CSC 222: Computer Programming II

Spring 2005

Sorting and algorithm analysis

- insertion sort
- algorithm analysis: worst vs. average vs. best case
- selection sort
- merge sort
- big-Oh notation

Dictionary revisited

recall the `Dictionary` class

- assumed the input file was in alphabetical order
- so, could use binary search to check for word

for HW3, you are adding an `add` method

- must add each word in correct alphabetical position to ensure binary search still works

```
public class Dictionary
{
    private ArrayList<String> dict;

    public Dictionary(String dictFile)
    {
        dict = new ArrayList<String>();

        try {
            Scanner infile = new Scanner(new File(dictFile));
            while (infile.hasNext()) {
                String word = infile.next();
                dict.add(word);
            }
        }
        catch (FileNotFoundException exception) {
            System.out.println("NO SUCH FILE FOUND");
        }
    }

    public boolean contains(String word)
    {
        return (Collections.binarySearch(dict, word) >= 0);
    }

    public int size()
    {
        return dict.size();
    }

    public void display()
    {
        for (int i = 0; i < dict.size(); i++) {
            System.out.println(dict.get(i));
        }
    }
}
```

Inserting dictionary words

the simple constructor works because the dictionary file is already sorted

if not, could use the add method to add each dictionary word in sequence

```
public Dictionary(String dictFile)
{
    dict = new ArrayList<String>();

    try {
        Scanner infile = new Scanner(new File(dictFile));
        while (infile.hasNext()) {
            String word = infile.next();
            add(word);
        }
    }
    catch (FileNotFoundException exception) {
        System.out.println("NO SUCH FILE FOUND");
    }
}
```

how efficient would this be?

- how fast could it build a dictionary of 100 words? 1,000 words? 117,777 words?

In the worst case…

suppose words are added in reverse order: "zoo", "moo", "foo", "boo"

zoo	

to add "moo", must first shift "zoo" one spot to the right

moo	zoo	

to add "foo", must first shift "moo" and "zoo" each one spot to the right

foo	moo	zoo	

to add "boo", must first shift "foo", "moo" and "zoo" each one spot to the right

boo	foo	moo	zoo	

Worst case (in general)

if inserting N items in reverse order

- 1st item inserted directly
- 2nd item requires 1 shift, 1 insertion
- 3rd item requires 2 shifts, 1 insertion
- . . .
- Nth item requires N-1 shifts, 1 insertion

(1 + 2 + 3 + … + N-1) = N(N-1)/2 shifts, N insertions

the approach taken by `Dictionary` is called "insertion sort"

- insertion sort builds a sorted list by repeatedly inserting items in correct order

since an insertion sort of N items can take roughly N^2 steps,
it is an $O(N^2)$ algorithm

Timing the worst case

`System.currentTimeMillis` method accesses the system clock and returns the time (in milliseconds)

- we can use it to time repeatedly `add` on sequences of varying lengths

```
public class TimeDictionary
{
  public static final int SIZE = 1000;

  public static void main(String[] args)
  {
    Dictionary dict = new Dictionary();

    ArrayList<String> temp = new ArrayList<String>();
    for (int i = 0; i < size; i++) {
      String word = "0000000000" + i;
      temp.add(word.substring(word.length()-10));
    }

    long startTime = System.currentTimeMillis();
    for (int i = SIZE-1; i >= 0; i--) {
      dict.add(temp.get(i));
    }
    long endTime = System.currentTimeMillis();

    System.out.println(endTime-startTime);
  }
}
```

list size (N)	time in msec
1000	80
2000	200
4000	530
8000	1652
. . .	. . .
16000	10004
32000	36893
64000	141764

$O(N^2)$ performance

as the problem size doubles, the time can quadruple

makes sense for an $O(N^2)$ algorithm

- if X items, then X^2 steps required
- if 2X items, then $(2X)^2 = 4X^2$ steps

QUESTION: why is the factor of 4 not realized immediately?

list size (N)	time in msec
1000	80
2000	200
4000	530
8000	1652
. . .	. . .
16000	10004
32000	36893
64000	141764

