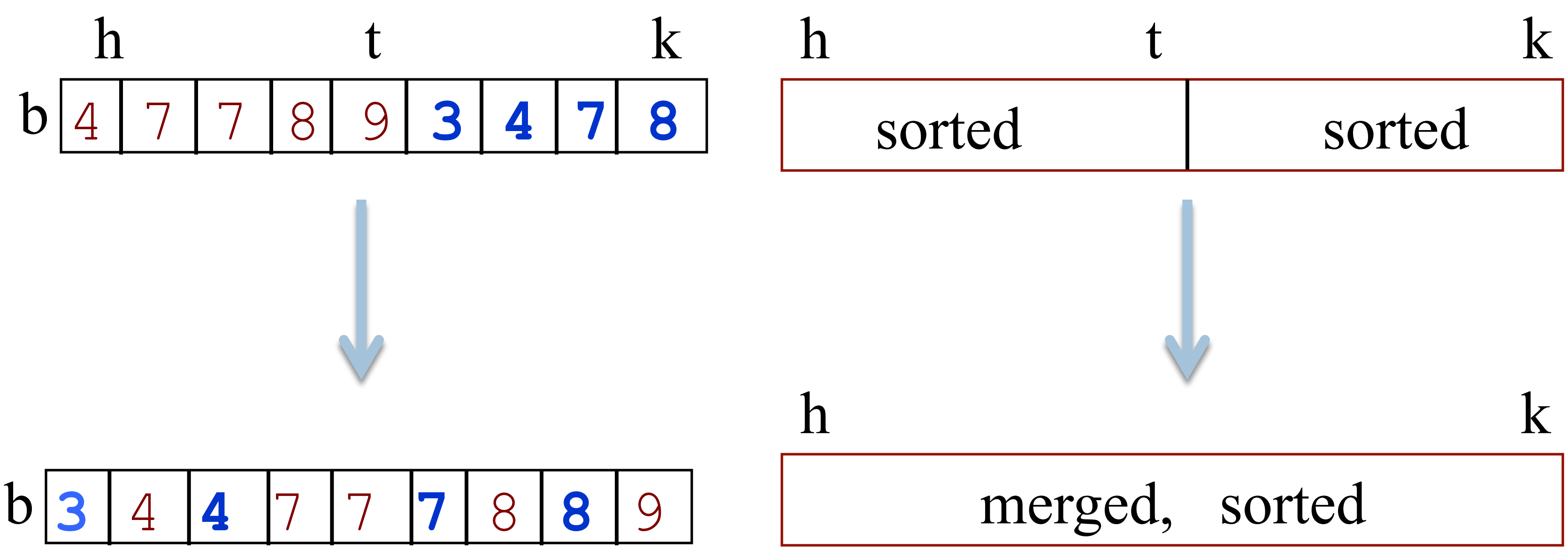
SEARCHING, SORTING, AND ASYMPTOTIC COMPLEXITY

Lecture 10

CS2110 – Fall 2015

Merge two adjacent sorted segments


```
/* 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


```
/* 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..]
}
```

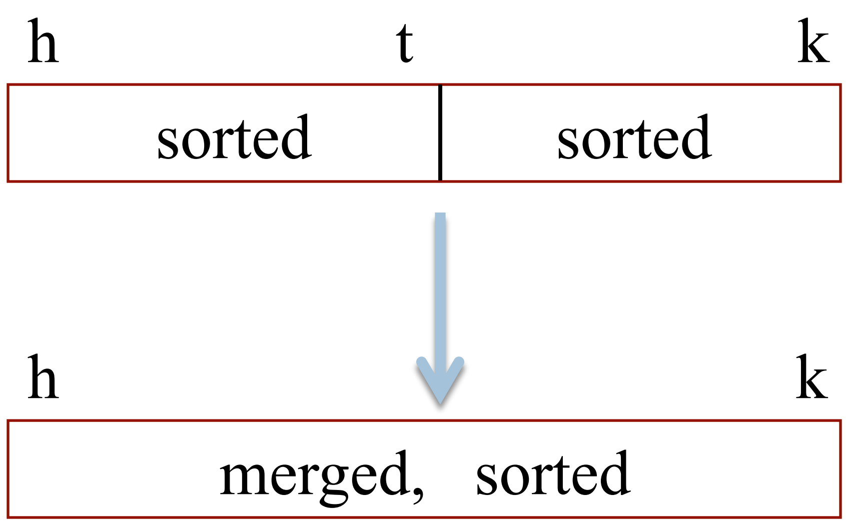
c	4	7	7	8	9				
	h				t				k
b	4	7	7	8	9	**3**	**4**	**7**	**8**
b	**3**	4	**4**	7	7	**7**	8	**8**	9

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].

Mergesort


```
/** 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


```
/** 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 shown (later) that time taken is proportional to n log n

But space is also proportional to n!

QuickSort versus MergeSort


```
/** 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

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

What Makes a Good Algorithm?

