Lecture 2: Efficiency, Analysis, and Order

CS3310

Math Refresher

Logarithms

- $\log_{10} 1000 = ?$

Math Refresher

Logarithms

- A *logarithm* (usually abbreviated *log*) is the inverse of exponentiation.
  - $\log_{10} 1000 = 3$
  - The log of a number is the exponent to which another number must be raised to produce that number.
    - In the above logarithmic equation, we call 10 the *base*.
  - $\log_{10} 1000 = 3$ is the inverse of $10^3 = 1000$

Math Refresher

Logarithms

- In algorithm analysis, we frequently see logarithms with a base of 2. This is called a *binary logarithm*
- Example:
  - $\log_2 64 = ?$

Math Refresher

Logarithms

- In algorithm analysis, we frequently see logarithms with a base of 2. This is called a *binary logarithm*
- Example:
  - $\log_2 64 = 6$, since $2^6 = 64$
- Binary logarithms are common, so we have a simpler notation for them:
  - **$\lg 64 = 6$**
- $\lg x = y$ is equivalent to $\log_2 x = y$
  - i.e. $2^y = x$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = \ ?$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
  - This is ***always*** true for every b, since $b^0 = 1$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
- $x^{\lg y} = ?$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
- $x^{\lg y} = y^{\lg x}$
  - Suppose $x = 2$ and $y = 4$.
  - We have $2^{\lg 4} = 4^{\lg 2}$, or $2^2 = 4^1$, or $\mathbf{4 = 4}$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
- $x^{\lg y} = y^{\lg x}$
- $(\lg n)^2 = ?$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
- $x^{\lg y} = y^{\lg x}$
- $(\lg n)^2 = \lg n * \lg n$

Math Refresher

Other Important Logarithm Facts

- $\log_b 1 = 0$
- $x^{\lg y} = y^{\lg x}$
- $(\lg n)^2 = \lg n * \lg n$
- $\lg a + \lg b = \lg(a * b)$
- $\log_b x^a = a * \log_b x$
  - i.e. $\lg n^2 = 2\lg n$ (since $\lg n^2 = \log_2 n^2$)

Math Refresher

$$\sum_{i=1}^{n} i$$

What does this notation mean?

Math Refresher

$$\sum_{i=1}^{n} i$$

What does this notation mean?

- The sum of the first n positive integers.
- If $n = 100$, this notation represents:
  $1 + 2 + \ldots + 100$

Math Refresher

Equation for the sum of the first n numbers, starting at 1:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Equation for the geometric sum:

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}$$

Algorithm Efficiency

- When writing an algorithm, it's important to verify it solves a given problem.
    - However, we are also interested in how *efficiently* it solves the problem.

- Why do we care about efficiency? Computers are getting faster!
    - As we will see, algorithms can be so inefficient that they would take years or **centuries** to complete, even on superfast computers!
        - Regardless of how fast computers get, these algorithms are still unviable.

- Therefore, when we measure efficiency, we don't calculate how fast an algorithm will run on a specific computer.
    - Instead, we measure how efficient an algorithm is in relation to other algorithms.
    - i.e. we might say Algorithm A is 100 times less efficient than Algorithm B

Algorithm Efficiency

Problem: is the key x in a sorted array S of n keys?

Parameters: positive integer n, array of keys S indexed from 1 to n, a key x

Outputs: The location of x in S, or 0 if x is not in S

There are many ways to solve this problem. Two well-known ways:

Sequential Search

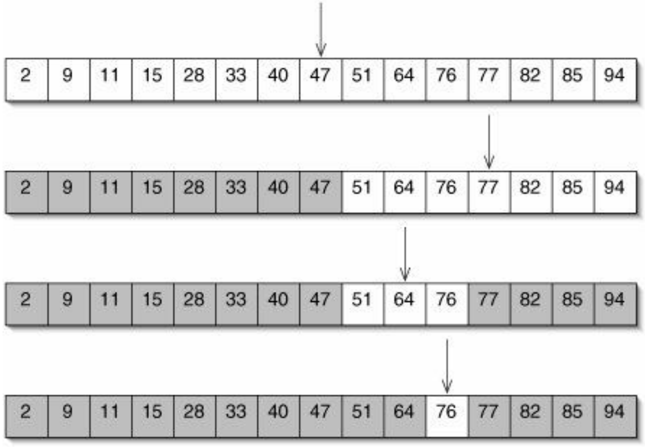
Start at the beginning of the array and check each index until the key is found or the end of the array is reached.

