University of Southern California

Viterbi School of Engineering

EE599 – Special Topics: Software Design and Optimization for Electrical Engineers

Introduction to Algorithms

Reference: Online resources, research papers, Professor Mark Redekopp's EE355 Course Materials

Definition

- **An Algorithm is an effective procedure with well-defined set (i.e., a rigid sequence) of steps to process certain inputs, and calculate certain outputs**
- **Example: Find Min**

  Set MINI to first number

  For each number x in list L

  If x <MINI => MINI = x
- **Implementation:**

```
double MINI = x[0];
for(i=1;i<x[i];i++) {
if(x[i]< MINI)  MINI =x[i];
}
printf("\n The MIN is %f \n", MINI);
```

Al-Kharizmi (780 - 850 AD)

Formal Definition

- For a computer, "algorithm" is defined as...
  - …an ordered set of unambiguous, executable steps that defines a terminating process
- Explanation:
  - Ordered Steps: the steps of an algorithm have a particular order, not just any order
- Unambiguous: each step is completely clear as to what is to be done
- Executable: Each step can actually be performed
- Terminating Process: Algorithm will stop, eventually. (sometimes this requirement is relaxed)

Algorithm Representation

- An algorithm is not a program or programming language
- Just as a story may be represented as a book, movie, or spoken by a story-teller, an algorithm may be represented in many ways
  - Flow chart
  - Pseudocode (English-like syntax using primitives that most programming languages would have)
  - A specific program implementation in a given programming language

Algorithms – SW or HW?

- Algorithms are at the heart of computer systems, both in HW and SW
  - They are fundamental to Computer Science and Electrical Engineering

SW programs: implement collection of Algorithms to perform tasks

How about HW components?

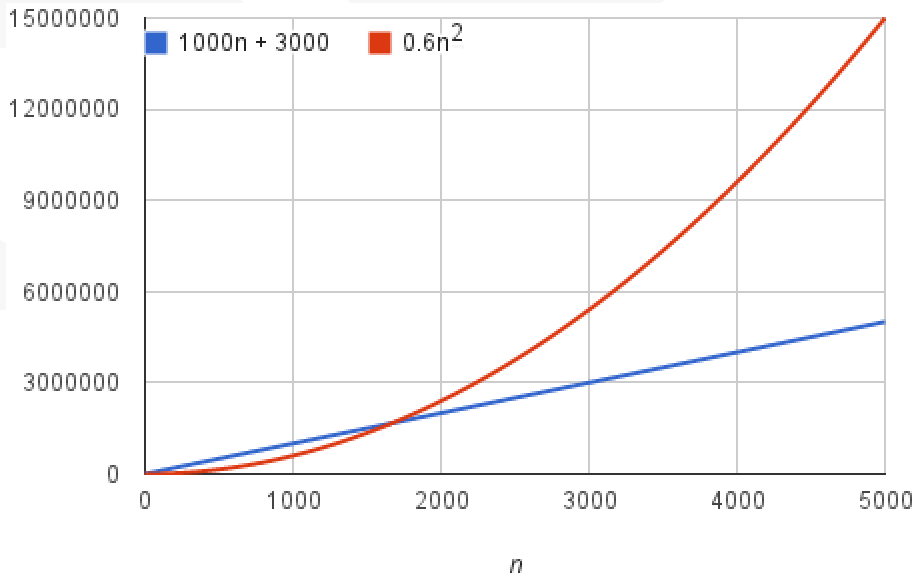
Benefits of Algorithms

- Problem solving
  - Algorithms help identifying the processes, key decision points and variables required for solving the corresponding problems
  - They help dividing the tasks into more manageable smaller subtasks and convert a problem (that would normally have been impossible or difficult to resolve) into a series of much smaller and solvable
  - They help classifying real life problems and using our knowledge to modify the existing algorithms or develop new ones to solve them
- Efficiency
  - Choosing the right algorithm can often lead to a dramatic increase in performance
- Clarity
- Question: Is the relation between algorithms and coding related to design or verification steps?

Algorithm Design

- **Correctness**: Does the algorithm do what it is supposed to do?

