

Biostatistics 615/815 Lecture 6: Linear Sorting Algorithms and Elementary Data Structures

Hyun Min Kang

Januray 25th, 2011

Announcements

A good and bad news

Announcements

A good and bad news

- Homework #2 will be announced in the next lecture

Announcements

A good and bad news

- Homework #2 will be announced in the next lecture

815 projects

- 5-6 team pairs in total
- Team assignment will be made during this week
- Each team should set up a meeting with the instructor to kickstart the project

Recap on sorting algorithms : Insertion sort

Algorithm description

- 1 For each $j \in [2 \cdots n]$, iterate element at indices $j - 1, j - 2, \cdots 1$.
- 2 If $A[i] > A[j]$, swap $A[i]$ and $A[j]$
- 3 If $A[i] \leq A[j]$, increase j and go to step 1

Insertion sort is loop invariant

At the start of each iteration, $A[1 \cdots j - 1]$ is loop invariant iff:

- $A[1 \cdots j - 1]$ consist of elements originally in $A[1 \cdots j - 1]$.
- $A[1 \cdots j - 1]$ is in sorted order.

Time complexity of Insertion sort

- Worst and average case time-complexity is $\Theta(n^2)$

Recap on sorting algorithms : Mergesort

Algorithm MERGESORT

Data: array A and indices p and r

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

if $p < r$ **then**

$q = \lfloor (p + r) / 2 \rfloor$;

 MERGESORT(A, p, q);

 MERGESORT($A, q + 1, r$);

 MERGE(A, p, q, r);

end

Time complexity of Mergesort

- Worst and average case time-complexity is $\Theta(n \log n)$

Recap on sorting algorithms : Quicksort

Algorithm QUICKSORT

Data: array A and indices p and r

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

if $p < r$ **then**

$q = \text{PARTITION}(A, p, r);$

$\text{QUICKSORT}(A, p, q - 1);$

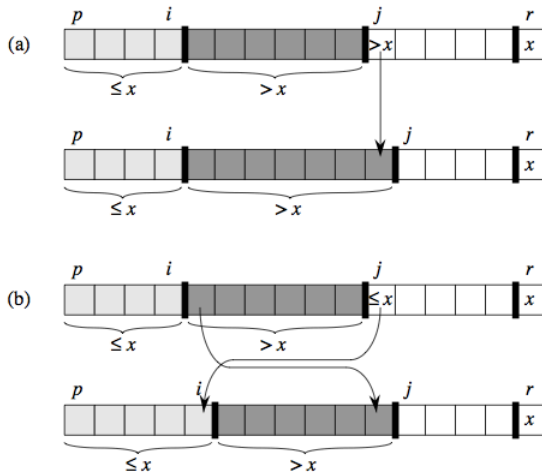
$\text{QUICKSORT}(A, q + 1, r);$

end

Time complexity of Quicksort

- Average case time-complexity is $\Theta(n \log n)$
- Worst case time-complexity is $\Theta(n^2)$, but practically faster than other $\Theta(n \log n)$ algorithms.

How PARTITION algorithm works



Performance of sorting algorithms in practice

Running example with 100,000 elements (in UNIX or MacOS)

```
user@host:~/> time cat src/sample.input.txt | src/stdSort > /dev/null
real 0m0.430s
user 0m0.281s
sys 0m0.130s
```

```
user@host:~/> time cat src/sample.input.txt | src/insertionSort > /dev/null
real 1m8.795s
user 1m8.181s
sys 0m0.206s
```

```
user@host:~/> time cat src/sample.input.txt | src/mergeSort > /dev/null
real 0m0.898s
user 0m0.755s
sys 0m0.131s
```

```
user@host:~/> time cat src/sample.input.txt | src/quickSort > /dev/null
real 0m0.427s
user 0m0.285s
sys 0m0.129s
```

Lower bounds for comparison sorting

CLRS Theorem 8.1

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

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.

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

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.

Lower bounds for comparison sorting

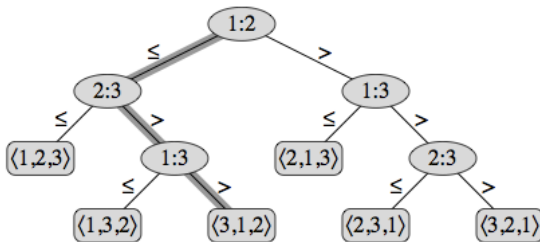
CLRS Theorem 8.1

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



Finding faster sorting methods

Sorting faster than $\Theta(n \log n)$

- Comparison-based sorting algorithms cannot be faster than $\Theta(n \log n)$
- Sorting algorithms NOT based on comparisons may be faster

Finding faster sorting methods

Sorting faster than $\Theta(n \log n)$

- Comparison-based sorting algorithms cannot be faster than $\Theta(n \log n)$
- Sorting algorithms NOT based on comparisons may be faster

Linear time sorting algorithms

- Counting sort
- Radix sort
- Bucket sort

A linear sorting algorithm : Counting sort

A restrictive input setting

- The input sequences have a finite range with many expected duplication.
- For example, each elements of input sequences is one digit number, and your input sequences are millions.

