

# Biostatistics 615/815 Lecture 5: Divide and Conquer Algorithms, Basic Data Structures

Hyun Min Kang

September 18th, 2012

## Example submission of Homework 1

Subject: [BIOSTAT615] Homework 1 - John Doe

Dear Dr. Kang,  
 Attached please find the tarball source code (.tar.gz) of the problem 1 and problem 3 for the submission of homework 1. The google document containing the additional copy of source codes, screenshots, and the explanation of problem 2 can be found at  
<https://docs.google.com/a/umich.edu/document/...>

- Send the email both to hmkang@umich.edu and atks@umich.edu,
- Allow access to the google document both addresses
- Make sure (1) to use proper title, (2) to attach .tar.gz file, and (3) to include the link to google document in one submission.
- You will receive an email when the grading is done. If you did not submit your homework in an expected format, you will be notified from the instructor during the grading period.

## Quick Poll

How many students did visit last Friday's office hours?

- 521048 0
- 521049 1
- 521050 2
- 521051 3
- 521321 4
- 521342 5

Submit the code (in blue) to <http://pollev.com>.

## STL strings

What is the expected output from the following code?

```
#include <iostream>
#include <string>
int main (int argc, char** argv) {
  char* p = "Hello";
  char* q = p;
  std::string s = p;
  p[0] = 'h';
  std::cout << q << " " << s << std::endl;
  return 0;
}
```

Submit the code (in blue) to <http://pollev.com>.

- 523649 Hello Hello
- 523650 Hello hello
- 523651 hello Hello
- 523655 hello hello

## Using Classes and Pointers

Which function(s) behave as expected?  
(i.e. creates a new point object and returns its address)

```
Point* createPoint1(double x, double y) {
    Point p(x,y);
    return &p;
}

Point* createPoint2(double x, double y) {
    Point* pp = new Point(x,y);
    return pp;
}
```

Submit the code (in blue) to <http://pollev.com>.

- 523672 createPoint1() only
- 523673 createPoint2() only
- 523674 Both
- 523675 None

## Using STLs

sortedEcho.cpp from last week

```
#include ... // assume all necessary headers are included
int main(int argc, char** argv) {
    std::vector<std::string> vArgs;
    for(int i=1; i < argc; ++i) { vArgs.push_back(argv[i]); }
    std::sort(vArgs.begin(),vArgs.end());
    for(int i=0; i < (int)vArgs.size(); ++i) { std::cout << " " << vArgs[i]; }
    std::cout << std::endl;
    return 0;
}
```

What is the expected output of the following run?

```
% ./sortedEcho hello 1 2 123
```

Submit "523671 expected\_output" to <http://pollev.com>.

## Passing STL objects as reference

```
// print each element of array to the standard output
void printArray(std::vector<int>& A) {
    // call-by-reference to avoid copying large objects
    for(int i=0; i < (int)A.size(); ++i) {
        std::cout << " " << A[i];
    }
    std::cout << std::endl;
}
```

## Divide-and-conquer algorithms

- Solve a problem recursively, applying three steps at each level of recursion
  - Divide** the problem into a number of subproblems that are smaller instances of the same problem
  - Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
  - Combine** the solutions to subproblems into the solution for the original problem

## Binary Search

```
// assuming a is sorted, return index of array containing the key,
// among a[start...end]. Return -1 if no key is found
int binarySearch(std::vector<int>& a, int key, int start, int end) {
    if ( start > end ) return -1; // search failed
    int mid = (start+end)/2;
    if ( key == a[mid] ) return mid; // terminate if match is found
    if ( key < a[mid] ) // divide the remaining problem into half
        return binarySearch(a, key, start, mid-1);
    else
        return binarySearch(a, key, mid+1, end);
}
```

