Midterm solution

1.

Give asymptotic upper and lower bounds for $T(n) = T(\sqrt{n}) + 1$. Assume that T(n) is constant for $n \leq 2$

Use changing variable technique. Let $m = \lg n$, so $n = 2^m$

$$T(n) = T(\sqrt{n}) + 1$$

$$\Rightarrow \quad T(2^m) = T\left(2^{\frac{m}{2}}\right) + 1$$

Let $S(m) = T(2^m) = T(n)$

$$T(2^m) = T\left(2^{\frac{m}{2}}\right) + 1$$

$$\Rightarrow \quad S(m) = S\left(\frac{m}{2}\right) + 1$$

By master theorem case 2, we could know that

$$S(m) = \theta(\lg m)$$

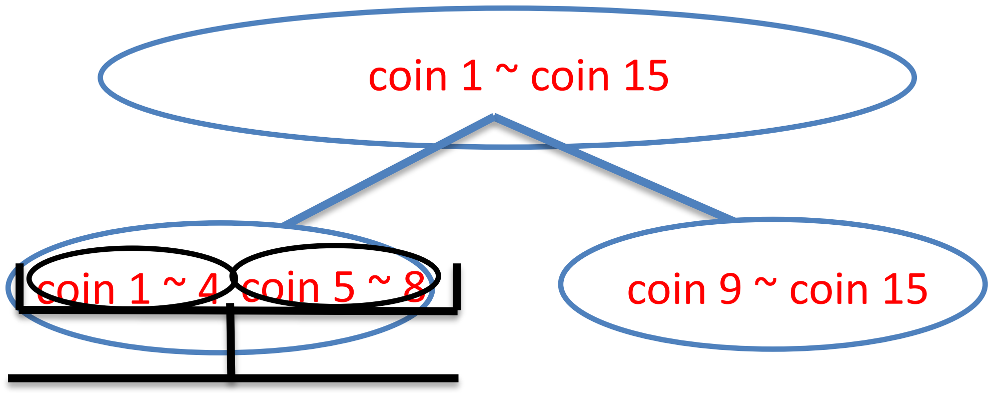
$$\Rightarrow \quad T(n) = \theta(\lg \lg n)$$

Time complexity: $\theta(\lg \lg n)$

2.

Suppose we have 15 identical looking coins numbered 1 through 15 and only one of them is counterfeit(偽造) coin whose weight is different with the others. Suppose further that you have one balance scale (天平秤). Develop a method for finding the counterfeit coin with minimum number of weighings (量稱).

The main idea is to divide 15 coins into two sets (There are 8 coins in the left side and 7 coins in the right side), then we try to determine which set contains the counterfeit coin. In order to determine which set contains the counterfeit coin, we divide the left side into two groups (group A and group B) and put them on the balance scale.



If the balance scale is not balance, then the counterfeit coin is in the left set (coin1~coin8). Otherwise, it's in the right set(coin9~coin15). Therefore, the size of subproblem is either 8 or 7, the solution for the subproblem is the same as that been described in last quiz. So the minimum number of weighings is $4 = \lceil \lg 15 \rceil$.

3.

What is the running time of Heapsort on an array of length n that is already sorted in increasing order? That about decreasing order? (You need use a few sentences to explain your answer.)

No matter it is in increasing or decreasing order, the time complexity is still $O(n \lg n)$.

Assume we maintain a minimum heap. Heapifying the input array needs $O(n)$. Then we extract each element from the minimum heap at each step. So the extraction needs $O(n \lg n)$. The bottleneck is extraction operation. Therefore, the time complexity is still $O(n \lg n)$.

4.

Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to α, where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don't worry about integer round-off.)

The minimum depth occurs for the path that always takes the smaller portion of the split, i.e., the nodes that takes α proportion of work from the parent node. The first node in the path(after the root) gets α proportion of the work(the size of data processed by this node is αn), the second one get α^2 so on. The recursion bottoms out when the size of data becomes 1. Assume the recursion ends at level h, we have

$$\alpha^h n = 1$$

$$h = \log_\alpha 1/n = \lg(1/n) / \lg \alpha = -\lg n / \lg \alpha$$

Maximum depth m is similar with minimum depth

$$(1 - \alpha)^m n = 1$$

$$m = \log_{1-\alpha} 1/n = \lg(1/n) / \lg(1 - \alpha) = -\lg n / \lg(1 - \alpha)$$

5.