Key idea

- ① Scan through each input sequence and count number of occurrences of each possible input value.
- ② From the smallest to largest possible input value, output each value repeatedly by its stored count.

Another linear sorting algorithm : Radix sort

Key idea

- Sort the input sequence from the last digit to the first repeatedly using a linear sorting algorithm such as COUNTINGSORT
- Applicable to integers within a finite range

329		720		720		329
457		355		329		355
657		436		436		436
839>>>>	457>>>>	839>>>>	457
436		657		355		657
720		329		457		720
355		839		657		839

Implementing radixSort.cpp

```
// use #[radixBits] bits as radix (e.g. hexadecimal if radixBits=4)
void radixSort(std::vector<int>& A, int radixBits, int max) {
    // calculate the number of digits required to represent the maximum number
    int nIter = (int)(ceil(log((double)max)/log(2.)/radixBits));
    int nCounts = (1 << radixBits); // 1<<radixBits == 2^radixBits == # of digits
    int mask = nCounts-1;          // mask for extracting #(radixBits) bits
    std::vector< std::vector<int> > B; // vector of vector, each containing
        // the list of input values containing a particular digit
    B.resize(nCounts);
    for(int i=0; i < nIter; ++i) {
        // initialize each element of B as a empty vector
        for(int j=0; j < nCounts; ++j) { B[j].clear(); }
        // distribute the input sequences into multiple bins, based on i-th digit
        radixSortDivide(A, B, radixBits*i, mask);
        // merge the distributed sequences B into original array A
        radixSortMerge(A, B);
    }
}
```

Implementing radixSort.cpp

```
// divide input sequences based on a particular digit
void radixSortDivide(std::vector<int>& A,
                    std::vector< std::vector<int> >& B, int shift, int mask) {
    for(int i=0; i < (int)A.size(); ++i) {
        // (A[i]>>shift)&mask takes last [shift .. shift+radixBits-1] bits of A[i]
        B[ (A[i] >> shift) & mask ].push_back(A[i]);
    }
}

// merge the partitioned sequences into single array
void radixSortMerge(std::vector<int>& A, std::vector< std::vector<int> >&B ) {
    for(int i=0, k=0; i < (int)B.size(); ++i) {
        for(int j=0; j < (int)B[i].size(); ++j) {
            A[k] = B[i][j]; // iterate each bin of digit and concatenate all values
            ++k;
        }
    }
}
```

Bitwise operation examples

```
shift=3, radixBits=1, A[i] = 117
```

```
57 = 1110101
```

```
-----=-
```

```
57>>3 = 110
```

```
mask = 1
```

```
-----
```

```
ret = 0
```

Bitwise operation examples

shift=3, radixBits=1, A[i] = 117

```
57      = 1110101
-----=--
57>>3  =      110
mask    =       1
-----
ret     =       0
```

shift=3, radixBits=3, A[i] = 117

```
57      = 1111001
-----
57>>3  =      110
mask    =     111
-----
ret     =     110
```

Radix sort in practice

```
user@host:~/> time cat src/sample.input.txt | src/stdSort > /dev/null
real 0m0.430s
user 0m0.281s
sys 0m0.130s
```

```
user@host:~/> time cat src/sample.input.txt | src/insertionSort > /dev/null
real 1m8.795s
user 1m8.181s
sys 0m0.206s
```

```
user@host:~/> time cat src/sample.input.txt | src/quickSort > /dev/null
real 0m0.427s
user 0m0.285s
sys 0m0.129s
```

```
user@host:~/> time cat src/sample.input.txt | src/radixSort 8 > /dev/null
real 0m0.334s
user 0m0.195s
sys 0m0.129s
```


Elementary data structure

Container

A container T is a genetic 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	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
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 on your own

myArray.h

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

myArray.cpp

```
#include "myArray.h"  
void myArray::insert(int x) { // function body goes here  
    ...  
}
```

Main.cpp

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

Building your program

Individually compile and link

```
user@host:~/> g++ -c myArray.cpp
user@host:~/> g++ -c Main.cpp
user@host:~/> g++ -o myArrayTest Main.o myArray.o
```

Or create a Makefile and just type 'make'

```
all: myArrayTest # binary name is myArrayTest

myArrayTest: myArray.o Main.o # link two object files to build binary
    g++ -o myArrayTest myArray.o Main.o # must start with a tab

Main.o: Main.cpp myArray.h # compile to build an object file
    g++ -c Main.cpp

myArray.o: myArray.cpp myArray.h # compile to build an object file
    g++ -c myArray.cpp

clean:
    rm *.o myArrayTest
```

Designing a simple array - myArray.h

```
// myArray.h declares the interface of the class, and the definition is in myArray.
#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 genetic type
    int size;    // number of elements in the container
    int nalloc; // # of objects allocated in the memory
public:       // abstract interface visible to outside
    myArray();    // default constructor
    ~myArray();    // desctructor
    void insert(const T& x); // insert an element x
    int search(const T& x); // search for an element x and return its location
    bool remove(const T& x); // delete a particular element
};
```

