Please select *all* the correct answer(s) to question 1 to 10. Note that there could be 0 to 5 correct answers. If none of the answer is correct, answer "none". If you do not wish to answer a question, leave it blank. All 10 answers must be written on the first page of your answer book, and the answer to question 1 must be in the first line, the answer to question 2 must be in the second line, and so on. If you fail to follow these rules then your answers will be *ignored*.

1. (3 points) Let $f(n)$ be the number of additions in the following algorithm.

```
for i = 1 to n do
  for j = 1 to i do
    for k = 1 to j do
      C[i][j][k] = A[i][j][k] + B[i][j][k];
    end for
  end for
end for
```

   1. $f(n) = O(n^2\sqrt{n})$
   2. $f(n) = O(n^2 \log n)$
   3. $f(n) = O(n^3)$
   4. $f(n) = O(n^3 \log n)$
   5. $f(n) = O(n^{100})$

2. (3 points) Given two sorted list A and B (in increasing key order), the following recursive algorithm merges A and B into a sorted list C (also in increasing key order).

   - If either A or B is *empty* then the result is the other list.
   - If both A or B are *not* empty, we compare the keys of the first nodes of A and B, and select the *smaller* one (denoted as s), and remove it from the list. Then we *recursively* merge the remaining parts of A and B into a new sorted list C', then we concatenate s with C' into the final sorted list C.

   Let $f(n, m)$ be the *minimum* number of comparisons of this algorithm, where n and m are the numbers of nodes in A and B respectively. $f(n, m)$ is?

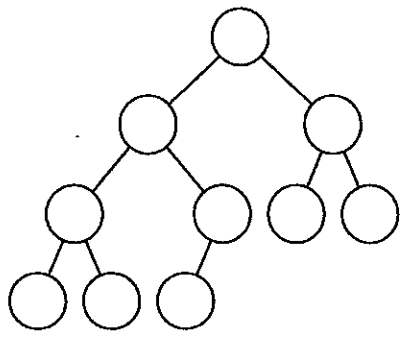
   1. $\Omega(n)$
   2. $\Omega(m)$
   3. $\Omega(n + m)$
   4. $\Omega(mn)$
   5. $\Omega(mn(\log m + \log n))$

3. (3 points) We now use the merge algorithm in the previous question to sort a set of keys. We first make each key a list of a single node. Then we merge the first and the second lists into a sorted list of length two, the third and the fourth lists into a sorted list of length two, and so on. If we start with n keys now we have roughly $n/2$ sorted list of length two now. We then merge these lists of length two into roughly $n/4$ sorted list of length four, and so on. Finally we will have a sorted list of length n. Let $f(n)$ denote the *maximum possible* total number of comparisons of this algorithm, then $f(n)$ is? Note that all the answers are in little-o notation.

見背面

1. $o(n)$
2. $o(n\sqrt{n})$
3. $o(n \log n)$
4. $o(n(\log n)^2)$
5. $o(n^2)$

4. (3 points) A *binary minimum heap* is a binary tree in which every level is completely full, except the last level, which is filled from the left to right. Here the *level* of a node is its distance to the root, so the root has level 0 and the children of the root will have level 1, and so on. Also the key of a parent is less than those of its children. For ease of discussion we assume that all keys in the heap are *distinct*. Now which of the following descriptions about a binary minimum heap of *10* nodes (see the figure below) are correct?



1. The second smallest key is always in level 1.
2. The largest key is always in a leaf, i.e., a node without any children.
3. The second largest key is always in a leaf.
4. Let n be the number of nodes and we consider the case of arbitrary n, then the height of the tree is $\Theta(\log n)$.
5. Let n be the number of nodes and we consider the case of arbitrary n, then we can find the *third* smallest key of the heap by $O(1)$ key comparisons.