Binary Search

Start in the middle of the array. If the key we are looking for is less than the middle value, binary search the left-half of the array. Otherwise, binary search the right-half of the array.

Algorithm Efficiency

Binary Search



Binary search is an example of a **divide-and-conquer** algorithm.

- In this example, binary search finds 76 in 4 steps.
- Sequential search requires 11 steps.

Algorithm Efficiency

- Binary search eliminates half of the remaining values each step.
  - Sequential search checks each item, only eliminating one value each step.
- Binary search, in most cases, is significantly more efficient than sequential search.
- If a computer performs one operation per nanosecond, Sequential Search would take 4 seconds with an input of 4 billion. Binary Search would be instantaneous!

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

Algorithm Efficiency

While 4 seconds is slow, it might seem somewhat reasonable. However, some algorithms are so inefficient they would never finish in a lifetime:

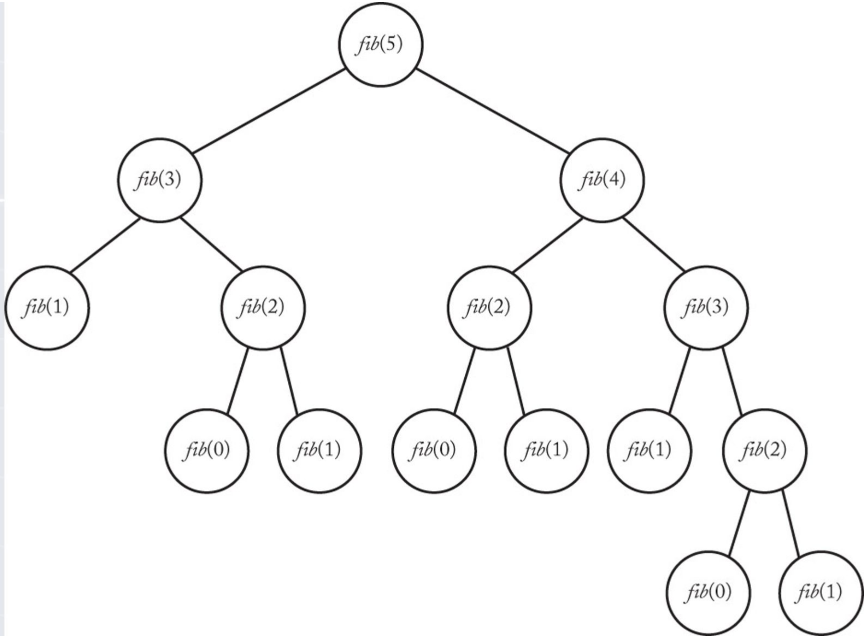
Problem: Calculate a fibonacci number
Parameters: positive integer *n*
Outputs: Return the *n*th fibonacci number

One algorithm to solve this problem is the following recursive one:

```
int fib (int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Algorithm Efficiency

- When calculating fib(5), we recalculate several values multiple times.



n	Recursive Calls
0	1
1	1
2	3
4	9
5	15
6	25

The algorithm's growth is *exponential*. If n = 200, the result would take 40,000 years to compute! This is clearly unviable, regardless of how fast computers get!

Algorithm Efficiency

There are usually many ways to solve a problem, and we are always searching for better ones. Just because a solution exists doesn't mean we stop there!

Let's consider three different solutions to the problem of finding the greatest common divisor of two integers.

Problem: What is the gcd of two numbers, m and n?
Parameters: Positive int m, positive int n
Outputs: The gcd of m and n

GCD

Algorithm 1:

1. Find all prime factors of m and n
2. Identify all common prime factors of both.
3. Multiply these common factors

$60 = \underline{2} \times \underline{2} \times \underline{3} \times 5$

$24 = \underline{2} \times \underline{2} \times 2 \times \underline{3}$

Therefore, gcd $= 2 \times 2 \times 3 = 12$

This algorithm, while easy to understand, is very difficult when m and n become big

GCD

Algorithm 2:

```
int gcd(int m, int n)
{
     int t = min(m, n);
     while (t > 0)
          if (m mod t == 0) and (n mod t == 0)
               return t;
          t--;
}
```

How does this algorithm find the gcd of m and n when $m = 60$ and $n = 24$?

GCD

Algorithm 2:

```
int gcd(int m, int n)
{
    int t = min(m, n);    // min of 60 and 24 is 24 so t = 24
    while (t > 0)
        if (m mod t == 0) and (n mod t == 0)
            return t;
        t--;
}
```

$t = 24$

60 mod 24 = 12, decrement t

$t = 23$

60 mod 23 = 14, decrement t

This process continues until $t = 12$

GCD

Algorithm 2:

```
int gcd(int m, int n)
{
    int t = min(m, n);
    while (t > 0)
        if (m mod t == 0) and (n mod t == 0)
            return t;
        t--;
}
```

The loop iterates 13 times to find gcd(60, 24). This is horribly inefficient if *m* and *n* are big and the gcd is small.

GCD

Algorithm 3:

```
int euclid(int m, int n)
{
    while (n != 0)
        int r = m mod n;
        m = n;
        n = r;
    return m;
}
```

The `while` only loop iterates <u>twice</u> to find gcd(60, 24)

m	n	r
60	24	12
24	12	0
12	0	

Analysis of Algorithms

- When analyzing performance, we are mostly concerned with how an algorithm *grows*
  - i.e. as the input size to an algorithm increases, how is performance affected?

Algorithm a: $x = x + y$;
Algorithm b: **for** i **from** 1 **to** n **do**
$x = x + y$;

- Regardless of what machine we use, ***b*** takes n times as long as ***a***
  - If $n = 1$, both take the same amount of time: only one operation is performed.
  - If $n = 100$, ***b*** loops 100 times while ***a*** still performs a single operation
    - In this case, ***b*** takes 100 times as long as ***a***
- Thus, we say that ***a*** performs 1 operation and ***b*** performs n operations.

Analysis of Algorithms

- We first pick an instruction (or group of instructions) such that the amount of work done by the algorithm is roughly proportional to the # of times this instruction is performed.
  - We call this instruction the **basic operation**.
  - Picking the basic operation can require a bit of intuition.

- We want the basic operation to be independent of programmer and language, so we don’t use overhead instructions (incrementing loop variables, setting pointers, etc)

- The # of times the basic operation is performed is the **frequency count.** This often grows larger as the **input size** increases.
  - i.e. a bigger array in sequential search increases the frequency count.

Analysis of Algorithms

There are several measures of complexity for algorithms.

- Some algorithms always have the same frequency count for every instance of size n.
  - In this case, we determine $T(n)$: Every-case time complexity.
- Some algorithms have different frequency counts for different instances of size n.
  - In this case, we determine the following:
    - $W(n)$: Worst-case time complexity
    - $A(n)$: Average-case time complexity
    - $B(n)$: Best-case time complexity

Analysis of Algorithms

```
number addArrayItems(int n, const number S[])
{
     index i;
     number result;

     result = 0;
     for (i = 1; i <= n; i++)
          result = result + S[i]
     return result;
}
```

Basic Operation:

Analysis of Algorithms

```
number addArrayItems(int n, const number S[])
{
      index i;
      number result;

      result = 0;
      for (i = 1; i <= n; i++)
            result = result + S[i]
      return result;
}
```

Basic Operation: the addition within the loop: `result = result + S[i]`

Does this algorithm always have the same frequency count for every instance of size *n*?

Analysis of Algorithms

