CSC 222: Computer Programming II

Spring 2004

Sorting and recursion

- insertion sort, $O(N^2)$
- selection sort
- recursion
- merge sort, O(N log N)

SortedList revisited

recall the behavior of the SortedList class

- the Add member function adds a new entry into the already sorted list
- to construct a sorted list of N items, add one at a time

how efficient is this approach?

```
template <class ItemType>
class SortedList : public List<ItemType>
{
  public:
    SortedList<ItemType>()
    // constructor (note: List constructor implicitly called first)
    {
      // does nothing
    }

    void Add(const ItemType & item)
     // Results: adds item to the list in order
     {
       items.push_back(item);
       int i;
       for (i = items.size()-1; i > 0 && items[i-1] > item; i--) {
         items[i] = items[i-1];
       }
       items[i] = item;
     }

    bool IsStored(const ItemType & item) const
     // Returns: true if item is stored in list, else false
     {
       int left = 0, right = items.size()-1;
       while (left <= right) {
         int mid = (left+right)/2;
         if (item == items[mid]) {
           return true;
         }
         else if (item < items[mid]) {
           right = mid-1;
         }
         else {
           left = mid + 1;
         }
       }
       return false;
     }
};
```

In the worst case…

suppose numbers are added in reverse order: 4, 3, 2, 1, …

4	

to add 3, must first shift 4 one spot to the right

3	4	

to add 2, must first shift 3 and 4 each one spot to the right

2	3	4	

to add 1, must first shift 2, 3 and 4 each one spot to the right

1	2	3	4	

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

$(1 + 2 + 3 + \ldots + N\text{-}1) = N(N\text{-}1)/2$ shifts, N insertions

the approach taken by `SortedList` 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 insertion sort (worst case)

the `ctime` library contains a `clock()` function that returns the system time (in milliseconds)

list size (N)	time in msec
100	50
200	180
400	451
800	1262
1600	4657
3200	17686
6400	70231

```
// sort1.cpp            Dave Reed

#include <iostream>
#include <ctime>
#include "SortedList.h"
using namespace std;

int main()
{
   int listSize;
   cout << "Enter the list size: ";
   cin >> listSize;

   SortedList<int> slist;
   clock_t start = clock();
   for (int i = listSize; i > 0; i--) {
        slist.Add(i);
   }
   clock_t stop = clock();

   cout << "Insertion sort required "
        << stop-start << " milliseconds"
        << endl;

   return 0;
}
```


O(N^2) performance

note pattern from timings

- as 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 increase smaller when the lists are smaller?

list size (N)	time in msec
100	50
200	180
400	451
800	1262
1600	4657
3200	17686
6400	70231

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=100: 4950 shifts + 100 inserts + … overhead cost is significant

N=6400: 20476800 shifts + 6400 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
800	10
1600	20
3200	30
6400	60
12800	130

on average, might expect to shift only half the time

- $(1 + 2 + \ldots + N-1)/2 = N(N-1)/4$ 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 `Die` object to pick random numbers (in range 1 to `INT_MAX`), and insert

list size (N)	time in msec
100	30
200	100
400	321
800	721
1600	2353
3200	8933
6400	34811

```
// sort2.cpp              Dave Reed

#include <iostream>
#include <climits>
#include <ctime>
#include "Die.h"
#include "SortedList.h"
using namespace std;

int main()
{
    int listSize;
    cout << "Enter the list size: ";
    cin >> listSize;

    Die d(INT_MAX);

    SortedList<int> slist;
    clock_t start = clock();
    for (int i = 0; i < listSize; i++) {
        slist.Add(d.Roll());
    }
    clock_t stop = clock();
    cout << "Insertion sort required "
         << stop-start << " milliseconds"
         << endl;

    return 0;
}
```