- **Efficiency**: Does the algorithm have a runtime complexity that is polynomially bounded? Is it as fast as possible?



Algorithm Checks

- **Prove the algorithm is correct:**
  - **Again, keep it simple: briefly say why the algorithm works in general and then focus on the non-obvious parts**
  - **Assume you are trying to convince one of your classmates**

- **Analyze its efficiency:**
  - **Ensure that it runs in polynomial time**
  - **Then try to give the best possible, worst-case upper bound on how many steps it will take**
  - **Check how much memory it needs to avoid memory explosion**

Algorithm Description – How Detailed?

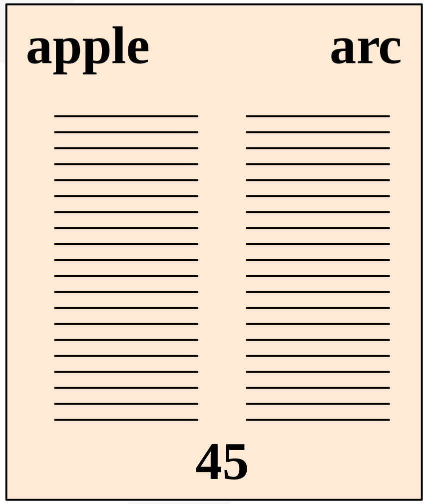
- Keep it as simple as possible, but no simpler. Difficult algorithms require more detail than intuitively obvious ones. No need to “write assembly code”: high-level statements that can obviously be implemented are fine
- Context vs detail level
- Examples:
  - If S is a set, we can generally assume we can iterate its elements, test if p is in S, etc. In some contexts we can assume calculating $S1 \cap S2$ is obviously easy. In others, we have to spell out how to do this, E.g. suppose S is the set of primes)
- Knowing the level of detail to present is a bit of an art. The solved exercises in textbooks and online resources can help

Humans and Computers

- Humans understand algorithms differently than computers
- Humans easily tolerate ambiguity and abstract concepts using context to help
  - "Add a pinch of salt." How much is a pinch?
  - "Michael Jordan could soar like an eagle"
  - "It's a bull market"
- Computers only execute well-defined instructions (no ambiguity) and operate on digital information which is definite and discrete (everything is exact and not "close to")

Exercise

- How to find a word in a dictionary
  - Describe an "efficient" method
  - Assumptions / Guidelines
    - Let *target_word* = word to lookup
    - N pages in the dictionary
    - Each page has the *start* and *last* word on that page listed at the top of the page
    - The user understands how to perform alphabetical ("lexicographic") comparison (e.g., "abc" is smaller than "acb" or "abcd")



Central Dogma of Computer Science

- **Difficulty** is not necessarily proportional to **size of the search space**

Example

Code	Cost (Time require for line (Units))	Repeatation (No. of Times Executed)	Total (Total Time required in worst case)
`int sumOfList( int A[ ], int n)`			
`{`			
`int sum = 0, i;`	1	1	1
`for(i = 0; i < n; i++)`	1 + 1 + 1	1 + (n+1) + n	2n + 2
`sum = sum + A[i];`	2	n	2n
`return sum;`	1	1	1
`}`			
			4n + 4 Total Time required

$\Rightarrow O(n)$

Example

```
void printFirstItem(const vector<int>& vectorOfItems)
{
    cout << vectorOfItems[0] << endl;
}
```

Example

```
void printAllItems(const vector<int>& vectorOfItems)
{
    for(int item : vectorOfItems) {
        cout << item << endl;
    }
}
```

Example

```
void printAllPossibleOrderedPairs(const vector<int>& vectorOfItems)
{
  for (int firstItem : vectorOfItems)  {
    for (int secondItem : vectorOfItems)  {
      cout << firstItem << ", " << secondItem <<  endl;
    }
  }
}
```

Example – Factoring

- Find all factors of a natural number, n
  - What is a factor?
  - What is the range of possible factors?

```
i ← 1
while(i <= n) do
F    if (remainder of n/i is zero) then
T      List i as a factor of n
     i ← i+1
```

- An improvement

```
i ← 1
while(i <= sqrt(n) ) do
   if (remainder of n/i is zero) then
     List i and n/i as a factor of n
   i ← i+1
```

Example – Ordered Search

- Searching an ordered list (array) for a specific value, k
- Sequential Search
  - Start at first item, check if it is equal to k, repeat for second, third, fourth item, etc.

