1

SEARCHING, SORTING, AND ASYMPTOTIC COMPLEXITY

Lecture 11
CS2110 – Fall 2016

A3 and Prelim

2

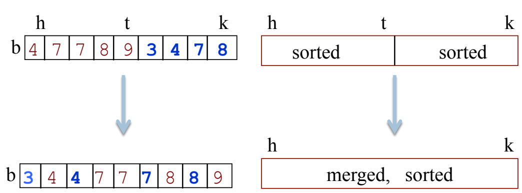
- Deadline for A3: tonight. Only two late days allowed (Wed-Thur)
- Prelim: Thursday evening. 74 conflicts!

  If you filled out P1conflict and don't hear from us, just go to the time you said you could go.
- BRING YOUR ID CARDS TO THE PRELIM. Won't get in without it.
- We are in THREE rooms. Olin 155/255 and Phillips 101. We will tell you on Thursday which one to go to.

Merge two adjacent sorted segments

3

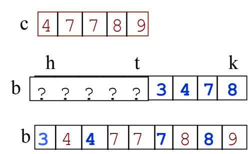
```
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */
public static merge(int[] b, int h, int t, int k) {
}
```



Merge two adjacent sorted segments

4

```
/* Sort b[h..k]. Precondition: b[h..t] and b[t+1..k] are sorted.  */
public static merge(int[] b, int h, int t, int k) {
    Copy b[h..t] into another array c;
    Copy values from c and b[t+1..k] in ascending order into b[h..]
}
```



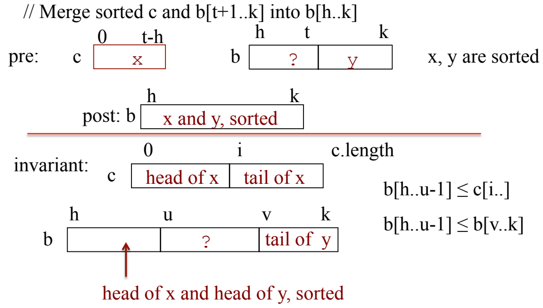
We leave you to write this method. Just move values from c and b[t+1..k] into b in the right order, from smallest to largest.

Runs in time linear in size of b[h..k].

Merge two adjacent sorted segments

5

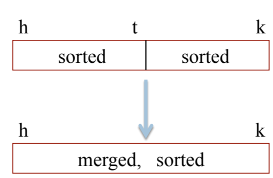
// Merge sorted c and b[t+1..k] into b[h..k]



Mergesort

6

```
/** Sort b[h..k] */
public static void mergesort(int[] b, int h, int k]) {
    if (size b[h..k] < 2)
        return;
    int t= (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}
```



Mergesort

7

```
/** Sort b[h..k] */
public static void mergesort(
        int[] b, int h, int k]) {
   if (size b[h..k] < 2)
       return;
   int t= (h+k)/2;
   mergesort(b, h, t);
   mergesort(b, t+1, k);
   merge(b, h, t, k);
}
```

Let n = size of b[h..k]

Merge: time proportional to n

Depth of recursion: log n

Can therefore show (later) that time taken is proportional to n log n

But space is also proportional to n

QuickSort versus MergeSort

8

```
/** Sort b[h..k] */
public static void QS
      (int[] b, int h, int k) {
   if (k – h < 1) return;
   int j= partition(b, h, k);
   QS(b, h, j-1);
   QS(b, j+1, k);
}
```

```
/** Sort b[h..k] */
public static void MS
      (int[] b, int h, int k) {
   if (k  – h < 1) return;
   MS(b, h, (h+k)/2);
   MS(b, (h+k)/2 + 1, k);
   merge(b, h, (h+k)/2, k);
}
```

One processes the array then recurses.
One recurses then processes the array.

Readings, Homework

9

- Textbook: Chapter 4
- Homework:
  - Recall our discussion of linked lists and A2.
  - What is the worst case time for appending an item to a linked list? For testing to see if the list contains X? What would be the best case time for these operations?
  - If we were going to talk about time (speed) for operating on a list, which makes more sense: worst-case, average-case, or best-case time? Why?

What Makes a Good Algorithm?

10

Suppose you have two possible algorithms that do the same thing; which is *better*?

What do we mean by *better*?
- Faster?
- Less space?
- Easier to code?
- Easier to maintain?
- Required for homework?

How do we measure time and space of an algorithm?

