Homework 5
Fundamental Algorithms, Spring 2003, Professor Yap

Due: Wed May 7, in class
Solutions prepared by: Chee Yap with help from T.A.'s.

INSTRUCTIONS:

- Please read the questions carefully. When in doubt, please ask.

---

1. (20 Points) Recursive Dynamic Programming
Equation (1) Lecture VII (p.12) describes the Dynamic Programming solution for the optimal triangulation problem. The "bottom-up" implementation of this equation as a triply-nested for-loop was given in page 14. In this question, we want to solve this "top-down", by using a recursive algorithm. We assume that the weight $W(i, j, k)$ is easily computed for any $1 \le i < j < k \le n$.
(i) Briefly explain how a naive recursive algorithm would be exponential.
(ii) Describe an efficient recursive algorithm for optimal triangulation. You will need to use some global data structure for sharing information across subproblems.
(iii) Briefly analyze the complexity of your solution.
(iv) Does your algorithm ever run faster than the bottom-up implementation? Can you make it run faster on some inputs?

SOLUTION:
(i) Let $T(n)$ be the cost of solving a problem of size n. In dynamic programming, we solve *at least one* subproblems of size $n-1, n-2, \ldots, 1$. The cost to combine the answers is at least n. Hence

$$\begin{array}{lcll} T(n) & \ge & n + \sum_{i=1}^{n-1} T(i) & \text{— (a)} \\ T(n-1) & \ge & n-1+\sum_{i=1}^{n-2} T(i) & \text{— (b)} \\ T(n) - T(n-1) & = & 1 + T(n-1) & \text{— (a)-(b)} \\ T(n) & = & 1 + 2T(n-1). & \end{array}$$

It is easy to see by induction that $T(n) > 2^n$ (assuming $T(0) = 1$.

(ii) We maintain a matrix C where $C[i, j]$ $(1 \le i < j \le n)$ is the optimal cost of the subproblem $P(i, j)$. Initially, $C[i, j] = \infty$ for all i, j. The recursive algorithm $R(i, j)$ returns the optimal cost $C(i, j)$ while updating the array C.

TRIANGULATION ALGORITHM:
1. Initialize C with $C[i, j] = \infty$ for all $1 \le i < j \le n$.
2. Return $R(1, n)$.

RECURSIVE ALGORITHM $R(i, j)$:
1. If $i = j - 1$, then $C[i, j] = 0$ and return 0.
2. If $C[i, j] < \infty$ then return $C[i, j]$.
3. For each $k = i + 1, \ldots, j - 1$,
4. $\quad C[i, j] = \min\{C[i, j], W(i, k, j) + R(i, k) + R(k, j)\}$.
5. Return $C[i, j]$.

(iii) Complexity Analysis: Each entry is filled in at most once. To fill an entry, we use $O(n)$ time. Since there are $O(n^2)$ entries, the overall time is $O(n^3)$.

(iv) Improvements: As implemented above, the recursive solution is no better than the bottom up method. But there is the potential in a recursive solution to avoid solving certain subproblems. This requires introducing some "bounding function". Here is a simple way to do it. Suppose we maintain another matrix $E[i,j]$ in parallel to $C[i,j]$. The entries $E[i,j]$ is a LOWER bound on the cost of $C[i,j]$. For instance, we could set $E[i,j] = \min\{W(i,k,j) : i < k < j\}$ but this does not seem easy to compute without additional properties about the functions $W(i,k,j)$. In the lecture notes, we noted several special cases, e.g., $W(i,k,j) = a_i + a_k + a_j$, etc. In some of these cases we can easily compute $E[i,j]$ in $O(n^2)$ time. Once this is done, we can modify line 4 in the above recursive algorithm as follows: "if $C[i,j] \geq W(i,j,k) + E[i,k] + E[k,j]$ then do X" where X is the assignment in the current line 4.