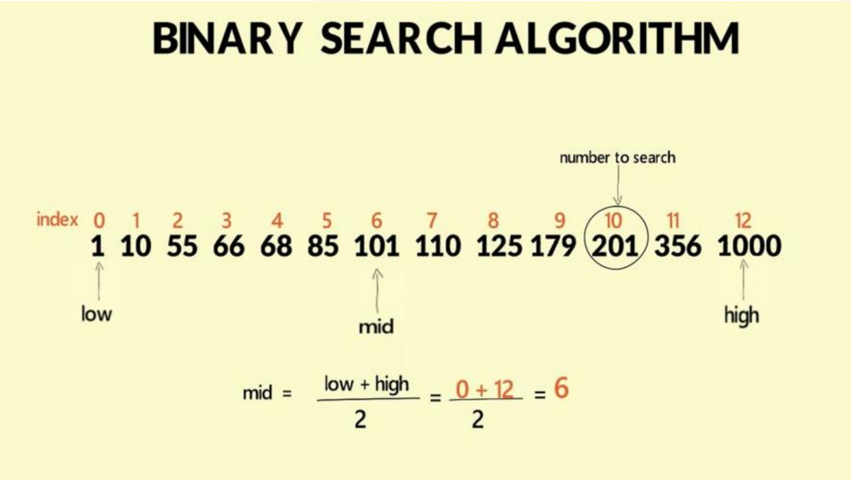
```
i ← 0
while ( i < length(myList) ) do
    if (myList[i] equal to k) then stop
    else i ← i+1
if (i == length(myList) ) then k is not in myList
else k is located at index i
```

myList	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

Example – Search (cont.)

- **Sequential search does not take advantage of the ordered nature of the list**
  - **Would work the same (equally well) on an ordered or unordered list**
- **Binary Search**
  - **Take advantage of ordered list by comparing k with middle element and based on the result, rule out all numbers greater or smaller, repeat with middle element of remaining list, etc.**

Example:



Example:

k = 6

List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

Start in middle

6 < 9

List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

6 > 4

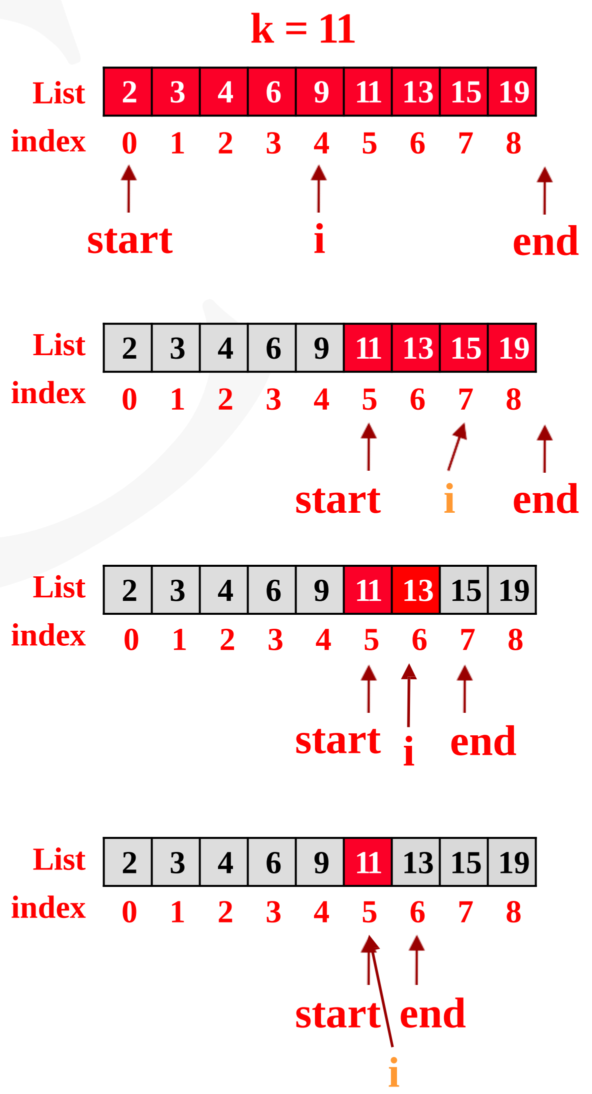
List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

6 = 6

Binary Search (cont.)

