Contents

Lesson 1: Fundamentals

1.1 Data Structures

Definition

- Data structure is way of storing data in a computer so it can be used efficiently (by algorithms)
- Data structures are the building blocks from which all programs are built

Why do we care?

- Right data structure results in well performing program
- Wrong data structure results in poorly performing program
- Each data structure has benefits and drawbacks
  - Key is to choose data structure that will perform well for problem you are solving

Examples

- Basic type
  - e.g. int, long, double, char
  - Most basic of all data structures
  - Single object of a single data type
- Struct
  - Grouping of objects of heterogeneous data types
  - Stored contiguously in memory
  - Cannot grow or shrink
  - Good for grouping set of known attributes
- Arrays
  - Grouping of objects of homogenous data type
  - Stored contiguously in memory
  - Cannot grow or shrink
  - Fast random access
  - Slow inserts – requires creating new array (example later this class)
  - Slow deletes – requires creating new array
  - Good for problems requiring a lot of random reads
- Lists
  - Stored non-contiguously in memory

        - Slow random access
        - Fast inserts
        - Fast deletes
        - Good for problems requiring a lot of inserts / deletes
    - Graphs
        - General data structure used to solve many problems

3 goals of good data structures

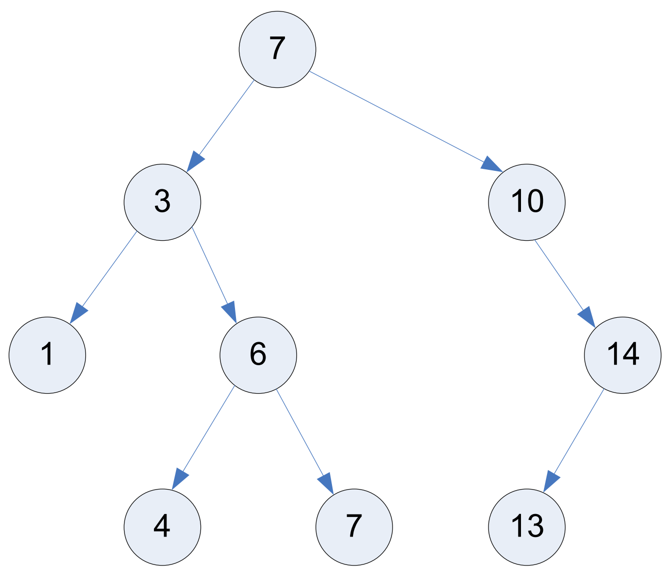
    - Efficiency
        - Well organized data allows algorithms to perform well
        - Example – Searching unsorted array

Unsorted Array

7	3	10	1	6	14	4	7	13

            - Worst case – had to traverse all elements
        - Example – Searching binary search tree

Binary Search Tree



            - Worst case – only had to traverse portion of elements
    - Abstraction
        - Well defined interface makes complex data easier to work with

        - Example – array
            - Don't need to know implementation details to be able to obtain length, access indices, etc.
    - Reusability
        - Generic implementation allows one implementation to be reused with multiple types
        - Example – array
            - Arrays works the same regardless of element data type

What we'll learn about each data structure

- How it works
- Where it performs well
- Where it performs poorly
- The types of problems it is well suited to help solve

1.2 Algorithms

Definition

- Algorithm is a well-defined procedure for solving a problem
- Algorithms are the functions that do work with data

Why do we care?

- Right algorithm results in well performing program
- Wrong algorithm results in poorly performing program

Examples

- Sorting
    - Orders elements in a data structure
    - Useful for preparing data for other algorithms to perform more efficiently
    - Example: Sorting an array allows a binary search algorithm to be performed on the array
    - Various sorting algorithms exist
        - Each has benefits & drawbacks for dealing with different data structures and element types
- Searching
    - Looks for matching element in a data structure
    - Various searching algorithms exist
        - Each has different benefits & drawbacks

3 goals of good algorithms

- Efficiency
    - Researchers have found efficient solutions to common programming problems
    - Reusing the work of others allows you take advantage of this
- Abstraction
    - Well-defined algorithm can be learned and understood without having to think about its inner working

        - Commonly known algorithm provides higher level for discussing solutions to problems
    - Reusability
        - Many algorithms can be applied to different problem domains or data types
        - Example: Determining if two polygons overlap can be simplified to the problem of determining if any of the sides intersect. Since an algorithm exists to solve the latter problem, the former problem may be solved.