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
- . . .

```
template <class Comparable> void SelectionSort(vector<Comparable> & nums)
{
    for (int i = 0; i < nums.size()-1; i++) {
        int indexOfMin = i;                         // traverse the list to
        for (int j = i+1; j < nums.size(); j++) {   // find the index of the
            if (nums[j] < nums[indexOfMin]) {       // next smallest item
                indexOfMin = j;
            }
        }

        Comparable temp = nums[i];                  // swap the next smallest
        nums[i] = nums[indexOfMin];                 // item into its correct
        nums[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

But first… recursion

merge sort & quick sort are naturally defined as recursive algorithms

a *recursive algorithm* is one that refers to itself when solving a problem

- to solve a problem, break into smaller instances of problem, solve & combine

Fibonacci numbers:

1st Fibonacci number = 1
2nd Fibonacci number = 1
Nth Fibonacci number = (N-1)th Fibonacci number + (N-2)th Fibonacci number

Factorial:

iterative (looping) definition: N! = N * (N-1) * …* 3 * 2 * 1
recursive definition: 0! = 1
N! = N * (N-1)!

Recursive functions

```
int Fibonacci(int N)
// Assumes: N >= 0
// Returns: Nth Fibonacci number
{
  if (N <= 1) {
    return 1;
  }
  else {
    return Fibonacci(N-1) +
           Fibonacci(N-2);
  }
}
```

```
int Factorial(int N)
// Assumes: N >= 0
// Returns: N!
{
  if (N == 0) {
    return 1;
  }
  else {
    return N * Factorial(N-1);
  }
}
```

these are classic examples, but pretty STUPID

- recursive approach to Fibonacci has huge redundancy
- can instead compute number using a loop, keep track of previous two numbers

- recursive approach to Factorial is not really any simpler than iterative version
- can computer number using a loop i=1..N, accumulate value by multiplying each i

Euclid's GCD algorithm

a more interesting recursive algorithm is credited to Euclid (3rd cent., BC)

to find the Greatest Common Divisor (GCD) of numbers a and b (a ≥ b)

- if a % b == 0, the GCD = b
- otherwise, GCD of a & b = GCD of b and (a % b)

note: defines GCD of two numbers in terms of GCD of smaller numbers

```
int GCD(int a, int b)
// Assumes: a >= b
// Returns: greatest common div.
{
  if (a % b == 0) {
    return b;
  }
  else {
    return GCD(b, a%b);
  }
}
```

```
int GCD(int a, int b)
// Returns: greatest common div.
{
  if (a < b) {
    return GCD(b, a);
  }
  else if (a % b == 0) {
    return b;
  }
  else {
    return GCD(b, a%b);
  }
}
```


Understanding recursion

every recursive definition has 2 parts:

BASE CASE(S): case(s) so simple that they can be solved directly

RECURSIVE CASE(S): more complex – make use of recursion to solve *smaller* subproblems & combine into a solution to the larger problem

```
int Fibonacci(int N)
// Assumes: N >= 0
// Returns: Nth Fibonacci number
{
  if (N <= 1) {    // BASE CASE
    return 1;
  }
  else {           // RECURSIVE CASE
    return Fibonacci(N-1) +
           Fibonacci(N-2);
  }
}
```

```
int GCD(int a, int b)
// Assumes: a >= b
// Returns: greatest common div.
{
  if (a % b == 0) { // BASE CASE
    return b;
  }
  else {            // RECURSIVE
    return GCD(b, a%b);
  }
}
```

to verify that a recursive definition works:

- convince yourself that the base case(s) are handled correctly
- ASSUME RECURSIVE CALLS WORK ON SMALLER PROBLEMS, then convince yourself that the results from the recursive calls are combined to solve the whole

Avoiding infinite(?) recursion

to avoid infinite recursion:

- must have at least 1 base case (to terminate the recursive sequence)
- each recursive call must get *closer* to a base case

