

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

```
Search - left: 0 right: 7
Middle: 3 middleValue: 6
Search - left: 4 right: 7
Middle: 5 middleValue: 10
Search - left: 6 right: 7
Middle: 6 middleValue: 12
```

Query Item #2: 3

```
Search - left: 0 right: 7
Middle: 3 middleValue: 6
Search - left: 0 right: 2
Middle: 1 middleValue: 2
Search - left: 2 right: 2
Middle: 2 middleValue: 4
Search - left: 2 right: 1 // hits base case, stops, returns -1
```

Turn the binary search function into a templated binary search function so that the function can be used on arbitrary input types. Only write down the lines that you believe need to be updated.

```
template <class Type>
int binarySearch(Type* list, int left, int right, Type input)
{
    Type middleValue;
```

Which operators should the arbitrary input types you define 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).

< operator, == (equality) operator, = operator (assignment)

Here's a simple function for computing the power of a number X to another number n (finding  $x^n$ ), defined recursively:

```
int power(int x, int n)
{
    int returnValue;
    cout << "power: x: " << x << " n: " << n << endl;
    if (n == 0) returnValue = 1;
    else returnValue = x * power(x,n-1);

    cout << "Returning: " << returnValue << endl;
    return returnValue;
}
```

Verify that this function is correct by tracing its execution for input values:  
 $x = 2$  and  $n = 2$ .

```
power: x: 2 n: 2
power: x: 2 n: 1
power: x: 2 n: 0
Returning 1
Returning 2
Returning 4
```

Rewrite the power function recursively so that it is faster by taking advantage of the fact that in general:

$x^n = x^{n/2} * x^{n/2}$  (verification:  $2^6 = 64$  and  $2^6 = 2^3 * 2^3 = 8 * 8 = 64$ )

given that  $n/2$  can be performed with double division (ie  $2^5 = 2^{2.5} * 2^{2.5}$ ). We can't use double division however - our function specifies only integers and integer division. Accordingly, be creative in considering how to handle odd values of  $n$ .

```
int power(int x, int n)
{
    if (n == 0) return 1;
    int value = power(x,n/2);
    if ((n % 2) == 0) return value * value;
    else return x * value * value;
}
```

1. Here's a slightly modified, iterative implementation of binary search for a phone book. Assume the normal  $<$  relationship for 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 address book arrays from a random cellphone which have been exported in sorted order on the names. Trace the cout statements for the binary search function when called as:

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

```
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"};
```

**Searching for name: StaciaCell**

**Left: 0 Right: 8 Middle: 4**

**Left: 5 Right: 8 Middle: 6**

**Left: 7 Right: 8 Middle: 7**

**Returning number: 919-600-3512**

- 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?

**KatieHome**

**1 step**

2. 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). Trace the selection sort algorithm printArray statement on the following input array. Assume printArray(inputArray, arraySize) prints the entire contents of the array, 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]
[1,3,5,6,4,2]
[1,2,5,6,4,3]
[1,2,3,6,4,5]
[1,2,3,4,6,5]
[1,2,3,4,5,6]

```

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.

```

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],inputArray[index]);
    }
}
```

```
// go through with a while loop, while value a < value a+1, keep moving up, back up a
//small number of spaces because some new items may have been put on in order (max of
//size of new set you would have to back up)
//that will give you the appropriate k to start with (can imagine as only working on a
//smaller array which happens to be next to a big already sorted array)
```

```
// similarly, could store index of array from last sort, indicating where you need to start
//from.
```