1.3 Data Structures & Algorithms Work Together

Data structures

- Hold the data

Algorithms

- Operate on the data

Conclusion

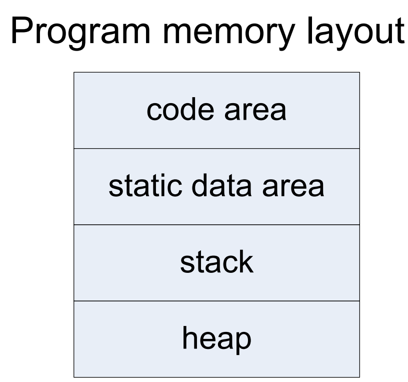
- One without the other is useless
- Choosing the right data structure with the right algorithm allows for efficient, elegant solutions to complex problems

1.4 Anatomy of a Running Program

Important to understand how a running program looks in memory

- Allows us to understand performance and limitations of some algorithms

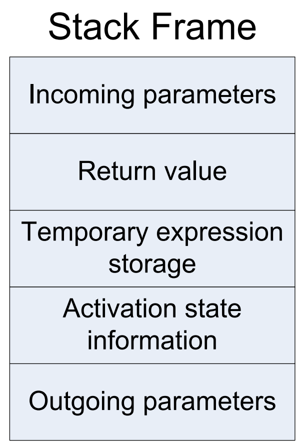
Running program has 4 logical locations in memory



- Code (text)
    - Compiled source code that gets executed as program runs
    - Read-only
- Static data
    - Static and global variables
    - Size pre-determined when program is compiled
- Heap
    - Dynamically allocated memory (malloc, calloc, realloc)
    - Total memory used changes as program allocates / deallocates dynamic objects
- Stack
    - Function call data
        - Local variables
        - Additional info related to managing function calls (described below)

1.5 Function Call Details

- When function called a stack frame (activation frame) is pushed onto the stack



- Incoming parameters
  - Parameters passed to this function
- Outgoing parameters
  - Parameters passed to the function called from this function
- Temporary expression storage
  - Space for expressions executed in this function
- Activation state
  - Info used to return this function call to the previous function call
    - Stack pointer – points to previous functions stack frame
    - Instruction pointer – points to previous function’s code in the Text section where execution should resume when this function returns
- Return value
  - The value returned by this function

So, which algorithms depend on anatomy of a running program?

- Recursive algorithms (those that call themselves)
- They consume stack space
  - If unchecked can cause stack overflow
  - Stack is limited in size (e.g. 1MB)

1.6 Recursion

Recursion

- Recursive function is one that calls itself
- Recursive calls result in activation records on call stack

Example - Recursive factorial

```
int fact(int n) {
      if (n < 0)
            return 0;
      else if (n == 0)
            return 1;
      else if (n == 1)
            return 1;
      else
            return n * fact(n – 1);
}
```