```
int Fibonacci(int N)
// Assumes: N >= 0
// Returns: Nth Fibonacci number
{
  if (N <= 1) {    // BASE CASE
    return 1;
  }
  else {           // RECURSIVE CASE
    return Fibonacci(N-1) +
           Fibonacci(N-2);
  }
}
```

with each recursive call, the number is getting smaller → closer to base case (≤ 1)

```
int GCD(int a, int b)
// Assumes: a >= b
// Returns: greatest common div.
{
  if (a % b == 0) { // BASE CASE
    return b;
  }
  else {            // RECURSIVE
    return GCD(b, a%b);
  }
}
```

with each recursive call, a & b are getting smaller → closer to base case (a % b == 0)

Recursion vs. iteration

it wouldn't be difficult to code these algorithms without recursion

```
int GCD(int a, int b)
{
  while (a % b != 0) {
    int temp = b;
    b = a % b;
    a = temp;
  }
  return b;
}
```

in fact, any recursive function can be rewritten iteratively (i.e., using a loop)

- but sometimes, a recursive definition is MUCH clearer, comparable in efficiency
- recursion is essential to traversing non-linear data structures (CSC427)
- recursion is essential to understanding and implementing fast sorting algorithms

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

```
template <class Comparable>
void Merge(vector<Comparable> & nums, int low, int high)
{
    vector<Comparable> copy;
    int size = high-low+1, middle = (low+high+1)/2;
    int front1 = low, front2 = middle;
    for (int i = 0; i < size; i++) {
        if (front2 > high || (front1 < middle && nums[front1] < nums[front2])) {
            copy.push_back(nums[front1]);
            front1++;
        }
        else {
            copy.push_back(nums[front2]);
            front2++;
        }
    }

    for (int k = 0; k < size; k++) {
        nums[low+k] = copy[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)
- . . .

```
template <class Comparable>
void MergeSort(vector<Comparable> & nums, int low, int high)
{
    if (low < high) {
        int middle = (low + high)/2;
        MergeSort(nums, low, middle);
        MergeSort(nums, middle+1, high);
        Merge(nums, low, high);
    }
}


template <class Comparable> void MergeSort(vector<Comparable> & nums)
{
    MergeSort(nums, 0, nums.size()-1);
}
```


Big-Oh revisited

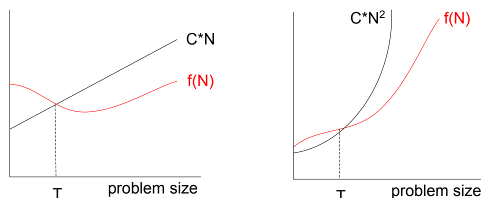
intuitively: an algorithm is O(f(N)) if the # of steps involved in solving a problem of size N has f(N) as the dominant term

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

…

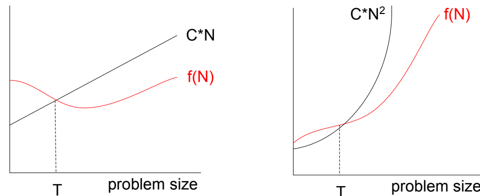
more formally: an algorithm is O(f(N)) if, *after some point*, the # of steps can be bounded from above by a scaled f(N) function

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 C = 2 and T = max(X, 1), then

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

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

$= \ldots$

$= N*Cost(1) + (C_1 + C_2/2 + C_3/4 + \ldots + C_{logN}/N)*N$

$= (C_0 + C_1 + C_2 + C_3 + C_{logN})*N$ — 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)

HW4

the next homework will involve comparing the performance of an $O(N^2)$ sort and an $O(N \log N)$ sort

- your program will generate random lists to sort
- will perform selection and merge sorts on the same lists and keep timings
- questions to ask:
  - ✓ does each sort exhibit the expected rate of growth?
  - ✓ is there a difference between worst case & average case?
  - ✓ does it improve performance if the list is "mostly sorted" to start with?
  - ✓ is it possible that selection sort might ever be faster?