Basic Step: One "constant time" operation

11

Basic step:
- Input/output of scalar value
- Access value of scalar variable, array element, or object field
- assign to variable, array element, or object field
- do one arithmetic or logical operation
- method call (not counting arg evaluation and execution of method body)

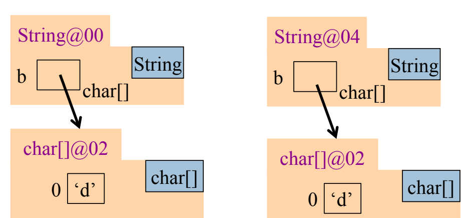
- If-statement: number of basic steps on branch that is executed
- Loop: (number of basic steps in loop body) * (number of iterations) –also bookkeeping
- Method: number of basic steps in method body (include steps needed to prepare stack-frame)

s= s + "c"; NOT a basic step, not constant time, takes time proportional to the length of s

Piazza question had this in it:

12

s.substring(0, 1) == s.substring(k, k+1)
Is it constant-time? If so, is it efficient?



Piazza question had this in it:

13

s.substring(0, 1) == s.substring(k, k+1)

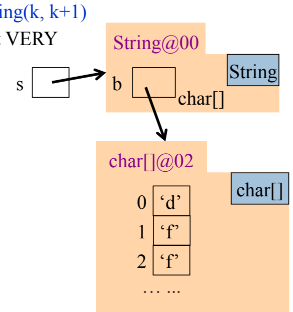
Is it constant-time? Yes. But VERY inefficient.

How about this instead?

s.charAt(0) == s.charAt(k)

Equivalent to

s.b[0] == s.b[k]



Counting basic steps in worst-case execution

14

Linear Search

Let n = b.length

```
/** return true iff v is in b */
static boolean find(int[] b, int v) {
  for (int i = 0; i < b.length; i++) {
    if (b[i] == v) return true;
  }
  return false;
}
```

worst-case execution

basic step	# times executed
i= 0;	1
i < b.length	n+1
i++	n
b[i] == v	n
return true	0
return false	1
Total	3n + 3

We sometimes simplify counting by counting only important things.
Here, it's the number of array element comparisons b[i] == v.
That's the number of loop iterations: n.

Sample Problem: Searching

15

Second solution:
Binary Search

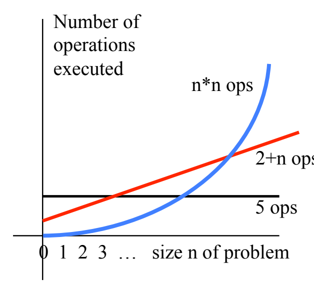
inv:
b[0..h] <= v < b[k..]

```
/** b is sorted. Return h satisfying
    b[0..h] <= v < b[h+1..] */
static int bsearch(int[] b, int v) {
   int h= -1;
   int k= b.length;
   //invariant: …
   while (h+1 != k) {
      int e= (h+ k)/2;
      if (b[e] <= v)  h= e;
      else k= e;
   }
   return h;
}
```

Number of iterations (always the same):
~log b.length
Therefore,
log b.length
arrray comparisons

What do we want from a definition of "runtime complexity"?

16



1. Distinguish among cases for large n, not small n

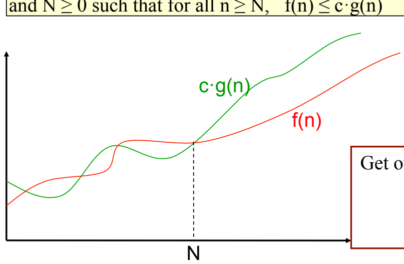
2. Distinguish among important cases, like
- n*n basic operations
- n basic operations
- log n basic operations
- 5 basic operations

3. Don't distinguish among trivially different cases.
- 5 or 50 operations
- n, n+2, or 4n operations

Definition of O(…)

17

Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)

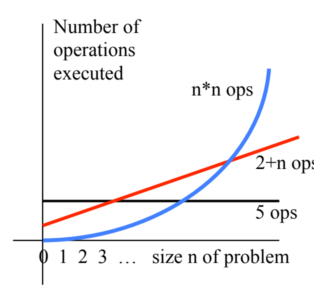


Graphical view

Get out far enough (for n ≥ N) f(n) is at most c·g(n)

What do we want from a definition of "runtime complexity"?

