CS 6150: Recurrences and Recursion

Resubmission deadline: Oct 8, 2014

This assignment has 8 questions, for a total of 100 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Recurrences and Recursion	25	
Pancakes, mmmm...	10	
Leaning Tower of Hanoi	10	
StoogeSort	15	
Finding $A[i] = i$	10	
Equivalence testing	10	
Medians	10	
Hidden Surface Removal	10	
Total:	100	

Question 1: Recurrences and Recursion .... **[25]**

Solve each of the following recurrences. You may use any method you like, but please show your work. In each recurrence, you may assume convenient starting values for $T(0)$ or $T(1)$ unless otherwise specified.. Note that c is an undetermined constant.

Solving a recurrence means that you provide a bound of the form $T(n) = O(f(n))$ for a specific f. Tight bounds get full credit: for example, if the recurrence is $T(n) = 2T(n/2) + cn$, the answer $T(n) = O(n^2)$ is correct but not precise enough, and you will not get full credit for it.

(a) **[3]** $T(n) = 7T(n/2) + cn^2$

Solution: Define $f(n) = cn^2$. Using the master's theorem, since $a = 7$ and $b = 2$,

$$af(n/b) = 7 * f(n/2) = \frac{7cn^2}{4}$$

Suppose n is very large that is $\forall n >> n_0$. So, when $c > 0$, $\exists m > 1$, making $af(n/b) \leq mf(n)$, and implying that $T(n) = \Theta(n^{\log 7})$.

(b) **[3]** $T(n) = T(n/2) + T(3n/10) + cn$

Solution: Let's have a try that $T(n) = O(n)$. Suppose $T(n) \leq mn$, then $T(n/2) \leq mn/2$ and $T(3n/10) \leq 3mn/10$. Hence,

$$\begin{aligned} T(n) &= T(n/2) + T(3n/10) + cn \\ &\leq mn/2 + 3mn/10 + cn = 4mn/5 + cn \leq mn \end{aligned}$$

Hence, the supposition that $T(n) = O(n)$ is right when $4mn/5 + cn \leq mn(m \geq 5c)$. And we could find m which is large enough. Hence $T(n) = O(n)$. Now let's have a try that $T(n) = O(\log n)$. Suppose $T(n) \leq m \log n$, then $T(n/2) \leq m \log(n/2)$ and $T(3n/10) \leq m \log(3n/10)$. Hence,

$$\begin{aligned} T(n) &\leq m \log n/2 + m \log 3n/10 + cn \\ &= m \log n - m \log 2 + m \log n - m \log 10/3 + cn \leq m \log n \end{aligned}$$

When n is very large, for $\forall n > n_0$, the constant items could be swiped away. Hence, $c \leq -m \log n/n (= 0)$. So when $c < 0$ or when c is small enough, $T(n) = O(\log n)$.

(c) **[3]** $T(n) = T(n-1)T(n-2)$ (hint: you can express your answer in terms of the n^{th} Fibonacci number $F(n)$)

Solution: For this case, we just need to expand this recursive equation again and again. We could make the observation of the exponents' variations.Finally, we will get constant items.

$$\begin{aligned} T(n) &= T(n-1)T(n-2) = T^2(n-2)T(n-3) = T^3(n-3)T^2(n-4) \\ &= T^5(n-4)T^3(n-5) = T^8(n-5)T^5(n-6) = ... = T^{F(k)}(n-k)T^{F(k-2)}(n-k-1) \\ &= ... = T^{F(n)}(1)T^{F(n-2)}(0) \leq (T(1)T(0))^{F(n)} \end{aligned}$$

F(n) represents the n^{th} Fibonacci number. $F(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$,and$C =_{def} T(1)T(0)$ is a constant number.Hence $T(n) = O(C^{F(n)})$.

(d) **[3]** $T(n) = n^{2/3}T(\sqrt{n}) + n^{1.2}$

Solution: Let's expand this recursive equation. We could get,

$$\begin{aligned}T(n) &= n^{\frac{2}{3}}T(n^{\frac{1}{2}}) + n^{1.2} = n^{\frac{2}{3}}[n^{\frac{1}{3}}T(n^{\frac{1}{4}}) + n^{1.2*1/2}] + n^{1.2} \\ &= n^{\frac{2}{3}+\frac{1}{3}}T(n^{\frac{1}{4}}) + n^{1.2*\frac{1}{2}+\frac{2}{3}} + n^{1.2} \\ &= n^{\frac{2}{3}+\frac{1}{3}+\frac{1}{6}}T(n^{\frac{1}{8}}) + n^{1.2*\frac{1}{2}*\frac{1}{2}+\frac{2}{3}+\frac{1}{3}} + n^{1.2*\frac{1}{2}+\frac{2}{3}} + n^{1.2} = ...\end{aligned}$$

From the observation of these expansions, we could find that when we make the k-th expansion, the coefficient of T will be $n^{\frac{4}{3}[1-(\frac{1}{2})^{k+1}]}$, and exponent of n in T will be $n^{\frac{1}{2^{k+1}}}$. And the other items will be $n^{1.2} + \sum_{i=1}^{k} n^{\frac{4}{3}-\frac{2}{15}(\frac{1}{2})^{k}}$. For any n, it can be expanded again and again as above, if the unknown item T in the right-hand side becomes known, then $T(n)$ can be represented as a function of n. Since $T(2)$ is easily figured out, it can be considered as a known constant. So if the unknown item T in the right-hand side becomes $T(2)$ in k^{th} expansion, $T(n)$ can be represented as a function of n and k.In addition, in the k^{th} expansion, $n^{\frac{1}{2^{k+1}}} = 2$, so k can be written in terms of n and all the items can be written as a function of n.

$$\begin{aligned}\frac{1}{2^{k+1}} &= \frac{1}{\log n} \\ &\to k = \log(\log n) - 1.\end{aligned}$$

So $k = \log(\log n) - 1$. When n is very large,

$$n^{\frac{4}{3}}[1 - (\frac{1}{2})^{k+1}] = n^{\frac{4}{3}}[1 - \frac{1}{2^{\log(\log n)}}] = n^{\frac{4}{3}}[1 - \frac{1}{\log n}] \approx n^{\frac{4}{3}}.$$

Define $n^{\frac{4}{3}} = 2^m$,then,

$$\begin{aligned}&\log n = \log 2^{\frac{3}{4}m} = \frac{3}{4}m. \\ &\sum_{i=1}^{k} n^{\frac{4}{3}-\frac{2}{15}(1/2)^i} = \sum_{i=1}^{i} n^{\frac{4}{3}[1-\frac{1}{10}(\frac{1}{2})^i]} \\ &= \sum_{i=1}^{k} 2^{m[1-\frac{1}{10}(\frac{1}{2})^i]} = 2^{\frac{19}{20}m} + 2^{\frac{39}{40}m} + 2^{\frac{79}{80}m} + ... + 2^{m[1-\frac{1}{5}(\frac{1}{2})^{k+1}]} \\ &= 2^{\frac{19}{20}m} + 2^{\frac{39}{40}m} + 2^{\frac{79}{80}m} + ... + 2^{m[1-\frac{1}{5}(\frac{1}{2^{\log(\log n)}})]} \\ &= 2^{\frac{19}{20}m} + 2^{\frac{39}{40}m} + 2^{\frac{79}{80}m} + ... + 2^{m[1-\frac{4}{15}\frac{1}{m}]} \\ &= 2^m[2^{-\frac{1}{20}m} + 2^{-\frac{1}{40}m} + ... + 2^{-\frac{4}{15}}].\end{aligned}$$

When n is large, m is large, hence,

$$2^{-\frac{1}{20}m} + 2^{-\frac{1}{40}m} + ... + 2^{-\frac{4}{15}} \approx \frac{2^{-\frac{4}{15}}}{1 - 2^{-\frac{4}{15}}} = C.$$

Here, C is a constant positive integer. Hence, $n^{1.2} + \sum_{i=1}^{k} n^{\frac{4}{3}-\frac{2}{15}(\frac{1}{2})^i} \approx n^{1.2} + 2^m C = n^{1.2} + n^{\frac{4}{3}}C = O(n^{\frac{4}{3}})$. So, when n is very large,

$$\begin{aligned}T(n) &= n^{\frac{4}{3}[1-(\frac{1}{2})^{k+1}]}T(n^{\frac{1}{2^{k+1}}}) + n^{1.2} + \sum_{i=1}^{k} n^{\frac{4}{3}-\frac{2}{15}(\frac{1}{2})^k} \\ &\approx n^{\frac{4}{3}}T(2) + n^{1.2} + Cn^{\frac{4}{3}} = O(n^{\frac{4}{3}}).\end{aligned}$$

So, $T(n) = O(n^{4/3})$.

