CSC230

Intro to C++ Lecture 25

Outline

- Introduction to Complexity
- Heap (next Tuesday)
- Final Exam Review (next Friday)

What Makes a Good Algorithm?

- Suppose you have two possible algorithms or data structures that basically do the same thing; which is *better*?

- Well… what do we mean by *better*?
  - Faster?
  - Less space?
  - Easier to code?
  - Easier to maintain?

- How do we measure time and space for an algorithm?

Program running time

When is the running time (waiting time for user) noticeable/important?

Program running time – Why?

When is the running time (waiting time for user) noticeable/important?

- web search
- database search
- real-time systems with time constraints

Sample Problem: Searching

- Determine if ***sorted*** array b contains integer v
- First solution: Linear Search (check each element)

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

Doesn't make use of fact that b is sorted.

Sample Problem: Searching

Second solution:
Binary Search

Still returning true if v is in a

Keep true: all occurrences of v are in b[low..high]

```
bool find (int* a, int length, int v) {
    int low= 0;
    int high= length - 1;
    while (low <= high) {
        int mid = (low + high)/2;
        if (a[mid] == v) return true;
        if (a[mid] < v)
            low= mid + 1;
        else   high= mid - 1;
    }
    return false;
}
```

Linear Search vs Binary Search

Which one is better?

- Linear: easier to program
- Binary: faster… isn't it?

How do we measure speed?

- Experiment?
- Proof?
- What inputs do we use?

- Simplifying assumption
- #1: Use ***size of input*** rather than input itself
  - For sample search problem, input size is n where n is array size

- Simplifying assumption
- #2: Count number of "***basic steps***" rather than computing exact times

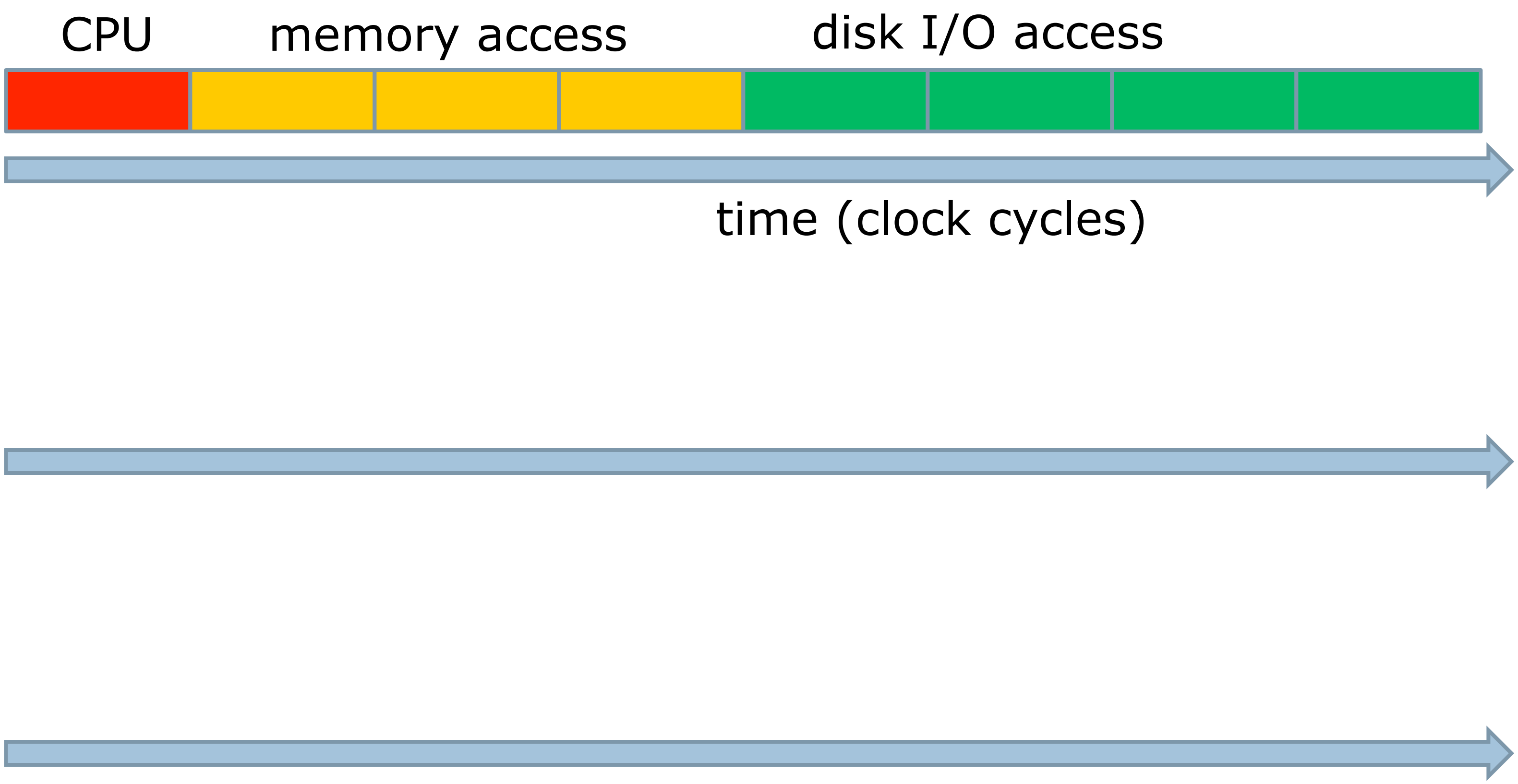
Factors that determine running time

- problem size: n
- basic algorithm / actual processing
- memory access speed
- CPU/processor speed
- # of processors?
- compiler/linker optimization?

