```
/**
 * isPerfect implemented by Matt Boutell on October, 2007.
 */
#include <stdio.h>
#include <float.h>
/* The constant DBL_MAX comes from float.h */
#include <math.h>

void doublingInt() {
	int i;
	for (i = 1; i > 0; i *= 2)
		printf("%12d\n", i);
	printf("Overflowed!\n");
	printf("%12d\n", i);
}

void doublingDouble() {
	double d;
	for (d = 1.0; d < DBL_MAX; d *= 2.0)
		printf("%5.3e\n", d);
}

int isPerfect(int n) {
	int i, sum = 1;
	for (i = 2; i <= sqrt(n); i++) {
		if (n % i == 0) {
			sum += (i + n/i);
		}
	}
	return (sum == n);
}

int main() {
	doublingInt();
	/* doublingDouble(); */

	int n;
	printf("Enter an integer (negative to quit): ");
	fflush(stdout);
	scanf("%d", &n);

	while (n != -1) {
		printf("%d is %sperfect\n\n", n, (isPerfect(n) ? "" : "not
"));
		printf("Enter an integer (negative to quit): ");
		fflush(stdout);
		scanf("%d", &n);
	}
	printf("Goodbye!");

	/* Print out the nth perfect number. We know the first 2 are 6
and 28,
	 * so we start after that.
	 */
	int nPerfect = 2;
	n = 29;
	while (nPerfect < 4) {
```

```c
		nPerfect += isPerfect(n); // adds 1 if n is perfect.
		if (isPerfect(n)) { printf("%d ", n); }
		n++;
	}
	n--; // to compensate for the last one added.
	printf("The fourth perfect number is %d\n", n);

	return 0;
}

/* Selection (and insertion) sorts */
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define AR_SIZE 1000

void swap(int* x, int* y);
void insertionSort(int ar[], int size);
void selectionSort(int ar[], int size);

int main() {
	srand(time(NULL));
	int i;
	int ar[AR_SIZE];
	for (i = 0; i < AR_SIZE; i++) {
		ar[i] = (int)rand();
	}

	for(i = 0; i < 10; i++) {
		printf("%d ", ar[i]);
	}
	printf("\n");
//	insertionSort(ar, AR_SIZE);
	selectionSort(ar, AR_SIZE);

	for(i = 0; i < AR_SIZE; i++) {
		printf("%d ", ar[i]);
	}
	printf("\n");
	return 0;
}


void selectionSort(int ar[], int size) {
	int n = size, i = 0, j = 0;
	for (i = n-1; i>0; i--) {
		int maxPos = 0;
		// find the largest
		for (j=1; j<=i; j++){
			if (ar[j]>ar[maxPos]) {
				maxPos = j;
			}
		}
		// move it to the end
		swap(&ar[i], &ar[maxPos]);
```

```c
	}
}

void insertionSort(int ar[], int size) {
	int i;
	for(i=1; i<size; i++){
		int temp = ar[i];
		int j = i;
		while (j>0 && temp<ar[j-1]) {
			ar[j] = ar[j-1];
			j--;
		}
		ar[j] = temp;
	}
}

void swap(int* x, int* y) {
	int tmp = *x;
	*x = *y;
	*y = tmp;
}

/* Array Lists */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct {
	int* ar;
	int size;
	int capacity;
} arrayList;

arrayList makeArrayList() {
	arrayList list;
	list.capacity = 5;
	list.size = 0;
	list.ar = (int*)malloc(sizeof(int) * list.capacity);
	return list;
}

void add(arrayList* list, int element) {
	if (list->size == list->capacity) {
		// double the size
		list->capacity *= 2;
		// create new space.