(e) [**3**] $T(n) = T(n-3) + 8^n$

Solution: For this case, we just need to expand it again and again. Because n decreases linearly, we will get constants items finally.

$$T(n) = T(n-3) + 8^n = T(n-6) + 8^{n-3} + 8^n = ...$$

So, when the unknown item becomes $T(k)$, $T(n) = T(k) + \sum_{i=1}^{(n-k)/3} 8^{3i}$. We can get,

$$T(n) = \begin{cases} T(0) + \sum_{i=1}^{n/3} 8^{3i}, n\%3 = 0 \\ T(1) + \sum_{i=1}^{n/3} 8^{3i}, n\%3 = 1 \\ T(2) + \sum_{i=1}^{n/3} 8^{3i}, n\%3 = 2 \end{cases}$$

So, $T(n) = c + 8^3 \frac{1-8^n}{1-8^3} = O(8^n)$, $c = T(0) or T(1) or T(2)$.

(f) [**3**] $T(n) = \sum_{k=1}^{n}(k \cdot T(k-1))$, where $T(0) = 1$.

Solution:

$$\begin{aligned} T(n) &= T(0) + 2T(1) + 3T(2) + ... + (n-1)T(n-2) + nT(n-1) \\ \Rightarrow T(n-1) &= T(0) + 2T(1) + 3T(2) + ... + (n-1)T(n-2) \\ \Rightarrow T(n) - T(n-1) &= nT(n-1) \\ \Rightarrow T(n) &= (n+1)T(n-1). \end{aligned}$$

Gauss $T(n) = (n+1)!$, then prove it using Induction Method:
Base Case:when $k = 0$, $T(0) = 1$ is true;
Hypothesis:when $k = n-1$, suppose $T(n-1) = n!$ is true;
Induction:when $k = n$, $T(n) = (n+1)T(n-1) = (n+1)!$.
In conclusion, the gauss is true. $T(n) = (n+1)!$.

(g) [**3**] $G(n) = \frac{G(n-1)}{G(n-2)}, G(0) = 1, G(1) = 2$.

Solution: Let's have a try to expand it.

$$\begin{aligned} G(n) &= \frac{G(n-1)}{G(n-2)} = \frac{G(n-2)}{G(n-3)G(n-2)} = \frac{1}{G(n-3)} \\ &= ... = G(n-6) = ... \end{aligned}$$

It's obvious that for whatever n, the value of function G at point n is equal to the value of G at point $n-6$ and the reciprocal of the value of G at point $n-3$. And because $G(0) = 1, G(1) = G(2) = 2$,$G(n)$ is equal to a constant number finally.So $G(n) = O(1)$.

(h) [**4**] $T(n) = n + \sum_{i=1}^{\log n} T(n/2^i)$.

Solution: Since the arguments of recursive equation is related to $(\frac{1}{2})^{exponent}$, we could have a try to find the relationship between $T(n)$ and $T(\frac{n}{2})$.

$$T(n) = n + T(n/2) + T(n/2^2) + ... + T(\frac{n}{2^{(\log n)-1}})(\approx T(2)) + T(\frac{n}{2^{\log n}})(\approx T(1)),$$

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

$$\Rightarrow T(n) - T(\frac{n}{2}) = \frac{n}{2} + T(\frac{n}{2}) \Rightarrow T(n) = 2T(\frac{n}{2}) + \frac{n}{2}.$$

According to master's theorem, in this case, $f(n) = \frac{n}{2}$, $a = b = 2$, so $2f(\frac{n}{2}) = n/2$. For $\forall n > n_0$, when n is large, $\exists c = 1, making 2f(n/2) = cf(n)$, so $T(n) = \Theta((n/2)\log n) = \Theta(n \log n)$.

Question 2: Pancakes, mmmm... . . . . **[10]**

Problem #4 from Erickson Lecture 1 (`http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/01-recursion.pdf`)

(a)

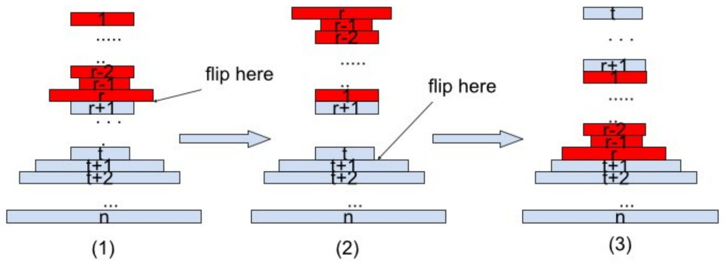

Figure 1

Solution: Define the algorithm FLIP1($A[1,...,n]$). What we need to do is to sort $A[1,...,n]$ by ascending actually. The operation we could do is FLIP(i).

FLIP1($A[1,...,n]$):

(1)Slide the flipper under the largest pancake among these not in the correct place. For example $A[t+1,..,n]$ are all in the correct place and they have been correctly sorted. $A[r]$ is the biggest among the $A[1,...,t]$;

(2)Call FLIP(r) to flip these pancakes($A[1,...,r]$);

(3)Then put the flipper above a pancake $A[t+1]$ which is the smallest among the pancakes in the correct place. If there is no such a pancake(the original $A[r]$ is the largest pancake), put the flipper at the bottom. Flip, then the largest pancake that was out of order is now in order($A[r]$ has been in the right place,i.e., $A(t)$ has became $A[r]$);

(4)If there is still pancakes not in the correct place, recursive call FLIP1($A[1,...,t-1]$). **Claim:**FLIP(i)

will flip the pancakes $A[1, ..., i]$. The steps of the algorithm is shown in Figure 1.

Proof. We could use the induction method to prove it. BASE CASE:For n=1, it's in the correct place, so we don't need to make any operation. This algorithm is right. ASSUMPTION:FLIP1($A[1, ..., t-1]$) will sort the pancakes$A[1, ..., t-1]$ correctly. INDUCTION:According to this algorithm, $A[t+1, ..., n]$ are all in the correct order and $A[r]$ is the largest among $A[1, ..., t]$ which are all out of order. It's obvious that $r \neq t$, because if $r = t$, then $A[r]$ is also in the correct order. And for $\forall A[i] \in A[t+1, ..., n]$ and $\forall A[j] \in A[1, ..., t]$, $A[i] \geq A[j]$. So our target is to place $A[r]$ to the $t-th$ place. FLIP(r) will flip $A[r]$ to the top, and now $A[1]$ has been the "largest". Then FLIP(t) will flip $A[1, ..., t]$, $A[1]$ will come to the $t-th$ place. Now $A[t, ..., n]$ have been sorted. In addition, FLIP1($A[1, ..., t-1]$) is correct. Hence, we could make a conclusion that this algorithm will sort $A[1, ..., n]$ correctly. □

Analysis. In the worst case, all the pancakes need to repeat this process described above except the last two ones. Say, we couldn't find such a solution that when we have placed $A[r]$ to the correct place t, then $A[t-1]$ is in the right place. So we have to call FLIP1(A) for $O(n)$ times. Every time, the number of FLIP(i) is $O(1)$. So the time complexity of this algorithm is $O(n)$. In the worst case, every process needs two flips. After the $(n-2)th$ pancake has been in the right place, if the pancake $A[1]$ is larger than $A[2]$, it only needs one operation to make it in the correct place. So the worst case needs $2(n-2)+1 = 2n-3$ flips.

