

CSC 427: Data Structures and Algorithm Analysis

Fall 2004

Searching and algorithm analysis

- sequential search, big-Oh analysis
- binary search, big-Oh analysis
- List class, SortedList class, inheritance

Sorting and algorithm analysis

- insertion sort, big-Oh analysis
- recursion
- merge sort, big-Oh analysis

1

Searching a list

suppose you have a list, and want to find a particular item, e.g.,

- lookup a word in a dictionary
- find a number in the phone book
- locate a student's exam from a pile

searching is a common task in computing

- searching a database
- checking a login password
- lookup the value assigned to a variable in memory

if the items in the list are unordered (e.g., added at random)

- desired item is equally likely to be at any point in the list
- need to systematically search through the list, check each entry until found

→ sequential search

2

Sequential search

sequential search traverses the list from beginning to end

- check each entry in the list
- if matches the desired entry, then FOUND
- if traverse entire list and no match, then NOT FOUND

note: the book first introduces searching/sorting algorithms using arrays
we will consider a more general approach: *templated functions on vectors*

```
template <class Type>
int sequentialSearch(const vector<Type> & items, Type desired)
// precondition: returns index of desired in items, else -1 if not found
{
    for(int k=0; k < items.size(); k++) {
        if (items[k] == desired) {
            return k;
        }
    }
    return -1;
}
```

3

How efficient is sequential search?

for this algorithm, the dominant factor in execution time is checking an item

- the number of checks will determine efficiency

in the worst case:

- the item you are looking for is in the last position of the list (or not found)
- requires traversing and checking every item in the list

- if 100 or 1,000 entries → NO BIG DEAL
- if 10,000 or 100,000 entries → NOTICEABLE

in the average case?

in the best case?

4

Big-Oh notation

to represent an algorithm's performance in relation to the size of the problem, computer scientists use *Big-Oh* notation

an algorithm is $O(N)$ if the number of operations required to solve a problem is proportional to the size of the problem

sequential search on a list of N items requires *roughly* N checks (+ other constants)
→ $O(N)$

for an $O(N)$ algorithm, doubling the size of the problem requires double the amount of work (in the worst case)

- if it takes 1 second to search a list of 1,000 items, then
it takes 2 seconds to search a list of 2,000 items
it takes 4 seconds to search a list of 4,000 items
it takes 8 seconds to search a list of 8,000 items
...

5

Searching an ordered list

when the list is unordered, can't do any better than sequential search

- but, if the list is ordered, a better alternative exists

e.g., when looking up a word in the dictionary or name in the phone book

- can take ordering knowledge into account
- pick a spot – if too far in the list, then go backward; if not far enough, go forward

binary search algorithm

- check midpoint of the list
- if desired item is found there, then DONE
- if the item at midpoint comes after the desired item in the ordering scheme, then repeat the process on the left half
- if the item at midpoint comes before the desired item in the ordering scheme, then repeat the process on the right half

6

Binary search

```
template <class Comparable>
int binarySearch(const vector<Comparable> & items, Comparable desired)
// precondition: returns index of desired in items, else -1 if not found
{
    int left = 0, right = items.size()-1; // maintain bounds on where item must be
    while (left <= right) {
        int mid = (left+right)/2;
        if (desired == items[mid]) { // if at midpoint, then DONE
            return mid;
        }
        else if (desired < items[mid]) { // if less than midpoint, focus on left half
            right = mid-1;
        }
        else { // otherwise, focus on right half
            left = mid + 1;
        }
    }
    return -1; // if reduce to empty range, NOT FOUND
}
```

note: each check reduces the range in which the item can be found by half

- see <http://www.creighton.edu/~davereed/book/Chapter8/search.html> for demo

7

How efficient is binary search?

again, the dominant factor in execution time is checking an item

- the number of checks will determine efficiency

in the worst case:

- the item you are looking for is in the first or last position of the list (or not found)

start with N items in list

after 1st check, reduced to N/2 items to search

after 2nd check, reduced to N/4 items to search

after 3rd check, reduced to N/8 items to search

...

after $\lceil \log_2 N \rceil$ checks, reduced to 1 item to search

in the average case?

in the best case?

