Alexander Molodyh

Class: CS361

Date: April 29, 2017

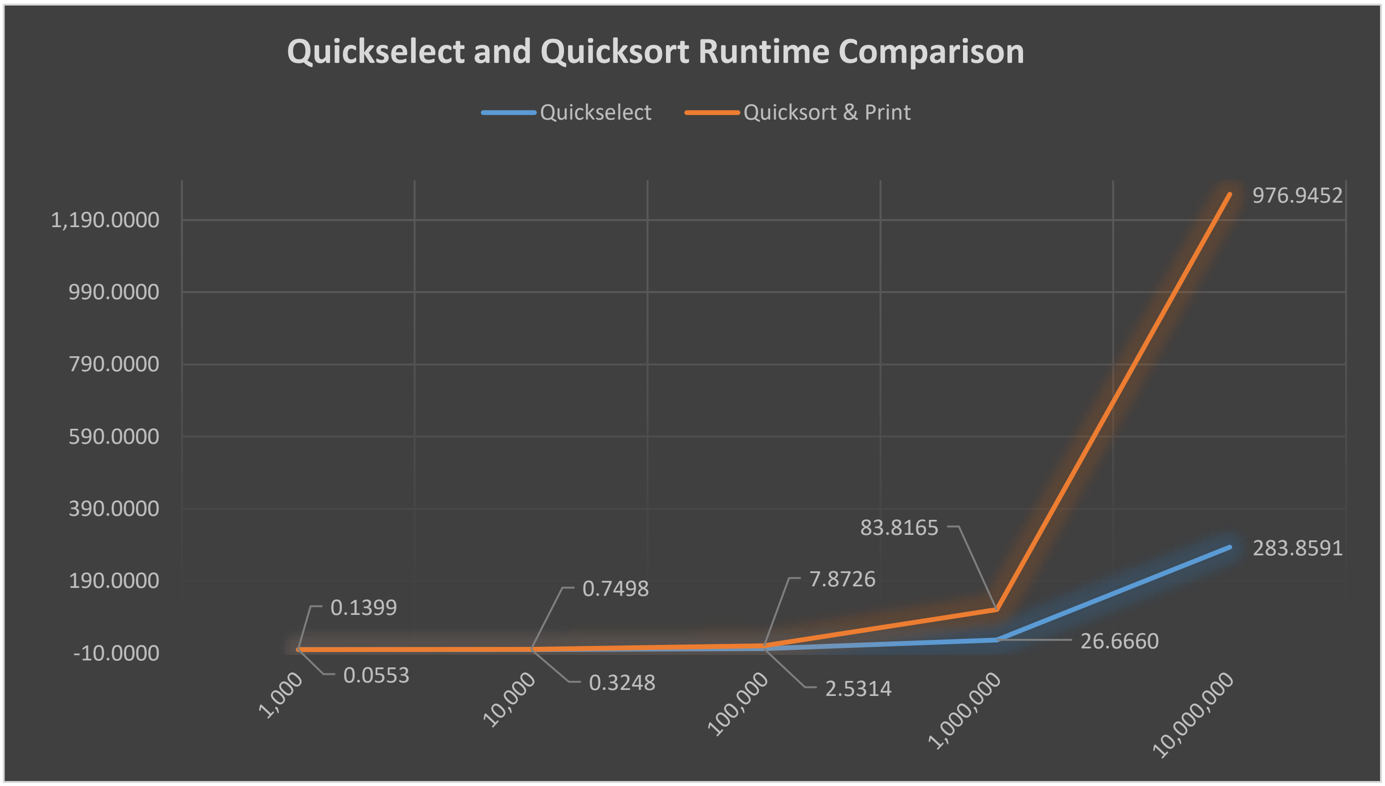
CS361 Lab2

While executing the Quickselect algorithm alongside the quicksort algorithm to find and print the largest 10 elements in a list. I have noticed that the Quickselect algorithm performs much better than actually sorting the list with quicksort. When the list becomes much larger, the Quickselect algorithm isn't affected like sorting the list then printing it out. While the quicksort has to sort the whole list before we can print out the elements, the Quickselect only partially sorts the list before it finds the element that we are looking for. The way that the Quickselect does this is by choosing a random pivot index to compare all other elements with. It then sorts all smaller elements on the left side of the pivot and larger elements on the right of the pivot. It continues this repeating this until it returns the index that we gave it to be compared with. While executing this algorithm, I had noticed that it would partition the list of size 10 about 3-10 times before it finds the element that we are searching for. When executing the algorithm on a list of size 10,000,000 it partitions the list about 16-18 times only until it finds the element we are looking for. The Quickselect is a divide-and-conquer algorithm which works extremely well for this type of search.

