CSC 120 Algorithms and Data Structures **Spring Semester, 2002**

Answer Sheet Review 1

1.
```
for (i=0; i<=n-1; i++){          // Sum n times 5.n.c ==> 5.n.n.c
   for (j=0; j< 5; j++) {        // Sum 5 times n.c  ==> 5.n.c
       for (k=0; k<n; k++) {     // Sum n times c ==> n.c
      loop body                  // c
}
for (i=0; i<=n-1; i++){          // Sum (i+1)c for all i from 0 to
n-1
   for (j=0; j<=i; j++){         // Sum i+1 times c ==> (i+1)c
      loop body                  // c
   }
}
```

- $$\begin{aligned} T(n) &= 5n^2c + \sum_{i=0}^{n-1}(i+1)c = \\ &= 5n^2c + c\sum_{i=0}^{n-1} i - nc = \\ &= 5cn^2 + \frac{(n-1)n}{2} - nc = \\ &= (5c+0.5)n^2 - (c+0.5)n = \Theta(n^2) \end{aligned}$$

2. Have to compare algorithms with worst-case time complexities estimated as follows: $\Theta(n^2)$, $O(n\sqrt{n}), \Omega(n \log n)$.

   • An algorithm that is bounded from above by $O(n\sqrt{n})$ is definitely better than the $\Theta(n^2)$ algorithm, so we have to choose between the last two algorithms.

   It is true that the function $n \log n$ has lower asymptotic growth rate than $n\sqrt{n}$. On the other hand the last algorithm is bounded from below by $n \log n$ which does not guarantee that it is better than the $O(n\sqrt{n})$ algorithm. In an extreme case, as an example, a $\Theta(2^n)$ algorithm is $\Omega(n \log n)$. Thus, with the information we have, the second algorithm, $O(n\sqrt{n})$, should be selected.

3. (a) Assume that f and g take only positive values. Is $f(n)+g(n) = O(\max\{f(n), g(n)\})$? Justify your answer.

   Note that you cannot do a prove, by "example" here, i.e. it is not enough to show that for some pair of functions the statement is true. You must show that for every pair, it is true.

   • For any n, $f(n) + g(n) \leq 2\max\{f(n), g(n)\}$.

   Thus if we choose $n_0 = 1$, and $C = 2$, the definition of Big-Oh is satisfied. Thus

   $$f(n) + g(n) = O(\max\{f(n), g(n)\})$$

   The answer is "yes".

(b) If $f(n) = O(g(n))$ then does it follow that $g(n) = O(f(n))$?

• "No", since there are functions for which $f(n) = O(g(n))$, and $g(n) \neq O(f(n))$. For example, take $f(n) = 1$ and $g(n) = n$.

Note that since we found one example for which

$$f(n) = O(g(n)) \text{ does not imply } g(n) = O(f(n)).$$

The answer is "No". Proof by counterexample is ok here.

4. What is the asymptotic time complexity of the following divide-and-conquer algorithm. You may assume that n is a power of 2. (NOTE: It doesn't matter what this algorithm does.)

Let `T(n)` be the run time of `foo()` on an input of size `n`.

```
foo(n,A){
  let B be an array of size n
  if (n==1){                          //  2, base case
    B[0] = 1;
    return B;
  }
  let AL be an array of size n/2
  let AR be an array of size n/2
  for (i=0; i <= (n/2)-1; i++)       //  Theta(n/2)=Theta(n)
    AL[i] = A[i];
  for (i=n/2; i <= n-1; i++)         //  Theta(n/2)=Theta(n)
    AR[i- (n/2)] = A[i];

  BL = foo(n/2,AL);                  // T(n/2)
  BR = foo(n/2,AR);                  // T(n/2)

  for (i=0; i<= n-1 ; i++)           // Theta(n)
     B[i] = 1;

  for (i=n/2; i <= n-1; i++)             // Theta(n^2)
     for (j=0; j <= (n/2)-1; j++)        // Theta(n/2) = Theta(n)
        if (A[i-n/2] > A[j])             // c
           B[i] = max(B[i],BR[i]+BL[j]);
  return B;                              // 1
}
```

• Note that when we add up Big-Thetas, the complexity of the sum is the same as the highest order Big-Theta.