Using a simple array Main.cpp

```
#include <iostream>
#include "myArray.h"
int main(int argc, char** argv) {
    myArray<int> A;
    A.insert(10);           // insert example
    if ( A.search(10) > 0 ) { // search example
        std::cout << "Found element 10" << std::endl;
    }
    A.remove(10);         // remove example
    return 0;
}
```

Implementing a simple array myArray.cpp

```
template <class T>
myArray<T>::myArray() { // default constructor
    size = 0;           // array do not have element initially
    nalloc = DEFAULT_ALLOC;
    data = new T[nalloc]; // allocate default # of objects in memory
}

template <class T>
myArray<T>::~~myArray() { // destructor
    if ( data != NULL ) {
        delete [] data; // delete the allocated memory before destorying
    } // the object. otherwise, memory leak happens
}
```


myArray.cpp : insert

```
template <class T>
void myArray<T>::insert(const T& x) {
    if ( size >= nalloc ) { // if container has more elements than allocated
        T* newdata = new T[nalloc*2]; // make an array at doubled size
        for(int i=0; i < nalloc; ++i) {
            newdata[i] = data[i]; // copy the contents of array
        }
        delete [] data; // delete the original array
        data = newdata; // and reassign data ptr
    }
    data[size] = x; // push back to the last element
    ++size; // increase the size
}
```

myArray.cpp : search

```
template <class T>
int myArray<T>::search(const T& x) {
    for(int i=0; i < size; ++i) { // iterate each element
        if ( data[i] == x ) {
            return i;           // and return index of the first match
        }
    }
    return -1;                 // return -1 if no match found
}
```

myArray.cpp : remove

```
template <class T>
bool myArray<T>::remove(const T& x) {
    int i = search(x);          // try to find the element
    if ( i > 0 ) {             // if found
        for(int j=i; j < size-1; ++j) {
            data[i] = data[i+1]; // shift all the elements by one
        }
        --size;                // and reduce the array size
        return true;           // successfully removed the value
    }
    else {
        return false;          // could not find the value to remove
    }
}
```

Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
    myArray<int> A;           // creating an instance of myArray
    A.insert(10);
    A.insert(20);
    myArray<int> B = A;      // copy the instance
    B.remove(10);
    if ( A.search(10) < 0 ) {
        std::cout << "Cannot find 10" << std::endl; // what would happen?
    }
    return 0;               // would to program terminate without errors?
}
```

Implementing complex data types is not so simple

```
int main(int argc, char** argv) {
    myArray<int> A;           // A is empty, A.data points an address x
    A.insert(10);            // A.data[0] = 10, A.size = 1
    A.insert(20);            // A.data[0] = 10, A.data[1] = 20, A.size = 2
    myArray<int> B = A;      // shallow copy, B.size == A.size, B.data == A.data
    B.remove(10);           // A.data[0] = 20, A size = 2 -- NOT GOOD
    if ( A.search(10) < 0 ) {
        std::cout << "Cannot find 10" << std::endl; // A.data is unwillingly modified
    }
    return 0; // ERROR : both delete [] A.data and delete [] B.data is called
}
```

How to fix it

A naive fix : preventing object-to-object copy

```
template <class T>
class myArray {
protected:
    T *data;
    int size;
    int nalloc;
    myArray(myArray& a) {}; // do not allow copying object
public:
    myArray() {...};       // allow to create an object from scratch
```

How to fix it

A naive fix : preventing object-to-object copy

```
template <class T>
class myArray {
protected:
    T *data;
    int size;
    int nalloc;
    myArray(myArray& a) {}; // do not allow copying object
public:
    myArray() {...};      // allow to create an object from scratch
```

A complete fix

- `std::vector` does not suffer from these problems
- Implementing such a nicely-behaving complex object is NOT trivial
- Requires a deep understanding of C++ programming language

A practical advice in implementing a C++ class

- When there are already proven implementations, always utilize them
 - Standard Template Library for basic data structures
 - Boost Library for more sophisticated data types (e.g. Graphs)
 - Eigen package for matrix operations
- ✓ *Always check the license carefully, especially if do not want to release your source code*

A practical advice in implementing a C++ class

- When there are already proven implementations, always utilize them
 - Standard Template Library for basic data structures
 - Boost Library for more sophisticated data types (e.g. Graphs)
 - Eigen package for matrix operations
 - ✓ *Always check the license carefully, especially if do not want to release your source code*
- If it is necessary to implement your own complex data types
 - Use STL (or other well-behaving) data types as member variables whenever possible
 - Keep the behavior simple and well-defined to reduce implementation overhead
 - However, if you spend your time to design your data type robust against many complex situations, your class will be very useful to others.

Next Lecture

Overview of elementary data structures

- Sorted array
- Linked list
- Binary search tree
- Hash table

Reading materials

- CLRS Chapter 10
- CLRS Chapter 11
- CLRS Chapter 12