Winding / unwinding of recursive factorial calls

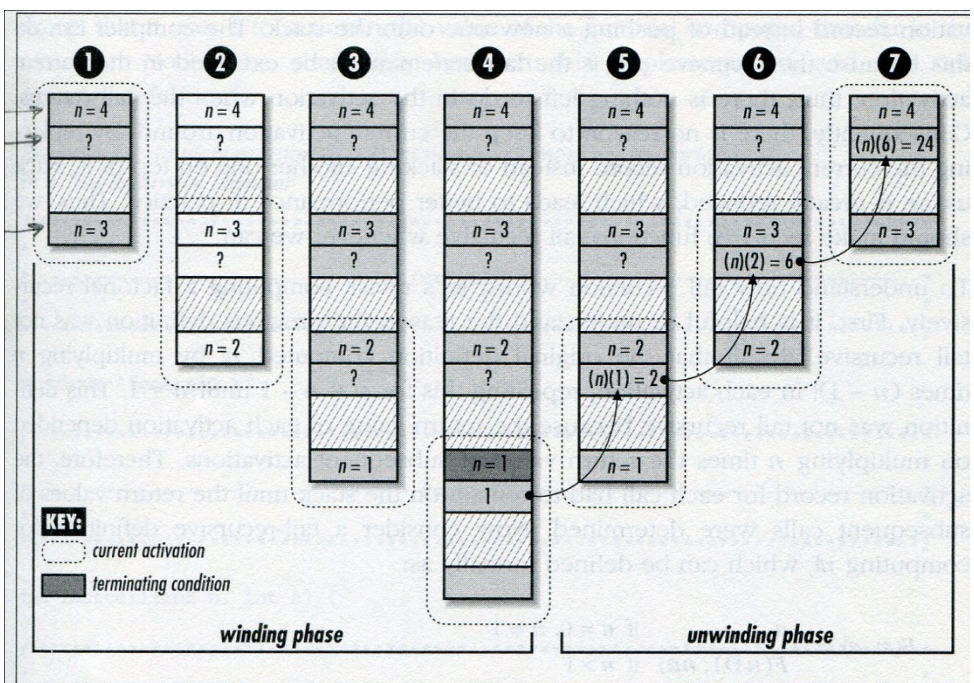


Figure 3-3. The stack of a C program while computing 4! recursively

- High value of n will cause many activation records
- Factorial not tail recursive

        - Result of factorial(n – 1) needed to be used in expression “n * factorial(n – 1)”

Tail recursion elimination

- If recursive call is last expression in function, no need to push new activation record
    - We need to eliminate multiplication so factorial(n – 1) is last expression
    - Called tail recursion elimination

Example – Tail-recursive factorial

```
int facttail(int n, int a) {
	if (n < 0)
		return 0;
	else if (n == 0)
		return 1;
	else if (n == 1)
		return a;
	else
		return facttail(n - 1, n * a);
}
```

- Tail recursive factorial
  - We now pass running factorial value as parameter instead of storing as local variable on stack
  - Compiler can optimize and have recursive call overwrite previous call's activation record
    - Possible because nothing from previous call is needed once recursive call is made

Call stack with tail recursion elimination

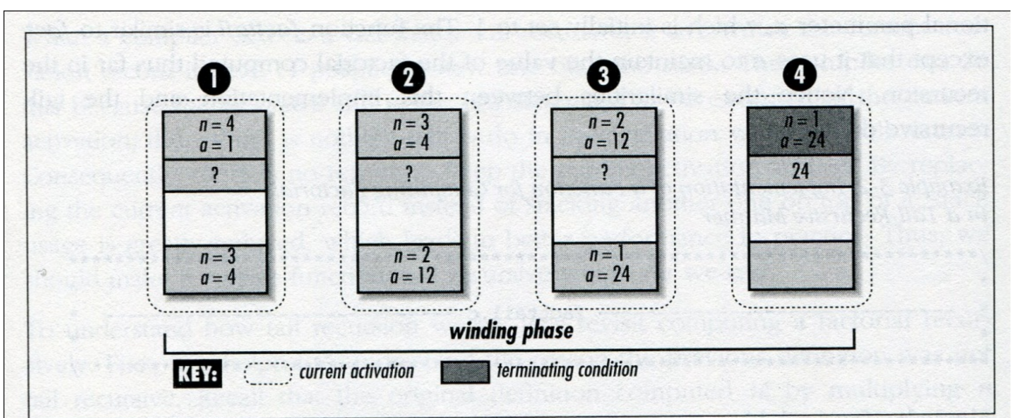


Figure 3-5. The stack of a C program while computing 4! in a tail-recursive manner