Suppose you have two possible algorithms or ADT implementations 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

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)

Counting basic steps in worst-case execution

Let n = b.length

Linear Search

```
/** 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

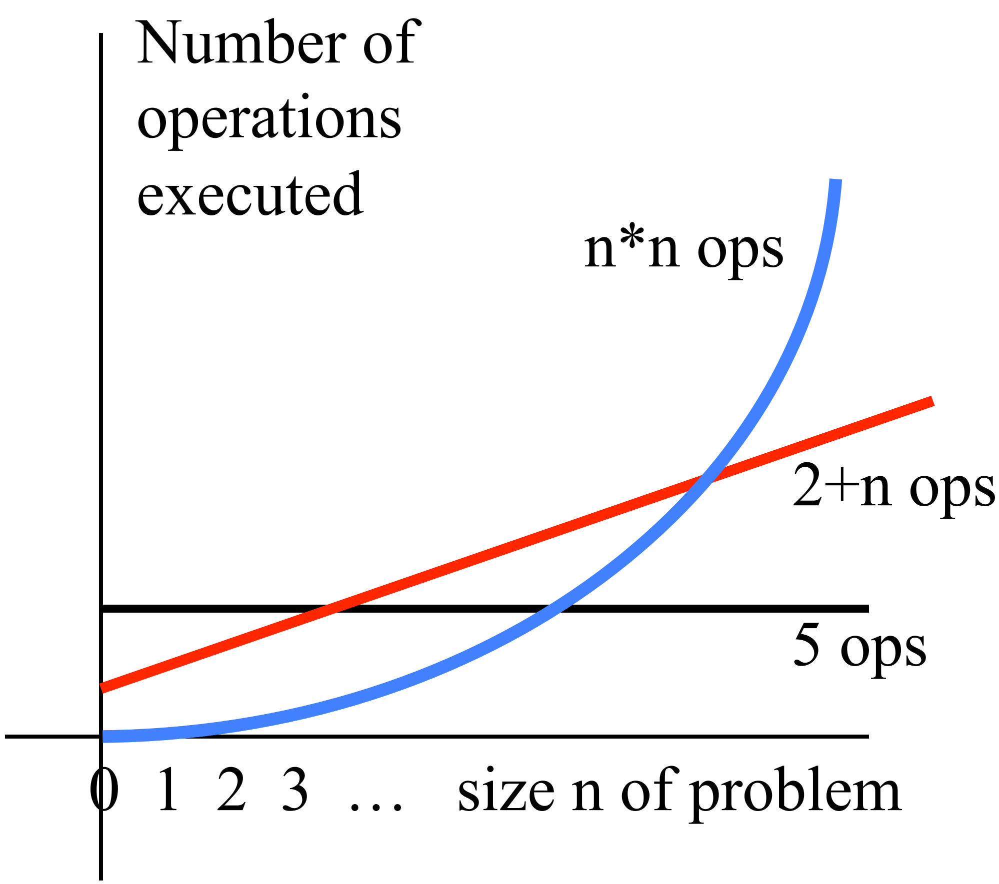
Second solution:
Binary Search

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

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