5. (3 points) We will follow the notations from the previous question. We can build a heap from an array of n keys by the following algorithm. A *heapify* operation for a binary minimum heap combines two heaps and a root into a new heap. For ease of discussion we assume that the array is indexed from 1 to n, and the left/right child of a node with array index i have array indices $2i$ and $2i+1$ respectively. We first set each key of the second half of the array (indexed from $n/2$ to n) as $n/2$ heaps, each has one node. Then we combine two of them and their parent into a heap of three nodes, and so on. If the root is *smaller* than the roots of *both* subtrees then we stop. Otherwise we exchange the root with the smaller root from the subtrees, and repeatedly heapify the subtree where the root has just been replaced. Next we heapify the key at $n/2-1$ and all the way back to the root (at index 1). The final result will be a binary minimum heap of n nodes.

1. The minimum number of key comparisons is $\Theta(n)$.
2. The maximum number of key comparisons is $\Theta(n)$.
3. Since each heapify on a key may compare it with all the keys down to a leaf, its number of comparisons is no more than the height of the heap, which is $O(\log n)$. Also we have $O(n)$ keys to heapify, so the total number of key comparisons is $O(n \log n)$.
4. The number of key comparisons is in the same order as the sum of (*original, before heapified*) levels (as defined in the previous question, the distance to the root) of all keys.

接次頁

5. The number of key comparisons is *maximized* if and only if the *largest* $n/2$ keys are in the *first* half of the array and in *decreasing* order.

6. (3 points) A *randomized quick sort* works as follows. For ease of discussion we assume that the sequence we want to sort, $K = (k_1, \ldots, k_n)$, is a *permutation* of 1 to n. We randomly and uniformly pick a key k from K. Then we partition K into two subsets – K_1 that has keys less than k and K_2 that has keys greater than k. Then we recursively sort K_1 and K_2, and combine the results with k to get the sorted sequence.

   1. The number of key comparisons will always be maximized when the keys in K are already sorted.
   2. Let $|K_1|$ and $|K_2|$ be the number of keys in K_1 and K_2 respectively. The expected value of $\max(|K_1|, |K_2|)$ is $\frac{2}{3}n$.
   3. The algorithm will compare two keys k_i and k_j at most once, so the number of key comparisons is $O(n^2)$.
   4. The probability that the algorithm will compare k_i and k_j $(n \geq i > j \geq 1)$ is $\frac{2}{i-j+1}$.
   5. Let $P = \{(i,j) | 1 \leq i, j \leq n, i \neq j\}$. The expected total number of key comparisons is $\sum_{(i,j)\in P} \frac{1}{|k_i - k_j|+1}$.

7. (3 points) We can use a linked list to implement a stack. We assume that we use structure `Node` with two members to represent a node of the linked list. The member `data` stores the data, and the member `next` points to the next node in the linked list. Also we use a variable `head` to point to the *first* node of the linked list. `head` is initialized to `NULL`.

   1. Pushing the value of a variable i into the stack can be implemented as follows.

      ```
      newHead = malloc(sizeof(Node));
      newHead->data = i;
      newHead->next = head;
      head = newHead;
      ```

   2. Popping the top of the stack into a variable i can be implemented as follows.

      ```
      free(head);
      i = head->data;
      head = head->next;
      ```

   3. To test if the stack is empty we can check if the `head` is a `NULL` pointer.
   4. We can improve the push/pop efficiency of this stack by adding another pointer `tail` that points to the *end* of the linked list, so we can access the *last* node in the list in $O(1)$ time.
   5. We can improve the push/pop efficiency of this stack by adding another pointer member into every node, so it becomes a *double linked list*. That is, we can traverse in *both* directions in this linked list.

8. (3 points) A *randomized* linked list sorting algorithm works as follows. We want to build a linked list in which the keys are in *increasing order*. That is, every node has a *smaller* key than its *successor* in the list. For ease of discussion we assume that the keys are *distinct* integers from 1 to n. The algorithm *randomly* picks a key from the remaining keys, and inserts it into the list. This process repeats until all keys are inserted. The inserted key k will *skip* those at the *beginning* of the list that are *smaller* than it, and will *stop* at the first key p that are *greater* than it. We then insert the key k *before* p. To ensure that all keys will stop we assume that initially the list has only one key, $n+1$. After we insert all keys we will have a sorted linked list from 1 to $n+1$.

   1. Every inserted key will stop exactly once.
   2. The smallest key 1 will not skip any keys.
   3. The largest key n will always skip $n-1$ keys.

