Algorithm and data structures analysis methods and practice

Comp Sci 1575 Data Structures

Missouri S&T Computer Science

"Simplicity is a great virtue, but it requires hard work to achieve it, and education to appreciate it. And to make matters worse: complexity sells better."
- Edsger Wybe Dijkstra
`https://en.wikipedia.org/wiki/Edsger_W._Dijkstra`

Outline

for() loops

How do we determine the order or growth rate of our code?

for() loops

- The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

```
sum = 0;

for(i = 1; i <= n; i++)
  sum += n;
```

- The first line is $\Theta(1)$.
- The for loop is repeated n times.
- The third line takes constant time, so the total cost for executing the two lines making up the for loop is $\Theta(n)$.
- Thus, the cost of the entire code fragment is also $\Theta(n)$.

Nested for() loops

- Analyze these inside out.
- Total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for( i = 0;  i < n; ++i )
  for( j = 0;  j < n; ++j )
    ++k;
```

$\Theta(n^2)$

Are double for loops always n^2?

Consecutive statements

- These just add (recall the sum rule)

```
for(i = 0; i < n; ++i)
  a[i] = 0;

for(i = 0; i < n; ++i)
  for(j = 0; j < n;++j)
    a[i] += a[j] + i + j;
```

$\Theta(n)$ followed by $\Theta(n^2)$, is just $\Theta(n^2)$

if/else

- Running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.
- Take greater complexity of then/else clauses

switch

- switch statement: Take complexity of most expensive case.

Recursion

Intuit the solution, or wait until taking algorithms to learn more, like:

- Substitution method
- Recursion-tree method
- Master method

Outline

Outline

Classifying functions
for() loops
Nested for() loops
Consecutive statements
if/else
switch
Recursion
Practice
Basics
Recursion
Logarithms
Logarithm review
Binary search
Euclid's algorithm
Exponentiation
Nested for loops always n^2 ?
Space complexity
Big picture
Data structures
Algorithms as technology
Problem versus algorithms

Assignment?


```
a = b;
```

$\Theta(?)$

Simple for loop?

Go line-by-line

```
sum = 0; // line 1?

for (i=1; i<=n; i++) // line 2?
  sum += n; // line 3?
```

$\Theta(?)$

Mess of for loops?

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum++;
for (k=0; k<n; k++)
   A[k] = k;
```

- Outer for loop is executed n times, but each time the cost of the inner loop is different with i increasing each time.
- During the first execution of the outer loop, $i = 1$.
- For the second execution of the outer loop, $i = 2$.
- Each time through the outer loop, i becomes one greater, until the last time through the loop when i = n:

$$\sum_{i=1}^{n} i = n(n-1)/2$$

$\Theta(?)$

Outline

Recursion

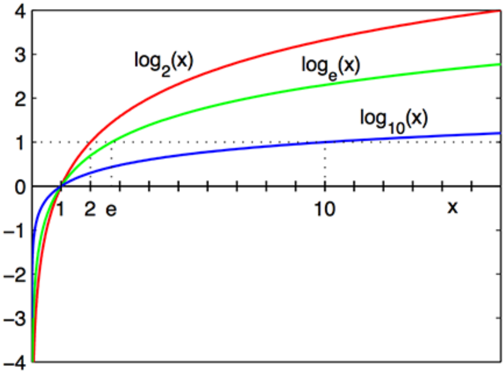
```
long fact(int n)
{
    if(n <= 1) return 1;
        return n * fact(n - 1);
}
```

- $T(n) = T(n-1) + 1$ for $n > 1$; $T(1) = 0$
- Which we can prove (later) is $T(n) = n - 1$

$\Theta(n)$

Outline

Log review: CS is mostly $\log_2 x$



- logarithm is the inverse operation to exponentiation
- $\log_b x = y$ and $b^y = x$ and $b^{\log_b x} = x$
- Example: $\log_2 64 = 6$ and $2^6 = 64$ and $2^{\log_2 64} = 64$
- $\log_2 x$ intersects x-axis at 1 and passes through the points with coordinates (2, 1), (4, 2), and (8, 3), e.g., $\log_2 8 = 3$ and $2^3 = 8$

Log review

	Formula	Example
product	$\log_b(xy) = \log_b(x) + \log_b(y)$	$\log_3(243) = \log_3(9 \cdot 27) = \log_3(9) + \log_3(27) = 2 + 3 = 5$
quotient	$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$	$\log_2(16) = \log_2\left(\frac{64}{4}\right) = \log_2(64) - \log_2(4) = 6 - 2 = 4$
power	$\log_b(x^p) = p\log_b(x)$	$\log_2(64) = \log_2(2^6) = 6\log_2(2) = 6$
root	$\log_b \sqrt[p]{x} = \frac{\log_b(x)}{p}$	$\log_{10}\sqrt{1000} = \frac{1}{2}\log_{10} 1000 = \frac{3}{2} = 1.5$

