Solutions to Problem Set 1

Rashmi Dwaraka

September 19, 2016

1 Recursion and induction in binary codes

Problem statement:
Digital transmission protocols transmit signals using binary codes. In order to minimize the effect of errors, it is often useful to select a code such that similar signals use similar codewords.
One such code is a list of 2n n-bit strings in which each string (except the first) differs from the previous one in exactly one bit. Let us call such a list a twiddling list since we go from one string to the next by just flipping one bit.
Consider the following recursive algorithm for listing the n-bit strings of a twiddling list. If n = 1, the list is 0,1. If n ¿ 1, first take a twiddling list of (n 1)-bit strings, and place a 0 in front of each string. Then, take a second copy of the twiddling list of (n 1)-bit strings, place a 1 in front of each string, reverse the order of the strings and place it after the first list. So, for example, for n = 2, the list is 00,01,11,10, and for n = 3, we get 000,001,011,010,110,111,101,100.
Prove by induction on n that (a) every n-bit string appears exactly once in the list generated by the algorithm, and (b) each string (except the first) differs from the previous one in exactly one bit.

Solution:
Let P(n) be the mathematical statement that

- (a) every n-bit string appears exactly once in the list generated by the algorithm, and
- (b) each string (except the first) differs from the previous one in exactly one

Base Case:
When n = 1, P(1) is a 1-bit string (0,1) are generated where every n-bit string appears only once and each string (except the first) differs from the previous one in exactly one and hence satisfy P(n) statement. So, P(1) is true.

Induction Hypothesis:
Assume that P(k) generated list of strings are true for some positive integer n=k.

Induction step:
We will now show that P(k+1) is true. P(k) generates k-bit unique strings with each string (except the first) differs from the previous one in exactly one bit. From Induction hypothesis, 2^k => $2^{k+1} = 2^k.2$ indicates that P(k+1) can generate $k+1$ bit strings. For each list of 2^k strings place a 0 in front of each string. Copy the strings in reverse order and place a 1 in front of each string. This generates 2^{k+1} strings. When combined, each string (except the first) differs from the previous one in exactly one and are unique.

Hence, By Mathematical induction, P(n) is true $\forall$ positive n value.

2 Holy Numbers

Problem statement:
We say a positive integer is holy if its prime factors only include 3 and 7. As some of you may know, the numbers 3 and 7 are both considered perfect numbers under the Hebrew tradition and hence we consider these two and all of their composites to be holy.
Long time ago a sinner goes to a sacred man and asks him how he can atone for his past crimes.

The sacred man told him if you find the 777th holy number, then all your sins will be forgiven and youll be like a newborn! The sinner then goes to every mathematician of his time but none of them could help him to find that number! The mathematicians had used the following algorithm (Call isHoly for every number until you reach the nth one. But note that most numbers are not holy.):

Algorithm 1 Get Nth Holy Number

```
1: procedure GetNthHolyNumber(n)
2:    let i = 1.
3:    let cur = 3.
4:    while i 6= n + 1 do
5:       if IsHoly(cur) then
6:          i  i + 1.
7:       cur  cur + 1
8:    return cur  1
```

Algorithm 2 Is Holy Number?

```
1: procedure IsHoly(n)
2:    if n < 3 then
3:       return false.
4:       while n  mod 7 == 0 do
5:          n = n/7
6:       while n  mod 3 == 0 do
7:          n = n/3
8:    return n == 1
```

- Implement these functions as a program (in the language of your choice) and time the results for n = 50, 100, 120, 124. Your main method should take n as input and print the elapsed time, then the result for that n.
- Report the running times that you get for the above numbers in your written solutions. If you think some of them take a long time you can write > 1hour or > 10mins.
- Derive the best lower bound you can on the running time of the algorithm in terms of n? Do you think it is practical to find 777th holy number with this algorithm?