- Binary Search
  - Compare k with middle element of list and if not equal, rule out ½ of the list and repeat on the other half
  - "Range" Implementations in most languages are [start, end)
    - Start is inclusive, end is non-inclusive (i.e. end will always point to 1 beyond true ending index to make arithmetic work out correctly)

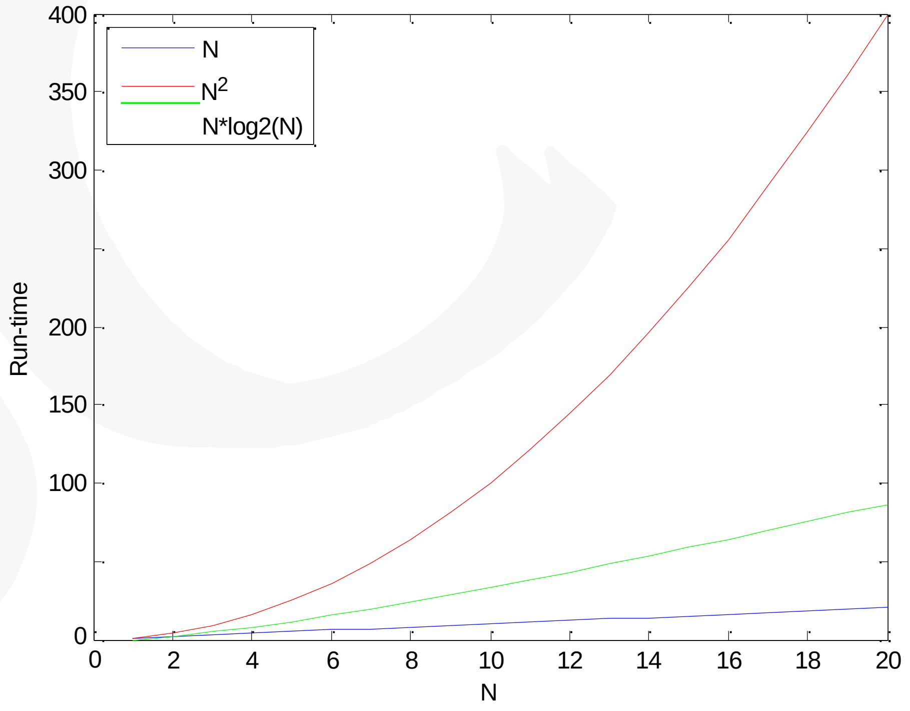
```
start ← 0; end ← length(List);
while (start < end) do
  i ← (end + start) / 2;
  if ( k == List[i] ) then return i;
  elseif ( k < List[i] ) then end ← i;
  else start ← i+1;
return -1
```



Search (cont.)

Complexity of Sort Algorithms

- **Bubble Sort**
  - **2 Nested Loops**
  - **Execute outer loop n-1 times**
  - **For each outer loop iteration, inner loop runs i times.**
  - **Time complexity is proportional to:**
    $N-1 + N-2 + N-3 + \ldots + 1 = N^2/2 = O(N^2)$