2. (20 Points) String Alignment Problem
Recall the string edit problem, where the operations are Insert, Delete and Replace. Given two strings X, Y, we want the minimum "cost" of modifying X so that the two strings becomes identical. Here is our definition of "cost":
(a) Each deletion or insertion costs 2 units.
(b) Each replacement costs 1 unit.
(c) Each "original" **match** involving a character from X and the same character in Y costs -2 units. What we mean by "original" match will be clear in an example. Note that this cost occurs in the absence of any operations!

In contrast, in the original string edit problem, matches have no cost while each of the other operations costs 1 unit. The minimum costs of such a sequence of operations (or non-operations) is called the **alignment cost** for X, Y and is denoted $A(X,Y)$. For instance, $X = cga$ and $Y = acaat$. We can first insert a in the front of X to get $X' = acga$ (costs 2), and delete t from Y to get $Y' = acaa$. To indicate the various copies of a letter, we use subscripts: $X' = a_1cga_2$ and $Y' = a_3ca_4a_5$. Now we make the obvious one-one correspondence between X' and Y': we get matches at positions $1, 2, 4$. The matches at positions 2 and 4 costs -2 units each. However, the match at position 1 (between a_1 with a_3) is a result of inserting a_1 and so it has already been counted. In other words, it is not an "original match". Since there is no match at 3, we need to replace g by an a to get a match – this replacement costs 1 unit. This shows that $A(X,Y) \leq 2 + 2 - 2 - 2 + 1 = 1$. [Can you show a lower cost alignment of X, Y?]

To summarize, here is how we perform a single alignment of X and Y: first perform a sequence of insertions and deletions until the strings have the same length. Then align the result in the obvious way, and find all the original matches, as well as perform any replacements as needed. Add up the cost of deletions/insertions, original matches and replacements. This problem arise in DNA sequencing in computational biology.

(i) Give the recursive formula for $A(X,Y)$ and justify it.
(ii) Compute $A(X,Y)$ where X, Y are the strings `AATTCCCGA` and `GCATATT`. You must organize this computation systematically as in the LCS problem.

SOLUTION:
(i) Suppose we have two strings X and Y and want to give a recursive version of the alignment cost A(X,Y). Suppose we transform X into X' and Y into Y' such that the alignment is optimal. Let's look at the last characters in X and Y, call them x and y. Similarly, let the last characters in X' and Y' be x' and y'. Since we do an optimal alignment, it is clear that at least one of x' and y' is from the initial string. (a position where both characters are newly inserted can clearly be removed in an optimal alignment). To simplify the analysis we suppose we only do insertions; this is without loss of generality, since a delete in one string is the same as an insert in the other one and the cost is the same. Hence there are three possibilities:
1. $(x = x')$ and $(y \neq y')$. This means an insertion of y' in Y. In this case the cost is $A(X,Y) = Cost(Insertion) + A(X_p, Y) = 2 + A(X_p, Y)$ where X_p is the prefix of X obtained by dropping the last character in X.
2. $(x \neq x')$ and $(y = y')$. This means an insertion of x' in X. In this case we get the recurrence $A(X,Y) = 2 + A(X, Y_p)$.
3. $(x = x')$ and $(y = y')$. This means the original characters at the end of X and Y are kept. There are two possibilities: (a) If $x = y$ then we get an original match with cost -2. The recurrence is $A(X,Y) = -2 + A(X_p, Y_p)$. (b) If $x \neq y$, the we need a replacement with cost 1, giving the recurrence $A(X,Y) = 1 + A(X_p, Y_p)$.