Big-Oh captures rate-of-growth behavior *in the long run*

- when determining Big-Oh, only the dominant factor is significant (in the long run)

cost = N(N-1)/2 shifts (+ N inserts + additional operations) → $O(N^2)$

N=1,000: 49,500 shifts + 1,000 inserts + … overhead cost is significant

N=100,000: 4,999,950,000 shifts + 100,000 inserts + … only N^2 factor is significant

Best case for insertion sort

while insertion sort can require ~N^2 steps in worst case, it can do much better

- BEST CASE: if items are added in order, then no shifting is required
- only requires N insertion steps, so O(N)
  → if double size, roughly double time

list size (N)	time in msec
16000	10
32000	20
64000	31
128000	50

on average, might expect to shift only half the time

- (1 + 2 + … + N-1)/2 = N(N-1)/4 shifts, so still $O(N^2)$

→ would expect faster timings than worst case, but still quadratic growth

Timing insertion sort (average case)

can use a `Random` object to pick random numbers (in range 1 to `SIZE`) and add to String

list size (N)	time in msec
1000	50
2000	140
4000	441
8000	1042
16000	4387
. . .	. . .

```
public class TimeDictionary
{
  public static final int SIZE = 1000;

  public static void main(String[] args)
  {
    Dictionary dict = new Dictionary();
    Random rand = new Random();

    ArrayList<String> temp = new ArrayList<String>();
    for (int i = 0; i < size; i++) {
      String word = "0000000000" + rand.nextInt(SIZE);
      temp.add(word.substring(word.length()-10));
    }

    long startTime = System.currentTimeMillis();
    for (int i = SIZE-1; i >= 0; i--) {
      dict.add(temp.get(i));
    }
    long endTime = System.currentTimeMillis();

    System.out.println(endTime-startTime);
  }
}
```


A more generic insertion sort

we can code insertion sort independent of the Dictionary class

- assume given an ArrayList of Comparable items
  (could just as easily be an array)

```
public void insertionSort(ArrayList<Comparable> items)
{
  ArrayList<Comparable> sorted = new ArrayList<Comparable>();

  for (int i = 0; i < items.size(); i++) {
    insert(items.get(i), sorted);      // inserts into correct spot (as in HW3)
  }

  for (int i = 0; i < items.size(); i++) {
    items.set(i, sorted.get(i));
  }
}
```

note: this insertion sort uses an additional list for temporary storage

- if tricky, can accomplish it without the extra storage
- HOW?

Other $O(N^2)$ sorts

alternative algorithms exist for sorting a list of items

e.g., selection sort:

- find smallest item, swap into the 1st index
- find next smallest item, swap into the 2nd index
- find next smallest item, swap into the 3rd index
- . . .

```
public void selectionSort(ArrayList<Comparable> items)
{
  for (int i = 0; i < items.size()-1; i++) {        // traverse the list to
    int indexOfMin = i;                             // find the index of the
    for (int j = i+1; j < items.size(); j++) {      // next smallest item
      if (items.get(j).compareTo(items.get(indexOfMin)) < 0) {
        indexOfMin = j;
      }
    }

    Comparable temp = items.get(i);                 // swap the next smallest
    items.set(i, items.get(indexOfMin));            // item into its correct
    items.set(indexOfMin, temp);                    // position
  }
}
```


O(N log N) sorts

there are sorting algorithms that do better than insertion & selection sorts

merge sort & quick sort are commonly used O(N log N) sorts

- recall from sequential vs. binary search examples:
  when N is large, log N is much smaller than N
- thus, when N is large, N log N is much smaller than N^2

N	N log N	N^2
1,000	10,000	1,000,000
2,000	22,000	4,000,000
4,000	48,000	16,000,000
8,000	104,000	64,000,000
16,000	224,000	256,000,000
32,000	480,000	1,024,000,000

Merge sort

merge sort is defined recursively

BASE CASE: to sort a list of 0 or 1 item, DO NOTHING!

RECURSIVE CASE:

1. Divide the list in half
2. Recursively sort each half using merge sort
3. Merge the two sorted halves together

12	9	6	20	3	15

1.

12	9	6

20	3	15

2.

6	9	12

3	15	20

3.

