COMP 4030/6030 - Spring 2017
Exam 3

Name: Madiian Pope

Note: Do not write on the backs; only front pages are digitized and graded.

20

1. (*20 points*) Use the Master's theorem to find the complexity of the following functions (assuming $T(1) = 1$):

1. $T(n) = 4n + 4 \cdot T(\frac{n}{4})$
2. $T(n) = 10n^2 + 7 \cdot T(\frac{n}{3})$

1. $a=4$, $b=4$, $d=1$; $\log_4 4 = 1$, $d=1$; $\Theta(n\log n)$

2. $a=7$, $b=3$, $d=2$; $\log_3 7 < 2$, $d=2$; $\Theta(n^2)$

2. (*15 points*) Determine (write it out clearly) the running time equation in terms of n, which is the number of items in the input list L. Based on that equation, find the running time complexity of the function **foo**. Hint: Pay attention to the running time of Python slicing.

$T(n) = c \times \frac{n}{2} + T(\frac{n}{2}) + b$
$= c \times n + T(\frac{n}{2})$

15

```
def foo(L):
  if len(L) <= 1:
    return 1
  A = L[0: len(L)//2]
  return len(L)//2 + foo(A)
```

b

n/2 → A = L[0: len(L)//2]

← $b + T(\frac{n}{2})$

$a=1$, $b=2$, $d=1$; $\log_2 1 = 0$, $d=1$; $\Theta(n)$

3. (*15 points*) Determine (write it out clearly) the running time equation in terms of n, which is the number of items between two indices *left* and *right* of the input list L. Based on that equation, find the running time complexity of the function **bar**. Hint: Pay attention to the running time of Python slicing.

$T(n) = c_1 \times n + c_2(\frac{n}{2}) + 2T(\frac{n}{2}) + b$
$= c \times n + 2T(\frac{n}{2})$

15

```
def bar(L, left, right):
  if left >= right:
    return 1
  sum = 0
  for i in range(left, right+1):
    sum = sum + i*i
  A = L[0: len(L)//2]
  return len(L)//2 + bar(A) * bar(A)
```

b

sum = sum + i*i ← c

A = L[0: len(L)//2] ← n/2

$L(\text{left}:(\text{left}+\text{right})//2)$

$b + 2T(\frac{n}{2})$

← bar(A, 0, len(A-1))

left?, (left+right)//2?

$a=2$, $b=2$, $d=1$; $\log_2 2 = 1$, $d=1$; $\Theta(n\log n)$

4. (*10 points*) Rewrite the function **bar** in the previous problem so that your revised version is faster. (Your revised function must, of course, do the same thing as **bar** does.) Use the Master's theorem to explain why your revised version is faster.

```
def bar(L, left, right):
  if left >= right:
    return 1
  sum = 0
  A = []
  for i in range(left, right +1):
    sum = sum + i*i
    A += L[i]
  return len(L)//2 + 2 * bar(A)
```

b

$b + T(\frac{n}{2})$

$T(n) = c \times n + T(\frac{n}{2}) + b$
$= c \times n + T(\frac{n}{2})$

$a=1$, $b=2$, $d=1$; $\log_2 1 = 0$, $d=1$; $\Theta(n)$

9

only calls bar(A) once instead of twice, this makes it quicker, and by multiplying by 2 get the same result

7. (*20 points*) The scenario for this problem is as follows. There are n people and n jobs. Each person has a ranking for the n jobs (smaller rank is better). The goal is to assign each person to each job so that (i) every person has a job and (ii) every job is assigned to exactly one person. The input of this problem is a list of preference list. P[i] is the list of preference of person i. To be consistent the ranking and numbering start from 0 and ends at n-1. Example:

1. P[0] = [1,0,2]. This means person 0 likes job 1 the most, followed by job 0, followed by job 2.
2. P[1] = [2,1,0]. This means person 1 likes job 2 the most, followed by job 1, followed by job 0.
3. P[2] = [0,2,1]. This means person 2 likes job 0 the most, followed by job 2, followed by job 1.