```
/** 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;
    while (h+1 != k) {
        int e= (h+ k)/2;
        if (b[e] <= v)  h= e;
        else k= e;
    }
    return h;
}
```

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



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

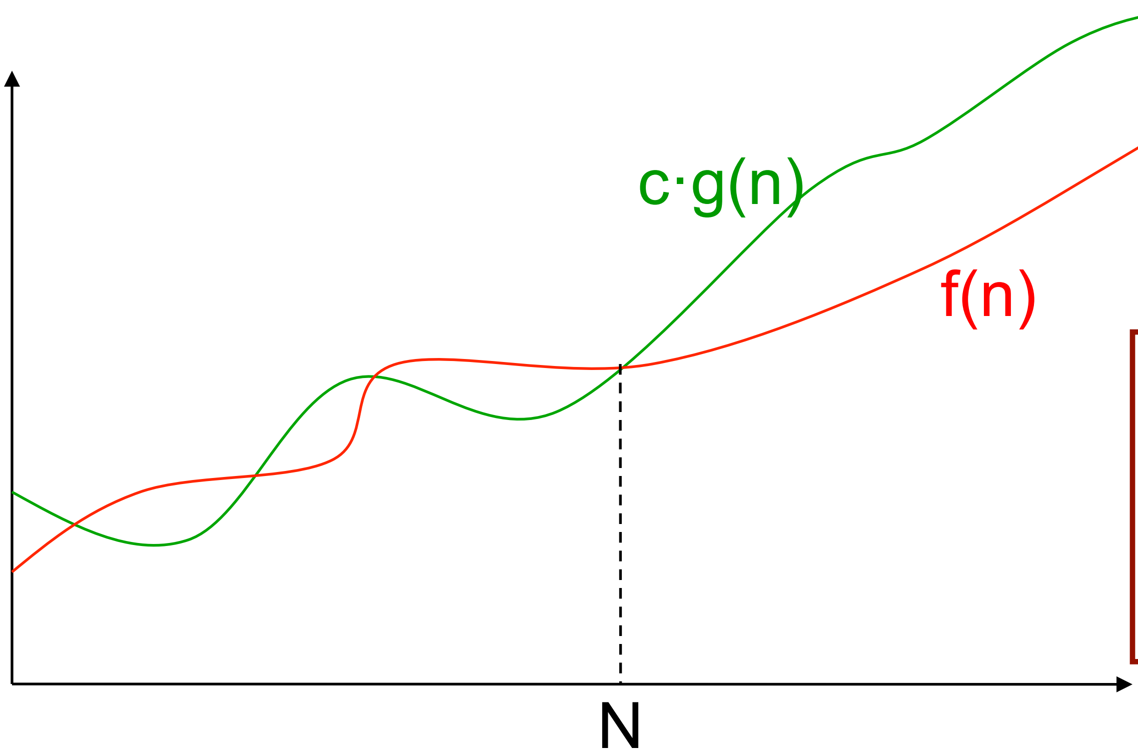
2. Distinguish among important cases, like
- n*n basic operations
- n basic operations
- log n basic operations
- 5 basic operations

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

Definition of O(…)

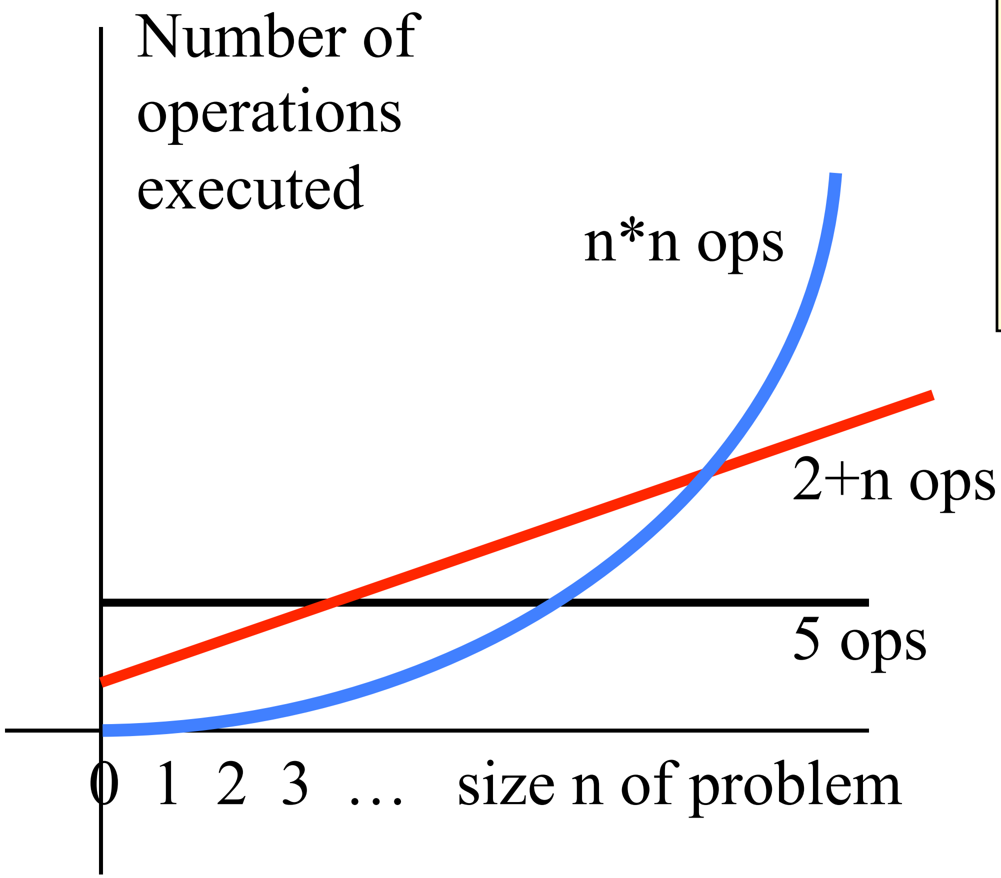
Formal definition: f(n) is O(g(n)) if there exist constants c and N such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Graphical view



Get out far enough –for n >=N– c·g(n) is bigger than f(n)

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



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

$f(n) \leq c \cdot g(n)$

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 $O(n^2)$

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

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 $O(n^2)$

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

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