見背面

4. The expected number of keys that the key i will skip is $\Omega(i)$.
5. The key i will skip j if and only if $i > j$ and i is inserted after j is inserted, so the expected value of the total number of key comparisons is $\Theta(n^2)$.

9. (3 points) A *binary search tree* is a tree where every node has at most *two* children. For ease of discussion we assume that every node has a *distinct* key. In addition, all keys in the left subtree are *smaller* than the key of root, and all keys in the right subtree are *larger* than the key of the root. A tree node is a *leaf* if it does *not* have any children. The *height* of the tree is the length of the *longest* path from the root to a leaf. The *successor* of a node x is the node right *after* x in increasing order, and the *predecessor* of a node x is the node right *before* x in increasing order.

   1. We can locate the *smallest* key in a binary search tree in $O(h)$ time in the worst case, where h is the height of the tree.
   2. The height of a binary search tree of n nodes is $O(\log n)$.
   3. We can locate the *largest* key in a binary search tree in $O(\log n)$ time in the worst case, where n is the number of the nodes in the tree.
   4. Assume that a node z has two children. We can remove z while still maintaining the sorted order by locating its *successor* y, connecting y's parent directly to the only child of y (if there is one, otherwise make it null), which effectively removes the node y from the tree, and replacing the key of z with that of y.
   5. Assume that a node z has two children. We can remove z while still maintaining the sorted order by locating its *predecessor* y, connecting y's parent directly to the only child of y (if there is one, otherwise make it null), which effectively removes the node y from the tree, and replacing the key of z with that of y.

10. (3 points) A red-black tree is a binary tree where every node has a color (red or black), and has the following properties.

    1. The root is black.
    2. All paths from the root to a leaf have the same number of black nodes.
    3. Every red node must have two black children.

    For ease of discussion we assume that all null pointers (a special pointer to "nothing") point to a special *black* node. As a result all counting of the second property ends at this special node.

    1. If we do *not* guarantee the first property, but still guarantee the second and the third properties, we will *not* be able to guarantee that the height of the tree is $O(\log n)$.
    2. It is possible that there are *no* red nodes in a red-black tree of 100000 nodes.
    3. We insert a key k into a *non-empty* red-black tree just like into an ordinary binary search tree as a leaf, attach to it two null pointers to the special black node, and then immediately *stop* without any adjustment. If we color k *red*, we may violate the second property above.
    4. Consider the scenario above. If we color k *red*, we may violate the third property above.
    5. Consider the scenario above. If we color k *black*, we will violate the second property above.

Please Turn Over (there are more questions on the next pages).

接次頁

Answer the following questions in details.

11. (5 points) Given an array of n integers $A = (a_1, a_2, ..., a_n)$ $(n > 0)$, you are asked to find the maximum and minimum elements with minimum comparisons. A simple solution is to iteratively compare each element with current max and min. It takes $2n$ comparisons in total.

    (a) (3 points) Devise an algorithm that requires less than $1.67n$ comparisons.

    (b) (2 points) How many comparisons does your algorithm take? Show your derivation.

12. (15 points) There are n stations along a coastal railway $(n > 0)$. You're planning to select some of them to open cafes. Three arrays S, L and R have been given, including

    - $S = (s_1, s_2, ..., s_n)$: the list of the stations from s_1 (first) to s_n (last).
    - $L = (l_1, l_2, ..., l_n)$: the locations of the stations, where l_i is the distance of s_i from the first station s_1. So $l_1 = 0$ and l_n is the length of the railway. $l_1 < l_2 < ... < l_n$.
    - $R = (r_1, r_2, ..., r_n)$: the revenues of the cafes, where r_i (> 0) is the revenue for opening a cafe in s_i.

    The only one constraint in your plan is that the distance of any pair of your selected stations should be longer than a given threshold T. If s_i and s_j $(i \neq j)$ are selected, the total revenue would be $l_i + l_j$. Different selection leads to different total revenue. Given S, L, R and T, your goal is to pick up a subset of the stations to maximize the total revenue under the constraint. Suppose that $f(n)$ returns the maximum total revenue for the cafes you select from the first n stations. Please answer the following questions. **No code is required (code will not be graded).**

    (a) (9 points) Give an $O(n^2)$ solution by defining a recurrence formula for $f(n)$. Clearly explain the meaning of your formula and why it can be computed in $O(n^2)$ time.