The base case is when $|X| = 0$ or $|Y| = 0$. In this case, $A(X,Y) = 2\max\{|X|, |Y|\}$. Hence the overall recurrence is

$$A(X,Y) = \begin{cases} 2\max\{|X|, |Y|\} & \text{if } |X|\cdot|Y| = 0, \\ \min\{2 + A(X_p, Y), 2 + A(X, Y_p), -2 + A(X_p, Y_p)\} & \text{if x=y}, \\ \min\{2 + A(X_p, Y), 2 + A(X, Y_p), 1 + A(X_p, Y_p)\} & \text{else} \end{cases}$$

We can simplify the recurrence in case $x = y$, by noticing that $\min\{2 + A(X_p, Y), 2 + A(X, Y_p), -2 + A(X_p, Y_p)\} = -2 + A(X_p, Y_p)$. To see this, we note that $A(X_p, Y_p) \le 2 + A(X_p, Y)$ since we can insert y at the end of Y_p to create Y. Hence $2 + A(X_p, Y) \ge A(X_p, Y_p) > -2 + A(X_p, Y_p)$. Similarly, $2 + A(X, Y_p) \ge A(X_p, Y_p) > -2 + A(X_p, Y_p)$.

$$A(X,Y) = \begin{cases} 2\max\{|X|, |Y|\} & \text{if } |X|\cdot|Y| = 0, \\ -2 + A(X_p, Y_p) & \text{if x=y}, \\ \min\{2 + A(X_p, Y), 2 + A(X, Y_p), 1 + A(X_p, Y_p)\} & \text{else} \end{cases}$$

(ii) Now that we know the recurrence we only have to fill in the usual table row by row.

	-	G	C	A	T	A	T	T
-	0	2	4	6	8	10	12	16
A	2	1	3	2	4	6	8	10
A	4	3	2	1	3	2	4	6
T	6	5	4	3	-1	1	0	2
T	8	7	6	5	1	0	-1	-2
C	10	9	5	7	3	2	1	0
C	12	11	7	6	5	4	3	2
C	14	13	9	8	7	6	5	4
G	16	12	11	10	9	8	7	6
A	18	14	13	9	11	7	9	8

The value we are interested in is at the bottom right corner of the table, so $A(X,Y) = 8$.

3. (20 Points) Probabilistic Counters
Recall the counter problem where, given a binary counter C which is initially 0, you can perform the operation `inc`(C) to increments its value by 1. Now we want to do **probabilistic counting**: each time you call `inc`(C), it will flip a fair coin. If heads, the value of C is incremented and otherwise the value of C is unchanged. Now, at any moment you could call `look`(C), which will return *twice* the current value of C. Let X_m be the value of `look`(C) after you have made m calls to `inc`(C).
(a) Note that X_m is a random variable. What is the sample space Ω here?
(b) Let $P_m(i)$ be the probability that `look`$(C) = 2i$ after m `inc`'s. State a recurrence equation for $P_m(i)$ involving $P_{m-1}(i)$ and $P_{m-1}(i-1)$.
(c) Give the exact formula for $P_m(i)$ using binomial coefficients. HINT: you can either use the model in (a) to give a direct answer, or you can try to solve the recurrence of (b). You may recall that binomial identity $\binom{m}{i} = \binom{m-1}{i} + \binom{m-1}{i-1}$.
(d) In probabilistic counting we are interested in the *expected* value of `look`(C), namely $E[X_m]$. What is the expected value of X_m? HINT: express $E[X_m]$ using $P_m(i)$ and do some simple manipulation involving binomial coefficients. If you do not see what is coming out, try small examples like $m = 2, 3$ to see what the answer is.

[NOTE: The expected value of X_m can be odd even when the actual value returned is always even. Using a generalization of these ideas, you can probabilistically count to 2^{2^n} with an n-bit counter.]