Running time of a program or transaction processing time

- amount of input: n → min. linear increase
- basic algorithm / actual processing → depends on algorithm!
- memory access speed → by a factor
- CPU/processor speed → by a factor
- # of processors? → yes, if multi-threading or multiple processes are used.
- compiler/linker optimization? → ~20%

Running time: a closer look



One Basic Step = One Time Unit

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 **invocation** (not counting arg evaluation and execution of method body)

- For conditional: number of basic steps on branch that is executed
- For loop: (number of basic steps in loop body) * (number of iterations)
- For method: number of basic steps in method body (include steps needed to prepare stack-frame)

Runtime vs Number of Basic Steps

Is this cheating?

- The runtime is not the same as number of basic steps
- Time per basic step varies depending on computer, compiler, details of code…

Well … yes, in a way

- But the number of basic steps is *proportional* to the actual runtime

Which is better?

- n or n^2 time?
- $100\ n$ or n^2 time?
- $10{,}000\ n$ or n^2 time?

As n gets large, multiplicative constants become less important

Simplifying assumption #3: Ignore multiplicative constants

Time Complexity

- measure of algorithm efficiency
- has a big impact on running time.
- Big-O notation is used.
- To deal with n items, time complexity can be $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, even $O(n^n)$.

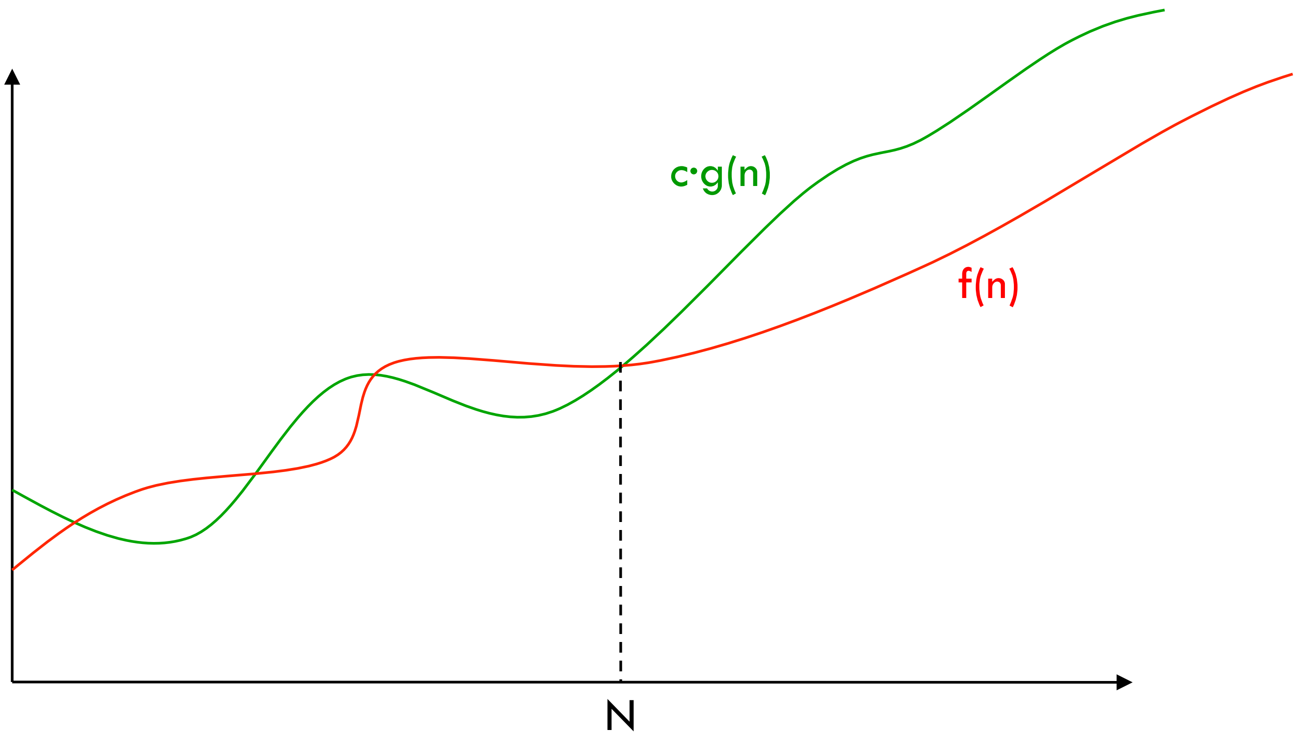
Using Big-O to Hide Constants

- We say f(n) is *order of* g(n) if f(n) is **bounded** by a **constant** times g(n)
- Notation: f(n) is O(g(n))
- Roughly, f(n) is O(g(n)) means that f(n) grows like g(n) or slower, to within a constant factor
- "Constant" means fixed and independent of n

- Example: $(n^2 + n)$ is ?
  - $O(n^2)$
- We know $n \leq n^2$ for $n \geq 1$
- So $n^2 + n \leq 2\ n^2$ for $n \geq 1$
- So by definition, $n^2 + n$ is $O(n^2)$ for c=2 and N=1

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

A Graphical View



To prove that f(n) is O(g(n)):

- Find N and c such that $f(n) \leq c\ g(n)$ for all n > N
- Pair (c, N) is a *witness pair* for proving that f(n) is O(g(n))

Big-O Examples

Claim: 100 n + log n is?

O(n)

We know log n ≤ n for n ≥ 1

So 100 n + log n ≤ 101 n

for n ≥ 1

So by definition,

100 n + log n is O(n)

for c = 101 and N = 1

Claim: $\log_B n$ is $O(\log_A n)$

since

$\log_B n = (\log_B A)(\log_A n)$

Question: Which grows faster: n or log n?

