

2. (24 points [4/4/5/3/8]) Here's a recursive function that implements the binary search algorithm:

```
int binarySearch(int* list, int left, int right, int input)
{
    cout << "search - left: " << left << " right: " << right: " << endl;
    int middle;
    int middleValue;
    if (left <= right)
    {
        middle = (left + right) / 2;
        middleValue = list[middle];
        cout << "middle: " << middle << " middleValue: " << middleValue << endl;
        if (input == middleValue) return middle;
        else if (input < middleValue) return binarySearch(list, left, middle-1, input);
        else return binarySearch(list, middle+1, right, input);
    }
    else return -1;
}
```

For the two query items listed below, trace the binary search algorithm on the following input array by printing the cout statements as they are reached:

Array: [0, 2, 4, 6, 8, 10, 12, 14]

Query Item #1: 12

Query Item #2: 3

Turn the binary search function into a templated binary search function so that the function can be used on arbitrary input types. You only need to write down any lines you would add to the function or change to make it be templated. Don't worry about copying the rest of the function.

Which operators should the arbitrary types that you define later in other programs overload if you want to be able to use your types with this templated binary search method? (Ignore the << operator and output statements as they are not part of the core binary search algorithm).

4. Here's a slightly modified, iterative implementation of binary search for a *PhoneBook* as implemented above. Assume the normal $<$ relationship for comparing strings (a dictionary ordering (cat $<$ dog)).

```
string binarySearch(string entryNames, string entryNumbers, int size, string name)
{
    cout << "Searching for name: " << name << endl;

    int middle;
    int left = 0;
    int right = size - 1;
    bool found = false;
    while ((left <= right) && (found == false))
    {
        middle =(left + right) / 2;
        cout << "Left: " << left << " Right: " << right << " Middle: " << middle <<
endl;
        if (name == entryNames[middle]) found = true;
        else if (name < entryNames[middle]) right = middle - 1;
        else left = middle + 1;
    }

    string numberToReturn = "000-000-0000";
    if (found == true) numberToReturn = entryNumbers[middle];
    cout << "Returning number: " << numberToReturn << endl;
    return numberToReturn;
}
```

a) Below are the *PhoneBook* arrays from a random cell phone which have been exported in sorted order on the names.

```
string entryNames[] = {"AmitHome", "AprilCell", "BryonyWork", "Dentist",  
"KatieHome", "Mom", "PatrickCell", "StaciaCell", "TanyaHome"}
```

```
string entryNumbers[] = {"803-442-2165", "803-554-2953", "770-434-2155", "803-221-  
1980", "843-298-3562", "803-447-1623", "410-339-1623", "919-600-3512", "803-659-  
1212"};
```

Trace the cout statements for the binary search function when called as:

```
string phoneNumber = binarySearch(entryNames, entryNumbers, 9, "StaciaCell");
```

b) Using binary search, what would be the easiest name to find from the above arrays and how many steps would it take to find?

5. Here's the selection sort algorithm:

```
void selectionSort(int* inputArray, int arraySize)
{
    int k;
    for (k = 0; k < arraySize-1; k++)
    {
        printArray(inputArray, arraySize);
        int index = minimumIndex(inputArray, k, arraySize-1);
        swap(inputArray, k, index);
    }
    printArray(inputArray, arraySize);
}
```

```
int minimumIndex(int* inputArray, int first, int last)
{
    int minIndex = first;
    for (int j = first + 1; j <= last; j++)
    { if (inputArray[j] < inputArray[minIndex]) minIndex = j; }
    return minIndex;
}
```

```
void swap(int* inputArray, int position1, int position2)
{
    int temp = inputArray[position1];
    inputArray[position1] = inputArray[position2];
    inputArray[position2] = temp;
}
```

a). Show the output of the selection sort algorithm printArray statements on the following input array. Assume printArray(inputArray, arraySize) prints the entire contents of the array from front to back (0 to arraySize-1), with brackets on both ends and commas between values (much like the Vector output from Labs 6 and 7).

[5,3,1,6,4,2]

b) Assume you work in a manufacturing plant where you maintain list of a jobs that need to be performed by a robotic welding tool. Each job has an integer job number associated with it. Periodically, the job list is sorted. Between sorts, however, new jobs may be added to the end of the list. These jobs are guaranteed to have job ID numbers greater than any number that was already in the recently sorted list, but are not guaranteed to be in sorted order themselves due to the fact that multiple people can add jobs to the list at the same time in a distributed fashion. The robot tool, however, only likes taking in sorted job numbers, so you have to do sorts every once in a while to make sure the whole list is sorted and the robot tool is happy.

Describe the changes you would make to the selection sort function below to take advantage of the fact that you know part of the list is already sorted. This change has the potential to allow significant speedups to the sorting process. Your description can be in English, or you can provide the code needed to perform the speedup.

Your changes must stay within the general architecture of selection sort. As an example, an answer of “Change from selection sort to insertion sort” would not be acceptable (nor as useful as some other changes).

```
void selectionSort(int* inputArray, int arraySize)
{
    int k;
    for (k = 0; k < arraySize-1; k++)
    {
        int index = minimumIndex(inputArray, k, arraySize-1);
        swap(inputArray, k, index);
    }
}
```