(1)

The running time of QUICKSORT when all elements of array A have the same value will be equivalent to the worst case running of QUICKSORT since no matter what pivot is picked, QUICKSORT will have to go through all the values in A. And since all values are the same, each recursive call will lead to unbalanced partitioning.

Thus the recurrence will be: $T(n) = T(n-1) + \Theta(n)$

Hence the running time of QUICKSORT in this case is $\Theta(n^2)$

(2)

before

	p

after

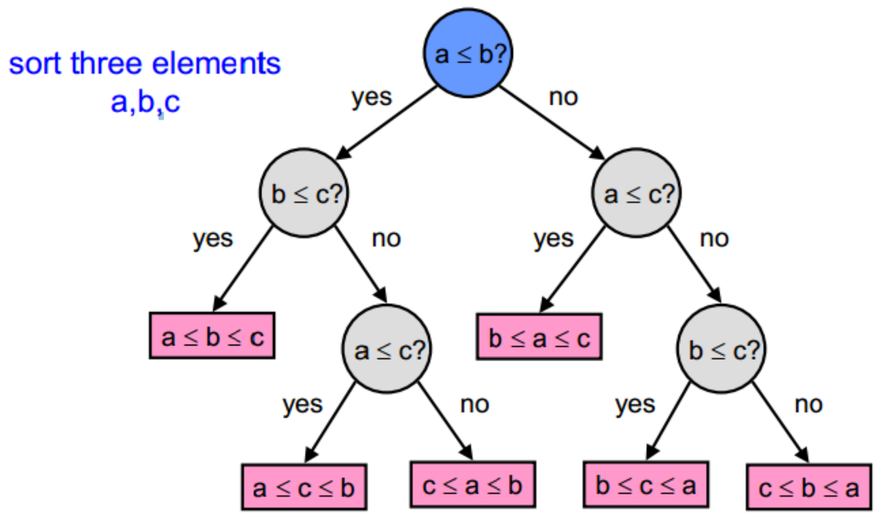
< p	= p	> p

Just have to do quick sort to <p part and >p part

You don't have to do sorting again to the =p part (whick is the same value to pivot)

Similar to 3-way partitioning .

6.



7.

Comparison sort only uses comparisons between items to gain information about the relative order of items , Consequently, any comparison sort can be viewed as performing in the following way:

1. Continuously gather relative ordering information between items
2. In the end, move items to correct positions

So, there is no contradiction with the fact that the lower bound for sorting is O(nlgn), because the linear sorting algorithm assume some properties on the input, and determine the sorted order by distribution.

The running time of the linear sorting algorithm is O(n + k).

Ps: k is the limitation of the input value

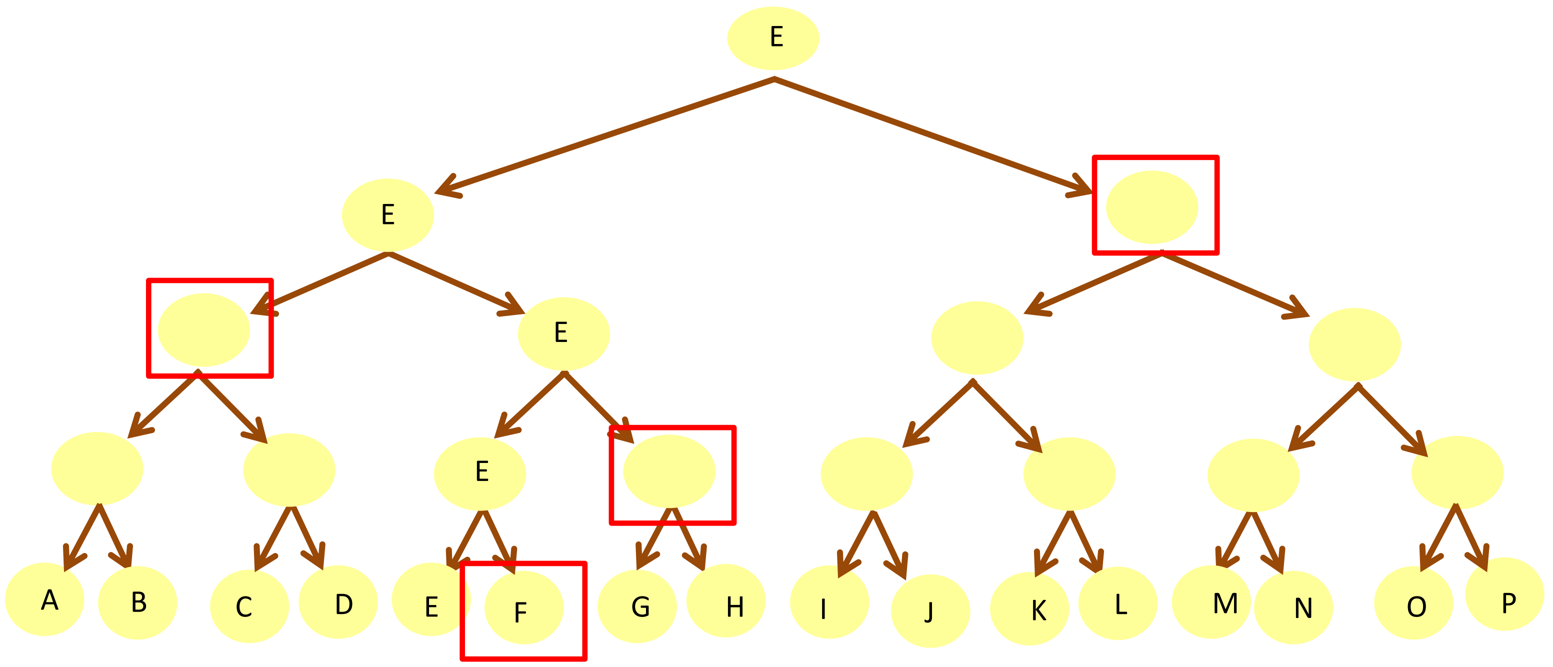
➔if k = O(n), time is (asymptotically) optimal