Big-O Examples

Let $f(n) = 3n^2 + 6n - 7$

- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^3)$
- $f(n)$ is $O(n^4)$
- …

$g(n) = 4\ n \log n + 34\ n - 89$

- $g(n)$ is $O(n \log n)$
- $g(n)$ is $O(\log n)$
- $g(n)$ 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

Problem-Size Examples

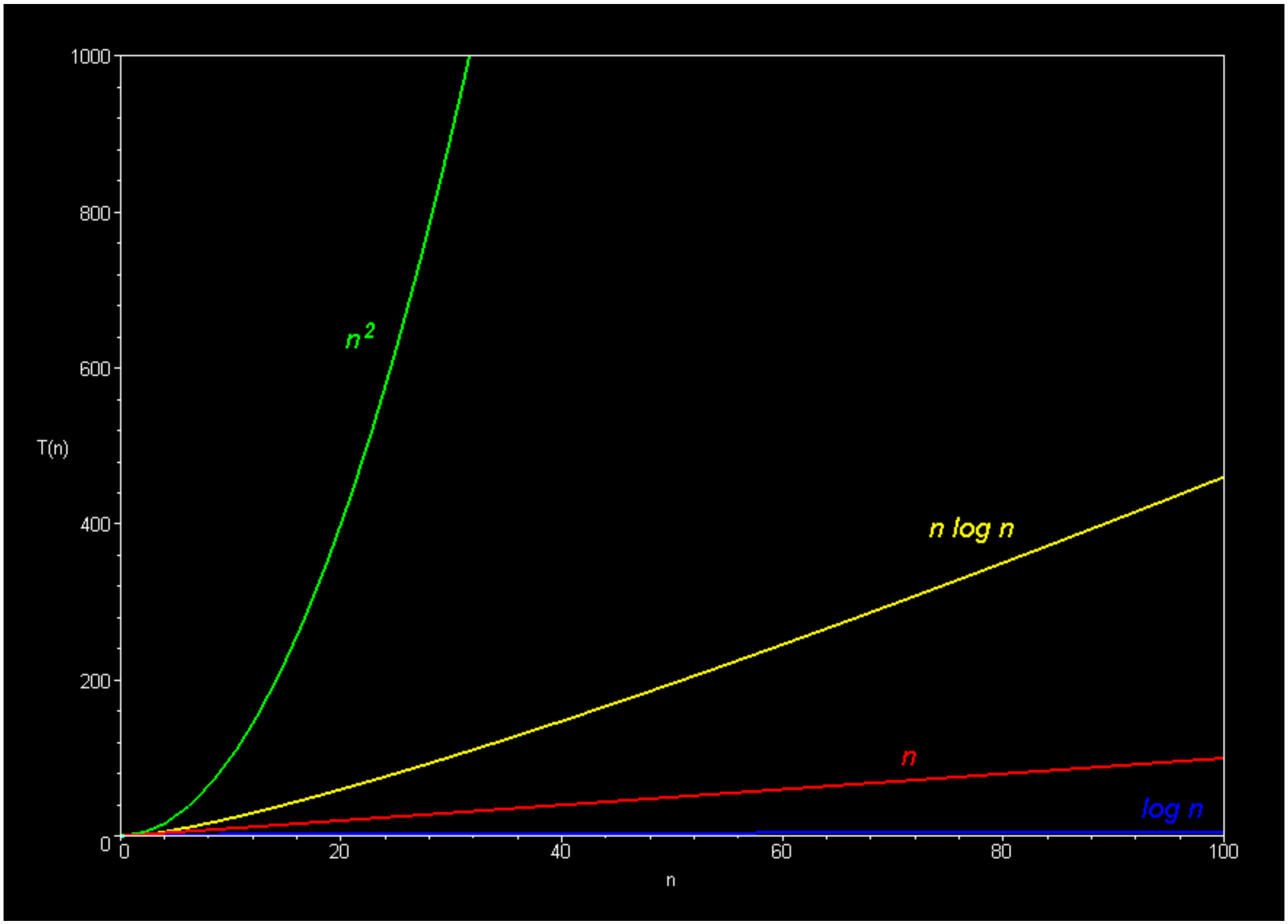
- Consisider a computing device that can execute 1000 operations per second; how large a problem can we solve?