The following table contains the results data from running the search, and sort & print data. The Quickselect data is from the recursive method that finds the 10 largest elements and prints them out. The Quicksort & Print data is from sorting the list with a quicksort and then printing the last 10 elements out.

Algorithm	**Array Size**	**Run #1**	**Run #2**	**Run #3**	**Run #4**	**Run #5**	**Run #6**	**Avg Runtime**
Quickselect	1,000	0.0478	0.0539	0.0432	0.0920	0.0529	0.0422	0.0553
Quickselect	10,000	0.2634	0.2632	0.3240	0.3728	0.3457	0.3797	0.3248
Quickselect	100,000	1.7639	2.5582	3.2475	3.0122	2.4310	2.1755	2.5314
Quickselect	1,000,000	29.0749	25.3933	22.5952	31.1769	29.1902	22.5656	26.6660
Quickselect	10,000,000	294.5764	320.7336	335.7071	218.3159	217.4988	316.3226	283.8591
Quicksort & Print	1,000	0.2192	0.1099	0.1206	0.1188	0.0879	0.1829	0.1399
Quicksort & Print	10,000	0.7573	0.7861	0.7884	0.7448	0.7233	0.6988	0.7498
Quicksort & Print	100,000	7.6320	8.0229	7.8517	7.8629	7.9023	7.9636	7.8726
Quicksort & Print	1,000,000	83.9004	83.4561	83.3907	85.2650	83.4553	83.4318	83.8165
Quicksort & Print	10,000,000	1,026.3146	1,015.2236	954.4790	954.6943	954.9141	956.0456	976.9452

The following chart represents the runtime for searching 10 largest items in 1,000, 10,000, 100,000, 1,000,000, and 10,000,000 items. The left column represents the time it took to search the size of the list. The horizontal axis represents the size of the list.



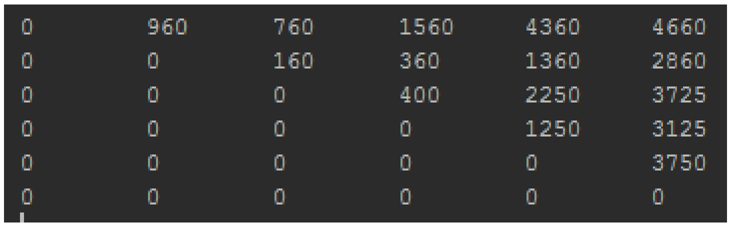
3) Show top right of the m array for your DP version of MCM algorithm for p being <30,4, 8, 5, 10, 25, 15>.