3	6	9	12	15	20

Merging two sorted lists

merging two lists can be done in a single pass

- since sorted, need only compare values at front of each, select smallest
- requires additional list structure to store merged items

```
public void merge(ArrayList<Comparable> items, int low, int high)
{
    ArrayList<Comparable> copy = new ArrayList<Comparable>();

    int size = high-low+1;
    int middle = (low+high+1)/2;
    int front1 = low;
    int front2 = middle;
    for (int i = 0; i < size; i++) {
        if (front2 > high ||
            (front1 < middle && items.get(front1).compareTo(items.get(front2)) < 0)) {
            copy.add(items.get(front1));
            front1++;
        }
        else {
            copy.add(items.get(front2));
            front2++;
        }
    }

    for (int k = 0; k < size; k++) {
        items.set(low+k, copy.get(k));
    }
}
```

Merge sort

once merge has been written, merge sort is simple

- for recursion to work, need to be able to specify range to be sorted
- initially, want to sort the entire range of the list (index 0 to list size – 1)
- recursive call sorts left half (start to middle) & right half (middle to end)
- . . .

```
private void mergeSort(ArrayList<Comparable> items, int low, int high)
{
    if (low < high) {
        int middle = (low + high)/2;
        mergeSort(items, low, middle);
        mergeSort(items, middle+1, high);
        merge(items, low, high);
    }
}


public void mergeSort(ArrayList<Comparable> items)
{
    mergeSort(items, 0, items.size()-1);
}
```

note: private helper method does the recursion; public method calls the helper with appropriate inputs

Big-Oh revisited

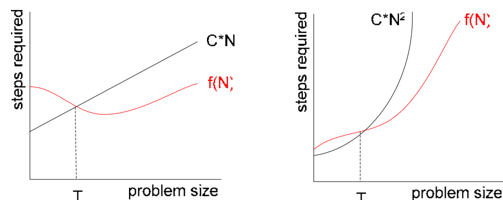
intuitively: an algorithm is O(f(N)) if the # of steps involved in solving a problem of size N has f(N) as the dominant term

O(N):	5N	3N + 2	N/2 – 20
$O(N^2)$:	N^2	$N^2 + 100$	$10N^2 – 5N + 100$

…

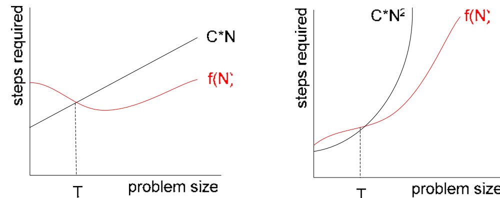
more formally: an algorithm is O(f(N)) if, *after some point*, the # of steps can be bounded from above by a scaled f(N) function

O(N): if number of steps can eventually be bounded by a line
$O(N^2)$: if number of steps can eventually be bounded by a quadratic
…



Technically speaking…

an algorithm is O(f(N)) if there exists a positive constant C & non-negative integer T such that for all $N \geq T$, # of steps required $\leq C*f(N)$



for example, insertion sort:

N(N-1)/2 shifts + N inserts + overhead = $(N^2/2 + N/2 + X)$ steps

if we consider $N \geq X$ (i.e., let T = X), then

$(N^2/2 + N/2 + X) \leq (N^2/2 + N^2/2 + N^2) = 2N^2 = CN^2$ (where C = 2) ➔ $O(N^2)$

Analyzing merge sort

cost of sorting N items = cost of sorting left half (N/2 items) +
cost of sorting right half (N/2 items) +
cost of merging (N items)

more succinctly: $Cost(N) = 2*Cost(N/2) + C_1*N$

$Cost(N) = 2*Cost(N/2) + C_1*N$	can unwind Cost(N/2)
$= 2*(2*Cost(N/4) + C_2*N/2) + C_1*N$	
$= 4*Cost(N/4) + (C_1 + C_2)*N$	can unwind Cost(N/4)
$= 4*(2*Cost(N/8) + C_3*N/4) + (C_1 + C_2)*N$	
$= 8*Cost(N/8) + (C_1 + C_2 + C_3)*N$	can continue unwinding
= …	
$= N*Cost(1) + (C_1 + C_2/2 + C_3/4 + \ldots + C_{logN}/N)*N$	
$= (C_0 + C_1 + C_2 + C_3 + \ldots + C_{logN})*N$	where $C_0 = Cost(1)$
$\leq (max(C_0, C_1,\ldots, C_{logN})*log N) * N$	
$= C * N log N$	where $C = max(C_0, C_1,\ldots, C_{logN})$
➔ O(N log N)	