		int* newAr = (int*)malloc(sizeof(int) * list->capacity);
/*		Here's an alternative to copying one element at a time:
		memcpy(newAr, list->ar, sizeof(int) * list->capacity); */
		int i;
		for (i = 0; i < list->size; i++) {
			newAr[i] = list->ar[i];
		}
		free(list->ar);
		list->ar = newAr;
	}
	list->ar[list->size] = element;
```

```
	(list->size)++;
}


int getSize(arrayList* list) {
	return list->size;
}

int getCapacity(arrayList* list) {
	return list->capacity;
}

int* getMemoryLocation(arrayList* list) {
	return list->ar;
}

/* Need to remember to free the string that is returned.*/
char* toString(arrayList* list) {
	char buff[10000] = "[";
	char intString[20]; // plenty big enough for a null-terminated
int string
	int i = 0;
	/* Add all the contests onto the output string. */
	for ( ; i < list->size-1; i++) {
		sprintf(intString, "%d,", list->ar[i]);
		strcat(buff, intString);
	}
	sprintf(intString, "%d]", list->ar[i]);
	strcat(buff, intString);
	/* Now that we know the size,we can only use the space that's
actually
	 * filled. But wouldn't it be nice to have a string class, just
like
	 * we've bascially done for arrays? */
	char * retValue = (char*)malloc(strlen(buff)+1);
	strcpy(retValue, buff);
	return retValue;
}

int main() {
	arrayList list = makeArrayList();
	int i, num = 0;

	FILE* in = fopen("ints.txt", "r");
	if (in == NULL) {
		fprintf(stderr, "ERROR opening input file\n");
		exit(1);
	}

	while (fscanf(in, "%d", &num) > 0) {
		/* Pass address so we can modify the list. */
		add(&list, num);
		/* Non toString */
		printf("stored at %p, with size = %d and capacity = %d,
contents = [",
		   getMemoryLocation(&list), getSize(&list),
getCapacity(&list)
```

```
		);
		for (i = 0; i < getSize(&list)-1; i++) {
			printf("%d,", *(list.ar+i));
		}
		printf("%d]\n", *(list.ar+getSize(&list)-1));
		/* Using toString */
		char* s = toString(&list);
		printf("Testing toString: %s\n", s);
		/* Free the dynamically-sized string */
		free(s);
	}
	return 0;
}



/* Basic Linked Lists */

#include <stdlib.h>
#include <stdio.h>
#include "LinkedListBasic.h"

Node* makeNode(int data, Node* next) {
/*	printf("Making a node using %d, %p\n", data, next);
	fflush(stdout);
*/
	Node* node = (Node*)malloc(sizeof(Node));
	node->data = data;
	node->next = next;
	return node;
}

LinkedList* makeList() {
	LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
	list->first = list->last = NULL;
	return list;
}

void addBack(LinkedList* list, int data) {
	if (list->last == NULL) {
		list->first = list->last = makeNode(data, NULL);

	} else {
		list->last->next = makeNode(data, NULL);
		list->last = list->last->next;
	}
}

void display(LinkedList* list) {
	Node * curr = list->first;
	while (curr !=  NULL) {
		printf("[%d] ",curr->data);
		//printf("[At %p, data=%d, next=%p]",curr, curr->data,
curr->next);
		curr = curr->next;
	}
	printf("\n");
```

```c
}

void addFront(LinkedList* list, int data) {
	Node* newNode = makeNode(data, list->first);
	// If the list were empty, we set last as well.
	if (list->first == NULL) {
		list->last = newNode;
	}
	list->first = newNode;
}

int removeFront(LinkedList* list) {
	if (list->first == NULL) {
		return -1;
	}

	Node * toRemove = list->first;
	list->first = toRemove->next;
	/* special case: if removing a node leaves first null, then
	 * the list is empty, so set last to null as well. */
	if (list->first == NULL) {
		list->last = NULL;
	}
	int returnValue = toRemove->data;
	free(toRemove);
	return returnValue;
}

/* Calculates the list's size (number of nodes) */
int getSize(LinkedList* list) {
	int count = 0;
	Node* curr = list->first;
	while (curr != NULL) {
		count++;
		curr = curr->next;
	}
	return count;
}

/* Sets the value at the node in position pos to the given data value.
 * If there is a header node, it is ignored; that is, the first node
with
 * data in it is in position 0.
 */
int setAt(LinkedList* list, int pos, int data) {
	int i = 0;
	Node * curr = list->first;
	// advance to node to change.
	while (curr != NULL && i < pos) {
		i++;
		curr = curr->next;
	}

	if (curr == NULL || pos < 0) return -1;

	curr->data = data;
	return 0;
```

```c
}

/* Removes the node in position pos and returns its data value.
 * If pos is out of bounds, returns -1. */
int removeAt(LinkedList* list, int pos) {
	int i = 0;
	Node * curr = list->first;

	/** Resets first if removing first node **/
	if (curr != NULL && pos == 0) {
		int returnValue = curr->data;
		list->first = curr->next;
		free(curr);
		return returnValue;
	}

	// advance before the node to change.
	while (curr != NULL && i < pos-1) {
		i++;
		curr = curr->next;
	}
	if (curr == NULL || curr->next == NULL) return -1;
	Node* toRemove = curr->next;
	curr->next = toRemove->next;
	int returnValue = toRemove->data;
	/*If we remove the last node, then we need to reset last. */
	if (toRemove->next == NULL) {
		list->last = curr;
	}
	free(toRemove);
	return returnValue;
}

/* Inserts the node in position pos. If pos is out of bounds, returns -
1.