SOLUTION:
(a) Each increment has one of two possible outcomes, denoted 0 or 1. So a sequence of m increments corresponds to a binary string of length m. Thus the sample space is $\Omega = \{0,1\}^m$.
(b) Let $E_{m,i}$ be the event that the counter value is i after m increments. F_m be the event that the mth

increment causes the counter value to increase by 1. Thus $E_{m,i} = (E_{m-1,i} \cap \overline{F_m}) \cup (E_{m-1,i-1} \cap F_m)$. But $\Pr(E_{m-1,i} \cap \overline{F_m}) = P_{m-1}(i)/2$ and $\Pr(E_{m-1,i-1} \cap F_m) = P_{m-1}(i-1)/2$. These are disjoint events, so we may add them up. $P_m(i) = \Pr(E_{m,i}) = (P_{m-1}(i) + P_{m-1}(i-1))/2$.
(c) The answer is $P_m(i) = 2^{-m}\binom{m}{i}$. One way to solve this is to interpret the probability in terms of the sample space in part (a). Thus $P_m(i)$ is just the number of sample points $w \in \Omega$ with exactly i one's in w. There are $\binom{m}{i}$ to choose these i one's. Each choice has 2^{-m} chance of occurring. Hence $P_m(i) = 2^{-m}\binom{m}{i}$. Another way is to solve the recurrence used (b) directly, recalling that this recurrence looks like

$$\binom{m}{i} = \binom{m-1}{i} + \binom{m-1}{i-1}.$$

We may guess that $P_m(i) = 2^{-m}\binom{m}{i}$, and then verify by induction.
(d)

$$\begin{aligned} E[X_m] &= \textstyle\sum_{i=0}^{m} 2iP_m(i) \\ &= \textstyle\sum_{i=1}^{m} 2i2^{-m}\binom{m}{i} \\ &= 2^{-m+1}\textstyle\sum_{i=1}^{m} m\binom{m-1}{i-1} \\ &= m2^{-m+1}\textstyle\sum_{i=0}^{m-1}\binom{m-1}{i} \\ &= m \end{aligned}$$

where the last equation comes from the fact that, for any $k \geq 0$, $2^k = (1+1)^k = \sum_{i=0}^{k}\binom{k}{i}$. After m increments, the expected value of $Look(C)$ is m! The odd thing is that, this is true even when m is an odd number, and yet the value $Look(C)$ is never odd.

4. (20 Points) Hashing

(a) Compute the sequence $\{\alpha\}, \{2\alpha\}, \ldots, \{n\alpha\}$ for $n = 10$ and $\alpha = \phi$ (= the golden ratio $(1+\sqrt{5})/2 = 1.618\ldots$). You may compute to just 4 decimal positions using any means you like.
(b) Let

$$\ell_0 > \ell_1 > \ell_2 > \cdots$$

be the new lengths of subsegments, in order of their appearance as we insert the points $\{n\phi\}$ (for $n = 0, 1, 2\ldots$) into the unit interval. For instance, $\ell_0 = 1, \ell_1 = 0.61803, \ell_2 = 0.38197$. Compute ℓ_i for $i = 0, \ldots, 10$.
(c) Using the multiplication method with $\alpha = \phi$, please insert the following set of 16 keys into a table of size $m = 10$. Treat the keys as integers by treating the letters `A, B, ..., Z` as $1, 2, \ldots, 26$, with the rightmost position having a value of 1, the next position with value 26, the third with value $26^2 = 676$, etc. Thus `AND` represents the integer $(1 \times 26^2) + (14 \times 26) + (4 \times 1) = 1044$. This is sometimes called the **26-adic** notation. To resolve collision, use separate chaining.

```
AND, ARE, AS, AT, BE, BOY, BUT, BY, FOR, HAD,
HER, HIS, HIM, IN, IS, IT
```

We just want you to display the results of your final hashing data structure.
(d) Use the division method on the same set of keys as (c), but with $m = 17$.

SOLUTION:

We solve this problem with the help of a C++ program which is included at the end of this file.

