

1. (20 points) Consider an integer in base  $b$  with  $n$  digits where the leading digit is non-zero.
  - (a) (5 points) What is the smallest such integer ?
  - (b) (5 points) What is the largest such integer ?
  - (c) (5 points) If all such integers are equally likely what's the probability of the integer where all the digits are equal to  $b - 1$  ?
  - (d) (5 points) If two such integers are chosen uniformly at random, what is the probability that their sum needs  $n + 1$  digits ?

**Solution:** Let  $x$  be the integer under consideration.  $x$  can therefore be written as

$$x = x_{n-1}b^{n-1} + x_{n-2}b^{n-2} + \dots + x_1b + x_0$$

Since  $x_{n-1} \neq 0$ , the smallest possible value for  $x$  is when  $x_{n-1} = 1$  and  $x_k = 0$  for  $0 \leq k \leq n - 2$ . The largest possible value for  $x$  is when  $x_k = b - 1$  for  $0 \leq k \leq n - 1$ . Therefore, we have  $x = x_{\min} = b^{n-1}$  as the smallest possible value and  $x = x_{\max} = b^n - 1$  as the largest possible value. The number of integers in the range  $[x_{\min}, x_{\max}]$  is then  $x_{\max} - x_{\min} + 1 = b^n - b^{n-1}$  and so the probability that  $x = x_{\max} = 1/(b^n - b^{n-1})$ .

Instead of finding  $\Pr(x + y \geq b^n)$  directly, we generalize the problem a little bit and ask for the probability that two integers  $X$  and  $Y$  chosen uniformly at random from  $[s, t]$  have the property that  $X + Y > t$ . This probability can be written as

$$\Pr(X + Y > t) = \sum_{x=s}^t \Pr(X + Y > t | X = x) \Pr(X = x)$$

Since we know that the integers are chosen uniformly at random, we have  $\Pr(X = x) = 1/(t - s + 1)$ . Also,

$$\Pr(X + Y > t | X = x) = \Pr(Y > t - x)$$

It's now easy to see that if  $x \leq t - s$ , then  $\Pr(Y > t - x) = x/(t - s + 1)$ ; otherwise, if  $x > t - s$ , then  $\Pr(Y > t - x) = 1$ . In short, we have

$$\Pr(X + Y > t) = \frac{1}{t - s + 1} \left( \sum_{x=s}^{t-s} \frac{x}{t - s + 1} + \sum_{x=t-s+1}^t 1 \right)$$

Simplification of the above expression and replacing  $t$  and  $s$  with  $b^n - 1$  and  $b^{n-1}$  respectively, we have

$$\Pr(X + Y > b^n - 1) = \frac{b^{2n} - 2b^{2n-2} - b^n + 2b^{n-1}}{2(b^n - b^{n-1})^2}$$

2. Consider the problem of efficiently adding an  $m$  digit base  $b$  number with no leading zero to an  $n$  digit base  $b$  number with no leading zero, where you are permitted to store the answer on top of the  $n$  digit number. Assume for this problem that  $n \geq m$ .
  - (a) (10 points) Give an efficient algorithm for this problem
  - (b) (10 points) Given an average time analysis of your algorithm when the input numbers are random in the sense of Question 1.

**Solution:** The algorithm might be as in Algorithm 1 below. Our average time analysis of the algorithm will be as follow. Since the for loop from line 1.2 to line 1.5 is executed irrespectively of the number  $X$  and  $Y$ , they are not relevant to our average time analysis. The while loop is entered only when there's a carry, that's if the addition of  $Y$  to the last  $m$  bits of  $X$  create a carry. From Question 1, we know that this is the probability that two numbers chosen uniformly at random from the range  $[0, b^m - 1]$  has sum at least  $b^m$ . This probability is then  $1/2 - b^{-m}$ . After updating  $X[i]$  for some  $i > m - 1$  in the body of the while loop, the while loop advances on to  $i + 1$  only if carry  $\neq 0$ . This is equivalent to  $X[i] = b - 1$  since the carry in any case is at most 1.

If we now let  $\rho = 1/2 - b^{-m}$  be the probability that the while loop is executed,  $p = 1/b$  be the probability that the while loop is continued, and  $q = 1 - p$ , then the probability that the while loop is executed  $k$  times is  $\rho p^{k-1} q$ .

```

1.1 carry ← 0 ;
1.2 for i = 0 to m - 1 do
1.3   temp ← X[i] + Y[i] + carry ;
1.4   X[i] ← temp mod b ;
1.5   carry = ⌊temp/b⌋ ;
1.6 end
1.7 while carry ≠ 0 and i < n - 1 do
1.8   temp ← X[i] + carry ;
1.9   X[i] ← temp mod b ;
1.10  carry = ⌊temp/b⌋ ;
1.11  i ← i + 1 ;
1.12 end
1.13 if carry = 0 then return X ;
1.14 else return carry · X