	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

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

Commonly Seen Time Bounds



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
- expected-case / 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
bool find(int* b, int length, int v) {
   for (int i = 0; i < length; i++) {
      if (b[i] == v) return true;
   }
   return false;
}
```

worst-case time: O(n)

Binary Search

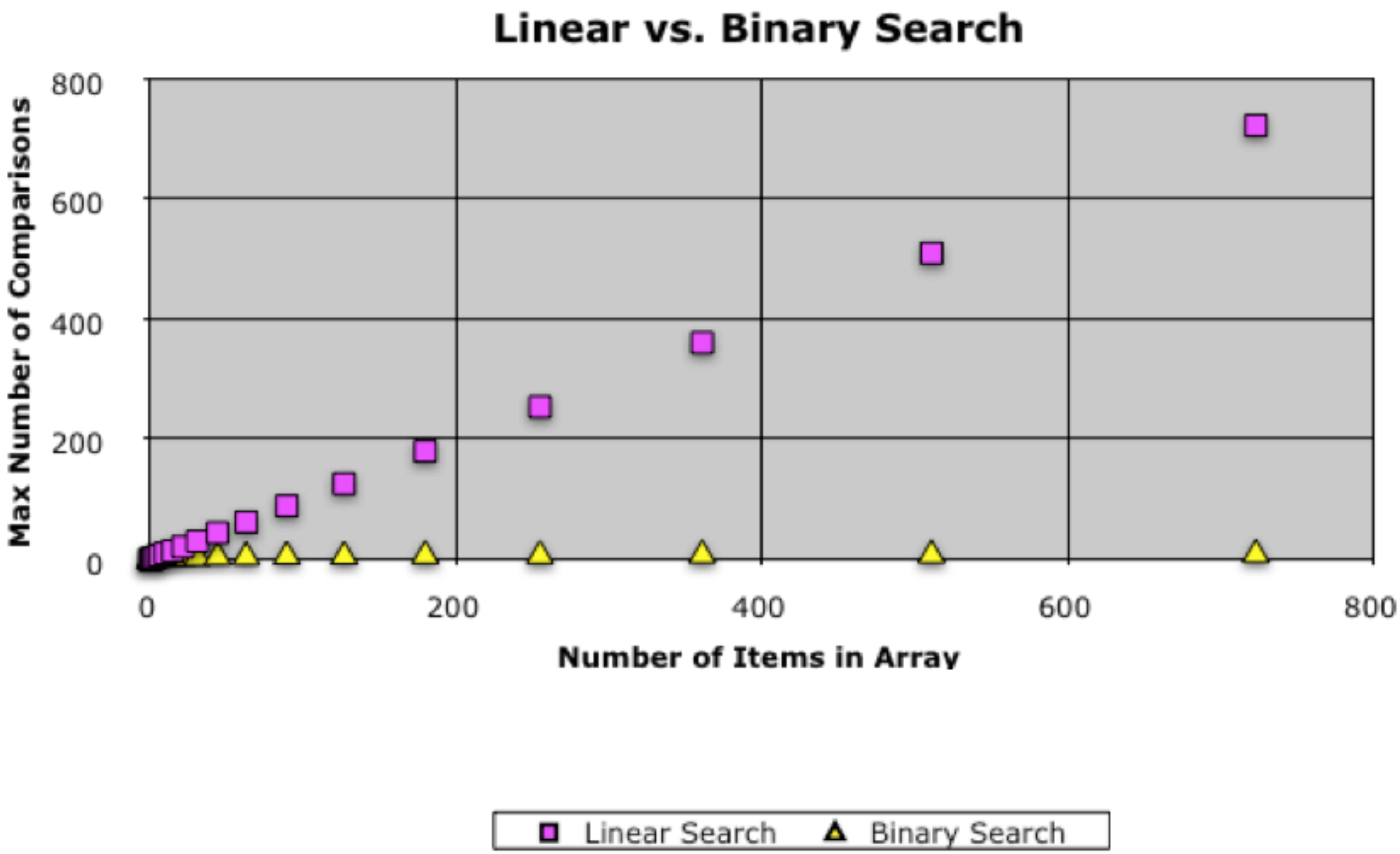
```
// Return h that satisfies
//      b[0..h] <= v < b[h+1..]
bool bsearch(int* b, int length, int v){
   int h= -1;  int t= length;
   while ( h != t-1 ) {
        int  e= (h+t)/2;
        if (b[e] <= v)  h= e;
        else t= e;
   }
}
```

Always takes ~(log n+1) iterations.

Worst-case and expected times:

O(log n)

Comparison of linear and 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];
   }
```

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

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 be 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

Example #1: carry n items from one room to another room

- How many operations?
- n pick-ups, n forward moves, n drops and n reverse moves → 4 n operations
- 4n operations = c. n = O(c. n) = O(n)
- Similarly, any program that reads n inputs from the user will have minimum time complexity O(n).

Example #2: Locating patient record in Doctor Office

What is the time complexity of search?

- Binary Search algorithm at work
- O(log n)
- Sequential search?
- O(n)

Example #3: Store manager gives gifts to first 10 customers

- There are n customers in the queue.
- Manager brings one gift at a time.
- Time complexity = O(c. 10) = O(1)
- Manager will take exactly same time irrespective of the line length.

Outline

Sorting complexity

QuickSort

6 5 3 1 8 7 2 4

Execution of logarithmic-space Quicksort


```
/** Sort b[h..k]. */
void QS(int b[], int h, int k) {
   int h1= h; int k1= k;
   // inv; b[h..k] is sorted if b[h1..k1] is
   while (size of b[h1..k1] > 1) {
        int j= partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
             {  QS(b, h, j-1);  h1=  j+1; }
       else
             {QS(b, j+1, k1);  k1=  j-1; }
   }
}
```

Call QS(b, 0, 11);


```
void QS(int b[], int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
     int j= partition(b, h1, k1);
     // b[h1..j-1] <= b[j] <= b[j+1..k1]
     if (b[h1..j-1] smaller than b[j+1..k1])
        {  QS(b, h, j-1);  h1=  j+1; }
     else {QS(b, j+1, k1);  k1=  j-1; }
  }
}
```

Initially, h is 0 and k is 11. The initialization stores 0 and 11 in h1 and k1. The invariant is true since h = h1 and k = k1.

j	?		
h	0	k	11
h1	0	k1	11

0											11
3	4	8	7	6	8	9	1	2	5	7	9

Call QS(b, 0, 11);


```
void QS(int b[], int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
     int j= partition(b, h1, k1);
     // b[h1..j-1] <= b[j] <= b[j+1..k1]
     if (b[h1..j-1] smaller than b[j+1..k1])
        { QS(b, h, j-1); h1= j+1; }
     else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

The assignment to j partitions b, making it look like what is below. The two partitions are underlined

0		j									11
2	1	3	7	6	8	9	4	8	5	7	9

j: 2

h: 0 k: 11

h1: 0 k1: 11

Call QS(b, 0, 11);