```
number addArrayItems(int n, const number S[])
{
     index i;
     number result;

     result = 0;
     for (i = 1; i <= n; i++)
          result = result + S[i]
     return result;
}
```

Basic Operation: the addition within the loop: `result = result + S[i]`

Does this algorithm always have the same frequency count for every instance of size n?

Yes! The loop <u>always</u> iterates from 1 to n, or n times.

T(*n*): ?   (every-case time complexity)

Analysis of Algorithms

```
number addArrayItems(int n, const number S[])
{
     index i;
     number result;

     result = 0;
     for (i = 1; i <= n; i++)
          result = result + S[i]
     return result;
}
```

Basic Operation: the addition within the loop: `result = result + S[i]`

Does this algorithm always have the same frequency count for every instance of size *n*?

Yes! The loop always iterates from 1 to *n*, or *n* times.

T(*n*): *n* iterations means the basic operation runs ***n*** times.

Analysis of Algorithms

```
void seqsearch(int n, const keytype S[], keytype x, index& location)
     location = 1;
     while (location <= n && S[location] != x)
          location ++;
     if (location > n)
          location = 0;
```

Basic Operation:

Analysis of Algorithms

```
void seqsearch(int n, const keytype S[], keytype x, index& location)
     location = 1;
     while (location <= n && S[location] != x)
          location ++;
     if (location > n)
          location = 0;
```

Basic Operation: `S[location] != x`

Does this algorithm always have the same frequency count for every instance of size *n*?

Analysis of Algorithms

```
void seqsearch(int n, const keytype S[], keytype x, index& location)
     location = 1;
     while (location <= n && S[location] != x)
          location ++;
     if (location > n)
          location = 0;
```

Basic Operation: `S[location] != x`

Does this algorithm always have the same frequency count for every instance of size *n*?

No! Since we might break out of the `while` loop before `location == n`, frequency count may differ.

Therefore, we can determine worst-case, best-case, and average-case complexities but **not** every-time case complexity.

Sequential Search Analysis

Worst-case time complexity W(n):

- What is the worst scenario that can happen with sequential search?

Sequential Search Analysis

Worst-case time complexity W(n):

- What is the worst scenario that can happen with sequential search?
  - x is the last item or it's not in the array at all
- How many times does the basic operation occur in this case?

Sequential Search Analysis

Worst-case time complexity W(n):

- What is the worst scenario that can happen with sequential search?
  - x is the last item or it's not in the array at all
- How many times does the basic operation occur in this case?
  - n times, since the `while` loop iterates until `location == n`

Sequential Search Analysis

Worst-case time complexity W(n):

- What is the worst scenario that can happen with sequential search?
  - x is the last item or it's not in the array at all
- How many times does the basic operation occur in this case?
  - n times, since the `while` loop iterates until `location == n`

Best-case time complexity B(n):

- What is the best scenario that can happen with sequential search?

Sequential Search Analysis

Worst-case time complexity W(n):

- What is the worst scenario that can happen with sequential search?
  - x is the last item or it's not in the array at all
- How many times does the basic operation occur in this case?
  - n times, since the `while` loop iterates until `location == n`

Best-case time complexity B(n):

- What is the best scenario that can happen with sequential search?
  - x is the first item
- How many times does the basic operation occur in this case?

Sequential Search Analysis

Worst-case time complexity $W(n)$:

- What is the worst scenario that can happen with sequential search?
  - x is the last item or it's not in the array at all
- How many times does the basic operation occur in this case?
  - n times, since the `while` loop iterates until `location == n`

Best-case time complexity $B(n)$:

- What is the best scenario that can happen with sequential search?
  - x is the first item
- How many times does the basic operation occur in this case?
  - Only 1 time, since we break out of the while loop on the first iteration.

$W(n) = n$, $B(n) = 1$

Sequential Search Analysis

Average-case time complexity A(n)

Average case is a little more difficult than the other two

There are 2 situations to consider:

1. We know x is always in the list
2. We don't know if x is in the list.

Sequential Search Analysis

Average-case time complexity A(n) case 1: x is always in the list.

- We could assign a probability to each index, indicating how likely it is x is in it
- However, to simplify, we assign equal probability to each index: $\frac{1}{n}$
- We then add up the odds that n is in each index:

$$A(n) = \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \cdots + \frac{1}{n} \cdot n \quad = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

On average we search about half of the list.

Sequential Search Analysis

Average-case time complexity A(n) case 2: x is not necessarily in the list.

- This is a little more difficult.
- We assume prob(x in array) = p
- prob(x not in array) = 1 - p and it takes n comparisons to know x is not in the array
- We then add up the odds that x is in each index:

$$A(n) = \frac{p}{n} \cdot \frac{n(n+1)}{2} + (1-p) \cdot n \quad = n \cdot (1 - \frac{p}{2}) + \frac{p}{2}$$

If $p = \frac{1}{2}$, $A(n) = \frac{3}{4} n + \frac{1}{4}$

About ¾ of the list is searched on average

Sequential Search Analysis

Now, the good news:

- We will usually not calculate average-case time complexity in this course!
- Although average-case is more descriptive than worst or best, it is much more complicated to calculate.
- It's important to know that average-case exists and that it should be used sometimes, but usually worst-case time complexity is informative enough.

Exchange Sort Analysis