Top Right: 4660

```
public ArrayList<int[][]> matrixChainOrder(int[] p)
{
    //The list that will hold the m and s arrays to be returned
    bothMS = new ArrayList<>();
    int n = p.length;
    //Create 2d arrays with the size of matrix dimensions length
    int[][] m = new int[n][n];
    int[][] s = new int[n][n];
    n--;

    //Set the 0'th chain to 0
    for(int i = 1; i <= n; i++)
        m[i][i] = 0;

    //This loop iterates through every chain
    for(int l = 2; l <= n; l++)
    {
        //Loop through every row in the table
        for(int i = 1; i <= n - l + 1; i++)
        {
            int j = i + l - 1;
            m[i][j] = Integer.MAX_VALUE;
            //Loops through every possible dimension for the current cell
            for(int k = i; k < j; k++)
            {
                //Set q to current calculation
                int q = m[i][k] + m[k + 1][j] + (p[i - 1] * p[k] * p[j]);

                //If the q result is the smallest from all others, store the result
                if(q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }

    bothMS.add(m);
    bothMS.add(s);

    return bothMS;
}
```

4) Show the m array for your memoization version of MCM algorithm for p being <30, 4, 8, 5, 10, 25, 15>.

M Array:

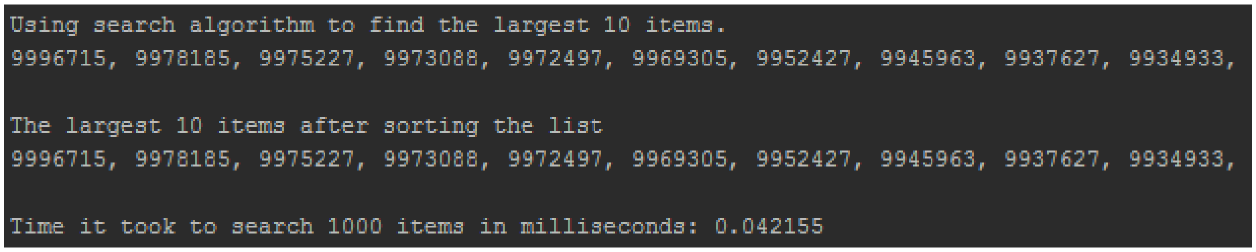


```
0       960     760     1560    4360    4660
0       0       160     360     1360    2860
0       0       0       400     2250    3725
0       0       0       0       1250    3125
0       0       0       0       0       3750
0       0       0       0       0       0
```

The code for the memoization version of MCM is the following:

```java
public int memoizedMC(int[] p)
{
    //Set length of matrix to be the size of the dimensions list
    int n = p.length;
    mArray = new int[n][n];

    //populate upper half of matrix with -1 values so we can track where we are in the matrix.
    for(int i = 1; i < n; i++)
    {
        for(int j = i; j < n; j++)
        {
            mArray[i][j] = -1;
        }
    }
    return lookupChain(p, 1, n - 1);
}

private int lookupChain(int[] p, int i, int j)
{
    //When we hit m[1, 1] or m[2, 2]...etc. Return the element which should be a zero or a
    //calculated value that has been stored previously needed for the first chain calculation
    if(mArray[i][j] > -1)
        return mArray[i][j];

    if(i == j)//When we hit m[1, 1] or m[2, 2]...etc for the first time, set it to zero
        mArray[i][j] = 0;
    else
    {
        //Loop through every possible k dimension for the current chain cell
        for(int k = i; k < j; k++)
        {
            //Calculate the current cell and store it in q
            int q = lookupChain(p, i, k) + lookupChain(p, k + 1, j) + (p[i - 1] * p[k] * p[j]);

            //If q is smaller than the previous stored value in m[i][j] then replace it with q
            if(mArray[i][j] == -1 || q < mArray[i][j])
                mArray[i][j] = q;
        }
    }
    return mArray[i][j];
}
```

5) Largest 10 numbers ran using search algorithm. Search algorithm is Quickselect.



```
Using search algorithm to find the largest 10 items.
9996715, 9978185, 9975227, 9973088, 9972497, 9969305, 9952427, 9945963, 9937627, 9934933,

The largest 10 items after sorting the list
9996715, 9978185, 9975227, 9973088, 9972497, 9969305, 9952427, 9945963, 9937627, 9934933,

Time it took to search 1000 items in milliseconds: 0.042155
```