```
void QS(int b[], int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
     int j= partition(b, h1, k1);
     // b[h1..j-1] <= b[j] <= b[j+1..k1]
     if (b[h1..j-1] smaller than b[j+1..k1])
        { QS(b, h, j-1); h1= j+1; }
     else {QS(b, j+1, k1); k1= j-1; }
  }
}
```

The left partition is smaller, so it is sorted recursively by this call. We have changed the partition to the result.

h	0	k	11
h1	0	k1	11

0 j 11

1	2	3	7	6	8	9	4	8	5	7	9

Call QS(b, 0, 11);


```
void QS(int b[], int h, int k) {
  int h1= h; int k1= k;
  // inv; b[h..k] is sorted if b[h1..k1] is
  while (size of b[h1..k1] > 1) {
     int j= partition(b, h1, k1);
     // b[h1..j-1] <= b[j] <= b[j+1..k1]
     if (b[h1..j-1] smaller than b[j+1..k1])
       {  QS(b, h, j-1);  h1=  j+1; }
     else {QS(b, j+1, k1);  k1=  j-1; }
   }
}
```

The assignment to h1 is done.

Do you see that the inv is true again? If the underlined partition is sorted, then so is b[h..k]. Each iteration of the loop keeps inv true and reduces size of b[h1..k1].

0		j									11
1	2	3	7	6	8	9	4	8	5	7	9

j	2		
h	0	k	11
h1	3	k1	11

Divide & Conquer!

It often pays to

- Break the problem into smaller subproblems,
- Solve the subproblems separately, and then
- Assemble a final solution

This technique is called *divide-and-conquer*

- Caveat: It won't help unless the *partitioning* and *assembly* processes are inexpensive

We did this in Quicksort: Partition the array and then sort the two partitions.

MergeSort

Ideal divide-and-conquer algorithm:

Divide array into equal parts, sort each part (recursively), then merge

Questions:

- Q1: How do we divide array into two equal parts?

  A1: Find middle index: length/2

- Q2: How do we sort the parts?

  A2: Call MergeSort recursively!

- Q3: How do we merge the sorted subarrays?

  A3: It takes linear time.

Merging Sorted Arrays A and B into C

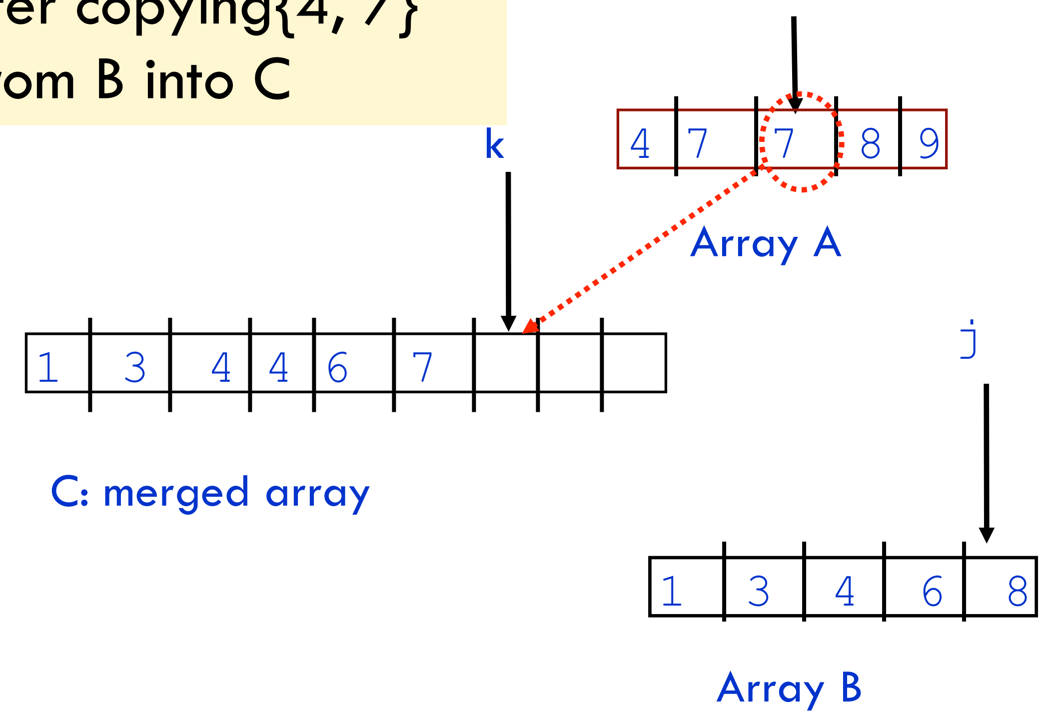
Picture shows situation after copying{4, 7} from A and {1, 3, 4, 6} from B into C

A[0..i-1] and B[0..j-1] have been copied into C[0..k-1].

C[0..k-1] is sorted.

Next, put a[i] in c[k], because a[i] < b[j].

Then increase k and i.



Merging Sorted Arrays `A` and `B` `into C`

- Create array C of size: size of A + size of B
- i= 0; j= 0; k= 0; // initially, nothing copied
- Copy smaller of A[i] and B[j] into C[k]
- Increment i or j, whichever one was used, and k
- When either A or B becomes empty, copy remaining elements from the other array (B or A, respectively) into C

This tells what has been done so far:

A[0..i-1] and B[0..j-1] have been placed in C[0..k-1].

C[0..k-1] is sorted.

MergeSort