8.

First , find the maximum element with n-1 running time.

Then, we find out the largest element which is compared with the maximum element before . we use the $\lceil lgn \rceil$-1 running time.

Thus , we need to use n+$\lceil lgn \rceil$-2 comparisons in the worst case.



9.

E(n)= O(n) + 1/n $\sum_{1\le k\le n}$ E(max{k-1, n-k})

= O(n) + 2/n $\sum_{\lfloor n/2\rfloor\le k\le n-1}$ E(k) 3%

Let E(K) ≤ ck

T(n)= O(n) + 2/n $\sum_{\lfloor n/2\rfloor\le k\le n-1}$ ck 5%

E(T(n)) ≤ 2c/n ($\Sigma_{k=1\ldots n}$ k - $\Sigma_{k=1\ldots(n/2-1)}$ k) + an

= 2c/n ((n-1)n/2 – (n/2 – 2)(n/2 – 1)/2) + an

= c/n (3n^2/4 + n/2 – 2) + an

= c(3n/4 + ½ – 2) + an

≤ cn – (cn/4 – c/2 – an) 8%

≤ an+cn ≤c_{2n}

By induction,T(n) ≤c_2n,so T(n) = O(n) 10%

10.

(1) Overlapping Subproblems 2%

The problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblem. (allows recursion)

(2) Optimal Substructure 5%

The problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblem.(allows speed up)

EX. Find longest path in directed graph 10%

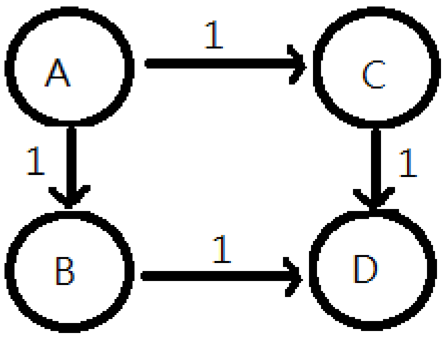
Longest path:

A=>C=>D : 2

A=>C : 3

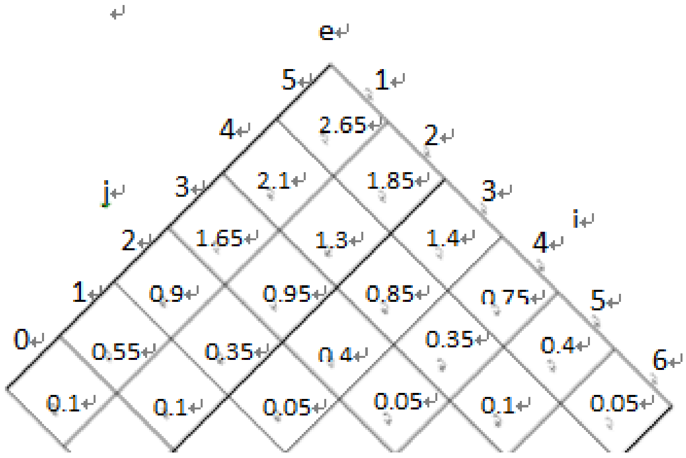
C=>D : 1

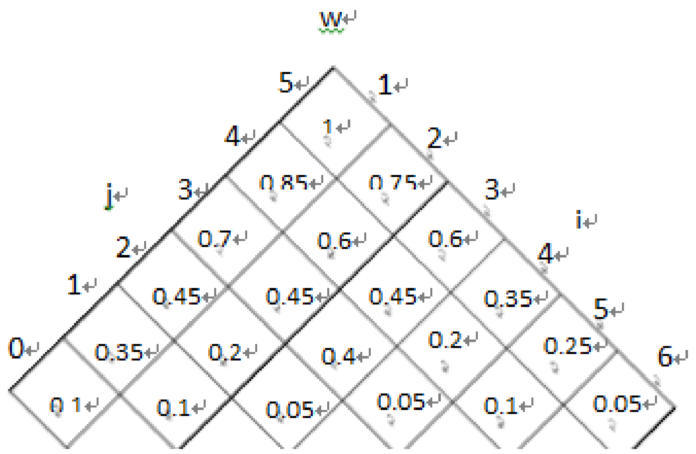
It can't be overlapping subproblem.

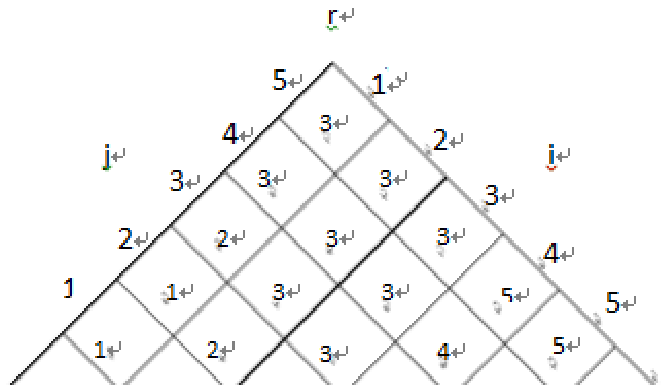


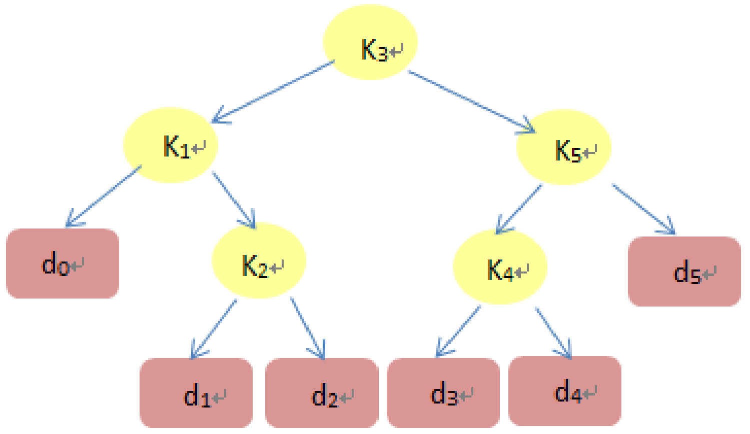
$$\begin{tikzcd}A \arrow[r, "1"] \arrow[d, "1"'] & C \arrow[d, "1"] \\ B \arrow[r, "1"] & D\end{tikzcd}$$

11.


e
j
i
5 4 3 2 1 0
1 2 3 4 5 6
2.65
2.1 1.85
1.65 1.3 1.4
0.9 0.95 0.85 0.75
0.55 0.35 0.4 0.35 0.4
0.1 0.1 0.05 0.05 0.1 0.05


w
j
i
5 4 3 2 1 0
1 2 3 4 5 6
1
0.85 0.75
0.7 0.6 0.6
0.45 0.45 0.45 0.35
0.35 0.2 0.4 0.2 0.25
0.1 0.1 0.05 0.05 0.1 0.05


r
j
i
5 4 3 2 1
1 2 3 4 5
3
3 3
2 3 3
1 3 3 5
1 2 3 4 5



12.

$F_0=1, F_1=1, F_2=2, F_3=4$ 2%

When $k > 3$, $F_k=F_{k-1}+F_{k-2}+F_{k-3}$ 6%

We can record every k in array, so we can computed F_k in O(k).

So F_n can be computed in O(n) 10%