Solution:

Python implementation for the algorithm:

```
import sys,time

def GetNthHolyNumber(n):
i = 1
cur = 3
```

```
while i != (n+1):
if IsHoly(cur):
i = i+1
cur = cur + 1
return cur-1

def IsHoly(n):
if n<3:
return false
while n%7==0:
n = n/7
while n%3==0:
n = n/3
return n==1

if __name__ == "__main__":
n = int(raw_input())
start_time = time.time()
print (GetNthHolyNumber(n))
print "elapsed time =", time.time() - start_time
```

The running time for n = 50,100,120,124

- n= 50, Holy Number is 531441, elapsed time= 0.195913076401s
- n=100, Holy Number is 234365481, elapsed time= 86.3721311092s
- n=120, Holy Number is 1640558367, elapsed time = 613.078603029s
- n=124, Holy Number is 2315685267, elapsed time= 888.786185026s

The best lower bound running time of the algorithm is approximately Ω(nlogn) for all positive n value.

The *GetNthHolyNumber* algorithm's best lower bound is Ω(n). It implies that the algorithm runs for at-least n iterations to find n-th Holy number ∀ positive n value. The *IsHolyNumber?* algorithm runs for Ω(1+ $log_7 n$ + c)) and for nth iteartion, isHolyNumber? runs for $\Omega(logn)$.
It is not practical to find $777th$ Holy number using the given algorithm as it requires at-least 2245 units

3 Ordering functions

Problem statement:
Arrange the following functions in order from the slowest growing function to the fastest growing function. Briefly justify your answers. (Hint: It may help to plot the functions and obtain an estimate of their relative growth rates. In some cases, it may also help to express the functions as a power of 2 and then compare.)

$$n^{\frac{1}{3}} \quad n + \log n \quad n(\log n)^5 \quad 2^{\sqrt{logn}}$$

Solution:

- Comparing $\quad n^{\frac{1}{3}}$ *and* $n + \log n$,

$$\begin{aligned}
\lim_{x\to\infty} \frac{n^{\frac{1}{3}}}{n+\log n} &= \frac{\frac{1}{3}\cdot n^{\frac{-2}{3}}}{\frac{n+1}{n}} \\
&= \frac{n^{\frac{1}{3}}}{3\cdot(n+1)} \\
&= \lim_{x\to\infty} \frac{n^{\frac{1}{3}}}{3\cdot(n+1)} \\
&= \frac{\frac{1}{3}\cdot n^{\frac{-2}{3}}}{3} \\
&= \frac{1}{9\cdot n^{\frac{2}{3}}} \\
&= 0
\end{aligned}$$

  This implies that $n^{\frac{1}{3}} < n + \log n$

- Comparing $\quad n + \log n$ *and* $n(\log n)^5$,

$$\begin{aligned}
\lim_{x\to\infty} \frac{n+\log n}{(n\log n)^5} &= \frac{1+\frac{1}{n}}{\frac{5n\cdot(\log n)^4}{n} + (\log n)^5} \\
&= \frac{n+1}{5n\cdot(\log n)^4 + (n\cdot\log n)^5} \\
&= \lim_{x\to\infty} \frac{n+1}{5n\cdot(\log n)^4 + (n\cdot\log n)^5} \\
&= \frac{1}{20\cdot(\log n)^3 + 5\cdot(\log n)^4 + 5\cdot(\log n)^4 + (\log n)^5} \\
&= 0
\end{aligned}$$

  This implies that $n + \log n < n(\log n)^5$

- Comparing $\quad n^{1/3}$ *and* $2^{\sqrt{\log n}}$,
  Let $k = \sqrt{\log n}$. Then,

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^{1/3}}{2^{\sqrt{\log n}}} &= \lim_{k\to\infty} \frac{2^{\frac{k^2}{3}}}{2^k} \\
&= \lim_{k\to\infty} 2^{k\cdot\ (\frac{k}{3}-1)} \\
&= \infty
\end{aligned}$$

  This implies that $n^{1/3} > 2^{\sqrt{\log n}}$

Based on the above calculations, the functions can be ordered as below

$$2^{\sqrt{logn}} < n^{\frac{1}{3}} < n + \log n < n(\log n)^5$$

4 Properties of asymptotic notation

Problem statement:
Let f(n), g(n), and h(n) be asymptotically positive and monotonically increasing functions. For each of the following statements, decide whether you think it is true or false and give a proof or a counterexample.