18



Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)

Roughly, f(n) is O(g(n)) means that f(n) grows like g(n) or slower, to within a constant factor

Prove that $(n^2 + n)$ is $O(n^2)$

19

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(n^2 + n)$ is $O(n^2)$

Methodology:

Start with f(n) and slowly transform into $c \cdot g(n)$:
- Use = and <= and < steps
- At appropriate point, can choose N to help calculation
- At appropriate point, can choose c to help calculation

Prove that $(n^2 + n)$ is $O(n^2)$

20

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Example: Prove that $(n^2 + n)$ is $O(n^2)$

f(n)
= <definition of f(n)>
$n^2 + n$
<= <for $n \geq 1$, $n \leq n^2$>
$n^2 + n^2$
= <arith>
$2*n^2$
= <choose $g(n) = n^2$>
2*g(n)

Choose
N = 1 and c = 2

Prove that 100 n + log n is O(n)

21

Formal definition: f(n) is O(g(n)) if there exist constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n)
= <put in what f(n) is>
100 n + log n
<= <We know $\log n \leq n$ for $n \geq 1$>
100 n + n
= <arith>
101 n
= <g(n) = n>
101 g(n)

Choose
N = 1 and c = 101

Do NOT say or write f(n) = O(g(n))

22

Formal definition: f(n) is O(g(n)) if there exist constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n) = g((n)) is simply WRONG. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don't read such things.

Here's an example to show what happens when we use = this way.

We know that n+2 is O(n) and n+3 is O(n). Suppose we use =

n+2 = O(n)
n+3 = O(n)

But then, by transitivity of equality, we have n+2 = n+3.
We have proved something that is false. Not good.

O(…) Examples

23

Let $f(n) = 3n^2 + 6n - 7$
- f(n) is $O(n^2)$
- f(n) is $O(n^3)$
- f(n) is $O(n^4)$
- …

$p(n) = 4 n \log n + 34 n - 89$
- p(n) is $O(n \log n)$
- p(n) is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$
h(n) is $O(2^n)$

a(n) = 34
- a(n) is O(1)

Only the *leading* term (the term that grows most rapidly) matters

If it's $O(n^2)$, it's also $O(n^3)$ etc! However, we always use the smallest one

Commonly Seen Time Bounds

24

O(1)	constant	excellent
O(log n)	logarithmic	excellent
O(n)	linear	good
O(n log n)	n log n	pretty good
$O(n^2)$	quadratic	OK
$O(n^3)$	cubic	maybe OK
$O(2^n)$	exponential	too slow

Problem-size examples

25

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

operations	1 second	1 minute	1 hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

Why bother with runtime analysis?

26

Computers so fast that we can do whatever we want using simple algorithms and data structures, right?

Not really – data-structure/algorithm improvements can be a *very big* win

Scenario:

- A runs in n^2 msec
- A' runs in $n^2/10$ msec
- B runs in 10 n log n msec

Problem of size $n=10^3$

- A: 10^3 sec ≈ 17 minutes
- A': 10^2 sec ≈ 1.7 minutes
- B: 10^2 sec ≈ 1.7 minutes

Problem of size $n=10^6$

- A: 10^9 sec ≈ 30 years
- A': 10^8 sec ≈ 3 years
- B: $2 \cdot 10^5$ sec ≈ 2 days

1 day = 86,400 sec ≈ 10^5 sec

1,000 days ≈ 3 years

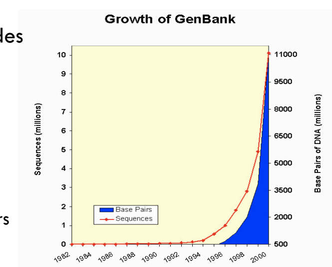
Algorithms for the Human Genome

27

Human genome
= 3.5 billion nucleotides
~ 1 Gb

@1 base-pair instruction/μsec

- $n^2 \rightarrow$ 388445 years
- n log n → 30.824 hours
- n → 1 hour



Worst-Case/Expected-Case Bounds

28

May be difficult to determine time bounds for all imaginable inputs of size n

Simplifying assumption #4:
Determine number of steps for either

- worst-case or
- expected-case or average case

- Worst-case
- Determine how much time is needed for the *worst possible* input of size n

- Expected-case
- Determine how much time is needed *on average* for all inputs of size n

Simplifying Assumptions

29

Use the size of the input rather than the input itself – n

Count the number of "basic steps" rather than computing exact time

Ignore multiplicative constants and small inputs (order-of, big-O)

Determine number of steps for either

- worst-case
- expected-case

These assumptions allow us to analyze algorithms effectively

Worst-Case Analysis of Searching

30

```
Linear Search
// return true iff v is in b
static bool find (int[] b, int v) {
  for (int x : b) {
    if (x == v) return true;
  }
  return false;
}
```