8

Big-Oh notation

an algorithm is $O(\log N)$ if the number of operations required to solve a problem is proportional to the logarithm of the size of the problem

binary search on a list of N items requires *roughly* $\log_2 N$ checks (+ other constants)
→ $O(\log N)$

for an $O(\log N)$ algorithm, doubling the size of the problem adds only a constant amount of work

- if it takes 1 second to search a list of 1,000 items, then
 - searching a list of 2,000 items will take time to check midpoint + 1 second
 - searching a list of 4,000 items will take time for 2 checks + 1 second
 - searching a list of 8,000 items will take time for 3 checks + 1 second
 - ...

9

Comparison: searching a phone book

Number of entries in phone book	Number of checks performed by sequential search	Number of checks performed by binary search
100	100	7
200	200	8
400	400	9
800	800	10
1,600	1,600	11
...
10,000	10,000	14
20,000	20,000	15
40,000	40,000	16
...
1,000,000	1,000,000	20

to search a phone book of the United States (~280 million) using binary search?

to search a phone book of the world (6 billion) using binary search?

10

Searching example

for small N , the difference between $O(N)$ and $O(\log N)$ may be negligible

consider the following large-scale application: a spell checker

- want to read in and store a dictionary of words
- then, process a text file one word at a time
 - if word is not found in dictionary, report as misspelled
- since the dictionary file is large (~60,000 words), the difference is clear

we could define a `Dictionary` class to store & access words

- constructor, `Add`, `IsStored`, `NumStored`, `DisplayAll`, ...

better yet, define a generic `List` class to store & access any type of data

- utilize template (as in `vector`) to allow for any type

```
List<string> words;           List<int> grades;
```

11

Templated List

note: templated classes must be defined in one file

the compiler can't compile anything until it knows what type to substitute into the template

```
template <class ItemType>
class List
{
public:
    List<ItemType>()
    // constructor, creates an empty list
    {
        //does nothing
    }

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

    bool isStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
        return (sequentialSearch(items, item) >= 0);
    }

    int numItems() const
    // Returns: number of items in the list
    {
        return items.size();
    }

    void displayAll() const
    // Results: displays all items in the list, one per line
    {
        for (int i = 0; i < items.size(); i++) {
            cout << items[i] << endl;
        }
    }
private:
    vector<ItemType> items;
};
```

12

Spell checker (spell1.cpp)

```
#include <iostream> // for cin, cout
#include <fstream> // for ifstream
#include <string> // for string
#include <cctype> // for ispunct
#include <cassert> // for assert
#include "List.h"
using namespace std;

const string DICTIONARY_FILE = "dict.txt";

void openFile(ifstream & myin);
void readDictionary(List<string> & dict);
string normalize(string word);

int main()
{
    List<string> dictionary;
    readDictionary(dictionary);
    ifstream textFile;
    openFile(textFile);

    cout << endl << "MISPELLED WORDS" << endl
         << "-----" << endl;

    string word;
    while (textFile >> word) {
        word = normalize(word);
        if (word != "" && !dictionary.isStored(word))
        {
            cout << word << endl;
        }
    }
    return 0;
}
```

```
void openFile(ifstream & myin)
// Results: myin is opened to the user's file
{
    string fname;
    cout << "Enter name of text file: ";
    cin >> fname;

    myin.open(fname.c_str());
    assert(myin);
}

void readDictionary(List<string> & dict)
{
    ifstream myDict(DICTIONARY_FILE.c_str());
    assert(myDict);

    cout << "Please wait while file loads... ";

    string word;
    while (myDict >> word) {
        dict.add(word);
    }
    myDict.close();
    cout << "DONE!" << endl << endl;
}

string normalize(string word)
{
    string copy;
    for (int i = 0; i < word.length(); i++) {
        if (!ispunct(word[i])) {
            copy += tolower(word[i]);
        }
    }
    return copy;
}
```