(a) First 10 numbers and (b) first 10 distinct lengths

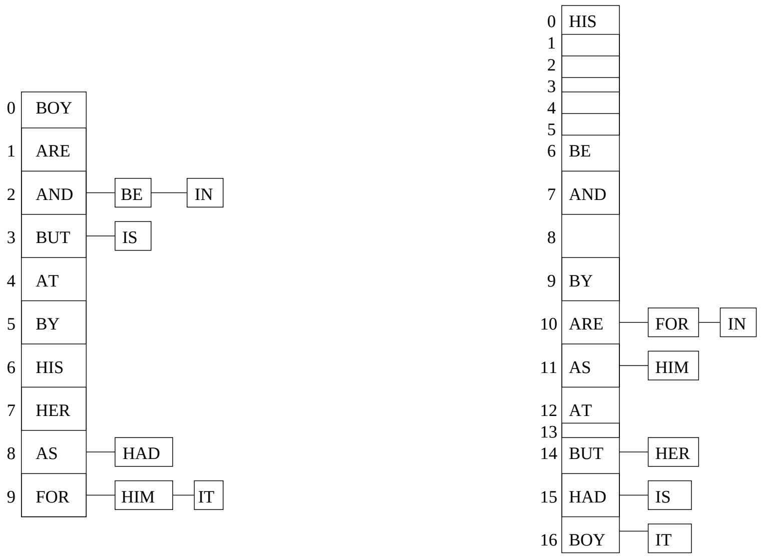
In the table below, r is the number $\{i\phi\} = i\phi - \lfloor i\phi \rfloor$, L_1 and L_2 are the lengths of the 2 new subintervals formed by the insertion of i-th number. The last column shows the newly created lengths ℓ_j's.

i	r	L_1	L_2	ℓ_j
1	0.6180	0.6180	0.3820	$\ell_1 = 0.6180, \ell_2 = 0.3820$
2	0.2361	0.2361	0.3820	$\ell_3 = 0.2361$
3	0.8541	0.2361	0.1459	$\ell_4 = 0.1459$
4	0.4721	0.2361	0.1459	
5	0.0902	0.0902	0.1459	$\ell_5 = 0.0902$
6	0.7082	0.0902	0.1459	
7	0.3262	0.0902	0.1459	
8	0.9443	0.0902	0.0557	$\ell_6 = 0.0557$
9	0.5623	0.0902	0.0557	
10	0.1803	0.0902	0.0557	
...	...	...	...	
13	0.0344	0.0344	0.0557	$\ell_7 = 0.0344$
...	...	...	...	
21	0.9787	0.0344	0.0213	$\ell_8 = 0.0213$
...	...	...	...	
34	0.0132	0.0132	0.0213	$\ell_9 = 0.0132$
...	...	...	...	
55	0.9919	0.0132	0.0081	$\ell_{10} = 0.0081$

(c), (d): Hashing 16 data items by multiplication ($m = 10$), and hashing 16 data items by division ($m = 17$): In the following table, *num* is the integer corresponding to data item in 26-adic notation, h_1 is the hash value for part (c), h_2 is the hash value for part (d). The hashing data structures are seen in Figure 1.

item	*num*	h_1	h_2
AND	1044	2	7
ARE	1149	1	10
AS	45	8	11
AT	46	4	12
BE	57	2	6
BOY	1767	0	16
BUT	1918	3	14
BY	77	5	9
FOR	4464	9	10
HAD	5438	8	15
HER	5556	7	14
HIS	5661	6	0
HIM	5655	9	11
IN	248	2	10
IS	253	3	15
IT	254	9	16

5. (20 Points) NP-Completeness
We guide you through Exercise 6.2 in Lecture XXX on NP-Completeness. The problem L is to recognize whether a given bigraph G is "triangular" or not. To show that L is Karp-reducible to SAT, you need to construct a Boolean formula $\phi(G)$ such that G is triangular iff $\phi(G) \in SAT$. Moreover, this construction must be done in polynomial time.
(i) If $G = (V, E)$ and $|V|$ is not divisible by 3 then there is no solution. What would you output as $\phi(G)$ in this case?
(ii) Suppose $|V| = 3m$. So our goal is to form m disjoint triangles from the vertices of G. Introduce the Boolean variable x_{ij} which corresponds to the proposition "Node i is in the jth Triangle". Here, $i \in V$ and $j = 1, \ldots, m$. Using these variables, you construct a Boolean formula $F_1(i)$ that is satisfiable iff i is in at least one of the m triangles?



