CS 410 Summer 2000
Solutions to Homework 1

1. Here is the pseudo-code which sorts A using selection sort. Note that it assumes the only data in each item is the key field. There are several other ways to do this. If you don't mind destroying array A you can replace each item placed in B with INFTY in A. Then you don't need to worry about the duplicates problem. Another possibility is to maintain a 3rd array which indicates if an element in A has been copied to B yet. All in all, I think the solution in part (c) is more natural.

```
procedure selection-sort(A)

prev_min=-INFTY
/* prev_min gives the value of the minimal element found in the last
phase. so the next minimal element will be the one which is the least
among all elements strictly greater than prev_min

index_b=0      /* this is the index of array B */

while (index_b < n) do  /* check whether all elements have been put into B */
begin
   j=1
   while (a[j]<=prev_min)
      ++j

   min=a[j]  /* find the first element which is greater than prev_min. this
                is a candidate for being the next smallest */
   no_duplicates=1  /* to record the number of duplicates*/

   for (k=j+1 to n) do
      if (a[k] > prev_min)
         if (a[k] < min)

            /* the min was not really the "min" so replace it */
            min=a[k];
            no_duplicates=1

         else
            if (a[k]==min)
               /* a duplicate has been found */
               ++no_duplicates

   /* copy the next minimum element found to B */
   for (l=1 to no_duplicates)
      b[index_b+l]=min;

   index_b=index_b + no_duplicates
      /* update index of b */
   prev_min=min /* update prev_min */
end
```

Time taken by selection sort is $\theta(n^2)$ because there are two nested loops (index_b and k), each of which perform a constant number of steps. The other loops (a[j] and l) are of lower order.

(a) *Advantage of Bubble sort :* Consider the following simple modification to bubble sort:

```
procedure bubble-sort(A)

   // iterate n-1 times (each iteration causes largest value to
   // "bubble" to end of array
   for i = 1 to n-1 do
      any_swaps=FALSE      // flag notes if any swaps occur in this iteration

      // for each position in the array...
      for j=1 to n-i do
      begin
         // if element at this position is larger than its right neighbor...
         if A[j] > A[j+1] then
            // swap them (and note that a swap has occurred)
            any_swaps=TRUE
            swap(A[j],A[j+1])
      end
      // no swaps this iteration means array is already sorted
      if any_swaps==FALSE then EXIT
```

After any number of iterations, if A happens to get sorted, then there won't be any swaps in the next iteration. Conversely, if no swaps are made in a certain pass, then A is sorted. The latter condition is detected using the *any_swaps* flag and hence bubble sort takes almost linear time when A is *almost* sorted.

Advantage of selection sort : Selection sort makes at most O(n) swaps irrespective of the input. Whereas bubble sort can make as much as $\theta(n^2)$ swaps in the worst case. (when A is in non-increasing order)

(b) Both are $\Theta(n^2)$ algorithms, so they're the same.

(c)

```
procedure selection-sort(A)

   // for each position in the array...
   for i = 1 to n-1 do
      // find the minimum value in the remainder (unsorted portion)
      // of the array
      min=i
      for j=i+1 to n do
      begin
         if A[min] > A[j] then min=j
      end
      // put the minimum value at the beginning of the unsorted portion
      // of the array.  This position is now part of the sorted portion.
      swap(A[i],A[min])
```

2. Let $f(n) = 2^n - 100n^2$. We are interested in the least positive of n such that f(n) is positive. It is easy to show that if $f(n_0) > 0$ then $f(n) > 0\ \forall n \geq n_0$. We have f(14)= -3216 and f(15)=10268. Therefore the least value of n is 15.

3. $\lg\lg n$, $\lg n$, $\lg n^2$, $\sqrt{n}$, $n/\lg n$, n , n^2 , $n^2 + n\lg n$, $10^{\sqrt{n}}$, 2^n , e^n
$\lg n = \theta(lgn^2)$
$n^2 = \theta(n^2 + n\lg n)$

4.(a) False.
Counterexample : $f(n) = n$ and $g(n) = n^2$

(b) False
Counterexample : $f(n) = n$ and $g(n) = n^2$ ($\forall n \geq 1$, min(f(n),g(n)) = f(n))

(c) True
Proof : max((f(n) , g(n)) $\leq$ f(n) + g(n) ,$\forall n$ (as f and g are positive functions) Therefore max(f(n),g(n)) = O(f(n) + g(n)) where c=1 and $n_0 = 0$.
Also, $2\max$(f(n),g(n)) $\geq$ f(n) + g(n) , $\forall n$. Therefore max(f(n),g(n)) = Ω(f(n) + g(n)) where c=1/2 and $n_0 = 0$.