Conclusion

- Recursive algorithms can safely recurse to any depth as long as we guarantee they are implemented in tail-recurse fashion

1.7 Pointers

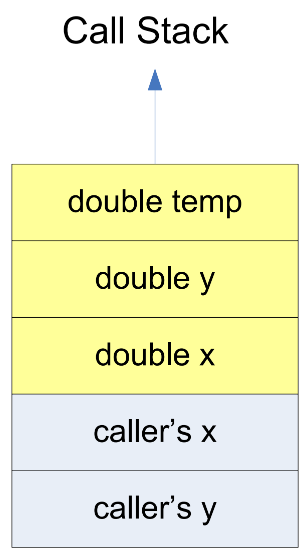
Right-left rule

- Method to determine the type of any variable
- Rules
    1. Start with the identifier (or in the case of a type cast, where the identifier would be)
    2. Look to the right for an attribute and if found, substitute its English equivalent and repeat this step
    3. Look to the left for an attribute and if found, substitute its English equivalent and repeat this step
    4. Continue steps 2 and 3, working your way out until the data type is reached on the left
- Examples
    - double (*cat)();
        - “cat is a pointer to a function returning double”
    - int *dog[];
        - “dog is an array of pointers to int”
    - char (*house)[10];
        - “house is a pointer to an array of 10 chars”
    - short car[][20];
        - “car is an array of arrays of 20 shorts”
    - float *(*(**(*pen)())[6][9])(int y);
        - “pen is a pointer to a function returning a pointer to a pointer to an array of 6 arrays of 9 pointers to functions taking an int and returning a pointer to float”

Pass by value

```
void badSwap(double x, double y) {
      double temp;
      temp = x;
      x = y;
      y = temp;
}
```

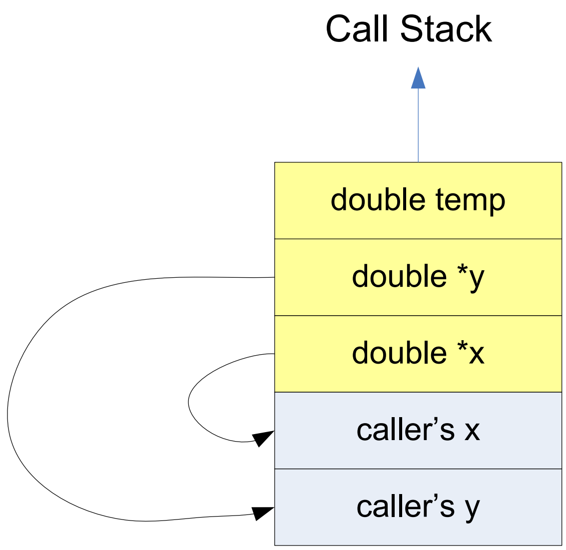
- Pass-by-value
    - All parameters in C are pass-by-value
    - Example: Bad swap



Pass by reference

```
void goodSwap(double *pX, double *pY) {
      double temp;
      temp = *pX;
      *pX = *pY;
      *pY = temp;
}
```

- Pass-by-reference can be simulated in C using pointers
- Example: Good swap



Pointer arithmetic

- p1 == p2
  - Yields whether p1 and p2 point to same location in memory
- p1 < p2, p2 > p1, p1 <= p2, p1 >= p2
  - Yields true / false based on memory addresses held in pointers
- p1 + (integral expression)
  - Yields address equal to p1 + size of type pointed to by p1 * integral expression
- p1 – p2
  - Yields number of elements between the pointers

1.8 Pointers to Functions

- Example: int (*f)(double);
  - "f is a pointer to a function that takes a double and returns an int"
- Allows behavior to be changed at run-time
- Example: Function pointer as parameter to function
  - void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *));
  - cmp parameter accepts two const void *'s and returns an int
    - Allows caller to define behavior used to compare to elements during sorting
- Example: Custom filtering using function pointers