 * Note that in a list of size 2, we should be able to insert a node at
pos=2
 * (which would do the same thing as addBack) */
int insertAt(LinkedList* list, int pos, int data) {
	int i = 0;
	Node * curr = list->first;

	/* New code. Adding to pos 0 is adding to front. */
	if (pos == 0) {
		addFront(list, data);
		return 0;
	}

	// advance before the node to change.
	while (curr != NULL && i < pos-1) {
		i++;
		curr = curr->next;
	}
	if ( curr == NULL || pos < 0) return -1;
	/* If we are going to create one after the last node, then we
need to update
	 * the pointer to last.
```

```
         */
        if (curr->next == NULL) {
              list->last = curr->next = makeNode(data, curr->next);
        } else {
              curr->next = makeNode(data, curr->next);
        }
        return 0;
}


/* Removes the first node with the given value. If there is none,
returns -1
 * but does not give an error. */
int removeValue(LinkedList* list, int data) {
        Node * curr = list->first;
        /** If removing first node, need to reset first. **/
        if (curr != NULL && curr->data == data) {
              int returnValue = curr->data;
              list->first = curr->next;
              free(curr);
              return returnValue;
        }

        // advance before the node to remove.
        while (curr != NULL && curr->next != NULL && curr->next->data !=
data) {
              curr = curr->next;
        }

        // return -1 if we hit the end of the list before finding the
element.
        if (curr == NULL || curr->next == NULL) return -1;
        Node* toRemove = curr->next;
        curr->next = toRemove->next;
        int returnValue = toRemove->data;
        /*If we remove the last node, then we need to reset last. */
        if (toRemove->next == NULL) {
              list->last = curr;
        }
        free(toRemove);
        return returnValue;
}


void displayRecursiveHelper(Node* node) {
        if (node == NULL) {
              printf("\n");
              return;
        } else {
              printf("[%d] ",node->data);
              //printf("[At %p, data=%d, next=%p]",node, node->data,
node->next);
              displayRecursiveHelper(node->next);
        }
}

void displayRecursive(LinkedList* list) {
```

```
	displayRecursiveHelper(list->first);
}

/* Deletes the entire list. Be sure to do it in the right order, so
every node
 * is freed! */
void deleteList(LinkedList* list) {
	Node * curr = list->first;
	while (curr != NULL) {
		Node* toRemove = curr;
		curr = curr->next;
		free(toRemove);
	}
	list->first = NULL;
	list->last = NULL;
	free(list);
	list = NULL;
}


void reverse(LinkedList* list) {
	if (list->first == NULL || list->first->next == NULL) return;
	list->last = list->first;
	Node* curr = list->first->next;
	list->first->next = NULL;
	while (curr->next != NULL) {
		Node* temp = curr->next;
		curr->next = list->first;
		list->first = curr;
		curr = temp;
	}
	curr->next = list->first;
	list->first = curr;
}

/* This would be a more elegant solution if we didn't have to worry
about the
 * pointer to the last element! */
void reverse2(LinkedList* list) {
	if (list->first == NULL || list->first->next == NULL) return;
	Node* reversed = NULL;
	Node* curr = list->first;
	while (curr != NULL) {
		Node* temp = curr->next;
		curr->next = reversed;
		reversed = curr;
		curr = temp;
	}
	list->first = reversed;
}

Node* reverseHelper(Node* ans, Node* remaining) {
	if (remaining == NULL) {
		return ans;
	} else {
		Node* next = remaining->next;
		remaining->next = ans;
```

```
            return reverseHelper(remaining, next);
        }
}

void reverseRecursive(LinkedList* list) {
        if (list->first == NULL || list->first->next == NULL) return;
        list->first = reverseHelper(NULL, list->first);
        // TODO: set list->last correctly.
        // pass 1 node in for ans (that is also list->last), and the rest
        // as remaining
}

/* Enhanced Liinked List */
#include <stdlib.h>
#include <stdio.h>
#include "linkedListEnhanced.h"


Node* makeNode(int data, Node* prev, Node* next) {
        Node* node = malloc(sizeof(Node));
        node->data = data;
        node->prev = prev;
        node->next = next;
        return node;
}

LinkedList* makeList() {
        LinkedList* list = malloc(sizeof(LinkedList));
        list->header = makeNode(-1, NULL, NULL);
        list->trailer = makeNode(-1, NULL, NULL);
        list->header->next = list->trailer;
        list->trailer->prev = list->header;
        list->size = 0;
        return list;
}

void addBack(LinkedList* list, int data) {
        Node* newNode = makeNode(data, list->trailer->prev, list-
>trailer);
        list->trailer->prev->next = newNode;
        list->trailer->prev = newNode;
        (list->size)++;
}

void display(LinkedList* list) {
        Node * curr = list->header->next;
        while (curr !=  list->trailer) {
            printf("[%d]",curr->data);
            curr = curr->next;
        }
        printf("\n");
}

void displayFull(LinkedList* list) {
        Node * curr = list->header;
        while (curr !=  NULL) {
            printf("[At %p, data=%d,prev=%p,next=%p]",
```

```
                        curr, curr->data, curr->prev, curr->next);
            curr = curr->next;
      }
      printf("\n");
}



void addFront(LinkedList* list, int data) {
      Node* newNode = makeNode(data, list->header, list->header->next);
      list->header->next->prev = newNode;
      list->header->next = newNode;
      (list->size)++;
}

int removeFront(LinkedList* list) {
      if (list->size == 0) {
            return -1;
      }
      Node * toRemove = list->header->next;
      toRemove->next->prev = list->header;
      list->header->next = toRemove->next;
      int returnValue = toRemove->data;
      free(toRemove);
      (list->size)--;
      return returnValue;
}

/* Returns the list's size (number of nodes) */
int getSize(LinkedList* list) {
      return list->size;
}

/* Sets the value at the node in position pos to the given data value.