- $\log(nm) = \log n + \log m$
- $\log(\frac{n}{m}) = \log n - \log m$
- $\log(n^r) = r \log n$
- $\log_a n = \frac{\log_b n}{\log_b a}$ (base switch)

Log general rule for algorithm analysis

- Algorithm is in $O(\log n)$ if it takes constant, $O(1)$, time to cut the problem size by a fraction (which is usually $\frac{1}{2}$).
- If constant time is required to merely reduce the problem by a constant amount, such as to make the problem smaller by 1, then the algorithm is in $O(n)$
- Caveat: with input of n, an algorithm must take at least $\Omega(n)$ to read inputs. Thus, $O(\log n)$ classification often assumes input is pre-read.

Binary search

```
// Return pos of element in sorted array "A" of
// size n with value K, or -1 if not found
int binary(int A[], int n, int K){
  int low = 0;
  int high = n - 1;
  while(low <= high){
    int mid = (low + high)/2; # int div
    if(K > A[mid]) low = mid + 1;
    if(K < A[mid]) high = mid - 1;
    else return mid;
  }
return -1;
}
```

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

Binary search: first approach

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

- Inside the loop takes $\Theta(c)$
- Loop starts with $high - low = n - 1$ and finishes with $high - low \leq -1$.
- Each iteration, $high - low$ must be at least halved from its previous value
- Number of iterations is at most $log(n-1)+2$
- For example, if $high - low = 128$, then the maximum values of $high - low$ after each iteration are $64, 32, 16, 8, 4, 2, 1, 0, -1$

$\Theta(\log n)$

Binary search: second approach

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Key	11	13	21	26	29	36	40	41	45	51	54	56	65	72	77	83

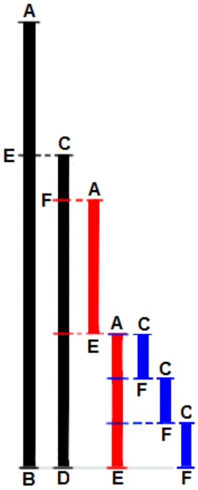
Though not a recurrent program, we can use a recurrent definition to calculate running time.

- $T(n) = T(n/2) + 1$ for $n > 1$; $T(1) = 1$
- Which is $T(n) = \log n$

$\Theta(\log n)$

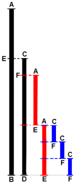
Euclid's algorithm

Euclid's algorithm efficiently computes the greatest common divisor (GCD) of two numbers (AB and CD below), the largest number that divides both without leaving a remainder (CF).



Proceeding left to right:

Euclid's algorithm




```
long long gcd(long long ab, long long cd){
  while(cd != 0){
    long long rem = ab % cd;
    ab = cd;
    cd = rem;
  }
  return ab;
}
```

- After 2 iterations, rem is at most half its original value
- If $ab > cd$, then $ab \% cd < ab/2$.
- Thus, number of iterations is at most $2 \log n = \Theta(\log n)$

Exponentiation

Compute b^n in how many multiplications?

```
// With a loop:
double simpleExpLoop(int b, int n){
  int result = 1;
  for(int c = 1; c <= n; c++){
    result *= b;
  }
  return result;
};

// or even recursively:
double simpleExpRecur(int b, int n){
  if(n == 0)  return 1;
  else return b * simpleExpRecur(b, n - 1);
}
```

$\Theta(?)$

Efficient exponentiation

To obtain b^n, do recursively:

- if n is even, do $b^{n/2} * b^{n/2}$
- if n is odd, do $b * b^{n/2} * b^{n/2}$
- with base case, $b^1 = b$

Note: $n/2$ is integer division

What is b^{62} ?

1. $b^{62} = (b^{31})^2$
2. $b^{31} = b(b^{15})^2$
3. $b^{15} = b(b^7)^2$
4. $b^7 = b(b^3)^2$
5. $b^3 = b(b^1)^2$
6. $b^1 = b$

What is b^{61} ?

1. $b^{61} = b(b^{30})^2$
2. $b^{30} = (b^{15})^2$
3. $b^{15} = b(b^7)^2$
4. $b^7 = b(b^3)^2$
5. $b^3 = b(b^1)^2$
6. $b^1 = b$

How many multiplications when counting from the bottom up?