Example – Pointers to functions

```
/*
 * Demonstrate function pointers by passing different
 * implementations of a filter function.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ID_TO_MATCH 13
#define NAME_TO_MATCH "Joe"

typedef struct Employee_
{
   int id;
   char name[256];
} Employee;

/* Return true if the Employee's ID matches */
int filterById(Employee *pEmployee)
{
   return pEmployee->id == ID_TO_MATCH;
}

/* Return true if the Employee's name matches */
int filterByName(Employee *pEmployee)
{
   return strcmp(pEmployee->name, NAME_TO_MATCH) == 0;
```

```c
}

/* Output all employees where the filter returns true */
void outputFilteredEmployees(Employee pEmployees[],
      size_t numEmployees,
      int (*filter)(Employee *pEmployee))
{
   size_t i;
   for (i = 0; i < numEmployees; ++i)
   {
      if (filter(&pEmployees[i]))
      {
         printf("Employee: ID = %d, Name = %s\n",
                pEmployees[i].id,
                pEmployees[i].name);
      }
   }
}

int main(void)
{
   /* Create array of Employees */
   Employee employees[] =
   {
      { 1, "Joe" },
      { 2, "Ray" },
      /* ... */
      { 13, "Sally" }
   };

   /* Output employees filtered by ID */
   outputFilteredEmployees(employees,
               sizeof(employees) / sizeof(employees[0]),
               filterById);

   /* Output employees filtered by name */
   outputFilteredEmployees(employees,
               sizeof(employees) / sizeof(employees[0]),
               filterByName);

   return EXIT_SUCCESS;
}
```

1.9 Void Pointers

- Can point to anything
- We will use extensively to make our data structures “Reusable”
- Will allow us to store elements of any type in our data structures

1.10 Pointers to Pointers

- Can be used to change what a pointer in the calling environment points to
- Example: Returning a pointer through a parameter

Example – Pointers to pointers

```
void allocateInt(int **ppInt) {
	*ppInt = (int *)malloc(sizeof(int));
	**ppInt = 7;
}

int main() {
	/* Pointer to int */
	int *pAge;

	/* Allocate an int and make pAge point to it */
	allocateInt(&pAge);

	/* Outputs 7 */
	printf("%d", *pAge);
}
```

1.11 Arrays

Array storage

- Storage contiguously in memory
- Size cannot be changed
  - Must allocate new array and copy old values to simulate growing array

Dynamic memory allocation

- malloc, calloc
  - Allocate block of memory in heap
- free
  - Frees block of memory in heap
- realloc
  - Attempts to reserve more memory at end of current block of memory
    - If available this is a very fast operation
    - If not available internally realloc:
      - Allocates new block of memory with new size
      - Copies values from old memory to new
      - Frees old memory
      - Slow

Array names as pointers

- Array names decay to pointers except in these cases:

```
int test[] = {5, 10, 15, 20, 25, 30};
```

  - As operand of sizeof

```
sizeof(test);           /* test type is "array of int" */
```

  - As operand of unary & operator

```
&test;                  /* test type is "array of int" */
```

  - As character string literal used to initialize array of char type

```
char foo[] = "abc";     /* literal is "array of char" */
```

- This means test decays to pointer to int in following:

```
test + 2          /* points to value "15" */
sizeof(test + 3)  /* sizeof ptr to int, not sizeof array */
test[5]           /* ptr arithmetic, same as *(test + 5) */
```

- Equivalent:
  - test + i   points to the same element as &test[i]
  - *(test + i)   references the same element as test[i]
- Array name decays to rvalue:
  - Array variable cannot be assigned

```
int array[] = {1, 2, 3, 4, 5};
int *pointer;
pointer = array;
array = pointer;  /* illegal: array name decays to rvalue */
```