f(n)

= <definition of f(n)>

$n^2 + n$

<= <for n >= 1, n <= 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)

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

f(n)

= <put in what f(n) is>

100 n + log n

<= <We know $\log n \le n$ for $n \ge 1$>

100 n + n

= <arith>

101 n

= <g(n) = n>

101 g(n)

Choose
N = 1 and c = 101

O(...) Examples

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

Problem-size examples

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

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

Commonly Seen Time Bounds

$O(1)$	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

Worst-Case/Expected-Case Bounds

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

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

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)

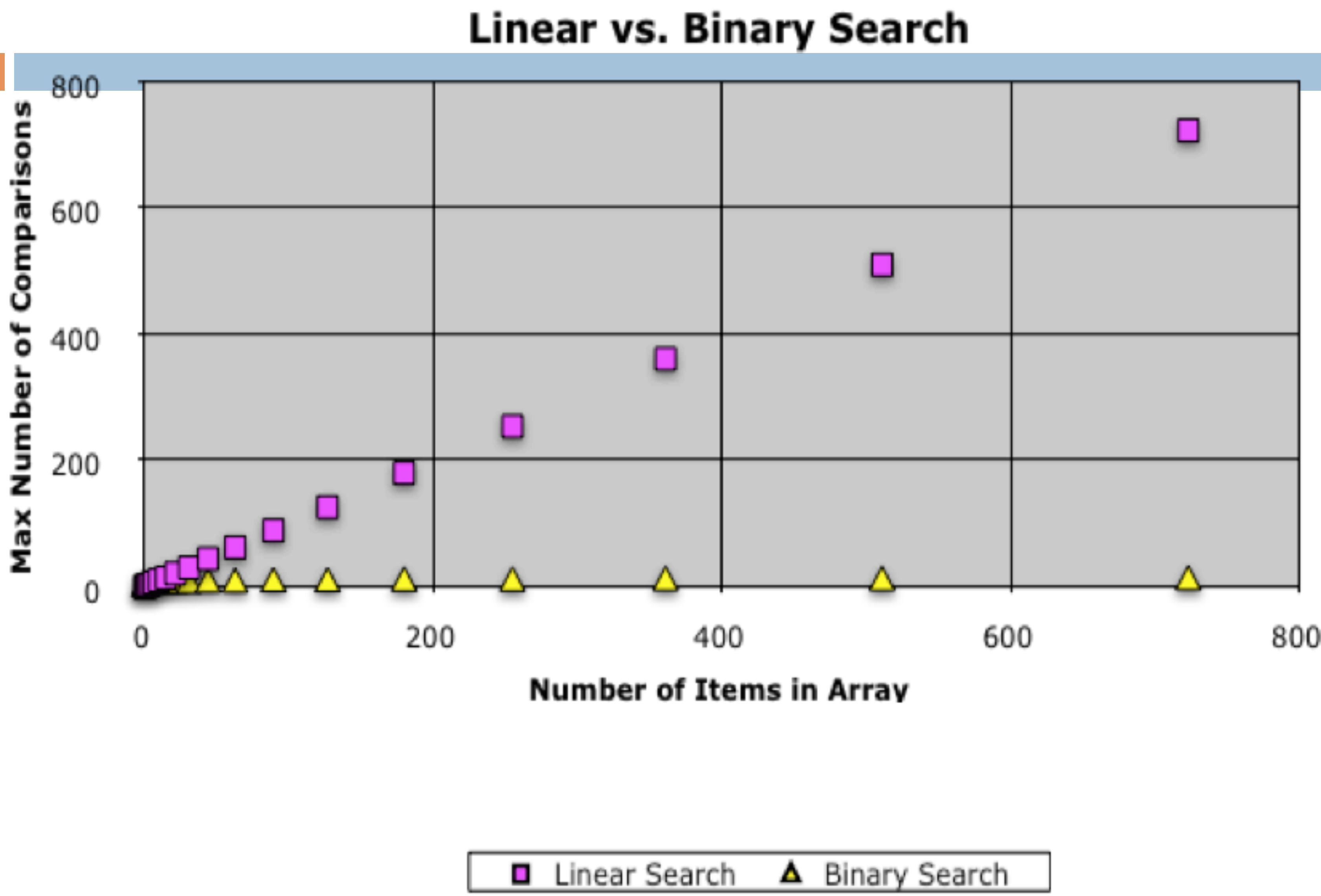
#b = size of b

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)


Linear vs. Binary Search
Max Number of Comparisons
800
600
400
200
0
0
200
400
600
800
Number of Items in Array
Linear Search
Binary Search

Analysis of Matrix Multiplication

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];
   }
```