- f(n) + g(n) = Θ(max{f(n), g(n)}).
- If f(n) = Ω(h(n)) and g(n) = O(h(n)), then f(n) = Ω(g(n)).
- If f(n) = O(g(n)), then $2^{f(n)}$ is $O(2^{g(n)})$.

Solution:

- f(n) + g(n) = Θ(max{f(n), g(n)}).
  Considering the case f(n) > g(n), as $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ which implies
  $$\lim_{n\to\infty} \frac{f(n)+g(n)}{f(n)} = \lim_{n\to\infty}(1+\frac{g(n)}{f(n)}) = 1$$
  Since the above value is finite, we can conclude the given statement is true.
- If f(n) = Ω(h(n)) and g(n) = O(h(n)), then f(n) = Ω(g(n)).

  f(n) = Ω(h(n)) can be restated as
  $$\exists\ c_1, n_0\ \textit{such that}\ f(n) \geq c_1 \cdot h(n)\ \forall\ n \geq n_0 \tag{1}$$
  g(n) = Ø(h(n)) can be restated as
  $$\exists\ c_2, n_0\ \textit{such that}\ g(n) \leq c_2 \cdot h(n)\ \forall\ n \geq n_1 \tag{2}$$
  Given (1) and (2), we need to prove that f(n) = $\Omega g(n)$
  f(n) = Ω(g(n)) can be restated as
  $$\exists\ c_3, n_0\ \textit{such that}\ f(n) \geq c_3 \cdot g(n)\ \forall\ n \geq n_2 \tag{3}$$
  From equation (1),(2) and (3), we can conclude that (3) is true based on transitive property.
- If f(n) = O(g(n)), then $2^{f(n)}$ is $O(2^{g(n)})$.

  f(n) = Ø(g(n)) can be restated as
  $$\exists\ c_1, n_0\ \textit{such that}\ f(n) \leq c_1 \cdot g(n)\ \forall\ n \geq n_0$$
  We need to prove that $2^{f(n)} = O(2^{g(n)})$ which can be restated as
  $$\exists\ c_2, n_1\ \textit{such that}\ 2^{f(n)} \leq c_2 \cdot 2^{g(n)}\ \forall\ n \geq n_1$$
  Consider the example, f(n)=2n and g(n)=n and $c_1 \geq 2$ which satisfies f(n) = O(g(n)).
  When f(n) and g(n) is substituted in $2^{f(n)} = O(2^{g(n)})$
  $$2^{2n}\ \leq c_2 \cdot 2^n\ is\ false\ since\ there\ is\ no\ constant\ that\ satisfies\ this\ equation$$
  Hence, If f(n) = O(g(n)), then $2^{f(n)}$ is not $O(2^{g(n)})$

5 Recurrences

Problem statement: Give asymptotically tight bounds for T(n) in each of the followingrecurrences. Assume that T(n) is constant for sufficiently small n.

- T(n) = 4T(n/2) + $n^{\frac{5}{2}}$
- T(n) = 2T(n/3) + T(n/4) + n.

Solution:

- T(n) = 4T(n/2) + $n^{\frac{5}{2}}$
  The given function is of the form T(n) = aT(n/b) + f(n) where, a≥1,b>1
  From the Master's theorem, a = 4, b=2, f(n)= $n^{\frac{5}{2}}$

$$n^{\log_b a} = n^{\log_2 4} = n^2 < n^{\frac{5}{2}}$$

  Considering the case f(n) is polynomial times bigger and hence by master's theorem,

$$f(n) = \Omega(n^{\log_b a+\epsilon})$$
$$T(n) = \Theta(f(n) = \theta(n^{\frac{5}{2}})$$

- T(n) = 2T(n/3) + T(n/4) + n.
  The given function is of the form T(n) = $\sum_{i=0}^{k} a_i T(n/b_i) + f(n)$ where, a≥1,b>1
  By Binomial theorem, we can say

$$\begin{aligned} T(n) &= \sum_{i=0}^{height} (\frac{1}{3} + \frac{1}{3} + \frac{1}{4})^i \cdot n \\ &= \sum_{i=0}^{height} (\frac{11}{12})^i \cdot n \\ &= \frac{1}{1-(\frac{11}{12})} \cdot n \qquad \textit{By G.P.} \\ &= 12n \\ T(n) &= \Theta(n) \end{aligned}$$

6 Modes and majority elements

Problem statement:
A mode of an array A[1..n] is an element of A that occurs most number of times in A. Thus, the mode of an array A = [7, 4, 12, 4, 1, 1, 4] is 4, which occurs three times.
(a) Given an array A[1..n] of n integers, show how a mode of A can be determined in O(n log n) time.