## Running time comparison : sorting algorithms

### Running example with 200,000 elements

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./insertionSort \\  
> /dev/null'
0:17.42 elapsed, 17.428 u, 0.017 s, cpu 100.0% ...
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./stdSort > /dev/null'
0:00.36 elapsed, 0.346 u, 0.042 s, cpu 105.5% ...
```

### Why is the speed so different?

- The time complexity of insertion sort is  $\Theta(n^2)$
- But the time complexity of STL's sorting algorithm is  $\Theta(n \log n)$ .

## Merge Sort

### Divide and conquer algorithm

**Divide** Divide the  $n$  element sequence to be sorted into two subsequences of  $n/2$  elements each

**Conquer** Sort the two subsequences recursively using merge sort

**Combine** Merge the two sorted subsequences to produce the sorted answer

## mergeSort.cpp - main()

```
#include <iostream>
#include <vector>
#include <climits>
void mergeSort(std::vector<int>& a, int p, int r); // defined later
void merge(std::vector<int>& a, int p, int q, int r); // defined later
void printArray(std::vector<int>& A); // same as insertionSort
// same to insertionSort.cpp except for one line
int main(int argc, char** argv) {
    std::vector<int> v;
    int tok;
    while ( std::cin >> tok ) { v.push_back(tok); }
    std::cout << "Before sorting: ";
    printArray(v);
    mergeSort(v, 0, v.size()-1); // differs from insertionSort.cpp
    std::cout << "After sorting: ";
    printArray(v);
    return 0;
}
```

## mergeSort.cpp - mergeSort() function

```
void mergeSort(std::vector<int>& a, int p, int r) {
  if ( p < r ) {          // terminating condition. nothing happens when p >= r
    int q = (p+r)/2;      // find a point to divide the problem
    mergeSort(a, p, q);   // divide-and-conquer
    mergeSort(a, q+1, r); // divide-and-conquer
    merge(a, p, q, r);    // combine the solutions
  }
}
```

## mergeSort.cpp - merge() function

```
// merge piecewise sorted a[p..q] a[q+1..r] into a sorted a[p..r]
void merge(std::vector<int>& a, int p, int q, int r) {
  std::vector<int> aL, aR; // copy a[p..q] to aL and a[q+1..r] to aR
  for(int i=p; i <= q; ++i) aL.push_back(a[i]);
  for(int i=q+1; i <= r; ++i) aR.push_back(a[i]);
  aL.push_back(INT_MAX); // append additional value to avoid out-of-bound
  aR.push_back(INT_MAX);
  // pick smaller one first from aL and aR and copy to a[p..r]
  for(int k=p, i=0, j=0; k <= r; ++k) {
    if ( aL[i] <= aR[j] ) {
      a[k] = aL[i];
      ++i;
    }
    else {
      a[k] = aR[j];
      ++j;
    }
  }
}
```

## Time Complexity of Merge Sort

If  $n = 2^m$

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

$$T(n) = \sum_{i=1}^m cn = cmn = cn \log_2(n) = \Theta(n \log_2 n)$$

For arbitrary  $n$

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn & \text{if } n > 1 \end{cases}$$

$$cn \lceil \log_2 n \rceil \leq T(n) \leq cn \lfloor \log_2 n \rfloor$$

$$T(n) = \Theta(n \log_2 n)$$

## Running time comparison

Running example with 200,000 elements

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./insertionSort \\  
> /dev/null'
```

0:17.42 elapsed, 17.428 u, 0.017 s, cpu 100.0% ...

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./stdSort > /dev/null'
```

0:00.36 elapsed, 0.346 u, 0.042 s, cpu 105.5% ...

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./mergeSort \\  
> /dev/null'
```

0:00.46 elapsed, 0.465 u, 0.019 s, cpu 102.1% ...

