does anything if there are at least two elements; that is why it checks, and if there is only one (it also checks that the arguments is a real length – element in N), it returns it unchanged, and a list of one is always sorted. That provides the the basis of the induction, $P(1)$.

Then we make the induction hypothesis. Here it suffices to assume that $P(m)$ holds, for some $m \geq 1$ (rather than the assumption of “strong induction,” that the property holds for an entire range, $P(k), 1 \leq k \leq m$).

The idea at the center of the induction step proof is a formulation of the operation of this procedure in the following way: “it goes along the elements, left to right, moving each as far to the left as needed, so it is larger than the one immediately to its left. All the skipped entries are shifted one position to the right.” This means that after it scanned the first $m$ elements (of a larger array), we can say that this is exactly what it would have done if the array size were $m$, and therefore, by assumption, those $m$ elements are properly sorted.
Then, when it comes to the last element of an array of size \( m + 1 \), and then \( j = m \) (remember our array starts at location 0, so this is the \( m + 1 \)st element), the loop over \( i \) examines the elements to its left, bumping one place to the right all those it is smaller than, and finally drops it to the left of the first element larger than it is. If I wanted to be really a stickler to formality, I mean – really tiresome, I would say that the activity I just described of the last element drifting to its correct position, because it does not describe directly each action of the procedure, but lumps them under terms like “moves them as far as...,” or “examines... and finally drops,” is not fully described, and therefore needs to be induction-based as well. However this loop is so simple, that I do not feel the urge is strong enough.

Done. In particular, since the operations involve just moving around elements, no value is changed, and no element is lost, and the claim of content invariance holds.

(b) Now look again at \( \text{rmin()} \), the procedure you worked on in problem 1. Explain why a similar induction proof will not be possible to prove its correctness.

No one of those I have seen so far got this part right. All kinds of interesting “excuses” why we cannot do induction, but not correct.

The one argument I saw more than once, and which is wrong, since it is simply irrelevant, was that because the procedure uses a random number generator, induction is powerless to handle it. Not so. And for illustration why this is the case, we look at a very similar procedure to \( \text{rmin} \), QuickSort. There, the pivot is also located at random, depending on the size of the element that was chosen (and we can make the choice random as well), but when it falls in position \( k, 0 \leq k \leq n - 1 \), in an array of size \( n \), QS proceeds to sort subarrays of sizes \( k \) and \( n - k - 1 \). The crucial point is that these arrays have always a size which is strictly less than \( n \). Therefore induction works! (though the strong induction, as described in part (a) above, is needed here).

And now you see the problem with our procedure here: if the rightmost element is chosen, \( \text{rmin} \) continues to call itself with array sizes 0 and \( n \)... no reduction – no induction.