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= 2T(n/2) + \Theta(n^2) \end{aligned}$$

We use the Master method, $a = 2 \geq 1, b = 2 > 1, k = 2 \geq 0, p = 0 \geq 0$, since $a < b^k$, we have case 3, and thus

$$T(n) = \Theta(n^2)$$

5. (a) Just so that we can write the algorithm in C++, we must decide on a data type for the array elements. Without loss of generality we assume that the elements of the array are integers.

```
//Returns the index of the min element in A between indices p and r.
// p <= r
int NewMinAlgo(int A[],int p,int r)
{
  // base case
  if (p==r)
     return p;  // if one element array, that element is the minimum

  // divide
  mid = (p+r)/2;

  //conquer
  lmin = NewMinAlgo(A,p,mid);   // get min in left half
  rmin = newMinAlgo(A,mid+1,q); // get min in right half

  //combine
  if (A[lmin]<A[rmin])
     return lmin;
  else
     return rmin;
}
```

(b) Let `T(n)` be the run time of `newMinAlgo()` on an input of size n.

```
int NewMinAlgo(int A[],int p,int r)
{
  if (p==r)                      // 1, base case
     return p;
  mid = (p+r)/2;                 // 1
  lmin = NewMinAlgo(A,p,mid);    // T(n/2)
  rmin = newMinAlgo(A,mid+1,q);  // T(n/2)
  if (A[lmin]<A[rmin])           // 1
     return lmin;
  else
     return rmin;
}
```

- $$T(n) = 2T(n/2) + \Theta(1)$$

  By the master method, with $a = 2 \geq 1, b = 2 > 1, k = 0 \geq 0, p = 0 \geq 0$, since $a = 2 > b^k = 1$,we have case 1. Thus

  $$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

(c) The worst case complexity of the brute force algorithm is

```
int BruteMinAlgo(int A[],n) {
  min=A[0];                    // 1
  for (i=0; i<n ; i++)         // Sum n times 1
     if (A[i] < min)           // 1
        min=A[i];
  return min;                  // 1
}
```

- $$T_{brute}(n) = 2 + \sum_{i=0}^{n-1} 1 = n + 2 = \Theta(n)$$

(d) • Thus the two algorithms have the same asymptotic time complexity. I would chose the brute force algorithm since it does not have big hidden constants, and it is not recursive, so it is more efficient in memory use.

6. (a) Let `T(n)` denote the run time of `Goo()` on an input of size n.

```
Goo(positive integer n)
  if (n==1)                    // 1, base case
     return 1;
  a = Goo(n-1) + Goo(n-1)      // 2T(n-1)+1
  return  a;                   // 1
```

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n-1) + 2 \end{aligned}$$

Notation: `2^i` means 2^i.

```
            2              level  #nodes     #statments in level
              T(n)           0       1          2.2^0
       2     /    \
        T(n-1)    T(n-1)     1       2          2.2^1
 2     /      \
  T(n-2)     T(n-2)  ....    2       2^2        2.2^2
    .          .             .       .          .
    .          .             .       .          .
 2  .          .             .       .          .
  T(2) ....                  k-1     2^(k-1)    2.2^(k-1)
1 /   \ T(1) T(1) ......     k       2^k        1.2^k
```

Summing up the number of statements in all levels (last column), we obtain

$$\begin{aligned} T(n) &= 2^k + \sum_{i=0}^{k-1}(2.2^i) = && (1) \\ &= 2^k + 2\sum_{i=0}^{k-1} 2^i = && (2) \\ &= 2^k + 2(2^k - 1) = && (3) \\ &= 3.2^k - 2 && (4) \end{aligned}$$

To find k we examine a leaf node $T(1) = T(n-k)$, solving for k,

$$1 = n - k$$

we obtain

$$k = n - 1.$$

Substitute, $k = n - 1$ in equation (4), the run time is

- $T(n) == 3.2^{n-1} - 2 = (3/2).2^n - 2.$

- Check:

$$\begin{aligned} \text{By recurrence :} \quad & \text{By solution :} && (5)\\ T(1) = 1 \quad & T(1) = 1 && (6)\\ T(2) = 2 + 2 = 4 \quad & T(2) = (3/2)2^2 - 2 = 4 && (7)\\ T(3) = 8 + 2 = 10 \quad & T(3) = (3/2)2^3 - 2 = 10 && (8) \end{aligned}$$

For the asymptotic time complexity we obtain,

- $T(n) = (3/2).2^n - 2 = \Theta(2^n).$

- We **cannot** use the master method to solve the recurrence since the recurrence does not use $T(n/b), b > 1$.