(d) True
Proof : f(n) = O(g(n))
$\Rightarrow f(n) \leq c\, g(n)\ \ \forall n \geq n_0$.......(1)
$\Rightarrow 2\, f(n) \leq c\, 2\, g(n)\ \ \forall n \geq n_0$
$\Rightarrow 2\, f(n) = O(2\, g(n))\ \ n_0$, c are the same as in (1).

(e) True
Proof : f(n) = $2\, f(n)/2\ \ \forall n$
$\Rightarrow f(n) \leq 2\, f(n)/2\ \ \forall n$
$\Rightarrow 2\, f(n) = O(f(n)/2)\ \ n_0 = 0$ and c=2.

(f) False
Counterexample : $f(n) = 2^{2\,n}$. Therefore $f(n/2) = 2^n$. If f(n)=O(f(n/2)) then there exists constants c and n_0 such that
$2^{2\,n} \leq c\, 2^n\ \ \forall n \geq n_0$
$\Rightarrow c \geq 2^n$
$\Rightarrow c$ is not a constant, which is a contradiction.

(g) True
Proof: $2^{n+1} = 2\, *\, 2^n\ \forall n$
$\Rightarrow 2^{n+1} \leq 2\, *\, 2^n$
$\Rightarrow 2^{n+1} = O(2^n)\, ,\ n_0 = 0,\ c = 2$.

(h) False
Counterexample : see (f)

5. (a) Number of operations is $(3.5*10^9)^2 = 3.5^2*10^{18}$. Therefore time needed is 3.5^2*10^9 seconds or approximately 388 years!

(b) Number of operations is roughly $111*10^9$. Therefore time needed is 111 seconds.

6. (a) Recall one of the definitions of “o” :
$f(n) = o(g(n) \iff \forall c > 0\, \exists n_0$ such that $0 \leq f(n) < c\, g(n)$.

The crucial difference w.r.t to "O" is the **for all quantifier**. Intuitively, the function $f(n)$ becomes *insignificant* to $g(n)$ after a certain time. Hence from the definition we can conclude $o(f(n)) \subseteq O(f(n))$. Also, f(n) = O(f(n)) and $f(n) \neq o(f(n))$. Therefore $o(f(n)) \subset O(f(n))$.

(b) $\lim_{n\to\infty} \frac{n^b}{a^n} = 0$ (By repeated use of L'Hospital's rule.) Therefore $n^b = o(a^n)$. (if $b \leq 0$, it is clear that the limit is 0, since the numerator decreases as the denominator increases. if $b > 0$ then if we differentiate $\lceil b \rceil$ times, it is easy to conclude the resulting fraction is 0.

REMARKS :

i. Many failed to notice the "for all" quantifier.

ii. It is **NOT** $\exists n_0 \, \forall c > 0 \ldots$. This would mean a single n_0 works for all $c > 0$. It's OK to choose different n_0's based on the c. Intuitively, as you decrease c, it takes longer to achieve this smaller ratio of $f(n)$ to $g(n)$, so we need larger n_0.

7. $\int_{x=0}^{n} x^3\, dx \ \leq \ \sum_{k=1}^{n} k^3 \ \leq \ \int_{x=1}^{n+1} x^3\, dx$