(a) (b)

Figure 1: (i) Hashing by multiplication with $m = 10$, (ii) Hashing by Division with $m = 17$

(iii) Similarly, construct $F_2(i)$ that is satisfiable iff i is in at most one triangle.
(iv) Construct a formula $F_3(j)$ that is satisfiable iff the jth triangle has at least three nodes.
(v) Construct a formula $F_4(j)$ that is satisfiable iff the jth triangle has at most three nodes.
(vi) Construct a formula $F_5(j)$ that is satisfiable iff each pair of vertices in the jth triangle has an edge in the graph G. [NOTE: this is the first time you are actually using specific information about the edges of G. You know G since it is in the input.]
(vii) Using the above formulas, describe a formula $\phi(G)$ that is satisfiable iff G is triangular. You must prove this claim about $\phi(G)$.
(viii) Conclude that L is Karp-reducible to SAT.

SOLUTION:
(i) If $|V|$ is not divisible by 3, let $\phi(G) = x \wedge \overline{x}$ (this is clearly not satisfiable). In the rest of this solution, assume $n = |V| = 3m$.
(ii) $F_1(i) = \bigvee_{j=1}^{m} x_{ij}$.
(iii) $F_2(i) = \bigwedge_{1\le j<j'\le m} (\overline{x_{ij}} \vee \overline{x_{ij'}})$.
(iv) $F_3(j) = \bigvee_{1\le i<i'<i''\le n} (x_{ij} \wedge x_{i'j} \wedge x_{i''j})$.
(v) $F_4(j) = \bigwedge_{1\le i<i'<i''<i''''\le n} (\overline{x_{ij}} \vee \overline{x_{i'j}} \vee \overline{x_{i''j}} \vee \overline{x_{i'''j}})$.
(vi) Let x be a new variable. For all $1 \le i < i' \le n$, let $E_{ii'}$ be equal to x is (i, i') is an edge of G, and $E_{ii'}$ be equal to $x \wedge \overline{x}$ otherwise. [Note that $E_{ii'}$ is either satisfiable or not satisfiable (as in part (i)). Then we define

$$F_5(j) = \bigwedge_{1\le i<i'\le n} (\overline{x_{ij}} \vee \overline{x_{i'j}} \vee E_{ii'}).$$

(vii) If $|V|$ is not divisible by 3 then let $\phi(G)$ be the formula in part (i). Otherwise, let $n = 3m$ and $\phi(G) = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5$ where $F_1 = \bigwedge_{i=1}^{n}(F_1(i)$, $F_2 = \bigwedge_{i=1}^{n} F_2(i))$, $F_3 \bigwedge_{j=1}^{m}(F_3(j)$, $F_4 \bigwedge_{j=1}^{m}(F_4(j)$ and $F_5 \bigwedge_{j=1}^{m}(F_5(j)$. Each F_i is called a "clause". We have used the formulas constructed in parts (ii)-(vi). The formula in part (i) is not needed (you could use it if you like, only as a "filter") We now prove the two properties needed of $\phi(G)$:
(A) If $\phi(G)$ is satisfiable, then G is triangular.
(B) If G is triangular, then $\phi(G)$ is satisfiable.
To see (A), suppose I is as assignment that makes $\phi(G)$ true. We can form m triangles as follows:

the jth triangle comprises those nodes i such that $I(x_{ij}) = 1$. Each of the clauses $F_1, \ldots, F_5$ in $\phi(G)$ ensures that G is truly triangular.
To see (B), suppose G is triangular. Let there be m triangles. We can use this information to construct an assignment I as follows: for each i, j, let $I(x_{ij}) = 1$ iff node i is in triangle j. Then we see that each of the clauses in $\phi(G)$ is satisfied by I.
(viii) Given G (encoded), we can compute $\phi(G)$ (encoded) in polynomial time. It is at most $O(n^5)$ time. Moreover, G is triangular iff $\phi(G) \in SAT$. This is exactly what is required by Karp reducibility.