Interesting variation

in general, insertion/selection sort is faster on very small lists

- the overhead of the recursion is NOT worth it when the list is tiny

can actually improve the performance of merge sort by switching to insertion/selection sort as soon as the list gets small enough

```
private final int CUTOFF = 10;              // WHERE SHOULD THE CUTOFF BE???

private void mergeSort(ArrayList<Comparable> items, int low, int high)
{
    if ((high - low + 1) < CUTOFF) {
        selectionSort(items, low, high);
    }
    else {
        int middle = (low + high)/2;
        mergeSort(items, low, middle);
        mergeSort(items, middle+1, high);
        merge(items, low, high);
    }
}
```


`Dictionary` revisited

recall that `Dictionary` maintains the list in sorted order

- searching is fast (binary search), but adding is slow
- N adds + N searches: $N*O(N) + N*O(\log N) = O(N^2) + O(N \log N) = O(N^2)$

if you are going to do lots of adds in between searches:

- could provide the option of adding without sorting → O(1)
- user could perform many insertions, then sort all at once
- N adds + sort + N searches: $N*O(1) + O(N \log N) + N*\log(N) = O(N \log N)$

we can redesign `Dictionary` to do this in secret

- provide `addFast` member function, which adds at end of the ArrayList
- add private data field `isSorted` that remembers whether the ArrayList is sorted
  - ✓ initially, the empty ArrayList is sorted
  - ✓ each call to `add` keeps the ArrayList sorted
  - ✓ each call to `addFast` is O(1), but means the ArrayList may not be sorted
- `IsStored` first checks the data field: if `!isSorted`, then call `mergeSort` first

Modified `Dictionary` class

from the user's perspective, implementation details are unimportant

what is the performance of `contains`?

```
public class Dictionary
{
  private ArrayList<String> dict;
  private boolean isSorted;

  public Dictionary(String dictFile) {
    dict = new ArrayList<String>();
    isSorted = true;

    // READ WORDS FROM FILE AS BEFORE
    }

    public void add(String word) {
      // ADD WORD IN ALPHABETICAL ORDER, AS BEFORE
    }

    public void addFast(String word) {
      dict.add(word);
      isSorted = false;
    }

    public boolean contains(String word) {
      if (!isSorted) {
        mergeSort(dict);
        isSorted = true;
      }
      return (Collections.binarySearch(dict, word) >= 0);
    }

    public int size() {
      return dict.size();
    }

    public void display() {
      for (int i = 0; i < dict.size(); i++) {
        System.out.println(dict.get(i));
      }
    }
}
```


Sorting summary

sorting/searching lists is common in computer science

a variety of algorithms, with different performance measures, exist

- $O(N^2)$ sorts: insertions sort, selection sort
- O(N log N) sort: merge sort

choosing the "best" algorithm depends upon usage

- if have the list up front, then use merge sort
  - sort the list in O(N log N) steps, then subsequent searches are O(log N)
  - keep in mind, if the list is tiny, then merge sort may not be worth it
- if constructing and searching at the same time, then it depends
  - if many insertions, followed by searches, use merge sort
    - do all insertions O(N), then sort O(N log N), then searches O(log N)
  - if insertions and searches are mixed, then insertion sort
    - each insertion is O(N) as opposed to O(N log N)

In-class exercise

we want to characterize the performance of `Collections.sort`

- is it O(N log N)? $O(N^2)$?
- how can we determine this? will the following program help?

```
import java.util.ArrayList;
import java.util.Collections;
import javax.swing.JOptionPane;

public class TimeSort
{
  public static void main(String[] args)
  {
    int size = Integer.parseInt(JOptionPane.showInputDialog("List size?"));

    ArrayList<Integer> nums = new ArrayList<Integer>();
    for (int i = 0; i < size; i++) {
      nums.add(i);
    }

    Collections.shuffle(nums);

    long startTime = System.currentTimeMillis();
    Collections.sort(nums);
    long endTime = System.currentTimeMillis();

    System.out.println(size + ": " + (endTime-startTime));
  }
}
```