In this example, the best assignment is [1,2,0], which means person 0 is assigned to job 1, person 1 is assigned to job 2, and person 2 is assigned to job 0. The total cost for this assignment is P[0][1]+P[1][2]+P[2][0] = 0, which is the best possible (a smaller rank is better implies a smaller cost is better).
Your task is to write a backtracking algorithm to find the best assignment (one with minimal total rank sum). Do that by completing the partially written code below.

```
P = get_preference()  # Assume this function is already defined.
N = len(P)
Solution = [-1]*N      # Hint: this is a permutation problem.
optimal = N*N          # N*N is worse than the worst solution.  So it's a safe initial value.

def FindJob(i):
    global optimal
    # Your code goes here
```

20

```
if promising(i):
    if i == N-1:
        # solution is filled
        cost=0
        for x in Solution:
            cost += P[x][Solution[x]]
        # x is person, Solution[x] is job they get
        if optimal > cost:
            optimal = cost

    else:
        for x in range(N):
            Solution[i+1] = x
            FindJob(i+1)
```

```
def promising(i):
# is job taken check
    for x in range(i):
        count=0
        for y in Solution:
            if Solution[y]==x:
                count += 1
        if count>1:
            return False
    return True
# if job appears in
# solution more than
# once, it is not promising
# solution
```

```
FindJob(-1)      # This will invoke the backtracking algorithm
print("Optimal solution is", optimal)
```

5. (*10 points*) Consider these three different algorithms:

1. Algorithm A takes as input a list L of n items. It makes two recursive calls on two sublists of L; each sublist has $\frac{n}{2}$ items. Then, Algorithm A combines the result returned by the two recursive calls in n steps.
2. Algorithm B takes as input a list L of n items. It makes four recursive calls on four sublists of L; each sublist has $\frac{n}{2}$ items. Then, Algorithm B combines the result returned by the four recursive calls in only 1 constant step.
3. Algorithm C takes as input a list L of n items. It makes only one recursive call on a sublist of L with $\frac{n}{2}$ items. Then, Algorithm C does additional processing in n^2 steps.

Explain which algorithm gives the fastest running time.

1. $T(n) = c \cdot xn + 2T(\frac{n}{2})$
$a=2$ $\log_2 2 = 1$
$b=2$
$d=1$ $d=1$
$\Theta(n\log n)$

2. $c + 4T(\frac{n}{2})$
$a=4$ $\log_2 4 = 2$
$b=2$
$d=1$ $d=1$
$\Theta(n^2)$

3. $c \times n^2 + T(\frac{n}{2})$
$a=1$ $\log_2 1 = 0$
$b=2$
$d=2$ $d=2$
$\Theta(n^2)$

10

$\Theta(n\log n)$ runs faster than $\Theta(n^2)$ so Algorithm 1 (A) has the fastest running time

6. (*20 points*) This problem is known as a *bin packing* problem. The input is a list of weights $w_1, \cdots, w_n$ and a size C. A valid packing is a set of weights that total up to exactly C. For example, given weights 3, 5, 10, 2 and $C = 7$, the output is True (because 5+2=7). If $C = 9$, the answer is False (note: no repetition). Your task is to write a backtracking algorithm to print all valid packings. Do that by completing the partially written code below.

```
Weight = get_weights()   # Assume get_weights is already defined. It returns a list of weights.
C = get_capacity()       # Assume get_capacity is already defined.
N = len(Weight)
Solution = [-1]*N

def k_pack(i):
    # Your codes go here
    if promising(i):
        if i == N-1:
            current_weight = 0
            for x in range(i):
                if Solution[x] == True:
                    current_weight += Weight[x]
            if current_weight == C:
                print(Solution)
        else:
            Solution[i+1] = True
            k_pack(i+1)
            Solution[i+1] = False
            k_pack(i+1)

def promising(i):
    current_weight = 0
    count = 0
    for x in range(i):
        if Solution[x] == True:
            current_weight = Weight[x]
    if current_weight > C:
        return False
    return True
```

20

```
k_pack(-1)   # this will print out all valid packings
```

1