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

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

template <class Comparable>
void MergeSort(vector<Comparable> & nums, int low, int high)
{
    if ((high - low + 1) < CUTOFF) {
        SelectionSort(nums, low, high);    // must modify to sort a range
    }
    else if (low < high) {
        int middle = (low + high)/2;
        MergeSort(nums, low, middle);
        MergeSort(nums, middle+1, high);
        Merge(nums, low, high);
    }
}
```

List & SortedList revisited

recall that SortedList maintains the list in sorted order

- searching is much faster (binary search), but adding is slower
- 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 SortedList to do this in secret

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

SortedList

from the user's perspective, details of the search are unimportant

what is the performance of `IsStored`?

note that we had to remove `const` from `IsStored`
WHY?

```
template <class ItemType> class SortedList : public List<ItemType>
{
  public:
    SortedList<ItemType>()
    // constructor, creates an empty list
    {
      isSorted = true;
    }

    void Add(const ItemType & item)
    // Results: adds item to the end of the list
    {
      // INSERTS INTO CORRECT POSITION, AS BEFORE
    }

    void AddFast(const ItemType & item)
    // Results: adds item to the end of the list
    {
      items.push_back(item);
      isSorted = false;
    }

    bool IsStored(const ItemType & item)
    // Returns: true if item is stored in list, else false
    {
      if (!isSorted) {
        MergeSort(items);
        isSorted = true;
      }

      // BINARY SEARCH, AS BEFORE
    }

  private:
    bool isSorted;
};
```

Timing the smart(?) SortedList

List size: 500
Simple sort/search: 621
Modified sort/search: 40

List size: 1000
Simple sort/search: 1903
Modified sort/search: 90

List size: 2000
Simple sort/search: 7420
Modified sort/search: 190

List size: 4000
Simple sort/search: 27830
Modified sort/search: 391

```
#include <iostream>
#include <climits>            // needed for INT_MAX constant
#include <ctime>              // needed for clock() function
#include "Die.h"
#include "SortedList.h"
using namespace std;

int main()
{
    int listSize;
    cout << "Enter the list size: ";
    cin >> listSize;

    Die d(INT_MAX);
    clock_t start, stop;
    bool found;
    SortedList<int> slist1, slist2;

    start = clock();
    for (int i = listSize; i > 0; i--) {
        slist1.Add(i);
    }
    found = slist1.IsStored(-1);
    stop = clock();
    cout << "Simple sort/search required " << stop-start
         << " milliseconds" << endl;

    start = clock();
    for (int i = listSize; i > 0; i--) {
        slist2.AddFast(i);
    }
    found = slist2.IsStored(-1);
    stop = clock();
    cout << "Modified sort/search required " << stop-start
         << " milliseconds" << endl;

    return 0;
}
```


Timing the smart(?) SortedList, again

this new version works very well if many adds between searches, but not always better

List size: 500
Simple: 621
Modified: 9474

List size: 1000
Simple: 1933
Modified: 41049

WHY?

```
#include <iostream>
#include <climits>            // needed for INT_MAX constant
#include <ctime>              // needed for clock() function
#include "Die.h"
#include "SortedList.h"
using namespace std;

int main()
{
    int listSize;
    cout << "Enter the list size: ";
    cin >> listSize;

    Die d(INT_MAX);
    clock_t start, stop;
    bool found;
    SortedList<int> slist1, slist2;

    start = clock();
    for (int i = listSize; i > 0; i--) {
        slist1.Add(i);
        found = slist1.IsStored(-1);
    }
    stop = clock();
    cout << "Simple sort/search required " << stop-start
         << " milliseconds" << endl;

    start = clock();
    for (int i = listSize; i > 0; i--) {
        slist2.AddFast(i);
        found = slist2.IsStored(-1);
    }
    stop = clock();
    cout << "Modified sort/search required " << stop-start
         << " milliseconds" << endl;

    return 0;
}
```

Sorting summary

sorting/searching lists is common in computer science

a variety of algorithms, with different performance measures, exist

- O(N2) 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)