Biostatistics 615/815 Lecture 5: Divide and Conquer Algorithms

Hyun Min Kang

September 20th, 2011
Recap - An example C++ class

```cpp
#include <iostream>
#include <cmath>

class Point {

public:

double x;

double y;

// A constructor can call constructor of each member variable
Point(double px, double py) : x(px), y(py) {} 
// equivalent to -- Point(double px, double py) : x(px), y(py) {} 

double distanceFromOrigin() { return sqrt( x*x + y*y ); }
};

int main(int argc, char** argv) {
    Point p(3,4); // calls constructor with two arguments
    std::cout << p.distanceFromOrigin() << std::endl; // prints 5
} 
```

Hyun Min Kang
Biostatistics 615/815 - Lecture 5
September 20th, 2011 2 / 30
Recap: STL in practice

```cpp
#include <iostream>
#include <string>
#include <vector>

int main(int argc, char** argv) {
    std::vector<std::string> vArgs;  // vector of strings
    for(int i=1; i < argc; ++i) {
        vArgs.push_back(argv[i]);    // append each arguments to the vector
    }
    std::sort(vArgs.begin(),vArgs.end());  // sort the vector in alphanumerical order
    std::cout << "Sorted arguments : ";   // print the sorted arguments
    for(int i=0; i < vArgs.size(); ++i) { std::cout << " " << vArgs[i]; }
    std::cout << std::endl;
    return 0;
}
```

A running example

```
user@host:~/> ./sortedEcho Hello, World! hello, world! 2 3 5 60 1
Sorted arguments : 1 2 3 5 60 Hello, World! hello, world!
```
Recap: Euclid's algorithm

Algorithm GCD

Data: Two integers \( a \) and \( b \)
Result: The greatest common divisor (GCD) between \( a \) and \( b \)

if \( a \) divides \( b \) then
    return \( a \)
else
    Find the largest integer \( t \) such that \( at + r = b \);
    return GCD\((r, a)\)
end

Function gcd()

```c
int gcd (int a, int b) {
    if (a == 0) return b; // equivalent to returning a when b % a == 0
    else return gcd(b % a, a);
}
```
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
/ 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);
}
Recursive Maximum

// find maximum within an a[start..end]
int findMax(std::vector<int>& a, int start, int end) {
    if ( start == end ) return a[start];  // conquer small problem directly
    else {
        int mid = (start+end)/2;
        int leftMax = findMax(a,start,mid);  // divide the problem into half
        int rightMax = findMax(a,mid+1,end);
        return ( leftMax > rightMax ? leftMax : rightMax );  // combine solutions
    }
}
#include <iostream>
#include <fstream>
#include <vector>

int main(int argc, char** argv) { // sorting software using std::sort
    int tok;
    std::vector<int> v;
    if (argc > 1) { // if argument is given, read from file
        std::ifstream fin(argv[1]);
        while (fin >> tok) { v.push_back(tok); }
        fin.close();
    }
    else { // read from standard input if no argument is specified
        while (std::cin >> tok) { v.push_back(tok); }
    }
    std::sort(v.begin(), v.end()); // Sort using the algorithm in STL
    for(int i=0; i < v.size(); ++i) {
        std::cout << v[i] << std::endl; // print out the content
    }
    return 0;
}
Reading from Files: `insertionSort.cpp`

```cpp
#include <iostream>
#include <fstream>
#include <vector>

void insertionSort(std::vector<int>& v); // insertionSort as defined before

int main(int argc, char** argv) {
    int tok;
    std::vector<int> v;
    if ( argc > 1 ) {
        std::ifstream fin(argv[1]);
        while( fin >> tok ) { v.push_back(tok); }
        fin.close();
    } else {
        while( std::cin >> tok ) { v.push_back(tok); }
    }
    insertionSort(v); // differs from stdSort in only this part
    for(int i=0; i < v.size(); ++i) {
        std::cout << v[i] << std::endl;
    }
    return 0;
}
```

STL Use in **INSERTIONSORT** Algorithm

---

**insertionSort.cpp - insertionSort() function**

```cpp
// perform insertion sort on A
void insertionSort(std::vector<int>& A) { // call-by-reference
    for(int j=1; j < A.size(); ++j) { // 0-based index
        int key = A[j]; // key element to relocate
        int i = j-1; // index to be relocated
        while( (i >= 0) && (A[i] > key) ) { // find position to relocate
            A[i+1] = A[i]; // shift elements
            --i; // update index to be relocated
        }
        A[i+1] = key; // relocate the key element
    }
}
```
Running time comparison

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
```
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()

```cpp
#include <iostream>
#include <vector>
#include <climits>

void mergeSort(std::vector<int>& a, int p, 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 - merge() function

```cpp
// 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;
        }
    }
}
```
void mergeSort(std::vector<int>& a, int p, int r) {
    if ( 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
    }
}
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 \( n \)

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

\[
-cn[\log_2 n] \leq T(n) \leq cn[\log_2 n]
\]

\[
T(n) = \Theta(n \log_2 n)
\]
Running time comparison

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
```
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** Parition (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
Quick sort 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

(a) 

(b)
Implementation of **Quicksort Algorithm**

```cpp
// 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;
            }
        }
        quickSort(A, p, i);
        quickSort(A, i+2, r);
    }
}
```
Running time comparison

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

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

**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 element 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
```
Sorting Algorithms

- Insertion Sort: $\Theta(n^2)$, loop invariant property
- Merge Sort: $\Theta(n \log n)$, straightforward divide and conquer, a little memory overhead
- Quicksort: $\Theta(n^2)$ worst case, $\Theta(n \log n)$ expected, partition algorithm, practically fast
- Count Sort: Linear algorithm, may require much memory
- Radix Sort: $\Theta(nk)$ with $k$ digits
Next Lecture

Sorting Algorithms
- Radix Sort

Overview of elementary data structures
- Array
- Sorted array
- Linked list
- Binary search tree
- Hash table