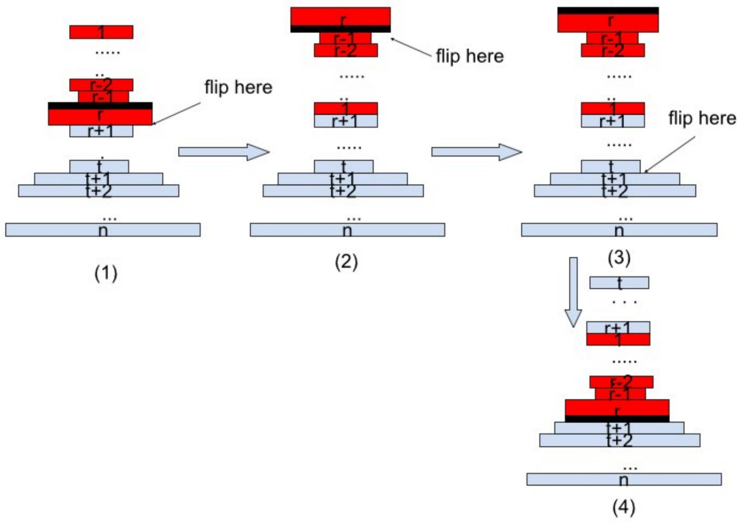

Figure 2

(b)

Solution: We only need to add at most one flip in each recursive process FLIP1($A[1, ..., n]$). If the burned side of the largest pancake out of order $A[r]$ facing down before FLIP(r), we don't need to add any operation. While, if its burned side is facing up, we should add FLIP(1) after $A[r]$ has been flipped to the top.
FLIP2($A[1, ..., n]$):

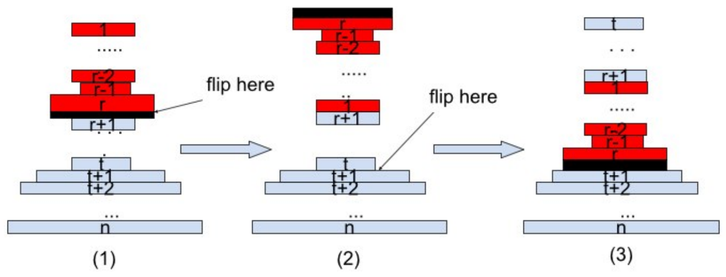

Figure 3

(1)Slide the flipper under the largest pancake among these not in the correct place. For example $A[t+1,..,n]$ are all in the correct place and they have been correctly sorted. $A[r]$ is the biggest among the $A[1,...,t]$;
(2)$judge \leftarrow 1$ if the burned side of $A[r]$ is facing up, otherwise $judge \leftarrow 0$ if $A[r]$ is facing down;
(3)Call FLIP(r) to flip these pancakes($A[1,...,r]$);
(4)Call FLIP(1) if $judge = 1$, otherwise, do nothing;
(5)Then put the flipper above a pancake $A[t+1]$ which is the smallest among the pancakes in the correct place. If there is no such a pancake(the original $A[r]$ is the largest pancake), put the flipper at the bottom. Flip, then the largest pancake that was out of order is now in order($A[r]$ has been in the right place,i.e., $A(t)$ has became $A[r]$);
(6)If there is still pancakes not in the correct place, recursive call FLIP1($A[1,...,t-1]$).
Claim:FLIP1($A[1,..,n]$) and FLIP(i) has been defined in (a). Figure 2 describes the algorithm if the burned side of $A[r]$ is facing up at first and Figure 3 describes the algorithm if the burned side of $A[r]$ is facing down at first.

Proof. If the burned side of the largest pancake out of order $A[r]$ is facing down at first, after the FLIP(r), its face is facing up and after FLIP(1), it's facing down again satisfying the requirement. And the other pancakes out of order will only go through two flips and their situations will not be influenced. If the burned side of $A[r]$ is facing up at first, after three flips, it will be facing down. And the additional is made when it's at the top, so the other pancakes out of order will still only go through two flips and their situations will not be influenced. So this idea is right. □

Analysis. The worst case is each pancake is facing up at first and all the pancakes except the last two ones need to repeat this process FLIP(A). After the $(n-2)th$ pancake has been in the right place and facing down, if the pancake $A[1]$ is larger than $A[2]$ and they are both facing down, $A[1]$ needs to 2 flips(FLIP(1) to face up and FLIP(2) to come to the right place and face down) to satisfy the requirement and then the new $A[1]$needs another flip to be facing down because it has been influenced by the flip of the original $A[1]$. It needs $3(n-2)+2+1 = 3n-3$ flips in total.

Question 3: Leaning Tower of Hanoi . . . . **[10]**
Problem #6 from Erickson Lecture 1 (`http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/01-recursion.pdf`)

Solution: We can find a $recursive pattern$- a pattern that uses information from previous steps to find the next step - for moving n disks from A to C. Define algorithm 1 transfers disks from A to C and $f(n)$ is the minimum steps needed for transferring n disks from A to C.

Algorithm1(n):
0:If($n \leq 0$) return;
1.First, call algorithm2(n-1) to transfer first $n-1$ disks above the n^{th} disk from A to B, and it uses at least $g(n-1)$ steps as a hypothesis;
2.Next, transfer the n^{th} disk from A to C, and it needs 1 step;
3.Finally, transfer the $n-1$ disks in B to C, according to the new rule, it only needs 1 step.

For the first step, we need to transfer the first $n-1$ disks above the n^{th} disk from A to B. Define algorithm 2 is to transfer disks from A to B or from C to B, and $g(n-1)$ is the minimum steps needed in total for transferring $n-1$ disks from A to B or from C to B.
Algorithm2(n-1):
0:If($n-1 \leq 0$) return;
1.First, call algorithm1(n-2) to transfer the first $n-2$ disks above the $(n-1)^{th}$ from A to C, which needs at least $f(n-2)$ steps as a hypotheis;
2.Transfer the $(n-1)^{th}$ disk from A to B;
3.Call algorithm(n-2) to transfer the $n-2$ disks from C to B, and the cost is at least $g(n-2)$ as a hypothesis;

Claim:A is the left needle, C is the right needle, and B is the leaning needle. $f(n)$ represents the number of moves needed to transfer of n disks from A to C(or from C to A) and $g(n)$ represents the number of moves needed to transfer n disks from A(or C) to B.

Proof. For this algorithm, $f(n) = g(n-1) + 2, n \geq 2, f(1) = 1$. And we can only find one another algorithm. That's because if we want to transfer the n^{th} disk from A to other needle, the first $n-1$ disks have to all be on B or on C, otherwise, the n^{th} disk can't move. Hence, there are only two cases. If the $n-1$ disks are all on B, the successive steps of algorithm 1 takes the fewest steps obviously. The algorithm below is the case when the first $n-1$ disks are all on the C. After the first $n-1$ disks are all on C, the algorithm below uses the fewest steps obviously.

1.Firstly, transfer the first $n-1$ disks from A to C, it needs at least $f(n-1)$ steps as a hypothesis;
2.Transfer the n^{th} disk from A to B which needs 1 step;
3.Transfer the $n-1$ disks in the C needle to B, and it needs at least $g(n-1)$ steps as a hypothesis;
4.Fianlly, transfer all the disks from B to C, which needs 1 step.

For this algorithm, it needs at least $g(n-1) + f(n-1) + 2$ steps. It implies that if moving the first $n-1$ disks from A to C first, it needs at least $g(n-1) + f(n-1) + 2$ steps. It's obvious that $g(n-1)+2 \leq g(n-1)+f(n-1)+2$. So the original algorithm 1 is best. Similarly, for moving $n-1$ disks from A to B, we can only find one other method. Since when moving the $(n-1)^{th}$ disk from A, the first $n-2$ disks have to be all on B or on C, otherwise, it can't move. When the first $n-2$ disks are all on C, and it takes the fewest steps as a hypotheses, the algorithm 2 takes fewest steps to finish the task obviously. The algorithm below takes the fewest steps obviously if the first $n-2$ disks are all on B.

1.Transfer the $n-1$ disks from A to B, and it costs at least $g(n-1)$ steps as a hypothesis;
2.Transfer the n^{th} disk from A to C;
3.Transfer the $n-1$ disks from B to A;
4.Transfer the n^{th} disk from C to B;
5.Finally, transfer the $n-1$ disks from A to B which at least costs $g(n-1)$ steps as a hypothesis.