$\Rightarrow n^4/4 \leq \ \sum_{k=1}^{n} k^3 \ \leq \ ((n+1)^4 - 1)/4$

$\Rightarrow \sum_{k=1}^{n} k^3 \ = \ \theta(n^4)$

8. Assume $T(n) \leq c\, n \ \ \forall n < m$. T(1)=1 $\Rightarrow c \geq 1$. Consider T(m).
$a_1 + a_2 < 1 \ \Rightarrow \ a_1\,, a_2 \ < 1 \ \Rightarrow \ a_1 m\,, a_2 m \ < m$.
Therefore by induction hypothesis, $T(a_1 m) \leq c\, a_1\, m$ and $T(a_2 m) \leq c\, a_2\, m$. Therefore
$T(m) \leq c\, a_1\, m + c\, a_2\, m + b\, m$
$\leq m(c\, a_1 \ + c\, a_2 \ + b)$
We want this to be $\leq c\, m$. It will be, if $c \geq c\,(a_1 + a_2) \ + \ b$

$c \geq c\,(a_1 + a_2) \ + \ b \iff$
$c \ - \ c\,(a_1 + a_2) \geq b \iff$
$c\,(1 - (a_1 + a_2) \geq b \iff$
$c \geq \frac{b}{1-(a_1+a_2)}$

To perform this last step we are making use of that fact $a_1 + a_2 < 1$, so we are dividing by a positive number, so the inequality doesn't change.
So, we can choose c (initially) so that $c \geq \max(1, \frac{b}{1-(a_1+a_2)})$, so
$T(m) \leq c\, m$

REMARKS : 1. Once you assume that $T(n) \leq c\, n$, it means that you have **fixed** c . You cannot modify c in the induction step. You can figure out how you need to choose c, but must not change it in the inductive step.
2. Many of you haven't bothered about the **base case**. While using the induction argument remember that n_0 will be the value of n at the base case.

9. Each interior node of the recurrence tree will have **4** children and the height of the tree $h = \lfloor \lg n \rfloor$. The number of leaves is 4^h. Work done at an interior node at a depth k,

$0 \leq k < h$ is $\leq \frac{n}{2^k}$. Work done at a leaf is 1. Total number of nodes at depth k is 4^k. Therefore total work done at depth k is $\leq 4^k \frac{n}{2^k}$. Therefore total work done is
$\leq$ work done at all the interior nodes + number of leaves
$\leq \sum_{k=0}^{k=h-1} 4^k \frac{n}{2^k} + 4^h$
$\leq n \sum_{k=0}^{k=h-1} 2^k + 4^h$
$\leq n(2^h - 1) + (2^h)^2$
$\leq n^2 + n^2$ (using the fact $2^h \leq n$)
$\leq 2n^2$

REMARKS : Most of you have got the right answer by just summing up from $i = 0$ to $i = \lg n$. This doesn't give you the right answer in the general case. The way the above recurrence is evaluated is
T(n) = *total work done at the internal nodes* + *total work done at the leaves*
So if we have T(n) = aT(n/b) + f(n) and T(1)= c , then $f(n)$ comes into play only at the internal nodes, but c comes into play at the leaves.

10. From the master theorem one can see that $T(n) = \theta(n^{\log_2 7})$. If $a > 16$, then case 1 of the master theorem will also hold for T', and $T'(n) = \theta(n^{\log_4 a})$, Therefore for $T'(n)$ to be asymptotically lesser than $T(n)$, we need to have
$\log_4 a < \log_2 7$
$\Rightarrow \log_4 a < \log_4 49$
$\Rightarrow a < 49$
$\Rightarrow a = 49$

11. Let $f(n)$ be the n^{th} Fibonacci number. Then from *page 37 of the text* we have
$f(n) = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$
$\leq \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$
$\Rightarrow f(n) = O((\frac{1+\sqrt{5}}{2})^n)$ where $c = \frac{1}{\sqrt{5}}$

The following 2 facts are easy to prove
$|(\frac{1-\sqrt{5}}{2})^n)| \leq 1$ (since $\sqrt{5} < 3$)
$(\frac{1+\sqrt{5}}{2})^n \geq 2 \ \forall n \geq 2$ (since $\frac{1+\sqrt{5}}{2} > 3/2$)
From the above 2 facts , it follows that
$f(n) \geq \frac{1}{2\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$
$\Rightarrow f(n) = \Omega((\frac{1+\sqrt{5}}{2})^n)$ where $c = \frac{1}{2\sqrt{5}}$ and $n_0 = 4$.
Hence $f(n) = \theta((\frac{1+\sqrt{5}}{2})^n)$