THE REMAINING QUESTIONS CARRIES NO CREDIT. BUT YOU ARE ENCOURAGED TO GO OVER THEM. IF YOU HAVE NO TIME, AT LEAST THINK TRY TO UNDERSTAND THE ISSUES AND HAVE A STRATEGY TO SOLVE THEM.

6. (0 Points)
(a) Suppose you have a random number generator, which is a function $rand()$ that returns a real number r in the range $0 \le r < 1$ with "uniform" probability (this means that for any $0 \le a < b \le 1$, the probability of r lying in $[a, b]$ is $b - a$). Given any n, how do you generate an integer in the range $\{0, 1, \ldots, n-1\}$ with equal probability?
(b) Given an array $A[0..n]$ of numbers ($A[i]$ is some number x_i). How do you compute a random permutation of these numbers?

SOLUTION: See Section 4 in Lecture 8 (page 11).

7. (0 Points)
To understand Coalesced Chaining, we ask you to provide the algorithms for LookUp(key k), Insert(key k, data d), and Delete(key k) for this form of hashing. As usual, we assume a hash function $h : U \to \mathbb{Z}_m$. The hash table is denote $T[0..m-1]$ where the ith entry is $T[i]$. We assume that $T[i]$ has three fields, $T[i]$.`Key`, $T[i]$.`Data` and $T[i]$.`next`. The value of $T[i]$.`Key` is either a key (an element of U) or the special values EMPTY or DELETED. The value of $T[i]$.`next` is either an element of $\mathbb{Z}_m$ or -1. Provide the algorithmic details for the three dictionary operations: LookUp, Insert and Delete.

SOLUTION:

Here each slot $T[i]$ is potentially the node of some chain, and all nodes are allocated from the hash table T. In this way, we avoid the dynamic memory management found in separate chaining. More precisely, we assume that $T[i]$ has three fields:

(a) $T[i]$.`Key` which either stores a key (element of U) or the special values EMPTY or DELETED, and
(b) $T[i]$.`next` which stores either an element of $\mathbb{Z}_m$ or -1.
(c) $T[i]$.`Data` which stores associated data. This is clearly important in practice, but its use is application dependent. As usual, we ignore this field in our discussions of the algorithms.

We use the `next` field to form the chains: If $T[i]$.`next` $\in \mathbb{Z}_m$, then it is a pointer to the next node in a chain; otherwise, $T[i]$ is the end of a chain and $T[i]$.`next` $= -1$. We also maintain a global variable n which is the number of keys currently in the hash table. Initially, $n = 0$ and $T[i]$.`Key` is EMPTY and $T[i]$.`next` $= -1$ for all i.

To lookup a key k, we first check $T[h(k)]$.`Key` $= k$. In general, suppose we have just checked $T[i]$.`Key` $= k$ for some index i. If this check is positive, we have found k and return i with success. If not, and $T[i]$.`next` $= -1$, we return a failure value. Otherwise, we let $i = T[i]$.`next` and continue the search.

To insert a key k, we first check to see if the n number of items in the table has reached the maximum value m. If so, we return a failure. Otherwise, we perform a lookup on k as before. If k is found, we also return a failure. If not, we must end with a slot $T[i]$ where $T[i]$.`next` $= -1$. In this case, we continue searching from i for the first j that does not store any keys (i.e., $T[j]$.`Key` is either EMPTY or DELETED. This is done sequentially: $j = i+1, i+2, \ldots$ (where all the index arithmetic is modulo m). We are bound to find such a j. Then we set $T[i]$.`next` $= j$, $T[j]$.`next` $= -1$, $T[j]$.`Key` $= k$ and increment n. We may return with success.