    (b) (6 points) Consider the special case that the distance difference between two consecutive stations is 1, i.e., $l_{i+1} - l_i = 1$ $(1 \leq i \leq n-1)$. Give an $O(n)$ solution by defining a recurrence formula for $f(n)$. Clearly explain the meaning of your formula and why it can be computed in $O(n)$ time.

13. (15 points) A disjoint-set data structure supports two operations. One is UNION(x, y), which merges the two roots of the trees containing x and y. The other is FIND(x), which returns the root of the tree containing x. Union-by-height is a union heuristic, which keeps track of the heights of the trees to attach the shorter tree to the root of the taller tree.

    (a) (9 points) Given an undirected graph $G = (V, E)$, where $V = \{v_1, ..., v_m\}$ $(m > 0)$ and $E = \{e_1, ..., e_n\}$ $(n > 0)$, devise an algorithm to check if the graph G is a tree or not by using UNION(v_i, v_j) with union-by-height and FIND(v_k) with no path compression, where v_i, v_j and $v_k \in V$. Clearly describe your solution and analyze the time complexity of your algorithm in the worst case. **No code is required (code will not be graded).**

    (b) (6 points) The union-by-height heuristic prevents the height of a tree from growing linearly. Consider another heuristic, called union-by-descendant, which always attaches the tree with fewer descendants to the root of the tree with more descendants. Suppose that path compression is not applied. Which of the two heuristics is asymptotically better? Why?

14. (20 points) A tree T is assumed to be simple, undirected, and with **positive edge-weights**. Let $d_T(u, v)$ denote the distance between u and v on T. For a vertex v, the *eccentricity* of v is the maximum of the distance to any vertex in the tree, i.e., $\max_{u \in V}\{d_T(v, u)\}$. The *diameter* of a tree is the maximum of the eccentricity of any vertex in the tree. (The term "diameter" is overloaded. It is defined as the maximum eccentricity and also as the path of length equal to the maximum eccentricity.) The *radius* of a tree is the minimum eccentricity among all vertices in the tree, and a *center* of a tree is a vertex with eccentricity equal to the radius.

    (a) (5 points) Prove or disprove that there exists a tree with three different diameters and two centers.

    (b) (5 points) Prove or disprove that there exists a tree with three centers.

見背面

(c) (10 points) Prove or disprove that any center of a given tree T must lie in the diameter of T.

15. (15 points) A Single Nucleotide Polymorphism (SNP, pronounced *snip*) is a single nucleotide variation in the genome that recurs in a significant proportion of the population of a species. The patterns of *Linkage Disequilibrium* (LD) observed in the human population reveal a block-like structure. LD refers to the association that particular alleles at nearby sites are more likely to occur together than would be predicted by chance. The entire chromosome can be partitioned into high LD regions interspersed by low LD regions. The high LD regions are usually called "haplotype blocks," and the low LD ones are referred to as "recombination hotspots." Since there is little or no recombination within a haplotype block, these SNPs are highly correlated. Consequently, a small subset of SNPs, called tag SNPs or haplotype tagging SNPs, is sufficient to categorize the haplotype patterns of the block. It has been shown that we can recast the tag SNP selection problem as **Problem W**, which is,"Given a universal set $\mathcal{U} = \{u_1, u_2, ..., u_n\}$ and a family $\mathcal{F} = \{F_1, F_2, ..., F_m\}$ of subsets of $\mathcal{U}$, find a minimum-size subset $\widehat{\mathcal{C}}$ of $\mathcal{F}$, such that every element of $\mathcal{U}$ belongs to at least one subset in $\widehat{\mathcal{C}}$."

(a) (5 points) Prove or disprove that a greedy approach always delivers an optimal solution for **Problem W**.

(b) (10 points) Prove or disprove that **Problem W** is NP-complete.

試題隨卷繳回