1.12 Analysis of Algorithms

Analyzing algorithmic performance

- Question: How does an algorithm perform on a given set of data?
- Performance metrics
  - Speed
    - How fast does an algorithm run
  - Memory usage
    - How much memory does an algorithm use
- Usually speed is the most important; we will focus on that in this class

Worst-case performance

- This is the metric we'll use to compare
- This is usually the metric by which algorithms compared because:
  - Worst case occurs frequently
    - Example, failing to find the item we're searching for
  - Best-case not informative
    - Example, finding an item on the first check doesn't tell us anything about an algorithm
  - Average case not always easy to determine
    - Sometimes it's even difficult to define what average means
    - A few algorithms (those having a randomized step) rarely exhibit worst case, so for those algorithms we'll use average case
  - Worst case gives upper bound
    - We're guaranteed an algorithm will never perform worse than worst case

O-notation (Big-O)

- Most common notation used to express an algorithm's performance
- Describes performance in terms of the size of the input

1.13 Big-O Example #1

Analyzing function performance

```
void foo(int array[], int size) {
      for (int i = 0; i < size; ++i) {    /* c1 * n */
            printf("%d", array[i]);       /* c2 */
      }
}
```

- Performance:
  - There are "n" elements in array
  - for loop costs constant time "c1" and it occurs n times
  - printf costs constant time "c2"
  - Performance of function
    - = O(c1 * n * c2)
    - = O((c1 + c2) * n)
- Big-O is Relative
  - O-notation's goal is to provide performance rating to be compared relatively to other algorithms
- Rules:
  - Constant terms are ignored
    - Reason: As the size of input grows their effect on the execution time becomes insignificant
    - Example:
      - Compare two functions T1 & T2
        - T1 = n + 50
        - T2 = n + 1000
      - The performance of both is O(n) since constant terms can be dropped
      - Dropping constant terms makes sense because as n becomes large the constant factors don't contribute to performance
    - **Formally: O(c) = O(1)**
  - Constant multipliers are ignored
    - Reason: As the size of input grows their effect on the execution time becomes insignificant
    - Example:
      - Compare two functions T1 & T2

            - T1 = 500n
            - T2 = n^2
        - The performance of T1 is worse when n is < 500 but once n becomes greater than 500 T1 performs better. As n becomes very large the 500 multiplier become unimportant when compared to the very poor performance of T2.
        - Dropping constant multipliers makes sense because as n becomes large the constant multipliers don't contribute to performance relative to other algorithms with higher powers of n.
        - Note, for two algorithms with the same Big-O the constant multipliers do determine which algorithm performs better for a given value of n. But, the constant multipliers are not included in Big-O because the goal of Big-O is for high level relative comparisons of algorithms to see if they fall into the same "category" of performance relative only to the input size.
    - **Formally: O(cT) = cO(T) = O(T)**
- Only consider highest order term
    - Reason: As the size of input grows the higher-order terms quickly outweigh the lower-order ones
    - Example:
        - T(n) = n^2 + n; as n becomes large the lesser-order term becomes insignificant; T is O(n^2)
    - **Formally: O(T1) + O(T2) = O(T1 + T2) = max (O(T1), O(T2))**
- Multiply nested loops
    - Reason: When one task causes another task to be executed for each iteration itself the inner task will be executed a number of times equal to the number of iterations of the outer task.
    - Example:
        - In a nested loop whose outer iterations are described by T1 and whose inner iterations are described by T2, if T1(n) = n and T2(n) = n, the result is O(n)O(n) = O(n^2)

- **Formally: O(T1)O(T2) = O(T1T2)**

1.14 Big-O Example #2

Analyzing performance of nested loops

```
for (int i = 0; i < n; ++i) {              /* O(n) */
  for (int j = 0; j < n / 2; ++j) {        /* O(n / 2) */
    for (int k = 0; k < n * 3; ++k) {      /* O(n * 3) */
      printf("%d", k);                     /* c */
    }
  }
}
Result:  O(n)O(n / 2)O(n * 3)c
    = O(n * n / 2)O(n * 3)c
    = O(n * n / 2 * n * 3)c
    = O((3/2)*n^3)c
    = O(n^3)c
    = O(n^3)
```

