CS24 Week 6 Lecture 2

Kyle Dewey

Overview

- More complexity analysis
- Recursion

More Complexity Analysis

Measuring Efficiency

- How might we determine the efficiency of a program?
  - Benchmarks tend to be too specific (new hardware? How big of inputs do we test?)
  - Better approach: define a formula in terms of the input size

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done once. Call this c_1.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done once. Call this c_2.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done `length` times. Call this c_3.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done `length` times. Call this c_4.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done `length` times. Call this `c5`.

Another Example

```
int sum2(int* arr, int length) {
   int s = 0, x, y;
   for (x = 0; x < length; x++) {
      for (y = 0; y < length; y++) {
         s += arr[x] + arr[y];
      }
   }
   return s;
}
```

Constant time, done `length * length` times.

Call this c_6.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done `length * length` times.

Call this `c7`.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done `length * length` times.

Call this c_8.

Another Example

```
int sum2(int* arr, int length) {
  int s = 0, x, y;
  for (x = 0; x < length; x++) {
    for (y = 0; y < length; y++) {
      s += arr[x] + arr[y];
    }
  }
  return s;
}
```

Constant time, done once. Call this c_9.

Putting it Together

- We are left with the following formula:

$$c_1 + c_2 + (\text{length} * c_3) + (\text{length} * c_4) + (\text{length} * c_5) + (\text{length} * \text{length} * c_6) + (\text{length} * \text{length} * c_7) + (\text{length} * \text{length} * c_8) + c_9$$

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

$$c_1 + c_2 + (\texttt{length} * c_3) + (\texttt{length} * c_4) + (\texttt{length} * c_5) + (\texttt{length} * \texttt{length} * c_6) + (\texttt{length} * \texttt{length} * c_7) + (\texttt{length} * \texttt{length} * c_8) + c_9$$

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

```
1 + 1 + (length * 1) + (length * 1) +
(length * 1) + (length * length * 1) +
(length * length * 1) +
(length * length * 1) + 1
```

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

```
3 + length + length + length + (length *
    length) + (length * length) +
          (length * length)
```

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

```
3 + 3(length) + 3(length * length)
```

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

```
1+ length + (length * length)
```

Putting it Together

- The specific values of constants are unimportant as long as they are positive
- We can replace all these with the value 1 as far as Big O notation is concerned

```
1+ length + length²
```

Putting it Together

- With sums, we always choose the larger sum
- A variable is always larger than a constant

$$\texttt{1+ length + length}^2$$

Putting it Together

- With sums, we always choose the larger sum
- A variable is always larger than a constant

$$\texttt{length} + \texttt{length}^2$$

Putting it Together

- With sums, we always choose the larger sum
- A variable is always larger than a constant

`length`^{2}

Putting it Together

- Observe that `length` is really N, the input size
- For this example, we are done

$\texttt{length}^2$

Putting it Together

- Observe that `length` is really N, the input size
- For this example, we are done

$O(N^2)$

Big O Heuristics

- A non-loop is often `O(1)`
- A single loop is often `O(N)`
- A singly nested loop is often $O(N^2)$
- Not always true though - we will see exceptions later in this class
  - Determining time complexity can be quite difficult in general

Recursion

Motivation

- A *lot* of problems are defined in terms of themselves (recursive)
  - You're already familiar with a lot!
- These demand solutions which are themselves recursive

Recursion

- Defining a problem in terms of:
  - Some simple trivial case
  - A more complex case which ultimately leads to the trivial case
- A way to define a problem in terms of itself

Trivial Case

- Often called the “base” case
- It represents a simple form of the problem

Recursive Case

- Defines problem in terms of itself
- Recursive cases should ultimately lead to base cases

My Two Cents on Recursion

- Phrased as a problem strictly with numbers, this seems magical and unintuitive
- Phrased as a problem over data structures, this makes more sense
  - Data structures themselves can have recursive structure
  - You’re been familiar with recursive data structures, for **many, many years**

Example: Arithmetic Expressions

```
1

1 + 1

1 * 1

1 + (1 + 1)

(1 * 1) + 1

(1 * (1 + 1)) - 1
```

Example: Arithmetic Expressions

```
n is an Integer
e is an Expression
op is an Operator

op ::= + | - | * | /
e ::= n | e1 op e2
```

```
1
1 + 1
(1 + 1) + (1 * 1)
```

Example: Arithmetic Expressions

```
n is an Integer
e is an Expression
op is an Operator

op ::= + | - | * | /
e ::= n | e1 op e2
```