Remarks

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

Why bother with runtime analysis?

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

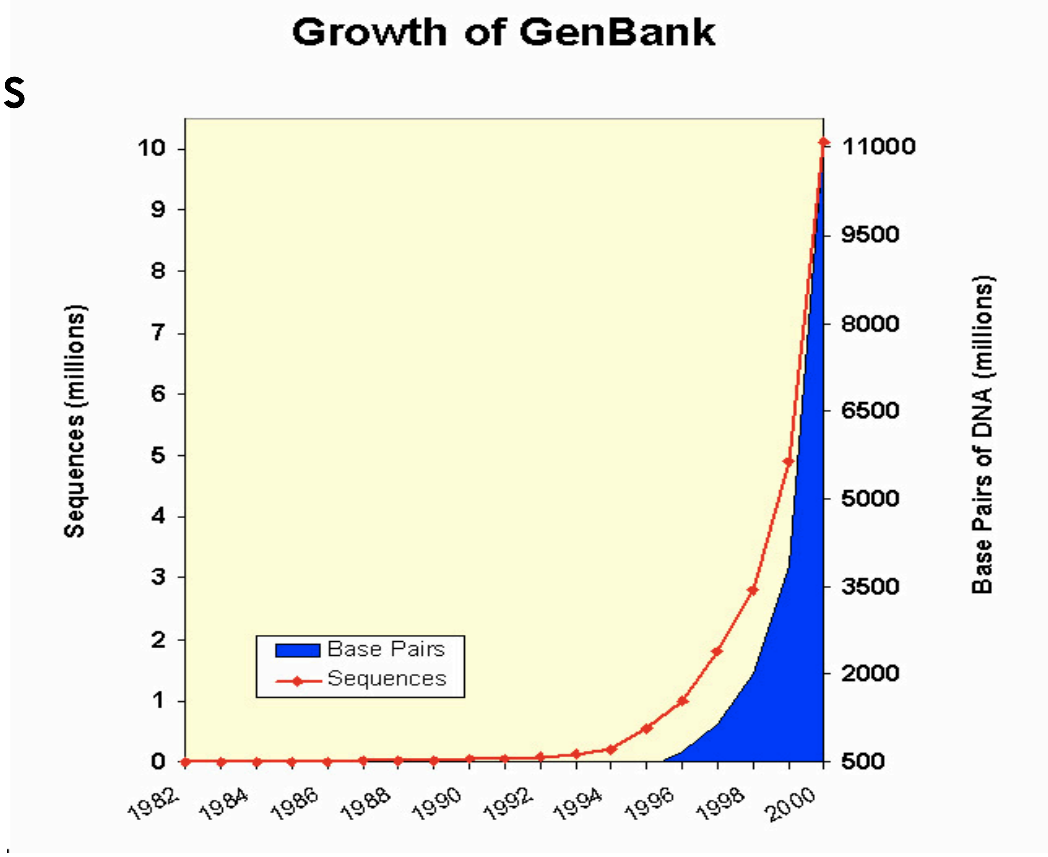
1,000 days ≈ 3 years

Algorithms for the Human Genome

Human genome
= 3.5 billion nucleotides
~ 1 Gb

@1 base-pair instruction/μsec

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



Limitations of Runtime Analysis

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

- 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

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

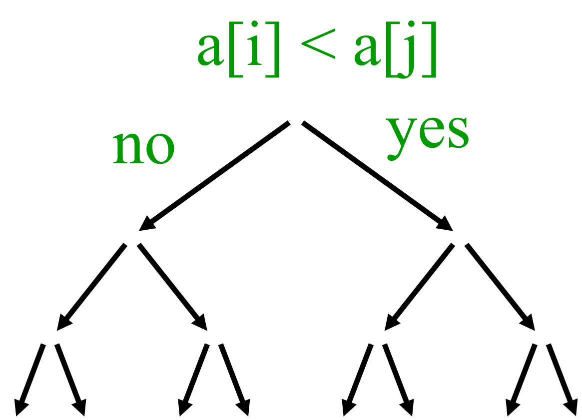
- How can we prove anything about the *best possible* algorithm?

- 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

- 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

- 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

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


```
/** 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