For our original method 2, if the unknown is n, then it needs $g(n-1)+1+f(n-1) = g(n-1)+g(n-2)+3$. For this new method, it needs at least $g(n) = 2g(n-1)+3$ steps. It implies that move the first $n-2$ disks are all on B before moving the $(n-1)^{th}$ disk, it needs at least $2g(n-1)+3$ steps. It's obvious that $g(n-2) < g(n-1)$, so algorithm 2 is best. □

Analysis. Now we could get that,

$$\begin{aligned}f(n+1)-2 = g(n) = g(n-1)+1+f(n-1) = f(n-1)+f(n)-1\\ \to f(n+1) = f(n)+f(n-1)+1.(n \geq 2, f(1)=1, f(0)=0).\end{aligned}$$

We could get some base values $f(2)=3, f(3)=5, f(4)=9$.Then,

$$\begin{cases} f(n+1) = f(n)+f(n-1)+1 \\ f(n+2) = f(n+1)+f(n)+1 \\ \hfill n \geq 2 \end{cases}$$

Hence,$f(n+2)-f(n+1) = f(n+1)-f(n)+f(n)-f(n-1)(n \geq 2)$. Define $S(n) = f(n)-f(n-1)$, so $S(n+2) = S(n+1)+S(n)(n \geq 2)$. Suppose $S(n) = q^n$. We could get that,

$$\begin{aligned}&S(n+2) = S(n+1)+S(n)\\ &\to q^{n+2} = q^{n+1}+q^n\\ &\to q^2-q-1 = 0\\ &\to q_1 = \frac{1+\sqrt{5}}{2}, q_2 = \frac{1-\sqrt{5}}{2}\end{aligned}$$

Then we could get a general solution:$S(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n(n \geq 2)$.Since $S(2) = f(2)-f(1) = 2andS(3) = f(3)-f(2) = 2$, we could get that $c_1 = \frac{5+\sqrt{5}}{10}$,$c_2 = \frac{\sqrt{5}-5}{10}$. So $S(n) = f(n)-f(n-1) = \frac{5+\sqrt{5}}{10}(\frac{1+\sqrt{5}}{2})^n + \frac{5-\sqrt{5}}{10}(\frac{1-\sqrt{5}}{2})^n(n \geq 2)$. Hence the total moves the algorithm needs is,

$$\begin{aligned}f(n) &= S(n)+S(n-1)+S(n-2)+...+S(2)+f(1)\\ &= S(n)+S(n-1)+S(n-2)+...+S(2)+1\\ &= \frac{5+\sqrt{5}}{5}[(\frac{1+\sqrt{5}}{2})^n-1] + \frac{\sqrt{5}-5}{5}[1-(\frac{1-\sqrt{5}}{2})^n]+1(n \geq 2, f(1)=1)\end{aligned}$$

We plug $n=1$ into this equation and find it's right as well. So the constrain could be $n \geq 1$.

Question 4: StoogeSort . . . . **[15]**

Problem #9 from Erickson Lecture 1 (`http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/01-recursion.pdf`)

(a) **[10]** Parts (a)-(d)

Solution:
(a)

Proof. We could use the Induction Method to prove it.
For the base case $n = 2$:the algorithm sorts its input.
When $n > 2$,we first suppose that for $k \leq n-1$,this algorithm is right. In other words, STOOGESORT$(A[0,...,k])(k \leq n-1)$ is right, Then for n:
1. Because $m = \lceil \frac{2n}{3} \rceil \leq n-1$. Hence, STOOGESORT$(A[0,...,m-1])$actually sorts the input $A[0,...,m-1]$;
2. Similarly, STOOGESORT$(A[n-m,...,n-1])$ sorts the input $A[n-m,....,n-1]$. After this process, $\forall y \in A[m,...,n-1]$, $\forall x \in A[0,...,m-1]$, $y \geq x$. Because if $\exists y \in A[n-m,...,m-1]$, $\exists x \in A[m,...,n-1]$,$x < y$, the algorithm STOORGESORT$(A[n-m,...,n-1])$ will make their in correct order. And if $\exists x \in A[0,...,n-m-1]$,$\exists z \in A[m,...,n-1]$,$z < x$, since we know the process of STOOGESORT$(A[0,...,m-1])$has made $\forall y \in A[n-m,...,m-1]$, $y > x$, $y > x > z$. However, the process of STOORESORT$(A[n-m,...,n-1])$ has made $\forall y \in A[n-m,...,m-1]$, $\forall z \in A[m,...,n-1]$, $y \leq z$. So by contradiction, we could get $\forall y \in A[m,...,n-1]$, $\forall x \in A[0,...,m-1]$, $y \geq x$. In addition, $A[m,...,n-1]$ has been sorted into a correct sort;
3.STOOGESORT$(A[0,...,m-1])$will make sort the smaller $\frac{1}{3}$ array and hence after this process, all the elements will be sorted. So by Induction, this algorithm still works well for n. Hence, this algorithm is correct. □

(b)
It's wrong. For example, when $n = 4$, $m = \lfloor \frac{2n}{3} \rfloor = 2$. Then the next three recursive process will become STOOGESORT$(A[0,...,1])$,STOOGESORT$(A[2,...,3])$ and STOOGESORT$(A[0,...,1])$. Hence it sorts the elements in $A[0,...,1]$ and $A[2,...,3]$ respectively. And the elements in $A[2,...,3]$ may be smaller than those in $A[0,...,1]$.
(c)
Define the number of comparisons $T(n)$. $T(n) = 3T(\frac{2n}{3}) + 1(n \geq 2), T(1) = 0, T(2) = 1$.
(d)
For this construct, we could use master's algorithm:
$f(n) = 1, a = 3, b = 3/2$. So $\exists c > 1$, c makes that $\forall n > n_0, 3f(2/3n) > cf(n)$. So $T(n) = O(n^{log^3_{3/2}}) = O(n^{2.71})$.

Proof. Let's try to expand it,

$$T(n) = 3T(\frac{2}{3}n) + 1 = 3(3T((\frac{2}{3})^2n) + 1) + 1 = 3^2T((\frac{2}{3})^2n) + 3 + 1$$
$$= ... = 3^kT((\frac{2}{3})^kn) + 3^{k-1} + ... + 3 + 1 = ...$$

When $(\frac{2}{3})^kn = 1$, we could get that $k = log^n_{1.5}$. At this time, $T(n) = 3^kT(1) + 3^{k-1} + ... + 1 = O(3^k) = O(3^{log^n_{1.5}}) = O(n^{\frac{log^3}{log^1.5}}) = O(n^{2.71})$. □

(b) [**5**] Part (e)

Solution: In the worst case, the array $A[0,...,n-1]$ sorts in a decreasing order. And the swap method is to swap the adjacent pair. Hence for $A[0]$, we should swap for $n-1$ times. For $A[1]$, we should swap for $n-2$ times. By induction, for $A[k]$, we should swap for $n-1-k$ times. So we need $\sum_{num=n-1}^{1} num = \frac{n(n-1)}{2} =$ swaps in total.
We could also calculate it by removing inversions. In the worst case, there are $(n-1) + (n-$

2) + ... + 2 + 1 = $\frac{n(n-1)}{2}$ inversions. Since for each swap, we could only remove one inversion, so we need $\frac{n(n-1)}{2}$ inversions in total.

Question 5: Finding $A[i] = i$ .... **[10]**

This is from Dasgupta/Papadimitriou/Vazirani, Exercise 2.17

You are given a sorted array of n elements. Determine if there's an i such that $A[i] = i$.

Solution:

Algorithm 1: Find value that $A[i] = i$

```
FindValue($A[1, ..., n]$):
if $1 > n$ then
    return $-1$;
end if
int $mid = (1 + n)/2$;
if $A[mid] > mid$ then
    return FindValue($A[1, ..., mid]$);
else
    if $A[mid] < mid$ then
        FindValue($A[mid + 1, ..., n]$);
    else
        return mid;
    end if
end ifp
```

Claim: This algorithm searches the target key using division and recursion.