Base case?
Recursive case?

```
1
1 + 1
(1 + 1) + (1 * 1)
```

Example: Arithmetic Expressions

```
n is an Integer
e is an Expression
op is an Operator
```

$$op ::= + \mid - \mid * \mid /$$

$$e ::= n \mid e_1 \; op \; e_2$$

Base

Recursive

```
1
1 + 1
(1 + 1) + (1 * 1)
```

Example: Arithmetic Expression Evaluation

- A number evaluates to itself
- To evaluate an operation (e_1 op e_2):
  - Evaluate e_1 to a number n_1
  - Evaluate e_2 to a number n_2
  - Evaluate n_1 op n_2

Example: Natural Language

- It is possible to take the majority of most natural languages and express them in a way that is similar to our arithmetic expression representation
- A clause containing another clause...

Example: Programming Languages

- Most programming languages work this way, too
- `if`s can be nested in `if`s...
  - At some point, we have to stop nesting the `if`s, or else we won't have a program

Example: Linked Lists

- A linked list is either:
  - An empty list
  - A node holding an item (`int` below) and a pointer to another list

```
List = Empty | int List
```

Relationship to Operations

- The recursive structure of applicable data structures often mirrors the recursive structure of operations on those data structures
- Which cases might be interesting for a linked list?

Relationship to Operations

- The recursive structure of applicable data structures often mirrors the recursive structure of operations on those data structures
- Which cases might be interesting for a linked list?
  - Empty list (e.g., `NULL`)
  - Non-empty list (a node)

Example Problem

- Say we want to calculate the length of a linked list recursively
- A list is represented as a `Node*`
  - Base case?
  - Length of `list` besides first element?
  - Recursive case?

```
int length(Node* list);
```

Example Problem

```
int length(Node* list) {
  if (list == NULL) {
    return 0; // base case
  } else {
    return (1 + // this node’s length
            // length of the rest of
            // the list
            length(list->getNext()));
  }
}
```

Revised Problem

- Say we want to determine the length of a list, but with a tweak: we also take the length of the list so far
  - Base case?
  - Length of `list` besides first element?
  - Recursive case?
- What does the initial call look like?

```
int firstCall(Node* list);
int length2(Node* list, int soFar);
```

```
int length2(Node* list, int soFar) {
  if (list == NULL) {
    return soFar; // base case
  } else {
    // get the length of the rest of
    // the list, and say that the
    // length so far is + 1
    return length2(list->getNext(),
                   soFar + 1);
  }
}

int firstCall(Node* list) {
  return length2(list, 0);
}
```

Relationship to Loops

- `length2` is more similar to an iterative implementation than it may seem at first
  - `while` dynamically inserts `if`s as many times as needed
  - Recursion dynamically inserts the body of a function as many times as needed
- After doing these expansions, they basically look the same!

Recursion With Arrays

Recursion With Arrays

- If we look at arrays in a similar way as linked lists, operations become more clear
- The index acts like a pointer to a particular node
  - What is the base case?
  - Recursive case?

Recursion With Arrays

- If we look at arrays in a similar way as linked lists, operations become more clear
- The index acts like a pointer to a particular node
  - What is the base case?
    - Index out of array
  - Recursive case?
    - Index in array

Example

- Determine the sum of an array of integers, starting from a particular index. An array containing no elements has a sum of 0.
  - Base case?
  - Recursive case?

```
int sumFromIndex(int* array,
                 int length,
                 int index);
```

Example

- Determine the sum of an array of integers, starting from a particular index. An array containing no elements has a sum of 0.
  - Base case? - index out of bounds (0)
  - Recursive case? - index in bounds (current element + sum of rest)

```
int sumFromIndex(int* array,
                 int length,
                 int index);
```

```
int sumFromIndex(int* array,
                 int length,
                 int index) {
  if (index >= length) return 0;
  else {
   int restSum =
     sumFromIndex(array, length,
                  index + 1);
   return restSum + array[index];
  }
}
```

Recursion Pros

- If your recursive case is always guaranteed to reach a base case, infinite recursion is impossible (appeals to induction)
  - No more infinite loops!
- Vital for more complex recursive data structures (e.g., trees)
- Easier to understand :)

Recursion Cons

- If you’re not careful, you can run out of stack space (a stack overflow)
  - Not written in a tail-recursive way
  - Compiler is too stupid to notice it’s tail-recursive
  - Very large input