Efficient exponentiation

To obtain b^n, do recursively:
if n is even, do $b^{n/2} * b^{n/2}$
if n is odd, do $b * b^{n/2} * b^{n/2}$

```
long long pow(long long x, int n){
    if(n == 0)
        return 1;
    if(n == 1)
        return x;
    if(isEven(n))
        return pow(x * x, n / 2);
    else
        return pow(x * x, n / 2) * x;
}
```

- Number of multiplications is at most $2 \log n$ and thus is in $\Theta(\log n)$

Nested for loops always n^2 ?


```
sum1 = 0;
for (k=1; k<=n; k*=2) // Do log n times
  for (j=1; j<=n; j++) // Do n times
   sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2) // Do log n times
  for (j=1; j<=k; j++) // Do k times
    sum2++;
```

- First outer for loop executed $\log n + 1$ times; on each iteration k is multiplied by 2 until it reaches n
  Inner loop is always n
  First block is $\sum_{i=0}^{\log n} n$, which is $\Theta(n \log n)$
- Second outer loop is $\log n + 1$
  Second inner loop, k, doubles each iteration
  Second block, with $k = 2^i$ yields $\sum_{i=0}^{\log n} 2^i$ which is $\Theta(n)$

$\Theta(?)$

Outline

Space complexity

Whereas for a single data structure, different operations can have different efficiencies, space requirements usually apply to the whole data structure itself. For example:

- What are the space requirements for an array of of n integers?
- To define binary connectivity between all elements with all other elements, we can use a fully connected matrix:

	a	b	c	d	e	f	g	h
a		•	•			•		
b	•			•		•		
c	•				•		•	•
d	•	•				•		
e			•				•	•
f	•	•		•				
g			•		•			•
h			•		•		•	

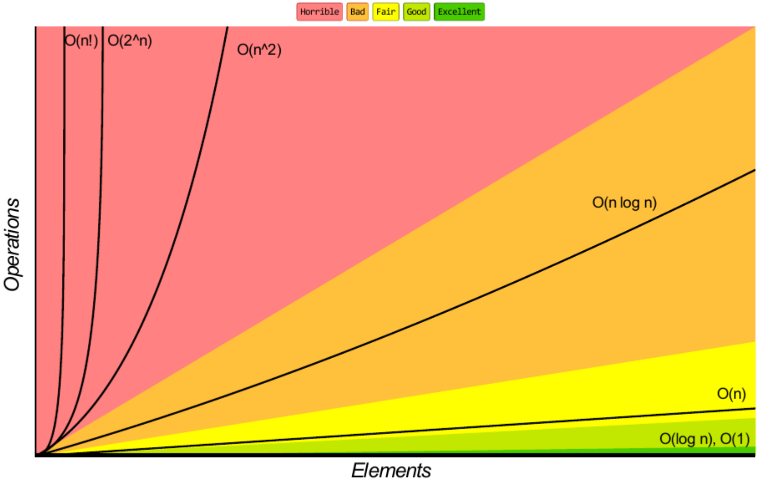
$\leftarrow$ dots imply connectivity

$\Theta(?)$
and can we do better for this kind of binary connectivity?

Outline

Outline

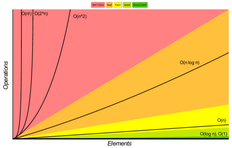
Data structures



Data structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Color key:



Classifying functions
for() loops
Nested for() loops
Consecutive statements
if/else
switch
Recursion
Practice
Basics
Recursion
Logarithms
Logarithm review
Binary search
Euclid's algorithm
Exponentiation
Nested for loops always n^2 ?
Space complexity
Big picture
Data structures
Algorithms as technology
Problem versus algorithms

Array sorting algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Outline

Algorithms as technology

Would you rather have a faster algorithm or a faster computer?

f(n)	**n**	**n′**	**Change**	**n′/n**
$10n$	1000	10,000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

Growth rate | old computer | 10x faster computer | Δ | ratio
This is a better key ↑

Outline

Problem complexity versus algorithm analysis

- **Analysis of algorithms** applies to **particular solutions** to **problems**, which themselves have **complexities** defined by the **entire set of their solutions**.
- Problem: E.g., what is the least possible cost for any sorting algorithm in the worst case?
  - Any algorithm must at least look at every element in the input, to determine that input is sorted, which would be be cn with $\Omega(n)$ lower bound.
  - Further, we can prove that any sorting algorithm must have running time in $\Omega(n \log n)$ in the worst case