- **Merge Sort**
  - **O(NlogN)**

Running Time

$> 10^{25}$ years

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

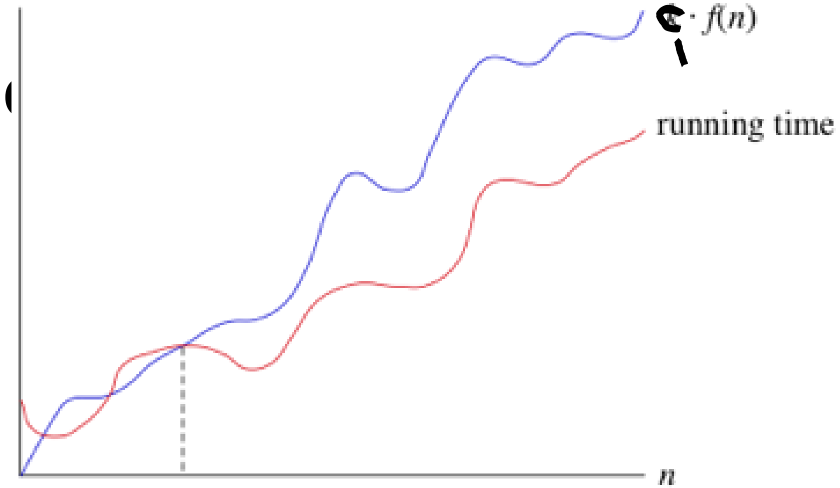
Asymptotic Upper Bounds

- **Example: A running time of $T(n) = n^2 + 4n + 2$ is usually too detailed. Rather, we're interested in how the runtime grows as the problem size grows**