```
void exchangesort (int n, keytype S[])
     index i, j;
     for (i = 1; i <= n; i++)
          for (j = i + 1; j <= n; j++)
               if (S[j] < S[i])
                    exchange S[i] and S[j];
```

Basic Operation:

Exchange Sort Analysis

```
void exchangesort (int n, keytype S[])
     index i, j;
     for (i = 1; i <= n; i++)
          for (j = i + 1; j <= n; j++)
               if (S[j] < S[i])
                    exchange S[i] and S[j];
```

Basic Operation: The comparison of S[j] and S[i] (we can consider the exchange an overhead)

Will this algorithm have the same frequency count for every instance of size n?

Exchange Sort Analysis

```
void exchangesort (int n, keytype S[])
     index i, j;
     for (i = 1; i <= n; i++)
          for (j = i + 1; j <= n; j++)
               if (S[j] < S[i])
                    exchange S[i] and S[j];
```

Basic Operation: The comparison of S[j] and S[i] (we can consider the exchange an overhead)

Will this algorithm have the same frequency count for every instance of size n? **Yes**!

- The outer loop iterates n times: in its first iteration, the inner loop iterates (n - 1) times. In its second iteration, the inner loop iterates (n - 2) times. In its final iteration the inner loop iterates once.

What is t(n)?

Exchange Sort Analysis

- The outer loop iterates n times: in its first iteration, the inner loop iterates $(n - 1)$ times. In its second iteration, the inner loop iterates $(n - 2)$ times. In its final iteration the inner loop iterates once.

$$\mathbf{T(\boldsymbol{n})} = (n - 1) + (n - 2) + (n - 3) + \ldots + 1 = \mathbf{[(\boldsymbol{n} - 1)\boldsymbol{n}] / 2}$$

Note: Since we are finding the sum of the first $n - 1$ integers rather than the first n integers, we have $[(n - 1)n] / 2$ instead of $[n(n + 1)] / 2$

In-Class Exercise

1. How many times faster is gcd(31415, 14142) by Euclid's algorithm than by algorithm 2 provided in the slides? (Use a calculator!) Compare the two based on the frequency count of the total number of mods performed.

2. What is the time complexity of the Matrix Multiplication algorithm?

```
void MatrixMult(int n, const number A[][], const number B[][], number C[][])
{
      index i, j, k;

      for (i = 1; i <= n; i++)
           for (j = 1; j <= n; j++)
                C[i][j] = 0;
                for (k = 1; k <= n; k++)
                     C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
```

In-Class Exercise

3. Consider the following algorithm for finding the distance between the two closest numbers in an array. (Note: the distance between two numbers a and b is measured by $|a - b|$) Make as many improvements as you can:

```
void MinDistance(int n, A[])
{
    minDist = ∞;
    for (i = 0 to n - 1)
        for (j = 0 to n - 1)
            if (i != j) and (|A[i] - A[j]| < minDist)
                minDist = |A[i] - A[j]|
    return minDist;
}
```