```

**Algorithm 1:** Addition Algorithm

Assuming that each execution of the steps in the while loop take  $c_2$  units of time and the last conditional test on line 1.13 take  $c_3$  units of time, we have the average running time  $A(m, n)$  of the above algorithm be

$$\begin{aligned}
 A(m, n) &= c_1 m + c_2 \rho \sum_{k=1}^{n-m} p^{k-1} q k + c_3 \\
 &= c_1 m + \frac{c_2 \rho q}{(1-p)^2} + c_3 \\
 &= c_1 m + \frac{c_2 \rho b}{b-1} + c_3
 \end{aligned}$$

3. (20 points) Consider a potential algorithm that is going to multiply three-digits numbers based on additions and multiplications of one-digit numbers. The intended algorithm will work with any positive integer base so that it can be used recursively, just as algorithm 5.2 was used.
- (a) (10 points) Let  $k$  be the number of one-digit multiplication used by the algorithm. What's the largest value for  $k$  such that the potential algorithm is more efficient than Algorithm 5.2 for large numbers ?
- (b) (10 points) Assume that the running time of the algorithm obeys the recurrence

$$T(n) = kT(n/3) + an + b \quad (*)$$

What is the running time when  $n$  is a power of 3 ? In your final answer, be sure to use the value of  $k$  from part a and give the answer in the simplest form.

**Solution:** After thinking for a little bit, we can see that if the intended algorithm needs 3 multiplication of one-digit numbers for every multiplication of two three-digits number, than the algorithm will be very fast and surely faster than algorithm 5.2's running time of  $O(n^{\log_2 3})$ . A quick look at the analysis of the running time of algorithm 5.2 will surely tell us that the running time of the intended algorithm should be of the form  $O(n^{\log_3 k})$ , and so the question reduces down to the largest value of  $k$  such that  $\log_3 k < \log_2 3$  and the answer is  $k = 5$ . Now if we use the value of  $k = 5$  in the recurrence equation (\*) for  $T(n)$ , we have

$$\begin{aligned}
 T(n) &= 5T(n/3) + an + b \\
 &= 25T(n/9) + 5an/3 + b + an + b \\
 &= 125T(n/27) + 25an/9 + b + 5an/3 + b + an + b \\
 &= 5^s T(n/3^s) + an \sum_{i=0}^{s-1} \left(\frac{5}{3}\right)^i + sb
 \end{aligned}$$

If we now let  $n = 3^s$  for some positive integer  $s$  and assume that  $T(1) = 1$

$$\begin{aligned}
T(n) &= 5^{\log_3 n} T(1) + an \frac{(5/3)^{\log_3 n} - 1}{5/3 - 1} + b \log_3 n \\
&= n^{\log_3 5} + an \frac{n^{\log_3 5} - 1}{2/3} + b \log_3 n \\
&= n^{\log_3 5} + \frac{3a}{2} (n^{\log_3 5} - n) + b \log_3 n
\end{aligned}$$

4. (20 points) This question ask you to modify algorithm 5.2 so that  $U$  has  $m$  bits,  $V$  has  $n$  bits,  $U_2$  has  $k$  bits and  $V_2$  has  $k$  bits.
- (a) (10 points) Give the code for an efficient algorithm based on these ideas.
- (b) (10 points) Analyze the running time for the algorithm. Indicate clearly for each term in your running time formula which part of the algorithm is associated with it. In your formulas, please use the following notation.  $M(m, n)$  is the time needed to multiply an  $m$  digit number by an  $n$  digit number.  $A(m, n)$  is the time needed to add or subtract an  $m$  digit number from an  $n$  digit number. To reduce your work while having uniform grading, you are required to make the following simplifying assumptions
- A No time is needed for anything except addition and multiplication
  - B Any sum or difference always leads to a result where the number of digits for the answer is equal to the number of digits for the larger input.
  - C Any product always leads to results where the number of digits for the answer is equal to the sum of the number of digits for each input.

**Solution:** Since  $U_2$  and  $V_2$  are of size  $k$  bits, we have

$$\begin{aligned}
U &= U_1 \times 2^k + U_2 \\
V &= V_1 \times 2^k + V_2
\end{aligned}$$

The product  $U \times V$  can then be written in the form

$$\begin{aligned}
U \times V &= U_1 \times V_1 \times 2^{2k} + (U_1 \times V_2 + U_2 \times V_1) \times 2^k + U_2 \times V_2 \\
&= U_1 \times V_1 \times 2^{2k} + \left[ (U_1 + U_2) \times (V_1 + V_2) - U_1 \times V_1 - U_2 \times V_2 \right] \times 2^k + U_2 \times V_2
\end{aligned}$$

The key idea then is to replace all occurrence of  $n$  in the pseudocode for algorithm 5.2 with  $k$ . Our algorithm then can be written as If we now analyze the running time of Algorithm 2 in terms of  $M(i, j)$  and  $A(i, j)$ , the time to