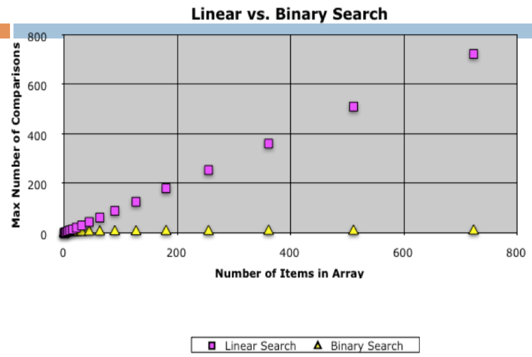
worst-case time: O(#b)
Expected time O(#b)

#b = size of b

```
Binary Search
// Return h that satisfies
//     b[0..h] <= v < b[h+1..]
static bool bsearch(int[] b, int v {
  int h= -1;  int t= b.length;
  while ( h != t-1 ) {
    int  e= (h+t)/2;
    if (b[e] <= v)  h= e;
    else t= e;
  }
}
```

Always ~(log #b+1) iterations.
Worst-case and expected times: O(log #b)

31



Analysis of Matrix Multiplication

32

Multiply n-by-n matrices A and B:

Convention, matrix problems measured in terms of n, the number of rows, columns

- Input size is really $2n^2$, not n
- Worst-case time: $O(n^3)$
- Expected-case time: $O(n^3)$

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    c[i][j] = 0;
    for (k = 0; k < n; k++)
      c[i][j] += a[i][k]*b[k][j];
  }
```

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    throw new Exception();
}}
```

Remarks

33

Once you get the hang of this, you can quickly zero in on what is relevant for determining asymptotic complexity

- Example: you can usually ignore everything that is not in the innermost loop. Why?

One difficulty:

- Determining runtime for recursive programs
  Depends on the depth of recursion

Limitations of Runtime Analysis

34

Big-O can hide a very large constant

- Example: selection
- Example: small problems

The specific problem you want to solve may not be the worst case

- Example: Simplex method for linear programming

Your program may not run often enough to make analysis worthwhile

- Example: one-shot vs. every day
- You may be analyzing and improving the wrong part of the program

- Very common situation
- Should use profiling tools

What you need to know / be able to do

35

- Know the definition of f(n) is O(g(n))
- Be able to prove that some function f(n) is O(g(n)). The simplest way is as done on two slides above.
- Know worst-case and average (expected) case O(…) of basic searching/sorting algorithms: linear/binary search, partition alg of quicksort, insertion sort, selection sort, quicksort, merge sort.
- Be able to look at an algorithm and figure out its worst case O(…) based on counting basic steps or things like array-element swaps

Lower Bound for Comparison Sorting

36

Goal: Determine minimum time *required* to sort *n* items

Note: we want *worst-case*, not *best-case* time

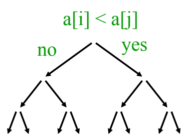
- Best-case doesn't tell us much. E.g. Insertion Sort takes O(n) time on already-sorted input
- Want to know *worst-case time* for *best possible* algorithm

- How can we prove anything about the *best possible* algorithm?
- Want to find characteristics that are common to *all* sorting algorithms
- Limit attention to *comparison-based algorithms* and try to count number of comparisons

Comparison Trees

37

- Comparison-based algorithms make decisions based on comparison of data elements
- Gives a *comparison tree*
- If algorithm fails to terminate for some input, comparison tree is infinite
- Height of comparison tree represents *worst-case number of comparisons* for that algorithm
- Can show: *Any* correct comparison-based algorithm must make at least n log n comparisons in the worst case



Lower Bound for Comparison Sorting

38

- Say we have a correct comparison-based algorithm
- Suppose we want to sort the elements in an array b[]
- Assume the elements of b[] are distinct
- Any permutation of the elements is initially possible
- When done, b[] is sorted
- But the algorithm could not have taken the same path in the comparison tree on different input permutations

Lower Bound for Comparison Sorting

39

How many input permutations are possible? $n! \sim 2^{n \log n}$

For a comparison-based sorting algorithm to be correct, it must have at least that many leaves in its comparison tree

To have at least $n! \sim 2^{n \log n}$ leaves, it must have height at least n log n (since it is only binary branching, the number of nodes at most doubles at every depth)

Therefore its longest path must be of length at least n log n, and that is its worst-case running time

Mergesort

40

```
/** Sort b[h..k] */
public static mergesort(
  int[] b, int h, int k]) {
    if (size b[h..k] < 2)
        return;
    int t= (h+k)/2;
    mergesort(b, h, t);
    mergesort(b, t+1, k);
    merge(b, h, t, k);
}
```

Runtime recurrence

T(n): time to sort array of size n
  T(1) = 1
  T(n) = 2T(n/2) + O(n)

Can show by induction that
  T(n) is O(n log n)

Alternatively, can see that T(n) is O(n log n) by looking at tree of recursive calls