```
/** Sort b[h..k] */
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);
}
```

merge 2 sorted arrays

QuickSort vs MergeSort


```
/** Sort b[h..k] */
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] */
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);
}
```

merge 2 sorted arrays

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

MergeSort Analysis

Outline

- Split array into two halves
- Recursively sort each half
- Merge two halves

Merge: combine two sorted arrays into one sorted array:

- Time: O(n) where n is the total size of the two arrays

Runtime recurrence

T(n): time to sort array of size n

$T(1) = 1$

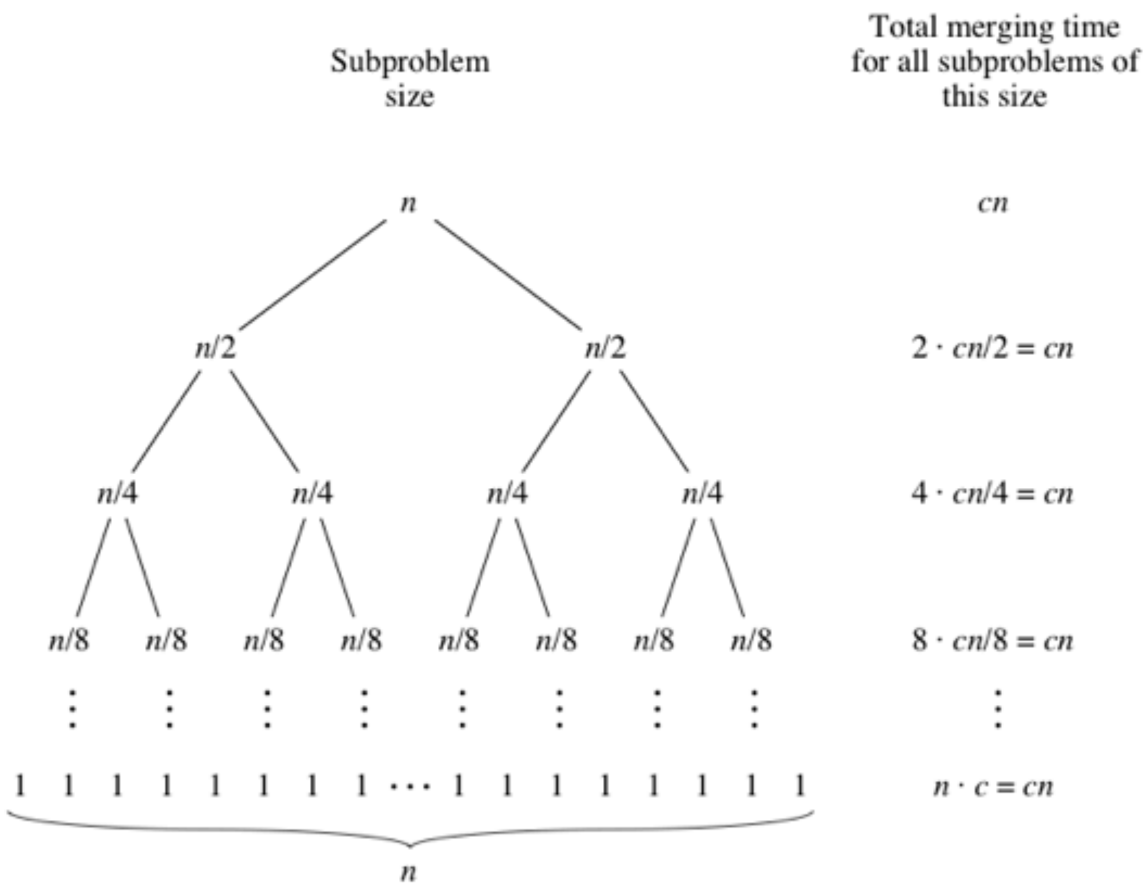
$T(n) = 2T(n/2) + O(n)$

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

MergeSort Analysis



MergeSort Notes

- Asymptotic complexity: O(n log n)