13

In-class exercise

copy the following files

- www.creighton.edu/~davereed/csc427/Code/Search.h
- www.creighton.edu/~davereed/csc427/Code/List.h
- www.creighton.edu/~davereed/csc427/Code/spell1.cpp
- www.creighton.edu/~davereed/csc427/Code/dict.txt
- www.creighton.edu/~davereed/csc427/Code/gettysburg.txt

create a project and spell check the Gettysburg address

- is the delay noticeable?

now download www.creighton.edu/~davereed/csc427/Code/doubledict.txt

and spell check the Gettysburg address with that

- `doubledict.txt` is twice as big (dummy value inserted between each word)
- does it take roughly twice as long?

14

SortedList class

could define a SortedList class similar to List

- add must add items in sorted order
- isStored can use binary search
- data field and other member functions identical to List

C++ provides a better mechanism: inheritance

- if have an existing class, and want to define a new class that extends it, can *derive* a new class from the existing (*parent*) class
- a *derived* class inherits all data fields and member functions from its *parent*
- can *add* new member functions or *reimplement* existing ones in the derived class
- inheritance defines an "IS A" *relationship*: an instance of the desired class IS A instance of the parent class, just more specific

a SortedList is a List

- inherit everything except add & isStored
- reimplement add to add in order, isStored to perform binary search

15

Inheritance

to define a derived class

- add

: public ParentClass

to class header to specify parent class

- define new or overridden data and functions
- parent class constructor is automatically called when constructing an object of derived class

```
template <class ItemType>
class SortedList : public List<ItemType>
{
public:
    SortedList<ItemType>()
        // constructor (note: List constructor implicitly called)
        {
            // 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
        {
            return (binarySearch(items, item) >= 0);
        }
};
```

16

Inheritance vs. reimplementation

using inheritance is preferable to reimplementing the class from scratch

- derived class is simpler/cleaner
- no duplication of code
- if change implementation of code in parent class, derived class automatically updated
- object of derived class is still considered to be an object of parent class
e.g., could pass a `SortedList` to a function expecting a `List`

inheritance approach does require some modifications to List class

- private data is hidden from derived class as well as client program
instead, use 3rd protection level: `protected`
`protected` data is accessible to derived classes, but not client program
- member functions to be overridden should be declared `virtual` in parent class
necessary if want to have a function that works on both `List` and `SortedList`

17

parent List class

`protected` specifies that the vector will be accessible to the `SortedList` class

`virtual` specifies that `Add` and `IsStored` can be overridden and handled correctly

- if pass a `SortedList` to a function expecting a `List`, should still recognize it as a `SortedList`

```
template <class ItemType>
class List
{
public:
    List<ItemType>()
    // constructor, creates an empty list
    {
        //does nothing
    }

    virtual void add(const ItemType & item)
    // Results: adds item to the end of the list
    {
        items.push_back(item);
    }

    virtual bool isStored(const ItemType & item) const
    // Returns: true if item is stored in list, else false
    {
        return (sequentialSearch(items, item) >= 0);
    }

    int numItems() const
    // Returns: number of items in the list
    {
        return items.size();
    }

    void displayAll() const
    // Results: displays all items in the list, one per line
    {
        for (int i = 0; i < items.size(); i++) {
            cout << items[i] << endl;
        }
    }
protected:
    vector<ItemType> items;
};
```

18

Spell checker (spell2.cpp)

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <cassert>
#include "SortedList.h"
using namespace std;

const string DICTIONARY_FILE = "dict.txt";

void openFile(ifstream & myin);
void readDictionary(List<string> & dict);
string normalize(string word);

int main()
{
    SortedList<string> dictionary;
    readDictionary(dictionary);
    ifstream textFile;
    openFile(textFile);

    cout << endl << "MISPELLED WORDS" << endl
         << "-----" << endl;

    string word;
    while (textFile >> word) {
        word = normalize(word);
        if (word != "" && !dictionary.isStored(word)) {
            cout << word << endl;
        }
    }
    return 0;
}
```