$T(n) = O(f(n))$ if $\exists\, n_0 \geq 0$ and $c_1 > 0$

such that $T(n) \leq c_1 f(n) \quad \forall\, n \geq n_0$

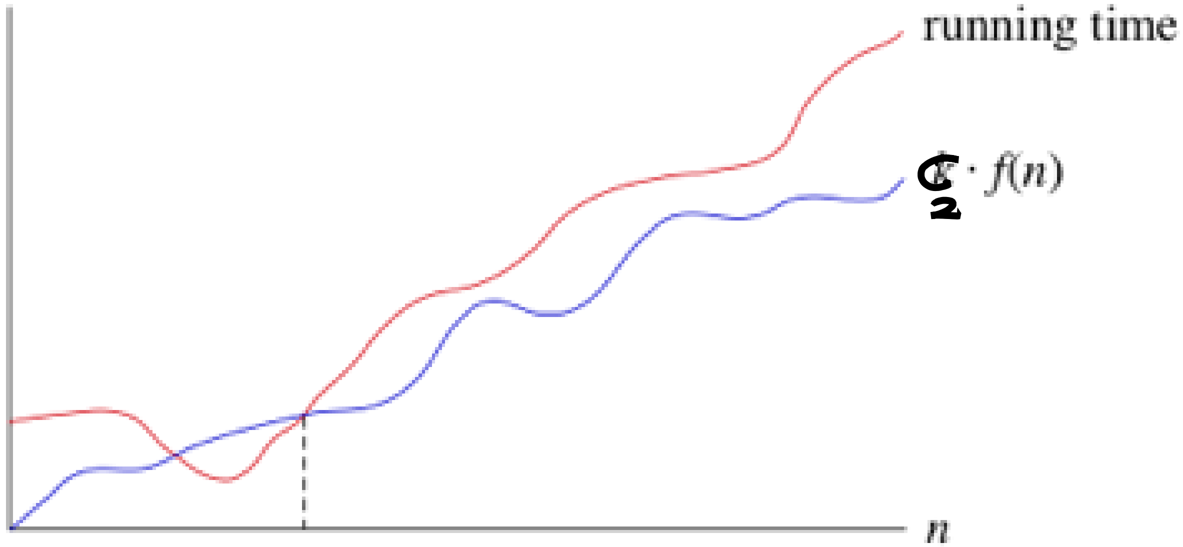
- **Question: Explain the ab**



Asymptotic Lower Bounds

$T(n) = \Omega(f(n))$ if $\exists\, n \geq 0$ and $c_2 > 0$

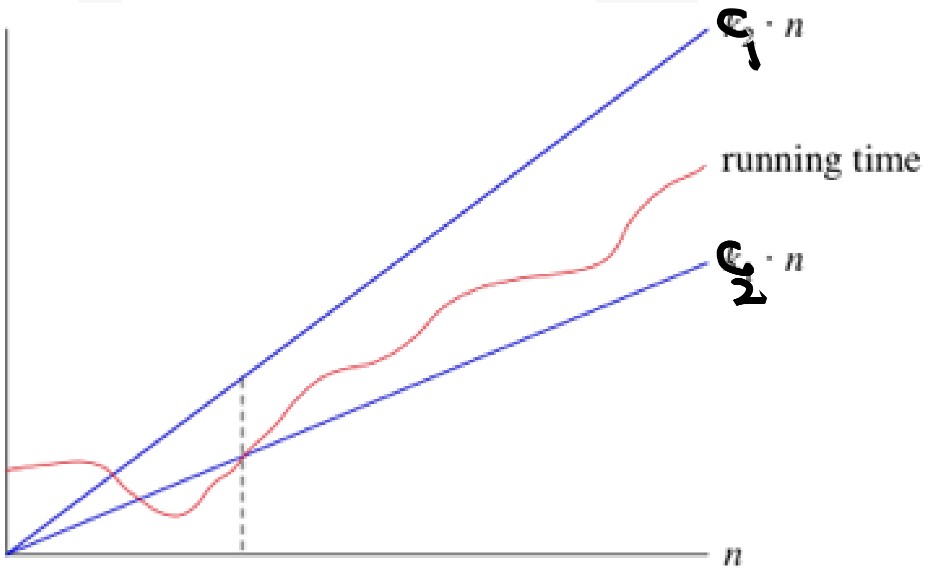
such that $T(n) \geq c_2 f(n) \;\; \forall\, n \geq n_0$



Tight Bounds

- If we know that T(n) is Θ(f(n)) then f(n) is the “right” asymptotic running time: it will run faster than O(f(n)) on all instances and some instances might take that long

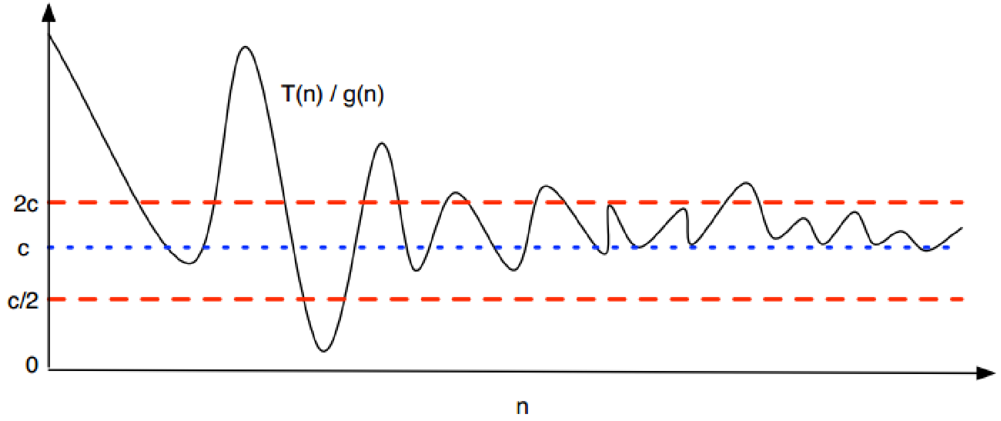
$T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$



Asymptotic Limit

Theorem

If $\lim_{n\to\infty} \frac{T(n)}{g(n)}$ *equals some* $c > 0$*, then* $T(n) = \Theta(g(n))$.



There is an n_0 such that $c/2 \leq T(n)/g(n) \leq 2c$ for all $n \geq n_0$.

Therefore, $T(n) \leq 2cg(n)$ for $n \geq n_0 \Rightarrow T(n) = O(g(n))$.

Also, $T(n) \geq \frac{c}{2}g(n)$ for $n \geq n_0 \Rightarrow T(n) = \Omega(g(n))$.