What about deletion? We look for the slot i such that $T[i]$.Key $= k$. If found, we set $T[i]$.Key $=$ $DELETED$. Otherwise deletion failed.

FOOD FOR THOUGHT: Why do we need DELETED values? What is the implication of this in terms of efficiency?

8. (0 Points)
Describe the algorithmic details of our offline Quicksort algorithm in the notes. You must make very explicit choices for the data structures (how the input is represented and how whether you are using linked lists, etc).

SOLUTION: Omitted.

```
/**************************************************
C++ PROGRAM for Question 4
Programmed by Igor Chikanian
*************************************************/

#include "math.h"

float find_pred(float r, float h[100], int n);
float find_succ(float r, float h[100], int n);
int hash1(const char word[3]);
int hash2(const char word[3]);
int num26(const char word[3]);
int code26(char c);

float g = (1.0 + sqrt(5))/2.0; // golden ratio

int main(int argc, char* argv[])

int i, k;
float f[100], r[100];
float pred, succ, new1, new2;
char* data[] = " ", // this is for data[0]
"AND","ARE"," AS"," AT"," BE","BOY","BUT"," BY",
"FOR","HAD","HER","HIS","HIM"," IN"," IS"," IT";
FILE* pf = fopen("hash.txt","w");
fprintf(pf,
"(a) First 60 numbers and (b) first 10 distinct lengths (new1,new2)\n");
fprintf(pf,
"-----------------------------------------------------------------\n");
for (i=1; i<=60; i++)
f[i] = i*g; // this is full i*g
k = f[i]; // this is floor(i*g)
r[i] = f[i] - k; // this is i*g = i*g - floor(i*g)
pred = find_pred(r[i],r,i);
succ = find_succ(r[i],r,i);
new1 = r[i] - pred;
new2 = succ - r[i];
fprintf(pf,
"i=
i, f[i], k, r[i], new1, new2);
```

```
fprintf(pf, "\n(c) Hashing 16 data items by multiplication, m=10\n");
fprintf(pf,
"-----------------------------------------------------\n");
for (i=1;i<=16;i++)
fprintf(pf,
"i=
i, data[i], num26(data[i]), hash1(data[i]));

fprintf(pf, "\n(d) Hashing 16 data items by division, m=17\n");
fprintf(pf,
"-----------------------------------------------------\n");
for (i=1;i<=16;i++)
fprintf(pf, "i=
i, data[i], num26(data[i]), hash2(data[i]));

fclose(pf);
return 0;


float find_pred(float r, float h[100], int n)
/* finds a proper predecessor of r in array h */
int i;
float curr_pred = 0.0;
for (i=1; i<=n; i++)
if ((h[i]<r)&&(curr_pred<h[i])) curr_pred = h[i];

return curr_pred;


float find_succ(float r, float h[100], int n)
/* finds a proper successor of r in array h */
int i;
float curr_succ = 1.0;
for (i=1; i<=n; i++)
if ((h[i]>r)&&(curr_succ>h[i])) curr_succ = h[i];

return curr_succ;


int hash1(const char word[3])
int k,i;
float f,r;
k = num26(word);
f = k*g; // this is full k*g
i = f; // this is floor(k*g)
r = f - i; // this is k*g = k*g - floor(k*g)
return 10*r; // this is floor(m*k*g), m=10


int hash2(const char word[3])
int k;
k = num26(word);
return (k
```

```
int num26(const char word[3])
return (26 * 26 * code26(word[0]))
+ (26 * code26(word[1]))
+ code26(word[2]);
```

```
int code26(char c)
if (c==' ') return 0;
else return c-64;
```