```
void openFile(ifstream & myin)
// Results: myin is opened to the user's file
{
    string fname;
    cout << "Enter name of text file: ";
    cin >> fname;

    myin.open(fname.c_str());
    assert(myin);
}

void readDictionary(List<string> & dict)
{
    ifstream myDict(DICTIONARY_FILE.c_str());
    assert(myDict);

    cout << "Please wait while file loads... ";

    string word;
    while (myDict >> word) {
        dict.add(word);
    }
    myDict.close();

    cout << "DONE!" << endl << endl;
}

string normalize(string word)
{
    string copy;
    for (int i = 0; i < word.length(); i++) {
        if (!ispunct(word[i])) {
            copy += tolower(word[i]);
        }
    }
    return copy;
}
```

19

In-class exercise

copy the following files

- www.creighton.edu/~davereed/csc427/Code/Search.h
- www.creighton.edu/~davereed/csc427/Code/List.h
- www.creighton.edu/~davereed/csc427/Code/SortedList.h
- www.creighton.edu/~davereed/csc427/Code/spell2.cpp
- www.creighton.edu/~davereed/csc427/Code/dict.txt
- www.creighton.edu/~davereed/csc427/Code/gettysburg.txt

create a project and spell check the Gettysburg address

- is it noticeably faster than sequential search?

now download www.creighton.edu/~davereed/csc427/Code/doubledict.txt

and spell check the Gettysburg address with that

- `doubledict.txt` is twice as big (dummy value inserted between each word)
- how much longer does it take?

20

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)
    {
        // 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
    {
        return (binarySearch(items, item) >= 0);
    }
};
```

21

In the worst case...

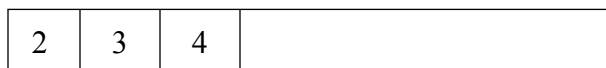
suppose numbers are added in reverse order: 4, 3, 2, 1, ...



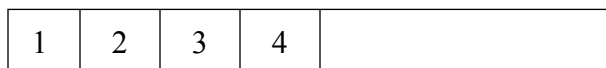
to add 3, must first shift 4 one spot to the right



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



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



22

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
- ...
- N^{th} item requires $N-1$ shifts, 1 insertion

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

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

23

Timing insertion sort (worst case)

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

<u>list size (N)</u>	<u>time in msec</u>
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;
}
```

24

O(N²) performance

note pattern from timings

- as problem size doubles, the time can quadruple

makes sense for an O(N²) algorithm

- if X items, then X² steps required
- if 2X items, then (2X)² = 4X² 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(N²)

N=100: 4950 shifts + 100 inserts + ...

overhead cost is significant

N=6400: 20476800 shifts + 6400 inserts + ...

only N² factor is significant

25

Best case for insertion sort

while insertion sort can require ~ N² 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 + ... + N-1)/2 = N(N-1)/4 shifts, so still O(N²)

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

26

Timing insertion sort (average case)

can use a `Die` object to pick random numbers (in range 1 to `INT_MAX`), and insert

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

```
// sort2.cpp          Dave Reed

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

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

    Die d(1000000);

    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;
}
```

27

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;
        for (int j = i+1; j < nums.size(); j++) {
            if (nums[j] < nums[indexOfMin]) {
                indexOfMin = j;
            }
        }
        Comparable temp = nums[i];
        nums[i] = nums[indexOfMin];
        nums[indexOfMin] = temp;
    }
}
```

28

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 N²

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

29

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

Factorial:

iterative (looping) definition: $N! = N * (N-1) * \dots * 3 * 2 * 1$

recursive definition: $0! = 1$
 $N! = N * (N-1)!$

Greatest Common Divisor of a and b ($a \geq b$):

$$\text{gcd}(a, b) = \begin{cases} b & \text{if } a \% b = 0 \\ \text{gcd}(b, a \% b) & \text{otherwise} \end{cases}$$

30

Recursive functions

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

```
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);
  }
}
```

these are classic examples, but pretty STUPID

- both could be implemented easily using loops

in general, 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

31

Merge sort

merge sort is commonly 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
---	---	---	----	----	----