## Summary: Merge Sort

- Easy-to-understand divide and conquer algorithms
- $\Theta(n \log n)$  algorithm in worst case
- Need additional memory for array copy
- Slightly slower than other  $\Theta(n \log n)$  algorithms due to overhead of array copy

## Quicksort

### Quicksort Overview

- Worst-case time complexity is  $\Theta(n^2)$
- Expected running time is  $\Theta(n \log_2 n)$ .
- But in practice mostly performs the best

### Divide and conquer algorithm

**Divide** Partition (rearrange) the array  $A[p..r]$  into two subarrays

- Each element of  $A[p..q-1] \leq A[q]$
- Each element of  $A[q+1..r] \geq A[q]$

Compute the index  $q$  as part of this partitioning procedure

**Conquer** Sort the two subarrays by recursively calling quicksort

**Combine** Because the subarrays are already sorted, no work is needed to combine them. The entire array  $A[p..r]$  is now sorted

## Quicksort Algorithm

### Algorithm QUICKSORT

**Data:** array  $A$  and indices  $p$  and  $r$

**Result:**  $A[p..r]$  is sorted

```

if  $p < r$  then
     $q = \text{PARTITION}(A, p, r)$ ;
    QUICKSORT( $A, p, q - 1$ );
    QUICKSORT( $A, q + 1, r$ );
end
    
```

## Quicksort Algorithm

### Algorithm PARTITION

**Data:** array  $A$  and indices  $p$  and  $r$

**Result:** Returns  $q$  such that  $A[p..q-1] \leq A[q] \leq A[q+1..r]$

$x = A[r]$ ;

$i = p - 1$ ;

**for**  $j = p$  **to**  $r - 1$  **do**

**if**  $A[j] \leq x$  **then**

$i = i + 1$ ;

        EXCHANGE( $A[i], A[j]$ );

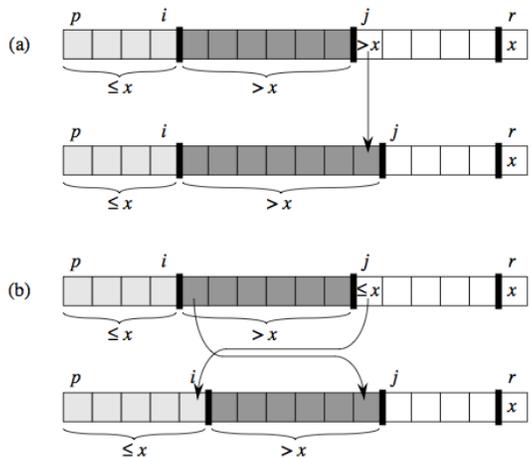
**end**

**end**

EXCHANGE( $A[i + 1], A[r]$ );

**return**  $i + 1$ ;

## How PARTITION Algorithm Works



## Implementation of QUICKSORT Algorithm

```
// quickSort function
// The main function is the same to mergeSort.cpp except for the function name
void quickSort(std::vector<int>& A, int p, int r) {
    if ( p < r ) { // immediately terminate if subarray size is 1
        int piv = A[r]; // take a pivot value
        int i = p-1; // p-i-1 is the # elements < piv among A[p..j]
        int tmp;
        for(int j=p; j < r; ++j) {
            if ( A[j] < piv ) { // if smaller value is found, increase q (=i+1)
                ++i;
                tmp = A[i]; A[i] = A[j]; A[j] = tmp; // swap A[i] and A[j]
            }
        }
        A[r] = A[i+1]; A[i+1] = piv; // swap A[i+1] and A[r]
        quickSort(A, p, i);
        quickSort(A, i+2, r);
    }
}
```

## Running time comparison

### Running example with 200,000 elements (in UNIX or MacOS)

```
user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./insertionSort \\  
> /dev/null'
0:17.42 elapsed, 17.428 u, 0.017 s, cpu 100.0% ...

user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./stdSort > /dev/null'
0:00.36 elapsed, 0.346 u, 0.042 s, cpu 105.5% ...

user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./mergeSort \\  
> /dev/null'
0:00.46 elapsed, 0.465 u, 0.019 s, cpu 102.1% ...

user@host:~$ time sh -c 'seq 1 200000 | ~hmkang/Public/bin/shuf | ./quickSort \\  
> /dev/null'
0:00.35 elapsed, 0.353 u, 0.018 s, cpu 102.8%...
```