Runtime Calculation

$$T(n) = aT(n/b) + f(n)$$

Master Theorem

$T(n) = aT(n/b) + f(n)$ $\quad$ n : nonnegative int

$a \geq 1, b > 1$ $\quad$ constant $\varepsilon > 0$

f is asymptotically positive

case 1 : $f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$

case 2 : $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$

case 3 : $f(n) = \Omega(n^{\log_b a + \varepsilon})$

and if $af(n/b) \leq cf(n)$ for some constant $c < 1$

and large n $\Rightarrow T(n) = \Theta(f(n))$

Example 1

$$T(n) = 2T(n/2) + \Theta(n)$$

$a = 2, b = 2$

$f(n) = \Theta(n)$

$n^{\log_b a} = n$

$\Rightarrow$ case 2 $\Rightarrow$ $T(n) = \Theta(n \log n)$

Example 2

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

$$a = 1 \quad , b = 2$$

$$f(n) = ?$$

Example 2

$$T(n) = T(2n/3) + 1$$

$$a = 1 \quad b = 3/2$$

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

$$\Rightarrow \text{case } 2 \Rightarrow T(n) = \Theta(\log n)$$

Q: How about $T(n) = T(n/3) + 2$?

Polynomial Time

$\exists\, c_1 > 0,\ c_2 > 0$ such that $T(n) < c_1 n^{c_2}\ \forall n$

constants

Assume $n \rightarrow Kn$ $\quad K > 0$

$T(n) \rightarrow K^{c_2}\, T(n)$

Linear Time

- Linear time usually means you look at each element a constant number of times
- Example 1: Finding the maximum element in a list:

```
max = a[1]
  for i = 2 to n:
     if a[i] > max then
    set max = a[i]
```

Constant amount of work per input

- This does a constant amount of work per element in array a
- Example 2: Merging sorted lists

How about adding them to one list and sorting them!

S1
S2 → S

constant work per input?

Merging (cont.)

- **Example 2: Merging sorted lists**

$S_1 = \{n, n+2, n+4, \ldots, 3n-2\}$ n: even

$S_2 = \{1, 3, 5, \ldots 2n-1\}$

compare n to $1, 3, 5, \ldots, n/2-1, n/2+1$

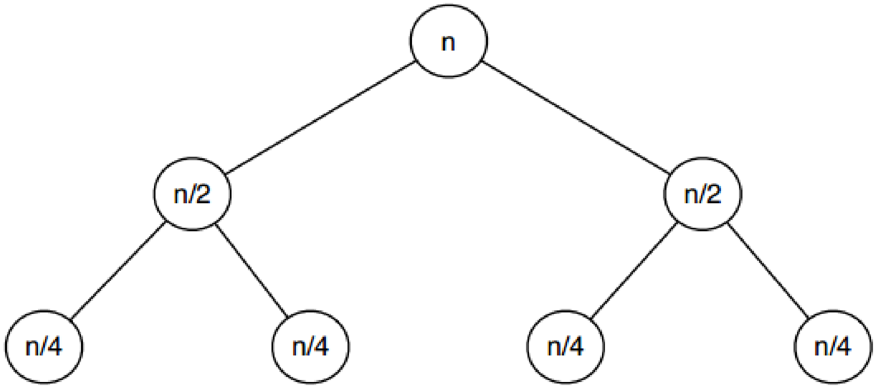
$\hookrightarrow O(n)$ comparisons per input

$\hookrightarrow O(n^2)$

O(n log n)

- O(n log n) time common because of sorting (often the slowest step in an algorithm)
- Example: Merge sort (based on divided and conquer concept)