Proof. For a given array, calculate its median - mid. If $A[mid] > mid$, since this is a sorted array and the values are distinct, $A[mid+index] \geq A[mid]+index > mid+index$. So the values after $A[mid]$ will not satisfy the requirement of $A[i] = i$. Hence we continue to find this element in $A[1, .., mid]$. Similarly, if $A[mid] < mid$, then we continue to find the element in $A[mid + 1, ..., n]$. This recursive process will stop until this element has been found or the array has been searched over(In this case, the target key doesn't exist in the array). □

*Analysis.*For each recursive process, the operation count is a constant, so the time complexity is $O(1)$. And after each recursive process, the array size will be halved in general situation and we only need to search the halved array then. So the recursive depth is $O(\log n)$, the time complexity is $O(\log n) \times O(1) = O(\log n)$.

Question 6: Equivalence testing .... **[10]**

this is Exercise 3 from KT chapter 5

Suppose you are given n objects, and a subroutine that can take two objects and decide if they are "equivalent" or not (for example, it outputs 1 if they are equivalent, and so on).

Design an algorithm that calls the equivalence tester $O(n \log n)$ times and decides whether the input has more than $n/2$ items that are equivalent to each other.

Solution: 1)The first method with time complexity $O(n)$:

Algorithm 2: The algorithm of finding the equivalent object I

```
global variable $num = n$;
EquivalenceTest1($object[1, ..., n]$):
if $n == 1$ then
    call EQUIVALENCE_TESTER n times to test it with all the original n objects and count the equivalent objects' number $equal\_num$ ;
    if $equal\_num > \frac{1}{2}num$ then
        return $object[n]$;
    else
        return $null$;
    end if
end if
if $n == 0$ then
    return $null$;
end if
if $n\%2 == 1$ then
    //if n is odd, evaluate the last ont to determine wether it's the target object
    call EQUIVALENCE_TESTER($object[i], object[n]$)$n-1$ times iteratively to figure out the number of objects equivalent to $object[n]$ //$1 \leq i < n$
    if $count > n/2$ then
        return $object[n]$
    else
        DROP($object[n]$) //if the last one is not the target object, just drop it
    end if
end if
$new\_object[\ ] \leftarrow empty$
while $object[\ ]! = empty$ do
    PICK($i, j$)//$i \neq j$
    if EQUIVALENCE_TESTER($object[i], object[j]$) $== 1$ then
        DROP($object[i]$)from $object[1..n]$
        DROP($object[j]$)from $object[1..n]$
        add $object[i]$ or $object[j]$ to $new\_object[\ ]$
    else
        DROP($object[i]$)from $object[1..n]$
        DROP($object[j]$)from $object[1..n]$//now the number of objects has been m($< n$)
    end if
end while
EquivalenceTest1($new\_object[1, ..., m]$)
```

Claim:EQUIVALENCE_TESTER($object[i], object[j]$)could take two objects and decide if they are "equivalent" or not and will return 1 if they are equivalent.DROP($object[i]$) could delete the $object[i]$ from the array. And function PICK(i, j) can select randomly two objects - $objcet[i], object[j]$ from these objects. In the proof below, I will use the item **"target object"** to represent the object whose number is larger than $\frac{1}{2}n$.

Proof. After each recursion of the algorithm, the number of target objects is still larger than $\frac{1}{2}$. The fact can be proved:
Suppose there are n objects at first. If n is odd, then we just need to judge the last object by comparing it with the first $n-1$ objects and count the number, if it's the target object, then it can be found. If it's not the target object, just drop it and it will not influence the result: If the target object exists, its number is still larger than $\frac{1}{2}$ of the total number and if the target object doesn't exist, then all objects' number is less than $\frac{1}{2}$ of the total number since the total number is odd and all objects' number takes account of at most $\frac{1}{2}$ after dropping the last object. Hence the

algorithm is trivially true for dealing with this situation. So, there are always even objects left if the last object is not target object. and the algorithm will always pick $\frac{n}{2}$ pairs of objects(i *and* j). If the target object exists, let's suppose the set of the target object is s1 and the others form set s2. $s1 + s2$ have n objects in total. There are three situations:

- $i, j \in s1$
- $i, j \in s2$
- $i \in s1, j \in s2$

Suppose the number of pairs for these three situations are x, y, z respectively. Hence, there are $2x + z$ target objects and $2x + 2y + 2z = n$. For the first situation, the algorithm will drop one of them and for the second situation, the algorithm will drop one of them if they are same or drop both of them if they are different and for the third situation, the algorithm will drop both of them. Define the number of remaining target objects is n_1 and the number of all remaining objects is m. After the algorithm, the target objects in the third situation will all be dropped obviously and there are no target object in the second situation, hence $n_1 = x$. In addition, the number of the remaining objects in total $m \leq x + y$ because there will be at most y objects left in the second situation and no object will be left in the third situation. What's more, Here, $2x + z$ is the number of the target objects in the original n objects and $2x + 2y + 2z$ is the total number of objects. so,

$$\begin{aligned} \frac{2x+z}{2x+2y+2z} > \frac{1}{2} &= \frac{2x+z}{4x+2z} \\ &\Rightarrow x > y \\ &\Rightarrow \frac{n1}{m} \geq \frac{x}{x+y} > \frac{1}{2}. \end{aligned}$$

Hence, after each recursion of the algorithm, the target object's number is still larger than $\frac{1}{2}$ of the total number. And the last one left is bound to be the target object obviously. Then the algorithm will check it and make sure it's the target object.
If the target object doesn't exist, the algorithm will check the last one left if there is one object left, hence it can find that the last one is not target object.
In conclusion, the algorithm can decide whether there exists such a target object and return it correctly. □

*Analysis.*In the worst situation, in each recursion, the algorithm should pick any two objects and then at most objects will be left, hence $m \leq \frac{1}{2}n$ and $T(n) \leq T(1/2n) + f(n)$. $f(n)$ represents the time complexity of comparing the $\frac{n}{2}$ pairs of objects or checking the last object left or checking the last object of all n objects if n is odd. So $f(n) = O(n) = cn(c > 0)$. Using master's theorem, $a = 1, b = 2, f(n) = cn$, $af(n/b) = cn/2$, $\exists m < 1$, making $af(n/b) = cn/2 < mcn = mf(n)$. So $T(n) = \Theta(f(n)) = \Theta(n)$.
2)The second Method with time complexity $O(nlogn)$:

Algorithm 3: The algorithm of finding the equivalent object II

```
EquivalenceTest2($object[1...n]$):
if $1 == n$ then
    return $object[1]$;
end if
if $n == 0$ then
    return 0;
end if
$mid \leftarrow (1 + n)/2$;
```

```
 9: $object\ tmp1 \leftarrow$ EquivalenceTest2($obejct[1..mid]$);
10: $object\ tmp2 \leftarrow$ EquivalenceTest2($obejct[mid+1..n]$);
11: if $tmp1! = null$ then
12:     $cur1 \leftarrow 0$;
13:     for $i \leftarrow 1$ to $n$ do
14:         if EQUIVALENCE_TESTER($object[i], tmp1$)$== 1$ then
15:             $cur1 \leftarrow cur1 + 1$;
16:         end if
17:     end for
18:     if $cur1 > \frac{n}{2}$ then
19:         return $tmp1$;
20:     end if
21: end if
22: if $tmp2! = null$ then
23:     $cur2 \leftarrow 0$;
24:     for $i \leftarrow 1$ to $n$ do
25:         if EQUIVALENCE_TESTER($object[i], tmp2$)$== 1$ then
26:             $cur2 \leftarrow cur2 + 1$;
27:         end if
28:     end for
29:     if $cur2 > \frac{n}{2}$ then
30:         return $tmp2$;
31:     end if
32: end if
33: return $null$;
```

EXPLANATION: The algorithm solves the problem using division and recursion. It divides the problem into two subproblem and then recursive call itself as Line 9 and Line 10 shows. Then it checks both the two returned values by comparing them with all the n objects and if one of them has a number more than half of the total number, the algorithm will return that object.
Claim:I will the item **"target object"** to represent the object whose number is more than $\frac{1}{2}$ of the total number.