(b) Here is the run time of 6.(b):

```
Pan(positive integer n)
  result=0                      // 1
  for (i=2; i<=n; i++)          //add (c.i+1) for i=2 to n, c is a constant
    result = result + Foo(i)    //  c.i + 1
  return result                 // 1
```

$$T(n) = 2+\sum_{i=2}^{n}(1+ci) = 2+(n-1)+c\sum_{i=2}^{n} i = 1+n-c+c\sum_{i=1}^{n} i = 1+n-c+c\frac{n(n+1)}{2}$$

And finally, the run time is

- $T(n) = 0.5cn^2 + (1 + 0.5c)n - c$

The asymptotic time complexity is

- $T(n) = \Theta(n^2)$

(c) Let `T(n)` be the run time of `Merge()` on an input of size n.

```
Merge(positive integer n)
  if ( n==1)                      // 1, base case
     return 1
  result = 0                      // 1
  temp = Merge(n/2) + Merge(n/2)// 2T(n/2) + 1
  for (i=1; i <= n; i++)          // n.1
        temp = temp + i
  Homer(n)                        // Theta(1)
  return temp                     // 1
```

The run time is

- $T(n) = 3 + n + \Theta(1) + 2T(n/2) = 2T(n/2) + \Theta(n),\ T(1) = 1$

This recurrence is exactly the same as the one for Closest pairs and Merge Sort. See the recursion trees we did in class for those.

For this case, we could also use the master method: $a = 2, b = 2, k = 1, p = 0$, case 2, thus the asymptotic time complexity is

- $T(n) = \Theta(n \log\, n)$.

7. Give tight asymptotic bounds for $T(n)$ in each of the following recurrences. Assume that $T(1) = 1$. Use the master method.

   (a) $T(n) = 16T\left(\frac{n}{2}\right) + \Theta(1)$

   - $a = 16, b = 2, k = 0, p = 0$, since $a = 16 > 1 = b^k$, case 1, thus

   $$T(n) = \Theta(n^{\log_2 16} = \Theta(n^4).$$

   (b) $T(n) = T\left(\frac{3n}{4}\right) + \Theta(\sqrt{n})$

   - $a = 1, b = 4/3, k = 1/2, p = 0$, since $a = 1 < \sqrt{4/3} = b^k$, case 3, thus

   $$T(n) = \Theta(\sqrt{n}).$$

   (c) $T(n) = 9T(n/9) + \Theta(n \log^2 n)$

   - $a = 9, b = 9, k = 1, p = 2$, since $a = 9 = 9 = b^k$, case 2, thus

   $$T(n) = \Theta(n \log_9^3 n).$$

8. • Merge sort:

```
                120, 3, 17, 11, 101, 15,  8, 10
                 (2, 8, 10, 11,  15, 17, 101, 120)
                            /     \
                           /       \


               120,  3, 17, 11       101, 15,  8, 10
                (3, 11, 17, 120)      (8, 10, 15, 101)
                 /        \                /        \
                /          \              /          \
           120, 3          17, 11     101, 15         8, 10
           (3,120)         (11,17)    (15,101)        (8,10)
          /       \        /    \      /     \        /     \
        120        3      17     11    101    15      8      10
```

• Quick sort. Each node in the tree is labeled with the collection of the array elements on which the call is made. The value of the pivot is written under the node (underlined with $\mathbb{E}$. At each node in parenthesis we give the sorted collection.

```
120, 3, 17, 11, 101, 15,  8, 10
(3, 8, 10, 11, 15, 17, 101, 120)
           10
           ^^^
        /         \
       /           \
     3,8        120, 17, 11, 101, 15
    (3,8)      (11, 15, 17, 101,120)
      8                  15
     ^^^                 ^^^
    /   \              /    \
   3    ""           11    120, 17, 101
                           (17, 101, 120)
                                101
                                ^^^
                               /   \
                              17   120
```