**input :** The  $n$  bits number  $U = U_1 \times 2^k + U_2$  and  $m$  bits number  $V = V_1 \times 2^k + V_2$

**output:** The product  $W$  of  $U$  and  $V$  where  $W = W_1 \times 2^{3k} + W_2 \times 2^{2k} + W_3 \times 2^k + W_4$

```

2.1  $T_1 \leftarrow U_1 + U_2$  /* Size  $T_1 = \max(n - k, k)$  */;
2.2  $T_2 \leftarrow V_1 + V_2$  /* Size  $T_2 = \max(m - k, k)$  */;
2.3  $W_3 \leftarrow T_1 \times T_2$  /* Size  $W_3 = \max(n - k, k) + \max(m - k, k)$  */;
2.4  $W_2 \leftarrow U_1 \times V_1$  /* Size  $W_2 = m + n - 2k$  */;
2.5  $W_4 \leftarrow U_2 \times V_2$  /* Size  $W_4 = 2k$  */;
2.6  $W_3 \leftarrow W_3 - W_2 - W_4$  /* Size  $W_3 = \max(m + n - 2k, m, n, 2k)$  */;
2.7  $C \leftarrow \lfloor W_4/2^k \rfloor$  and  $W_4 \leftarrow W_4 \bmod 2^k$  /* Size  $C = k$  */;
2.8  $W_3 \leftarrow W_3 + C$ ,  $C \leftarrow \lfloor W_3/2^k \rfloor$ , and  $W_3 \leftarrow W_3 \bmod 2^k$  /* Size  $C = \max(m + n - 3k, m - k, n - k, k)$  */;
2.9  $W_2 \leftarrow W_2 + C$ ,  $W_1 \leftarrow \lfloor W_2/2^k \rfloor$ , and  $W_2 \leftarrow W_2 \bmod 2^k$ 

```

**Algorithm 2:** Algorithm 5.2 modified

multiply or add two numbers of size  $i$  and  $j$ , then we have the following time decompositions. For the additions on lines 2.1, and 2.2 the time taken is  $A(n - k, k)$  and  $A(m - k, k)$ , respectively. The multiplication on lines 2.3, 2.4, and 2.5 is  $M(\max(n - k, k), \max(m - k, k))$ ,  $M(n - k, m - k)$  and  $M(k, k)$ , respectively. The two subtractions on line 2.6 takes  $A(m + n - 2k, 2k) + A(\max(n - k, k) + \max(m - k, k), \max(m + n - 2k, 2k))$ . Finally, the additions on

lines 2.8 and 2.9 take time  $A(\max(m+n-2k, m, n, 2k), k)$  and  $A(m+n-2k, \max(m+n-3k, m-k, n-k, k))$ , respectively. The total running time of the above algorithm is then

$$\begin{aligned} T(m, n) &= M(n-k, m-k) + M(k, k) + M(\max(n-k, k), \max(m-k, k)) \\ &\quad + A(\max(m+n-2k, m, n, 2k), k) + A(m+n-2k, \max(m+n-3k, m-k, n-k, k)) \\ &\quad + A(\max(n-k, k) + \max(m-k, k), \max(m+n-2k, 2k)) \\ &\quad + A(n-k, k) + A(m-k, k) + A(m+n-2k, 2k) \end{aligned}$$

5. (20 points) Simplify your answer to Question 4 by omitting the addition time. Also eliminate occurrence of max and min by doing a case analysis. In particular, give the simplified formula for each case below. For each case, also give the value of  $k$  that makes the multiplication-only time as small as possible. Clearly indicate which case gives the optimum value for  $k$ . Finally, give the optimum value of  $k$ .

(a)  $2k \leq m \leq n$

(b)  $m \leq 2k \leq n$

(c)  $m \leq n \leq 2k$

**Solution:** For case a, the time under consideration is then

$$T(m, n) = M(n-k, m-k) + M(k, k) + M(n-k, m-k) = 2M(n-k, m-k) + M(k, k)$$

Since we know that  $M(m, n)$  is of order  $\Omega(m+n)$ , i.e.  $M(m, n)$  takes at least linear time, therefore,

$$M(j+1, j+1) + M(n-j-1, m-j-1) < M(n-j, m-j) + M(j, j)$$

Therefore, the minimum of  $2M(n-k, m-k) + M(k, k)$  for  $2k \leq m \leq n$  occurs when  $2k = m$ , or  $k = m/2$ .

For case b, the time under consideration is

$$T(m, n) = M(n-k, m-k) + M(k, k) + M(n-k, k)$$

In this case, we want  $M(n-k, m-k) + M(k, k)$  to be minimum while still satisfying  $m \leq 2k \leq n$ , which means that  $k = m/2$  is again the best choice. For the case c, the time under consideration is

$$T(m, n) = M(n-k, m-k) + 2M(k, k)$$

In this case, we want to make  $M(k, k)$  as small as possible while still satisfying  $m \leq n \leq 2k$ , which means that  $k = n/2$  is the choice that will makes  $T(m, n)$  smallest. From the above case analysis, we see that optimal value of  $k$  is  $m/2$ .