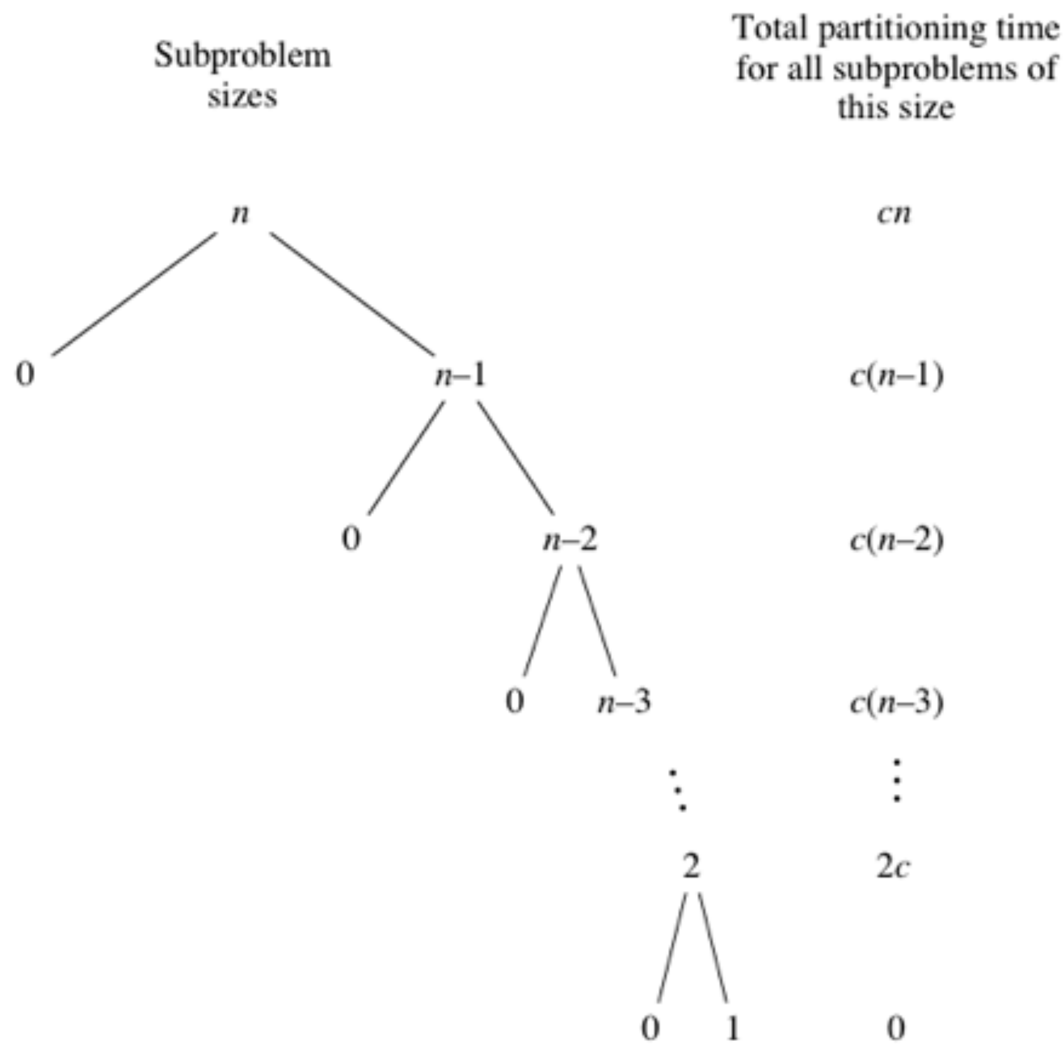
  Much faster than $O(n^2)$

- Disadvantage
  - Need extra storage for temporary arrays
  - In practice, can be a disadvantage, even though MergeSort is *near optimal for sorting*

- Good sorting algorithm that does not use so much extra storage?
  - Yes: QuickSort —when done properly, uses (log n) space.

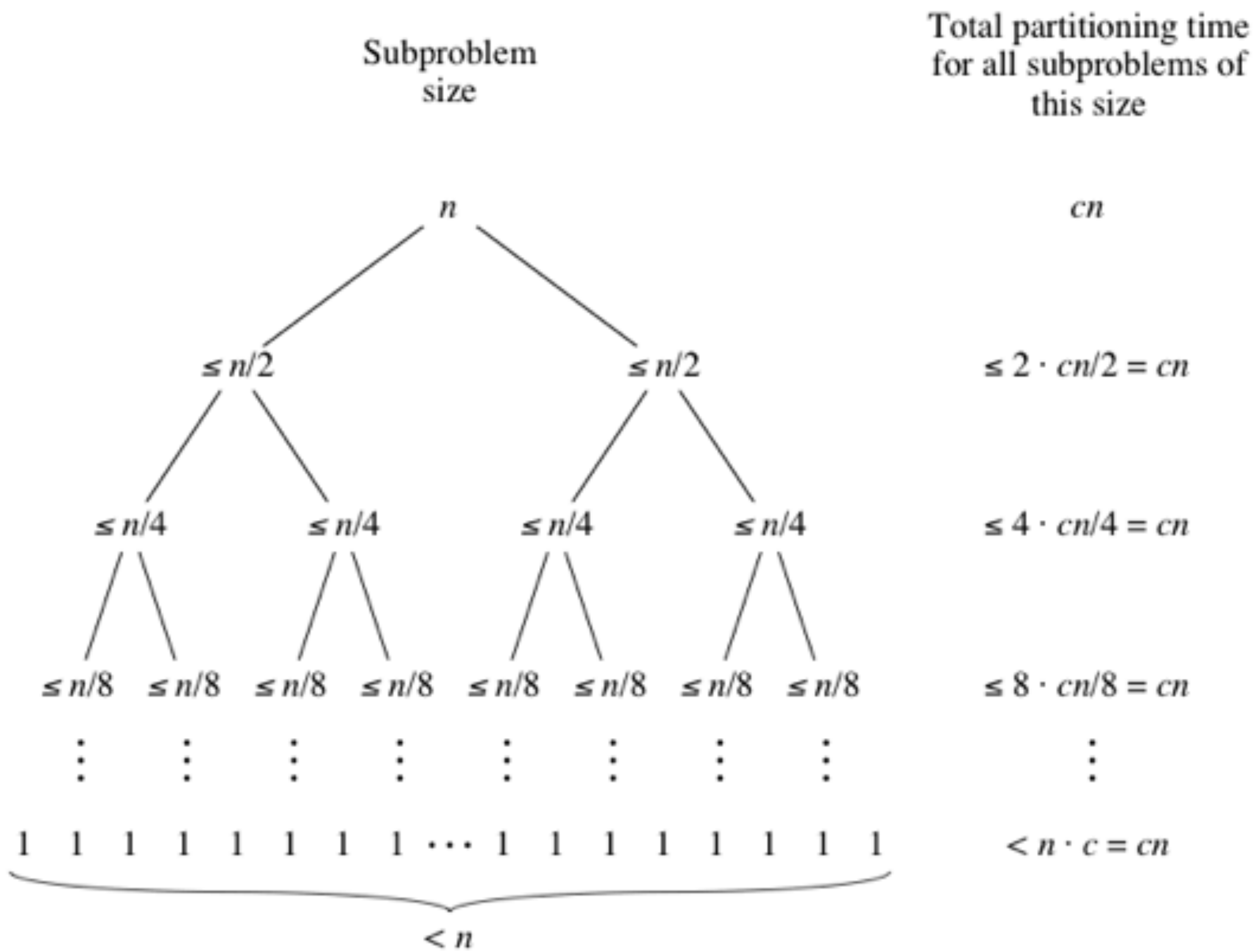
QuickSort

- Worst case?
  - O(n^2)



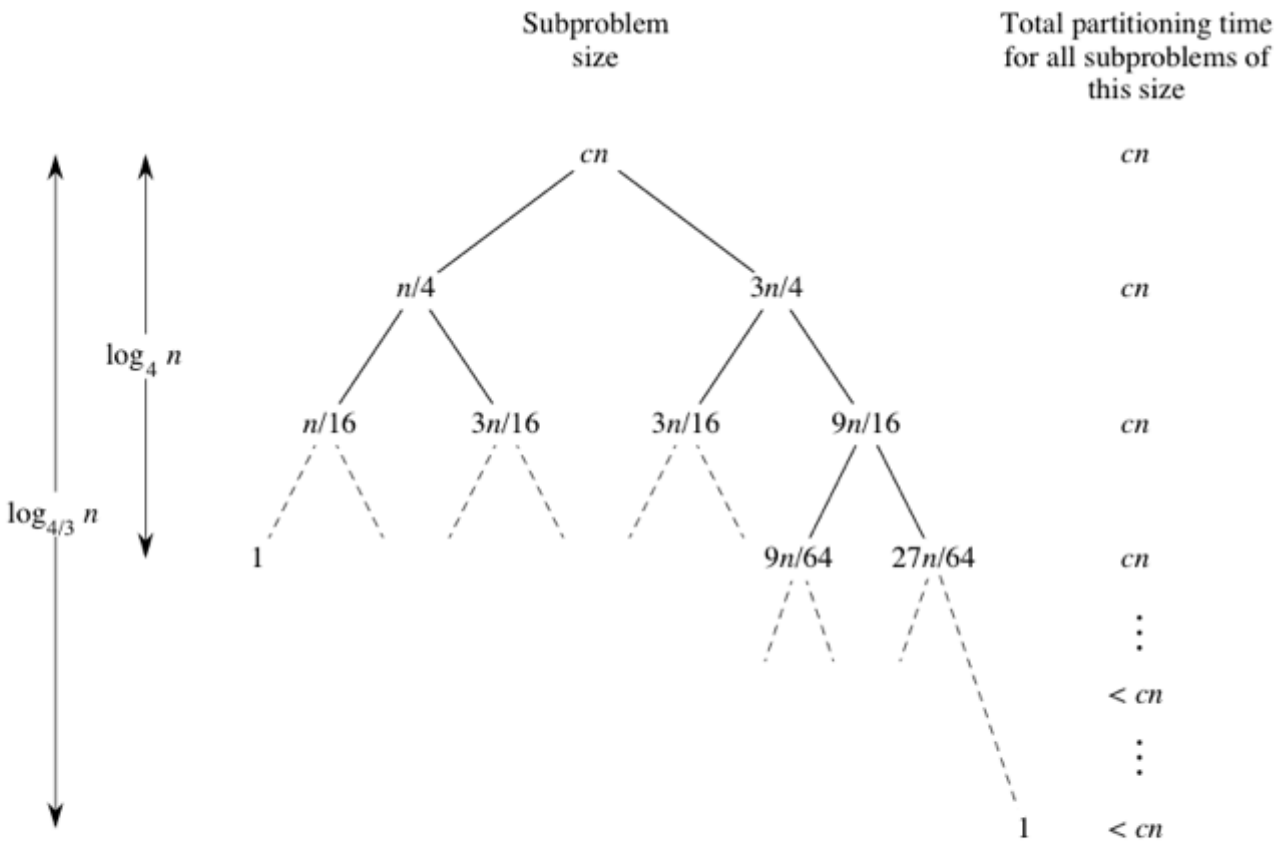
QuickSort

- Best case?
- O(nlogn)



QuickSort

- Average case?
  - O(nlogn)



QuickSort Analysis

Runtime analysis (worst-case)

- Partition can produce this:

p	**$\geq$ p**

- Runtime recurrence: $T(n) = T(n-1) + n$
- Can be solved to show worst-case $T(n)$ is $O(n^2)$
- Space can be $O(n)$ —max depth of recursion

Runtime analysis (expected-case)

- More complex recurrence
- Can be solved to show *expected* $T(n)$ is $O(n \log n)$

Improve constant factor by avoiding QuickSort on *small* sets

- Use InsertionSort (for example) for sets of size, say, ≤ 9
- Definition of *small* depends on language, machine, etc.

Sorting Algorithm Summary

We discussed

- InsertionSort
- SelectionSort
- MergeSort
- QuickSort

Other sorting algorithms

- HeapSort (will revisit)
- ShellSort
- BubbleSort
- RadixSort
- BinSort
- CountingSort

Why so many?

Stable sorts: **Ins, Sel, Mer**

Worst-case O(n log n): **Mer, Hea**

Expected O(n log n): **Mer, Hea, Qui**

Best for nearly-sorted sets: **Ins**

No extra space: **Ins, Sel, Hea**

Fastest in practice: **Qui**

Least data movement: **Sel**

A sorting algorithm is stable if: equal values stay in same order: b[i] = b[j] and i < j means that b[i] will precede b[j] in result