## Summary: Quicksort

- $\Theta(n \log n)$  algorithm on average (and most case)
- $\Theta(n^2)$  algorithm in worst case
- Divide conquer algorithms based on partitioning
- Slightly faster than other  $\Theta(n \log n)$  algorithms

## Lower bounds for comparison sorting

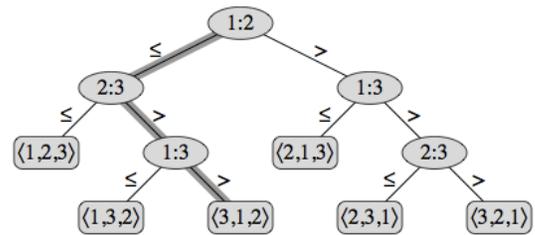
### CLRS Theorem 8.1

Any comparison-based sort algorithm requires  $\Omega(n \log n)$  comparisons in the worst case

### An informal proof

- Any comparison sort algorithm can be represented as a binary decision tree, where each node represents a comparison. Each path from the root to leaf represents possible series of comparisons to sort a sequence.
- Each leaf of the decision tree represents one of  $n!$  possible permutations of input sequences
- We have  $n! \leq l \leq 2^h$ , where  $l$  is the number of leaf nodes, and  $h$  is the height of the tree, equivalent to the # of comparisons.
- Then it implies  $h \geq \log(n!) = \Theta(n \log n)$

## Example decision-tree representing INSERTIONSORT



## Elementary data structure

### Container

A container  $T$  is a generic data structure which supports the following three operation for an object  $x$ .

- SEARCH( $T, x$ )
- INSERT( $T, x$ )
- DELETE( $T, x$ )

### Possible types of container

- Arrays
- Linked lists
- Trees
- Hashes

## Average time complexity of container operations

	SEARCH	INSERT	DELETE
Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
SortedArray	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
List	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Array or list is simple and fast enough for small-sized data
- Tree is easier to scale up to moderate to large-sized data
- Hash is the most robust for very large datasets

## Arrays

### Key features

- Stores the data in a consecutive memory space
- Fastest when the data size is small due to locality of data

### Using std::vector as array

```
std::vector<int> v; // creates an empty vector
// INSERT : append at the end, O(1)
v.push_back(10);
// SEARCH : find a value scanning from begin to end, O(n)
std::vector<int>::iterator i = std::find(v.begin(), v.end(), 10);
if ( i != v.end() ) { std::cout << "Found " << (*i) << std::endl; }
// DELETE : search first, and delete, O(n)
if ( i != v.end() ) { v.erase(i); } // delete an element
```

## Implementing data structure as a header file

### myArray.h

```
class myArray {
    int* data;
    int size;
    void insert(int x) { ... }
    ...
};
```

### myArrayTest.cpp

```
#include <iostream>
#include "myArray.h"
int main(int argc, char** argv) {
    ...
}
```

## Designing a simple array - myArray.h

```
#include <iostream>
#define DEFAULT_ALLOC 1024

template <class T> // template supporting a generic type
class myArray {
protected: // member variables hidden from outside
    T *data; // array of the generic type
    int size; // number of elements in the container
    int nalloc; // # of objects allocated in the memory
public:
    myArray(); // default constructor
    ~myArray(); // destructor
    void insert(const T& x); // insert an element x, const means read-only
    bool search(const T& x); // search for an element x and return its location
    bool remove(const T& x); // delete a particular element
    void print(); // print the content of array to the screen
};
```

## protected and public

```
#include <iostream>

class myClass {
protected:
    int x;
public:
    int getX() { return x; }
    void setX(int _x) { x = _x; }
};

int main(int argc, char** argv) {
    myClass c;
    c.x = 1; // invalid, accessing protected member
    c.setX(1); // valid, accessing public member
    std::cout << c.x << std::endl; // invalid
    std::cout << c.getX() << std::endl; // valid
}
```

There is also a private keyword, but we won't handle it in the class.