- Example: Linear vs. binary search using an array
  - **[Example will be drawn on board]**
- Performance comparisons

Table 4-2. The Growth Rates of the Complexities in Table 4-1 (continued)

	$n = 1$	$n = 16$	$n = 256$	$n = 4K$	$n = 64K$	$n = 1M$
$O(n^2)$	1.000E+00	2.560E+02	6.554E+04	1.678E+07	4.295E+09	1.100E+12
$O(2^n)$	2.000E+00	6.554E+04	1.158E+77	—	—	—
$O(n!)$	1.000E+00	2.092E+13	—	—	—	—

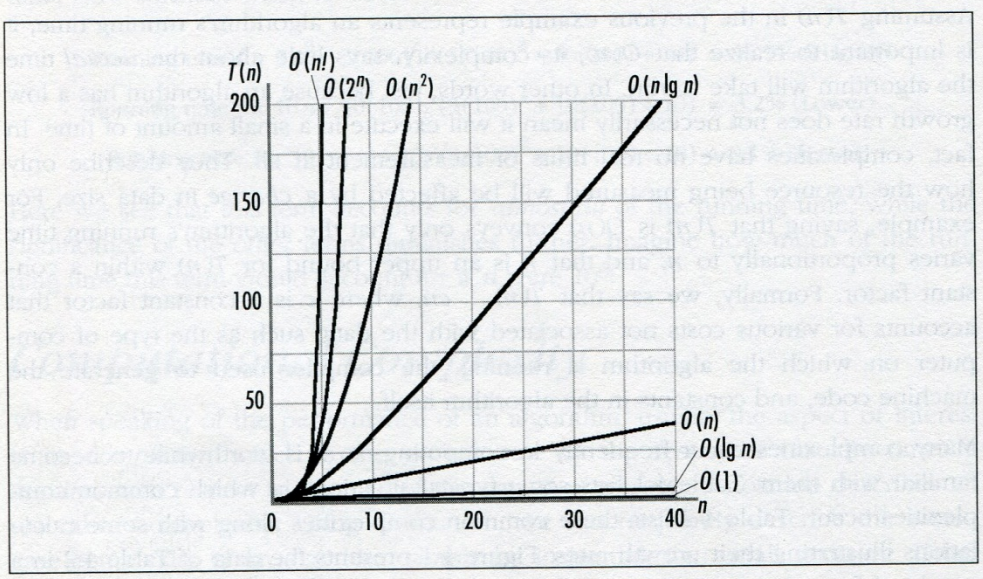


Figure 4-1. A graphical depiction of the growth rates in Tables 4-1 and 4-2

1.15 Profiling

- Profiling allows us determine how sections of our program are performing
- Example (available on course website)

```c
#include <stdio.h>
#include <time.h>

void doSomeLengthyOperation()
{
   int i;
   for (i = 0; i < 10000000; ++i)
      printf(".");
}

int main()
{
   clock_t startTicks;
   clock_t stopTicks;
   double elapsedSeconds;

   /* Get elapsed ticks prior to executing section of code */
   startTicks = clock();

   /* Execute the section of code */
   doSomeLengthyOperation();

   /* Get elapsed ticks after executing section of code */
   stopTicks = clock();

   /* Output time section of code took to complete */
   elapsedSeconds = (double)(stopTicks - startTicks) / CLOCKS_PER_SEC;
   printf("Operation took %g seconds to complete.\n", elapsedSeconds);
}

/*
 * Program output (on my MacBook Pro, 2.53 GHz Intel Core 2 Duo):
 *    Operation took 0.290777 seconds to complete.
 */
```

Lesson 2:
Linked Lists