Proof. (1)If there exists a target object among n objects, define its number m. If all the n objects are divided into two groups with size $n1$ and $n2$ respectively, then at least in one group, the target object occupies more than half of the objects in that group. Otherwise, suppose there are $m1$ and $m2$ target objects in the two groups respectively, then, $m = m1 + m2 \leq \frac{1}{2}n1 + \frac{1}{2}n2 = \frac{1}{2}n$, which implies that the number of the target objects is no more than half of the total number and it's a contradiction. The algorithm can correctly deal with the case and it can be proved by Induction Method:
Base Case: If there are $k = 1$ or $k = 0$ objects, then the algorithm is obviously true.
Hypothesis: Make a hypothesis that the algorithm can return the target object correctly if there are $k = m < n$ objects.
Induction: Now if there are $k = n$ objects, since there exists the target object, its number is more than a half at least in one of the two groups $object[1..mid]$ and $object[mid + 1..n]$ as proved above. Define the two subgroup group1 and group2 and suppose the target object is in the group1(it can represent the general case). Since its number in group1 is more than a half and it's still the target object among mid objects, EQUIVALENCE2($object[1..mid]$) can return the target object as a hypothesis above. Then the algorithm will check its number among all the n objects, since its number is more than a half among the n objects, hence the algorithm will return it correctly. For the return object of group2, even if its number is larger than a half of the total number in group2, the algorithm won't return it since it's not the target object, its number won't exceed the half among

the n objects. Hence the algorithm can correctly return the target object when there are $k = n$ objects by induction.
Conclusion1: The algorithm can correctly deal with the case that there exists target object.
(2)If the target object doesn't exist among the n objects, make a hypothesis that the algorithm can correctly find the target object when there are $k < n$ objects or just return null when there is no target object. Generally, suppose both EQUIVALENCE2($object[1..mid]$) and EQUIVALENCE2($object[mid+1..n]$) will return two objects $obj1$ and $obj2$. The algorithm will compare either one with all the n objects and can make the judgement that they are not the target object. For any other objects in the two groups, like object $obj3$, it can't be the target object as well. If it is the target object, then its number will occupy more than a half at least one of the two groups as proved above and then the fact is $obj3 = obj1$ or $obj3 = obj1$ which is a contradiction.
Conclusion2: The algorithm will return null when there is no target object. □

*Analysis.*Suppose the algorithm needs $T(n)$ time to find the target object, then line 9 and line 10 will both occupy $T(\frac{1}{2}n)$ time. Hence, $T(n) = 2T(\frac{1}{2}n) + 2n$. Using mater's theorem, here $f(n) = 2n$, $a = 2, b = \frac{1}{2}$, hence $\exists c = 1$ that makes $2f(\frac{n}{2}) = 2n = c{*}2n$ and hence $T(n) = \Theta(2nlogn) = \Theta(nlogn)$. The subroutine EQUIVALENCE will be called $\Theta(nlogn)$ times in total.

Question 7: Medians . . . . **[10]**

(a) [**5**] Why do we need to take groups of 5 elements ? Why not 3 ?

Solution: If we take groups of 3 elements, $T(n) = O(nlogn)$, which is more complex than the complexity that we take groups of 5 elements.

Proof. First suppose $T(n) = O(n)$ when we take groups of 3 elements. Now for each recursion, the median of medians is larger or smaller than $1/3n$ elements in the input array. Thus, in the worst case, the final recursive call searches an array of size $2/3n$. $T(n) = O(n) + T(1/3n) + T(2/3n)$. If $T(n) \leq cn$ and $O(n) = c^{'}n(c > 0 \ and \ c^{'} > 0)$, then $T(1/3n) \leq 1/3cn$ and $T(2/3n) \leq 2/3cn$. Then,

$$T(n) \leq 1/3cn + 2/3cn + c^{'}n = cn + c^{'}n > cn.$$

So the suppose is wrong. By contradiction, we could get that $T(n) = \Omega(n)$. In fact, we have proved the complexity is more complex than that in the situation taking groups of 5 elements. Now suppose $T(n) = O(nlogn)$. If $T(n) \leq cnlogn$ and $O(n) = c^{'}n(c > 0 \ and \ c^{'} > 0)$, then $T(1/3n) \leq c(1/3n)log(1/3n) and T(2/3n) \leq c(2/3n)log(2/3n)$. So,

$$T(n) = c(1/3n)log(1/3n) + c(2/3n)log(2/3n) + c^{'}n \leq cnlog(2/3n) + c^{'}n.$$

If $cnlog(2/3n) + c^{'}n \leq cnlogn$, then $c^{'} \leq \frac{\log n}{\log(2n/3)}c$. So we just need to use some c high enough to satisfy this requirement. Hence $T(n) = O(n \log n)$ is right. If we take groups of 3 elements, the time complexity is higher than taking group of 5 elements. □

(b) [**5**] Analyze the behavior of the median finding algorithm if we take groups of size $2k+1$ for any $k > 0$.

Solution: If we take groups of $2k+1$ elements, then for each recursion, the median of medians is larger or smaller than elements in the input array. Thus, in the worst case, the final recursive call researches an array of size $\frac{(3k+1)n}{4k+2}$. So $T(n) = O(n) + T(\frac{n}{2k+1}) + T(\frac{(3k+1)n}{4k+2})$.
Make an assumption that $T(n) \leq cn$ and $O(n) = c^{'}n(c > 0 \ and \ c^{'} > 0)$. Then $T(\frac{n}{2k+1}) \leq \frac{cn}{2k+1}$

and $T(\frac{(3k+1)n}{4k+2}) \leq \frac{(3k+1)cn}{4k+2}$. So $T(n) \leq c^{'}n + \frac{cn}{2k+1} + \frac{(3k+1)cn}{4k+2}$. If the assumption that $T(n) \leq cn$ is right, then,

$$T(n) = c^{'}n + \frac{cn}{2k+1} + \frac{c(3k+1)n}{4k+2} \leq cn$$
$$\Rightarrow c^{'} \leq \frac{k-1}{4k+2}c$$

when $k = 1$, this assumption is obviously wrong and we have proved that if taking groups of 3 elements, $T(n) = O(n \log n)$ in part(a). When $k > 1$, we just need to use some c large enough to satisfy this requirement. Hence the assumption that $T(n) \leq cn$ is right and $T(n) = O(n)$.

Question 8: Hidden Surface Removal .... **[10]**

This is exercise 5 from KT Chapter 5

You are given n lines $\ell_i : y = ax_i + b_i$. None of them are vertical (i.e $a_i < \infty$ for all i). A line ℓ_i is *uppermost* at $x = x_0$ if

$$a_i x_0 + b_i > a_j x_0 + b_j$$

for all $j \neq i$. In other words, ℓ_i is the first line a vertical line $x = x_0$ hits as it descends from $y = \infty$.

A line is *visible* if it's uppermost at some point. Intuitively this means that if all lines were colored with different colors, and you looked down from the "top" ($y = \infty$), you'd see the color for a visible line. Note that all lines might not be visible ! Give an algorithm running in $O(n \log n)$ time that takes n lines as input and returns the ones that are visible.
solution 1:

Solution:

Algorithm 4: The algorithm of Hidden Surface Removal I with time cost $O(n \log n)$ in general case and $O(n^2)$ in the worst case

```
1: QUICKSORT($A$)//According to their slopes
2: $right = +\infty$; $left = -\infty$;
3: $line\ result[\ ]$
4: FINDVISIBLE($line\ A[1, ..., n]$, $int\ left$, $int\ right$):
5: int $mid = (1+n)/2$ //select the median line $A[(1+n)/2]$
6: int $left\_bound = left, right\_bound = right$ // The variants represent the interval within which
   the median line is visible
7: line $A1[\ ], A2[\ ]$
8: for $i = 0$ to $n$ do
9:   if $i \neq mid$ then
10:      //Calculating the interval within which $a_{mid}x + b_{mid} \geq a_i x + b_i$ is true,
11:      CALCULATE_INTERVAL(mid,i)
12:      if $a_{mid}! = a_i$ and $\frac{b_i - b_{mid}}{a_{mid} - a_i}$ is an upper bound of the return interval then
13:         add i to $A1[\ ]$
14:         if $right\_bound > \frac{b_i - b_{mid}}{a_{mid} - a_i}$ then
15:            $right\_bound = \frac{b_i - b_{mid}}{a_{mid} - a_i}$
16:         end if
17:      else
18:         if $a_{mid}! = a_i$ and $\frac{b_i - b_{mid}}{a_{mid} - a_i}$ is a low bound of the return interval then
19:            add i to $A2[\ ]$
20:            if $left\_bound < \frac{b_i - b_{mid}}{a_{mid} - a_i}$ then
```

```
          $left\_bound = \frac{b_i - b_{mid}}{a_{mid} - a_i}$
        **end if**
      **end if**
    **else**
      **if** $a_{mid} == a_i$ and $b_{mid} < b_i$ **then**
        $left\_bound \leftarrow right\_bound + 1$;%set $left\_bound$,making it larger than $right\_bound$
      **else**
        no work%the situation that $a_{mid} = a_i$ and $b_{mid} \geq b_i$.
      **end if**
    **end if**
    **if** $left\_bound < right\_bound$ **then**
      break;
    **end if**
  **end if**
**end for**
//After comparing the median line with the others, we could get an interval whose low bound
  is left_bound and the upper bound is right_bound. In the interval, the median line is visible.
**if** $left\_bound > right\_bound$ **then**
  delete $A[mid]$ from A
  FINDVISIBLE($A$, $left$, $right$)
  return;
**else**
  **if** $right\_bound == +\infty || left\_bound == -\infty || left\_bound \leq right\_bound$ **then**
    add $A[mid]$ to result
  **end if**
**end if**
**if** $right\_bound < right$ **then**
  FINDVISIBLE($A1$, $right\_bound$, $right$)
  return;
**end if**
**if** $left\_bound > left$ **then**
  FINDVISIBLE($A2$, $left$, $left\_bound$)
  return;
**end if**
```

EXPLANATION: The algorithm FINDVISIBLE solves the problem using division and recursion. Firstly, the input $A[1..n]$ is sorted by their slopes, hence the median line $A[mid]$ has the median slope. Then the algorithm compares the median line with all the other lines and narrows the interval within which the median line is visible at the same time. Here, the median line is bounded by bounded by $left\_bound$ and $right\_bound$. The goal of comparing the median line with all the other lines is to figure out an interval within which the median line is visible actually. The function of CALCULATE_INTERVAL(mid,i) is to is to get an interval within which $a_{mid}x + b_{mid} \geq a_i x + b_i$ is true. If it's true, it means that the median line shadowed the i^{th} line. The function is rather simply, hence its code is omitted. If the return value $\frac{b_i - b_{mid}}{a_{mid} - b_i}$ is a upper bound of the returned interval, i.e., $[...\frac{b_i - b_{mid}}{a_{mid} - b_i}]$, then the algorithm updates $right\_bound$. As Figure 4 shows, the right bound of the interval that the median line(red line) shades the green line is on the left side of the old $right\_bound$, hence the algorithm will update $right\_bound$. However, for the grey line, $right\_bound$ will not be updated. If it's a low bound, then the algorithm updates $left\_bound$. As Figure 4 shows, the blue line will shadow the median line in the interval [*old left_bound...new left_bound*], hence the algorithm will update $left\_bound$, meanwhile, for the yellow line, $left\_bound$ will not be updated. If $a_i == a_{mid}$, it compares b_{mid} and b_i to update the bounds. If $b_{mid} < b_i$, then the target interval doesn't exist. Otherwise, the interval has no bound, so no work is needed. Figure 5 shows the two situations. In each comparison, the lines will be classified into two line sets $A1[\,]$ and $A2[\,]$ given the result of the inequality. As Figure 4 shows, the green line and grey line will be added to $A1[\,]$ and the yellow line and the blue line will be added to $A2[\,]$. Finally, the algorithm makes a judgement on the median line and recursive itself to find visible lines in the two sets $A1[\,]$ and $A2[\,]$ respectively.

Proof. In each recursion, the algorithm could judge whether the picked line is visible after comparing with all other lines and figuring out an interval $[left\_bound, right\_bound]$ within which its value is larger than all other lines. This is obviously correct. Then, according to this algorithm, these lines whose values are larger than the picked line in interval $[right\_bound, right]$ will be classified into set A1. The lines in A1 dont need to compare with these lines in A2, we just need to recursively call FindVisible($A1$, $right\_bound$, $right$). Because in the interval$[left, left_bound]$, the values of lines in A2 are larger than the value of the picked line and the value of picked line is larger than the values of the lines in A1 in the interval $[left, left\_bound]$. In addition, in the interval $[left\_bound, right\_bound]$, the lines in A1 is covered by the picked line. So if we want to judge whether the line in A1 is visible, we just need to compare it with the others in A1 in the interval $[right\_bound, right]$. This is similar to the situation of set A2. Hence, the algorithm is correct. □

*Analysis.*For each recursive process, the time complexity is $O(n)$. And since after each recursion, the array of lines will be classified into two sets and its a Divide and Conquer. Because the algorithm chooses the line having median slope, so the two sets are approximately equal size. Hence, we just need to recursively call the halved set. Define the time cost of the algorithm is $T(n)$, then $T(n) = 2T(\frac{n}{2}) + O(n)$. Given master's theorem, here $f(n) = O(n)$, $a = 2$ and $b = 2$ and it implies that $\exists c = 1$, making $2f(n/2) = cf(n)$, so the average time complexity of this algorithm is $O(n \log n)$. However, for each recursion, the picked line may be not visible or the division of the two sets A1 and A2 may be uneven, we have to deal with lines one by one. so in the worst case, the complexity is $O(n^2)$. We pick the line which has a median slope on purpose. Because it could avoid the second terrible situation effectively. However, it cant avoid the first terrible situation. For example, there is a situation that line1 has a positive slope and line2 has a negative line and the others slope is 0 as Figure 6 shows. In this case, the algorithm will pick the red lines and compare the red line with the others in each recursion and there will not have division and conquer, so the time cost is $O(n^2)$. For most cases, this algorithms complexity is $O(n \log n)$.

solution2:

Solution: Credit to the idea on a webpage $http://blog.sina.com.cn/s/blog_76f6777d010198al.html$ **and do by myself**.

Algorithm 5: The algorithm of Hidden Surface Removal II with time cost $O(n \log n)$

Require: $a[1...n]$, the slopes of the n lines, $b[1..n]$ the corresponding intercepts of the n lines;
Ensure: $ans[]$, a set of visible lines.

```
1: sort $a[1..n]$ in an increasing order and permute $b[1..n]$ correspondingly;
2: $ans[] \leftarrow empty$, $stack < int > st \leftarrow empty$;
3: if $n == 1$ or $n == 0$ then
4:     return a[n], b[n];
5: end if
6: push 1 to $st$;
7: $i \leftarrow 0$;
8: while $i \leftarrow 2$ to $n$ do
```