## Using friend

```
class mySignature {
protected:
    std::string message;
    friend class myManager;
};

class myManager {
public:
    mySignature s;
    bool verifySignature(std::string& m) {
        return s.message == m; // valid access
    }
};

class myGuest {
public:
    mySignature s;
    bool verifySignature(std::string& m) {
        return s.message == m; // invalid access
    }
};
```

## Using templates for generic class

### Allowing generic type for member variables or functions

```
class Point {
    double x, y; // what if I want to use int instead?
    ...
};
```

### Using template

```
template <class T>
class Point {
    T x, y; // T can be \texttt{int}, \texttt{double}, or any other type
    ...
};

Point<int> intPoint(3,4);
Point<double> doublePoint(3.5,4.5);
```

## Caveat of call-by-reference

```
#include <iostream>

int squareVal(int x) { return x*x; }

int squareRef(int& x) { return x*x; }

int main(int argc, char** argv) {
    int a = 2;
    std::cout << squareVal(a) << std::endl; // valid
    std::cout << squareRef(a) << std::endl; // valid
    std::cout << squareVal(2) << std::endl; // valid
    std::cout << squareRef(2) << std::endl; // invalid
    return 0;
}
```

## Using const T & instead of call-by-value

```
#include <iostream>

int squareVal(int x) { return x*x; }

int squareConstRef(const int& x) { return x*x; }

int main(int argc, char** argv) {
    int a = 2;
    std::cout << squareVal(a) << std::endl; // valid
    std::cout << squareConstRef(a) << std::endl; // valid
    std::cout << squareVal(2) << std::endl; // valid
    std::cout << squareConstRef(2) << std::endl; // valid
    return 0;
}
```

Passing by const reference should be always compatible to passing by value and avoids unnecessary copying of the object. However, its value cannot be updated.

## Revisiting myArray.h

```
#include <iostream>
#define DEFAULT_ALLOC 1024

template <class T> // template supporting a generic type
class myArray {
protected: // member variables hidden from outside
    T *data; // array of the generic type
    int size; // number of elements in the container
    int nalloc; // # of objects allocated in the memory
public:
    myArray(); // default constructor
    ~myArray(); // destructor
    void insert(const T& x); // insert an element x, const means read-only
    bool search(const T& x); // return true if searched an element x
    bool remove(const T& x); // delete a particular element
    void print(); // print the content of array to the screen
};
```

## Using a simple array - myArrayTest.cpp

```
#include <iostream>
#include "myArray.h"

int main(int argc, char** argv) {
    myArray<int> A;
    A.insert(10); // {10}
    A.insert(5); // {10,5}
    A.insert(20); // {10,5,20}
    A.insert(7); // {10,5,20,7}
    A.print();
    std::cout << "A.search(7) = " << A.search(7) << std::endl; // true
    std::cout << "A.remove(10) = " << A.remove(10) << std::endl; // {5,20,7}
    A.print();
    std::cout << "A.search(10) = " << A.search(10) << std::endl; // false
    return 0;
}
```

## Summary: Array

- Simplest container
- Constant time for insertion
- $\Theta(n)$  for search
- $\Theta(n)$  for remove
- Elements are clustered in memory, so faster than list in practice.
- Limited by the allocation size.  $\Theta(n)$  needed for expansion

## Summary

- ### Today
- Merge Sort
  - Quicksort
  - Array

- ### Next Lectures
- Sorted Array
  - Linked list
  - Binary search tree
  - Hash tables
  - Dynamic Programming