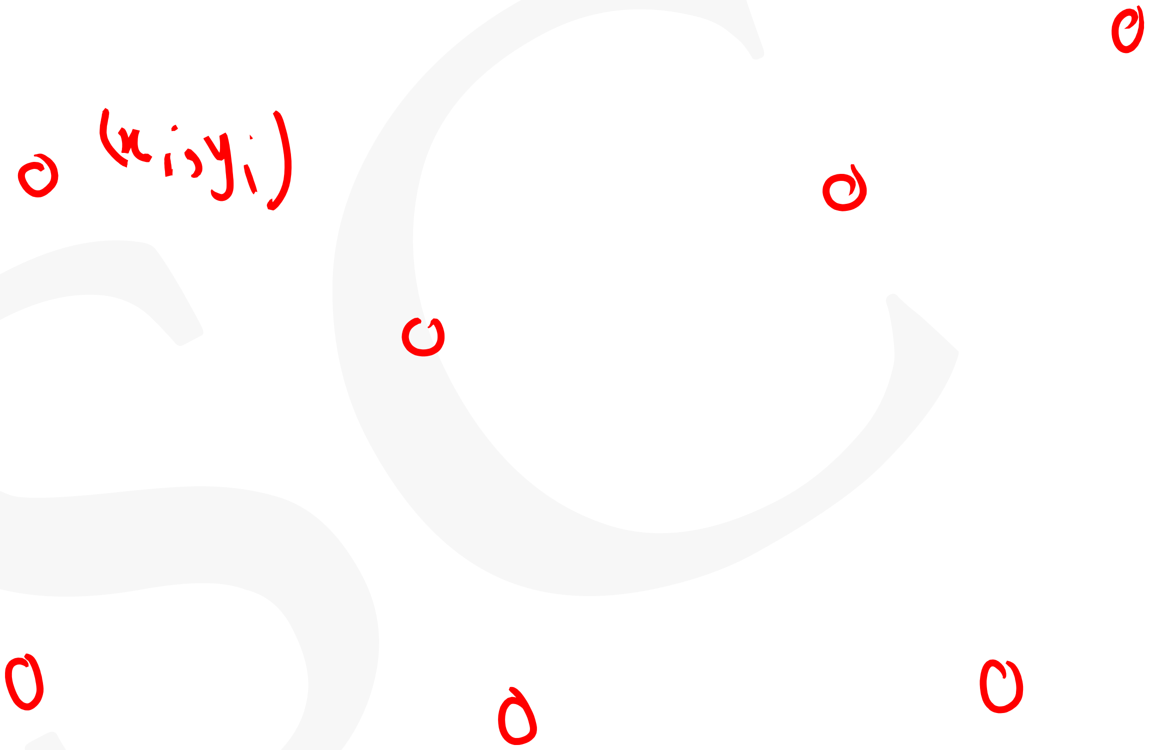
Where does the $O(n \log n)$ come from?



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

O(n^2)

Which two have the shortest distance?



$O\binom{n}{2}$

$O(n^3)$

- **Example:**
  - **Given n sets S1, S2, . . . , Sn that are subsets of {1, . . . , n}, is there some pair of sets that is disjoint?**

$O(n^k)$

A given graph of n nodes, find whether an independent set of size K

Question: How many K node sets are there?

Exponential Time

- What if we didn't limit ourselves to independent sets of size k, and instead want to find the largest independent set?
- Brute force algorithms search through all possibilities
- How many subsets of nodes are there in an n-node graph? 2^n
- What's the runtime of the brute force search for a largest independent set? $O(n^2 2^n)$

Sublinear Time

- Sublinear time means we don’t even look at every input
- Since it takes n time just to read the input, we have to work in a model where we count how many queries to the data we make
- Sublinear usually means we don’t have too look at every element
- Example: Binary search

Space Complexity

- **The total space taken by the algorithm with respect to the input size**
- **Auxiliary vs input**
- **Auxiliary space is the extra space or temporary space used by an algorithm**
- **Important note regarding interviews**
  - Example:

```
int square(int a)
{
return a*a;
}
```

Example

```
int sum(int A[], int n)
{
    int sum = 0, i;
        for(i = 0; i < n; i++)  sum = sum + A[i];
    return sum;
}
```

Space vs Time Tradeoff

- There can be a tradeoff between time and space complexities
  - As an engineer we have to decide which we should optimize for

- Example: Lookup table vs analytical forumulae to calculate the salary of employees as a function of their education level, # years of experience, and employee level

- Question: Which one is faster? Which one is more memory efficient?

Algorithms vs Heuristics

- **Algorithms**
  - **Well defined steps**
  - **Optimal solution**

- **Heuristics**
  - **Less predictable**
  - **Optimality not guaranteed**
  - **More popular in daily life**
  - **Examples:**
    - Educated guess
    - Draw a picture to understand the problem
    - If a solution cannot be found, work backward, i.e., assume you have a solution and try to derive properties from it