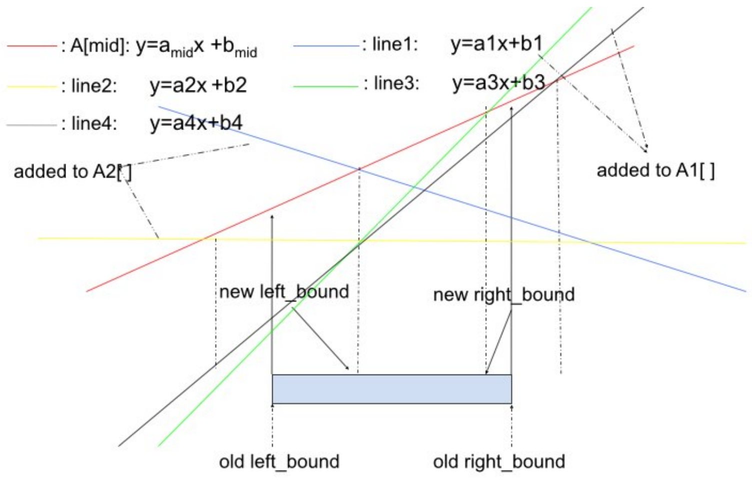

Figure 4

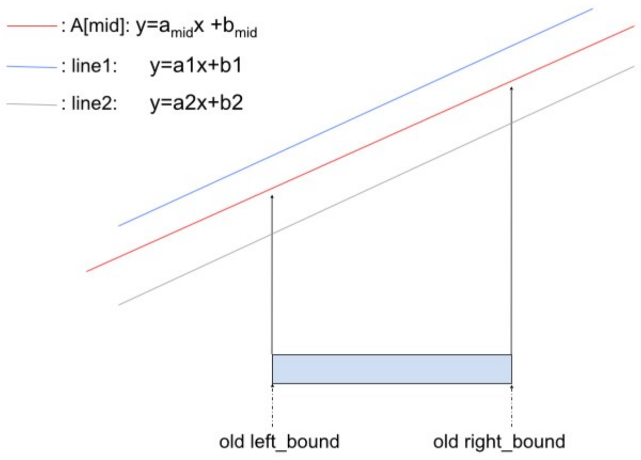

Figure 5

```
9:  **if** $a[st[st.top]] == a[i]$ and $b[st[st.top]] \leq b[i]$ **then**
10:     pop st[st.top] from st, st.top–;
11:     push i to st, st.top++;
12: **else**
13:     **if** $a[st[st.top]]! = a[i]$ **then**
14:         push i to st, st.top++;
15:         break the loop;
16:     **end if**
17: **end if**
```

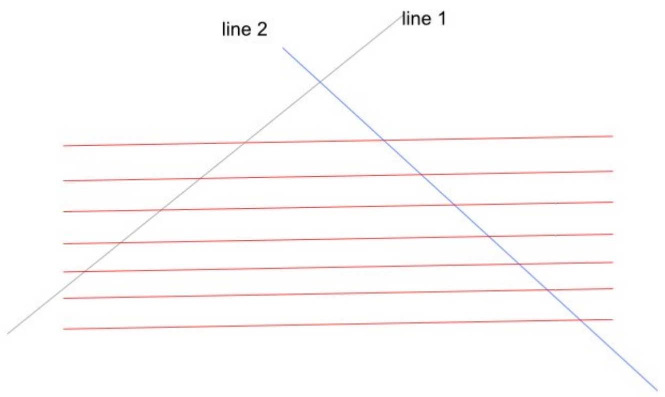

Figure 6

```
end while
for $k \leftarrow i+1$ to $n$ do
    if $st.size == 0$ then
        push k to st, st.top++;
        break the loop;
    end if
    if $st.size \geq 1$ then
        $j \leftarrow st[st.top]$;
        if $a[j] == a[k]$ then
            if $b[k] > b[j]$ then
                pop st[st.top] from st, st.top–;
                push k to st, st.top++;
                continue;
            else
                continue;
            end if
        end if
    else
        if $st.size == 1$ then
            push k to st and st.top++;
            continue;
        end if
    end if
    $i \leftarrow st[st.top-1]$;
    $x1 \leftarrow \frac{b[k]-b[j]}{a[j]-a[k]}$;
    $x2 \leftarrow \frac{b[i]-b[j]}{a[j]-a[i]}$;
    while $x1 \leq x2$ and $st.size \geq 2$ do
        pop st[st. top] from st, st.top–;
        $j \leftarrow st[st.top]$;
```

```
47:        $i \leftarrow st[st.top-1]$;
48:        $x1 \leftarrow \frac{b[k]-b[j]}{a[j]-a[k]}$;
49:        $x2 \leftarrow \frac{b[i]-b[j]}{a[j]-a[i]}$;
50:    end while
51:    push k to st, st.top++;
52: end for
53: add all the elements in st to $ans[]$;
54: return $ans[]$;
```

Claim:st is a stack, $st.top$ is the index of the top element in the stack, $st.size$ is the number of elements in the stack.

EXPLANATION: From Line 8 to Line 18, the algorithm tries to push the first two lines with different slopes into the stack. If the first two lines always have same slopes, push the one with a larger intercept into the stack and drop the other one. Then, try the next one in the sorted input until the line with a different slope is found and then push that line in the stack too in order to guarantee there are at least two lines with different slopes in the stack. From Line 19 to Line 52, the algorithm tries the sorted lines in turn. When try the line k, pick the element on the top of the stack as line j and then pick the element below it as line i. Figure out the intersection $x1$ between line k and line j and the intersection $x2$ between line i and line j. If $x1 \leq x2$, the algorithm will pop line j from the stack. If line k has the same slope with line j and has a larger intercept, pop line j from the stack and push line k into it in order to guarantee the lines' slopes in the stack are increasing from the stack bottom to top and different. Otherwise, if they have same slope but line k's intercept is smaller, just drop line k and try the new line among the n lines. If there is only one line in the stack, just push line k into the stack in order to guarantee there are at least two lines in the stack. Continuously repeat the process until $x1 > x2$ and then push line k into the stack, however, line k doesn't need compare its slope with the new line j. Since it's the lines in the stack have different slopes and are increasing from the bottom to top, and the new line j has a smaller slope comparing the first line j. Hence line k has a larger slope compared with the new line j. Then the algorithm repeats the process described above to try new line among the remaining sorted lines. Finally, the lines in the stack are just the set of visible lines.

Proof. Firstly, the lines' slopes in the stack are increasing from the bottom to the top. The algorithm sorts the lines in an increasing order at first and then push the lines in that increasing order. Make a hypothesis that the lines in the stack have increasing slopes from the bottom. it's obvious that the lines traversed by the algorithm always have a larger slope than all the lines in the stack. Hence, no matter how many lines are popped from the stack, when it's pushed into the stack, the line on the top has the largest slope among one and the lines are still in an increasing order.
Secondly, the lines in the stack maintained by the algorithm are all visible lines. It can be proved by induction method. Before that, given the case that two lines - line k and line j on the stack top have same slope. The line with a lager intercept is bound to cover the other line. Hence, the algorithm deals with case correctly. And in order to avoid troubles, it's safe to assume that all lines have different slopes without generality. If there are only two lines in the stack, they are obviously all visible. Make a hypothesis that the lines in the stack are all visible after trying the first $k = m - 1$ lines among the sorted n lines. When try the m^{th} line, it's obvious a visible among the first m lines since it has the largest slope. However, some lines in the stack may be covered by it. Define the top two lines in the stack line j, and line i, the new traversed one line k, and the intersection of line k and line j x_1 and the intersection of line i and line j x_2 and the intersection of line i and line k x_3. Since the lines in the stack have increasing slopes from the bottom, line j has a larger slope than line i, hence line j was covered by line i in the interval $(-\infty, x_2]$. Since the new traversed line k has a larger slope than line j, line j is covered by line k in the interval $[x_1, +\infty]$. Hence, if $x_1 \leq x_2$, line j is covered totally as Figure 7 shows. So it should be popped out from the stack. However, for line i,

it's still visible in the interval $[-\infty, min\{x_3, x_2\}]$ and won't be covered by line k and line j. Hence, repeat the process can pop out all the lines covered by line k. Then for the remaining lines, they were visible as a hypothesis and won't be covered by line k, hence they are still visible. And line k is obviously visible. Hence, the lines in stack are still visible when traverse the kth line among the n lines.
In conclusion, the algorithm is correct. □

*Analysis.*Suppose the time cost of the algorithm is $T(n)$. Since the result of a comparison is either a line will be popped out from the stack or a line will be pushed into the stack. Since each line will be pushed at most once and be popped at most once, so the total number of the operations including pushing and popping is at most $2n$. Hence the total number of comparisons is $O(n)$. Since the lines have to be sorted at first, and the fastest sort algorithm costs $O(n \log n)$ time, the total time cost is $O(n \log n)$ For the space cost, the major extra cost is the a stack containing the visible lines, so the space cost is $O(n)$.

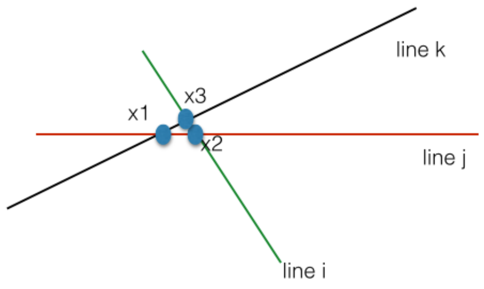

Figure 7

Figure 7