An array A[1..n] is said to have a majority element if more than half of its entries are the same. We would like to determine whether a given array A has a majority element, and if so, find the element. Unlike in part (a), however, we will not restrict the elements to be integers. In fact, we assume that the elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form is A[i] > A[j]?; only questions of the form is A[i] = A[j]? can be answered.

(b) Show how to solve the majority element problem in O(n log n) time. (Hint: Split the array into two arrays A1 and A2 of half the size. Use a divide-and-conquer approach that finds the majority element of A, if it exists, using the knowledge of majority elements of A1 and A2, if they exist.)

(c) Can you give a linear-time algorithm? (Hint: Another divide-and-conquer approach is as follows: (a) pair up the elements of A arbitrarily, to get n/2 pairs; (b) if two elements of a pair are different, then discard both of them, else keep just one of them. Show that after this procedure there are at most n/2 elements left, and that they have a majority element if A does.)

Solution:

- **Algorithm (6a): Find-Mode(A,p,q)**
  A is an Array, p and q are first and last indices

  ```
  1:  if p=q return A[p]
  2:  if p<q
  3:     Sort A[p..q] using Merge Sort
  4:     let counter=0, max-counter=0 and mode = A[0]
  5:     for i=p to q-1 iterate over A
  6:        if A[i] == A[i+1]
  7:           counter++
  8:        else if max-counter<counter
  9:              max-counter=counter
  10:             mode=A[i]
  11:             counter=0
  12:    return mode;
  ```

  The Merge sort runs in $O(n \log n)$ and the mode calculation requires $O(n)$ and approximately the entire algorithm takes $O(n \log n)$

- **Algorithm (6b): Majority-Element(A,p,q)**
  A is an Array, p and q are first and last indices

  ```
  1:  if p=q return A[p]
  2:  if p<q
  3:     m = $\left\lfloor \frac{p+q}{2} \right\rfloor$
  4:     l= Majority-Element(A,p,m)
  5:     r= Majority-Element(A,m+1,q)
  ```

```
6:      if l==r
7:          return l
8:      $lcount = count(A, p, q, l)$
9:      $rcount = count(A, p, q, r)$
10:      $if\ lcount > rcount\ and\ lcount > \frac{q-p}{2}$
11:          return l
12:      $else\ if\ rcount > lcount\ and\ rcount > \frac{q-p}{2}$
13:          return r
14:      else return No Majority
```

Algorithm (6b): count(A,p,q,e)

```
1: let count =0
2:    for i from p to q in A
3:        if A[i] == e
4:            count++
5: return count;
```

The Majority-Element runs for $O(nlogn)$ and the count runs in a linear time of $O(n)$ and overall the algorithm requires $O(nlogn + c)$ which approximates to $O(nlogn)$

- **Algorithm (6c): Majority-Element-Linear(A,p,q)**
  A is an Array, p and q are first and last indices

```
1:    m = SubMajority-Element-Linear(A)
2:    if count(m > $\frac{(p+q)}{2}$ + 1)
3:        return m
4:    else No Majority
```

Algorithm (6c): SubMajority-Element-Linear(A)

```
1:    if size of A == 1 return A[0]
1:    temp = Array of size A
2:    for i = 0 to size of A-1
3:        if A[i] == A[i+1]
```

```
4:         i++       **pair wise iteration
5:         append A[i] to temp
6:    return SubMajority-Element-Linear(temp)
```

Let 'm' be the majority element in A[] A is divided into pairs, where one type of pair formed are equal and other pair include different elements. The latter can have at most half the elements which are 'm'. This means that the former must contain 'm' as a majority. Hence, by only considering one of the element from same element pairs we will end up with the majority element 'm'