The following code is for the search algorithm:

```java
    public void printNLargest(int[] integerList, int n)
    {
        searchTime = System.nanoTime();

        for(int j = integerList.length - 1; j > integerList.length - (n + 1); j--)
            System.out.print(integerList[j] + ", ");

        searchTime = System.nanoTime() - searchTime;
    }


/**
 * select is a method part of the Quickselect algorithm. It finds the element at 'k' index and
 * returns it. If you select the last element in the list it will be the larges element in the
set.
 * If you select the first element in the list, it will be the smallest item in the list.
 *
 * @param a     The array to perform the search on.
 * @param left  The starting index of the array.
 * @param right The last index of the list.
 * @param k     The k'th largest or smallest element that you want to be returned.
 * @return An integer representing the element that you searched for.
 */
public int select(int[] a, int left, int right, int k)
{
    //Make a copy of the array so we don't modify the original
    int[] copyArr = Arrays.copyOf(a, a.length);
    Random r = new Random();//Random object to randomize the pivot starting point
    while(right >= left)
    {
        int pivotIndex = partition(copyArr, left, right, r.nextInt(right - left + 1) + left);

        //If the pivotIndex has reached the index of k then the k'th element is in place
        // and we can return it If pivot is equal to k, then we have found out k'th largest number
        if(pivotIndex == k)
            return copyArr[pivotIndex];//Return the largest element at k index
        else if(pivotIndex < k)//If pivot is less than k, then increment pivotIndex
            left = pivotIndex + 1;
        else//Otherwise decrement pivotIndex to move closer to the k'th element
            right = pivotIndex - 1;
    }
    searchTime = System.nanoTime() - searchTime;
    return 0;
}
```

```
     private int partition(int[] a, int left, int right, int pivotIndex)
     {
         //Set the current pivot value to a random element.
         int pivotValue = a[pivotIndex];
         //We swap the current pivotValue element with the end of the list
         swap(a, pivotIndex, right);
         int storeIndex = left;//Start the search from the left index

         //loop from the left index to the right index
         for(int i = left; i < right; i++)
         {
             /*
              * If the value in index i is less than the pivotValue, then we swap the value with
     the
         * storeIndex. Every time that the element in index i is not less than the pivot value,
         * storeIndex does not get incremented.
         */
        if(a[i] < pivotValue)
        {
            //Swap the i'th element with the storeIndex location. This is partially sorting the
           //list.
            swap(a, i, storeIndex);
            storeIndex++;
        }
    }
    swap(a, right, storeIndex);//Swap the storeIndex with the right index

    return storeIndex;
}

private void swap(int[] a, int p, int r)
{
    int temp = a[p];
    a[p] = a[r];
    a[r] = temp;
}
```

The following is the code I used to run the search algorithm.

```
int[] integerList;//This is the list that would have the search be performed on.
int n = 10;

for(int i = integerList.length - 1; i > integerList.length - (n + 1); i--)
{
    System.out.print(select(integerList, 0, integerList.length - 1, i) + ", ");
}
```

The following code was used to print out each matrix.

```
public void printMatrix(int[][] matrix)
{
    System.out.println();
    for(int i = 0; i < matrix.length; i++)
    {
        if(i > 0)//This makes sure we are not printing the 0 index row
        {
            for(int j = 0; j < matrix.length; j++)
            {
                //The following string and char[] will help keep the distance between
                // a cell with a zero in it and a cell with large values in it
                String number = "" + matrix[i][j];
                char[] space = {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '};

                //This loops through the string and replaces the s element from
                // the string into the char[] s element index
                for(int s = 0; s < number.length(); s++)
                    space[s] = number.charAt(s);
                String stringSpace = new String(space);
                if(j > 0)//This makes sure we are not printing the 0 index column
                    System.out.print(stringSpace);
            }
            System.out.println();
        }
    }
}
```