32

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];
    }
}
```

33

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);
}
```

34

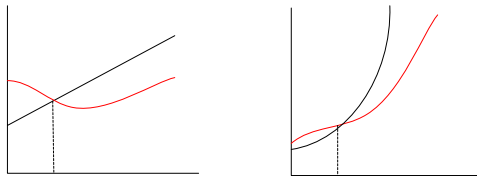
Big-Oh revisited

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

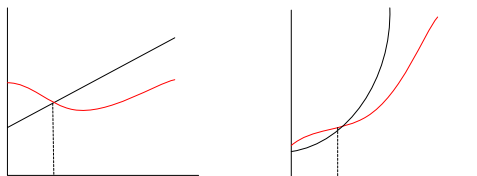
$O(N)$: if number of steps can eventually be bounded by a line
 $O(N^2)$: if number of steps can eventually be bounded by a quadratic
 ...



35

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 $\leq C \cdot f(N)$



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

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

36

$C \cdot N$

$f(N)$

T problem size

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: $\text{Cost}(N) = 2 * \text{Cost}(N/2) + C * N$

$\text{Cost}(N) = 2 * \text{Cost}(N/2) + C_1 * N$	can unwind $\text{Cost}(N/2)$
$= 2 * (2 * \text{Cost}(N/4) + C_2 * N/2) + C_1 * N$	
$= 4 * \text{Cost}(N/4) + (C_1 + C_2) * N$	can unwind $\text{Cost}(N/4)$
$= 4 * (2 * \text{Cost}(N/8) + C_3 * N/4) + (C_1 + C_2) * N$	
$= 8 * \text{Cost}(N/8) + (C_1 + C_2 + C_3) * N$	can continue unwinding
$= \dots$	
$= N * \text{Cost}(1) + (C_1 + C_2 + C_3 + \dots + C_{\log N}) * N$	
$= (\text{Cost}(1) + C_1 + C_2 + C_3 + C_{\log N}) * N$	
$\leq (\max(\text{Cost}(1), C_1, \dots, C_{\log N}) * \log N) * N$	
$= C * N \log N$	where $C = \max(\text{Cost}(1), C_1, \dots, C_{\log N})$
$\rightarrow O(N \log N)$	

37

Interesting combination: LazyList

suppose items are always added in large batches (e.g., from files)

- can get speed of binary search without the cost of insertion sort

a LazyList adds
items at the end \rightarrow
 $O(1)$ cost

a data field keeps
track of whether
currently sorted

before performing
binary search, sort if
not already sorted
 $\rightarrow O(N \log N)$ sort +
 $O(\log N)$ searches

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

    void add(const ItemType & item)
        // Results: adds item to the list in order
    {
        List<ItemType>::add(item);
        isSorted = false;
    }

    bool isStored(const ItemType & item)
        // Returns: true if item is stored in list, else false
    {
        if (!isSorted) {
            mergeSort(items);
            isSorted = true;
        }
        return SortedList<ItemType>::isStored(item);
    }

private:
    bool isSorted;
};
```

38

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 $O(N \log N)$

39

HW 1 solutions

at least it works: www.creighton.edu/~davereed/csc427/Code/skins1.cpp

- uses a `vector< vector<int> >` to store the players' scores
- ugly code – mixes algorithm & data structure details (e.g., index mapping)
- difficult to keep track of data

better: www.creighton.edu/~davereed/csc427/Code/skins2.cpp

- defines a `ScoreCard` class to encapsulate a single player's scores
- uses a `vector<ScoreCard>` to store the players' scores
- provides one level of data abstraction, simplifies program somewhat
- still has implementation details mixed with algorithm (e.g., player vector)

best: www.creighton.edu/~davereed/csc427/Code/skins3.cpp

- defines a `ScoreBoard` class to encapsulate all players' score cards
- uses a `ScoreBoard` to store the players' scores
- provides full data abstraction, main program is trivial

40