 * The header node is ignored; that is, the first node with
 * data in it is in position 0. If pos is out of bounds
 * (including trying to modify the header or trailer nodes). it returns
-1.
 */
int setAt(LinkedList* list, int pos, int data) {
      if (pos < 0 || pos >= list->size) return -1;

      int i = 0;
      Node * curr = list->header->next;
      // advance to node to change.
      while (i < pos) {
            i++;
            curr = curr->next;
      }
      curr->data = data;
      return 0;
}

/* Removes the node in position pos and returns its data value.
 * If pos is out of bounds, returns -1. */
int removeAt(LinkedList* list, int pos) {
      if (pos < 0 || pos >= list->size) return -1;
```

```
	int i = 0;
	Node * toRemove = list->header->next;
	// advance to node to remove.
	while (i < pos) {
		i++;
		toRemove = toRemove->next;
	}
	toRemove->next->prev = toRemove->prev;
	toRemove->prev->next = toRemove->next;
	int returnValue = toRemove->data;
	free(toRemove);
	(list->size)--;
	return returnValue;
}

/* Inserts the node in position pos. If pos is out of bounds, returns -
1. */
int insertAt(LinkedList* list, int pos, int data) {
	if (pos < 0 || pos > list->size) return -1;
	int i = 0;
	Node * curr = list->header;
	/* advance to node right before the one we are inserting.
	 * We do this by starting at header, not first. */
	while (i < pos) {
		i++;
		curr = curr->next;
	}
	Node* newNode = makeNode(data, curr, curr->next);
	newNode->next->prev = newNode;
	newNode->prev->next = newNode;
	(list->size)++;
	return 0;
}

/* Removes the first node with the given value. If there is none,
returns -1
 * but does not give an error. */
int removeValue(LinkedList* list, int data) {
	Node * toRemove = list->header->next;
	while (toRemove != list->trailer && toRemove->data != data) {
			toRemove = toRemove->next;
	}
	if (toRemove == list->trailer) return -1;

	/* So toRemove must be the node to remove. Proceed as before */
	toRemove->next->prev = toRemove->prev;
	toRemove->prev->next = toRemove->next;
	int returnValue = toRemove->data;
	free(toRemove);
	(list->size)--;
	return returnValue;
}

/* Deletes the entire list. Be sure to do it in the right order, so
every node
 * is freed! */
void deleteList(LinkedList* list) {
```

```c
	Node * curr = list->header;
	while (curr != NULL) {
		Node* toRemove = curr;
		curr = curr->next;
		free(toRemove);
	}
	free(list);
	list = NULL;
}

void displayRecursiveHelper(Node* node) {
	// This condition skips the trailer node.
	if (node->next == NULL) {
		printf("\n");
		return;
	} else {
		printf("[At %p, data=%d,prev=%p,next=%p]",node, node->data,
node->prev,node->next);
		displayRecursiveHelper(node->next);
	}
}

void displayRecursive(LinkedList* list) {
	displayRecursiveHelper(list->header->next);
}

void reverse(LinkedList* list) {
	if (list->size <= 1) return;

	/* We use a 3-node system, reversing the links on the middle
	 * one as we go. The first node to swap is the header. */
	Node* A = list->header->prev;
	Node* B = list->header;
	Node* C = list->header->next;
	/* This includes the trailer as well */
	while ( B != NULL ) {
		/* reverse links */
		B->next = A;
		B->prev = C;
		/* Move to next node */
		A = B;
		B = C;
		if (C != NULL) C = C->next; /* For trailer's next */
	}
	/* Only thing left is to swap the header and trailer nodes.
	 * We use B as a temp node for the swap. */
	B = list->header;
	list->header = list->trailer;
	list->trailer = B;
}
```