CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Solutions to Homework # 2

1. At each recursive step, the input size n is divided by p+1 and only 1 parallel key comparison is needed. For convenience, suppose that the input size at each recursive step is always divisible by $p+1$, i.e. $n = (p+1)^k$ for some k. Also, assume that in the base case, when $n \leq p$, exactly one parallel key comparison is needed to solve the problem. We can formulate a recurrence relation as follows:

$$R(n) = R(n/(p+1)) + 1; \quad R(1) = 1.$$

Using the iteration method, we get:

$$\begin{aligned} R(n) &= R(n/(p+1)) + 1 \\ &= R(n/(p+1)^2) + 1 + 1 \\ &= \quad ... \\ &= R(n/(p+1)^i) + i. \end{aligned}$$

Let $i = \log_{p+1} n$. Then $R(n) = R(1) + \log_{p+1} n$. Therefore,

$$R(n) = \log_{p+1} n + 1.$$

When n is not of the form $(p+1)^k$, the number of steps is by $\lceil \log_{p+1} n \rceil + 1$.

2. (a) A recursive algorithn to compute $x^e \bmod N$ is as follows:

```
F(x,e,N) {  /* x,e,N are non-negative binary numbers */
  if (e == 1)
     return (x mod N)
  else
     if (e is even) then
        temp = F(x,e/2,N);
        return (temp*temp mod N)
     else //then e is odd
        temp = F(x,(e-1)/2,N);
        return (temp*temp*x mod N)
```

(b) (i) Let $Mul(e)$ denote number of multiplications for computing F(x,e,N). Then, $Mul(e) = Mul(e/2) + 1$ if e is even and $Mul(e) = Mul((e-1)/2) + 2$ if e is odd. Also, $Mul(1) = 0$. In the worst case, the exponent e will be odd on every recursive call, for example, if e is of the form $2^i - 1$ for some i. In this case,

$$\begin{aligned} Mul(e) &= Mul(\lfloor e/2 \rfloor) + 2 \\ &= Mul(\lfloor e/4 \rfloor) + 2 + 2 \\ &= \quad ... \\ &= Mul(\lfloor e/2^i \rfloor) + 2 + \ldots + 2 \text{ (where the number of 2's is } i\text{).} \end{aligned}$$

Choose $i = \lceil \log_2 e \rceil$. Then $\lfloor e/2^i \rfloor) = 1$, and since $Mul(1) = 0$ we have $Mul(e) = 2\lceil \log_2 e \rceil$. If e is an n-bit number, it follows that $\lceil \log_2 e \rceil = n$ and so $Mul(e) = 2n$.

(ii) Let $Mod(e)$ denote number of mod operations for computing F(x,e,N). Then, $Mod(e) = Mod(e/2) + 1$ if e is even and $Mod(e) = Mod((e-1)/2) + 1$ if e is odd. Also, $Mod(1) = 1$. Solving in a similar manner to the recurrence of (i), we find that $Mod(e) = \lceil \log_2 e \rceil + 1$. If e is an n-bit number, it follows that $Mod(e) = n + 1$.

(c) Suppose that x, e, N are all n-bit binary numbers. Let $T(n)$ denote the running time of the algorithm F(x,e,N). Then $T(n)$ is proportional to the time for all the multiplications plus the time for all the mod operations. This is

$$nO(n^2) + (n+1)O(n^2) = O(n^3).$$

(d) The simpler algorithm repeats the same step e times, where the step involves one multiplication and one mod operation. If e is an n-bit number, in the worst case $e = 2^n - 1$. Therefore the worst case the running time is given by

$$T'(n) = O(en^2) = O(2^n n^2).$$

This is much worse than $T(n)$ when n is large enough.

3. If $0 < m \leq n$ we have that

$$\begin{aligned}
T(m,n) &= T(m, n-1) + 1 \\
&= T(m, n-2) + 2 \\
&= \quad \ldots \\
&= T(m, n-(n-m)) + (n-m) \text{ (this is the point where both } = \\
&= \text{ parameters have the same size)} \\
&= T(m,m) + n - m \text{ (rewriting the previous line)} \\
&= T(m, m-1) + 1 + (n-m) \\
&= T(m-1, m-1) + 2 + (n-m) \\
&= \quad \ldots \\
&= T(1,1) + 2(m-1) + (n-m) \\
&= T(1,0) + 1 + 2(m-1) + (n-m) \\
&= 1 + 2(m-1) + (n-m) \text{ (using the base case)} \\
&= n + m - 1.
\end{aligned}$$

4. (a) If we take the minimum size set of coins - call this set S - that sum to L and remove one coin, say c_i, then the number of coins left must be $P(L - c_i)$ and so the total number of coins in S is $P(L - c_i) + 1$. If $L > 0$ then at least one coin is in the set S. Hence,

$$P(L) = \min_{1 \leq i \leq n} P(L - c_i) + 1.$$

The base case can be $P(0) = 0$.

Some students suggested variants of the following recurrence:

$$P(L) = \min_{1 \le i \le n} (P(L \text{ div } c_i) + P(L \bmod c_i))$$

where here, L div ci is the integer part of L/ci. As it turns out, this may not yield the minimum solution. Consider the case where the coins have values 11, 7, and 1 and $L = 25$ to see why.

(b) From the recurrence relation, there is a natural recursive algorithm for computing $P(L)$:

```
function P(L)
   if L = 0 then output 0
   else output 1 + min_i (P(L-ci), where the min is taken over i in [1,n]
```

While this algorithm is correct, its running time is not $O(nL)$. The problem is that as the recursion unfolds, different branches of the algorithm may call the function on the same value. To avoid this problem one can instead store the values of the function in an array as they are being computed. An alternative algorithm is as follows:

```
function P(L)
    {the function uses an array p[0..L]}
    p(0) = 0
    for j = 1 to L do p(j) = 1 + min_i p[j-ci]
```

Now, the recursive calls are replaced by reads to an array and each array read takes constant time. Each iteration of the for loop takes $O(n)$ time, since the min of n numbers needs to be computed. The for loop is executed L times, and so the total running time of the algorithm is $O(nL)$.