COMSM1201 : Data Structures & Algorithms

Dr. Neill Campbell
Neill.Campbell@bristol.ac.uk

University of Bristol

November 2, 2021

Table of Contents

Simple Recursion

- When a function calls itself, this is known as recursion.
- This is an important theme in Computer Science that crops up time & time again.
- Can sometimes lead to very simple and elegant programs.
- Let's look at some toy examples to begin with.

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

#define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}

void strrev(char* s, int n);

int main(void)
{
   char str[] = "Hello World!";
   strrev(str, strlen(str));
   printf("%s\n", str);
   return 0;
}

/* Iterative Inplace String Reverse */
void strrev(char* s, int n)
{
   for(int i=0, j=n-1; i<j; i++, j--){
      SWAP(s[i], s[j]);
   }
}
```

Execution :

```
!dlroW olleH
```

Recursion for *strrev()*

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

#define SWAP(A,B) {char temp; temp=A;A=B;B=temp;}

void strrev(char* s, int start, int end);

int main(void)
{
    char str[] = "Hello World!";
    strrev(str, 0, strlen(str)-1);
    printf("%s\n", str);
    return 0;
}

/* Recursive : Inplace String Reverse */
void strrev(char* s, int start, int end)
{
    if(start >= end){
        return;
    }
    SWAP(s[start], s[end]);
    strrev(s, start+1, end-1);
}
```

Execution :

```
!dlroW olleH
```

- We need to change the function prototype.
- This allows us to track both the start and the end of the string.

The Fibonacci Sequence

A well known example of a recursive function is the Fibonacci sequence. The first term is 1, the second term is 1 and each successive term is defined to be the sum of the two previous terms, i.e. :

fib(1) is 1
fib(2) is 1
fib(n) is fib(n-1)+fib(n-2)

1,1,2,3,5,8,13,21, ...

Iterative & Recursive Fibonacci

```c
#include <stdio.h>

#define MAXFIB 24

int fibonacci(int n);

int main(void)
{

    for(int i=1; i<=MAXFIB; i++){
        printf("%d = %d\n", i, fibonacci(i));
    }

    return 0;

}

int fibonacci(int n)
{
    if(n <= 2){
        return 1;
    }
    int a = 1;
    int b = 1;
    int next;
    for(int i=3; i<=n; i++){
        next = a + b;
        a = b;
        b = next;
    }
    return b;
}
```

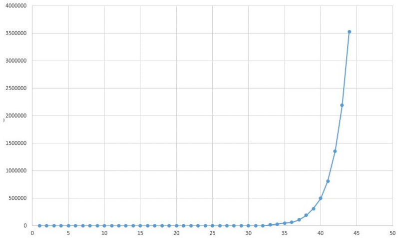
Execution :

```
1 = 1
2 = 1
3 = 2
4 = 3
5 = 5
6 = 8
7 = 13
8 = 21
9 = 34
10 = 55
11 = 89
12 = 144
13 = 233
14 = 377
15 = 610
16 = 987
17 = 1597
18 = 2584
19 = 4181
20 = 6765
21 = 10946
22 = 17711
23 = 28657
24 = 46368
```

Iterative & Recursive Fibonacci

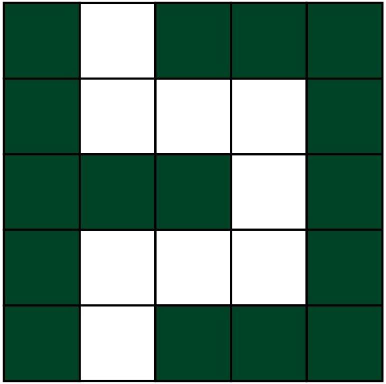
```c
#include <stdio.h>

#define MAXFIB 24

int fibonacci(int n);

int main(void)
{

    for(int i=1; i<=MAXFIB; i++){
        printf("%d = %d\n", i, fibonacci(i));
    }

    return 0;

}

int fibonacci(int n)
{
    if(n == 1) return 1;
    if(n == 2) return 1;
    return( fibonacci(n-1)+fibonacci(n-2));
}
```

It's interesting to see how run-time increases as the length of the sequence is raised.



Maze Escape

The correct route through a maze can be obtained via recursive, rather than iterative, methods.



```
bool explore(int x, int y, char mz[YS][XS])
{
  if  mz[y][x] is exit return true;

  Mark mz[y][x] so we don't return here

  if we can go up :
    if(explore(x, y+1, mz)) return true

  if we can go right :
    if(explore(x+1, y, mz)) return true

  Do left & down in a similar manner

  return false; // Failed to find route
}
```

Permuting

- Here we consider the ways to permute a string (or more generally an array)
- Permutations are all possible ways of rearranging the positions of the characters.

  Execution :

  ABC
  ACB
  BAC
  BCA
  CBA
  CAB

```c
// From e.g.  http://www.geeksforgeeks.org
#include <stdio.h>
#include <string.h>

#define SWAP(A,B) {char temp = *A; *A = *B; *B = temp;}

void permute(char* a, int s, int e);

int main()
{
     char str[] = "ABC";
     int n = strlen(str);
     permute(str, 0, n-1);
     return 0;
}

void permute(char* a, int s, int e)
{
    if (s == e){
      printf("%s\n", a);
      return;
    }
    for (int i = s; i <= e; i++){
       SWAP((a+s), (a+i)); // Bring one char to the front
       permute(a, s+1, e);
       SWAP((a+s), (a+i)); // Backtrack
    }
}
```

Self-test : Power

- Raising a number to a power $\mathtt{n} = 2\hat{\ }5$ is the same as multiple multiplications $\mathtt{n} = 2{*}2{*}2{*}2{*}2$.
- Or, thinking recursively, $\mathtt{n} = 2 * (2\hat{\ }4)$.

```c
/* Try to write power(a,b) to computer a^b
   without using any maths functions other than
   multiplication :
   Try (1) iterative then (2) recursive
   (3) Trick that for n%2==0, x^n = x^(n/2)*x^(n/2)

*/

#include <stdio.h>

int power(unsigned int a, unsigned int b);

int main(void)
{

   int x = 2;
   int y = 16;

   printf("%d^%d = %d\n", x, y, power(x,y));

}

int power(unsigned int a, unsigned int b)
{
}
```

Table of Contents

Sequential Search

- The need to search an array for a particular value is a common problem.
- This is used to delete names from a mailing list, or upgrading the salary of an employee etc.
- The simplest method for searching is called the sequential search.
- Simply move through the array from beginning to end, stopping when you have found the value you require.

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define NOTFOUND -1
#define NUMPEOPLE 6
typedef struct person{
        char* name; int age;
} person;

int findAge(const char* name, const person* p, int n);

int main(void)
{
   person ppl[NUMPEOPLE] = { {"Ackerby", 21}, {"Bloggs", 25},
               {"Chumley", 26}, {"Dalton",  25},
               {"Eggson", 22}, {"Fulton",  41} };

   assert(findAge("Eggson",  ppl, NUMPEOPLE)==22);
   assert(findAge("Campbell", ppl, NUMPEOPLE)==NOTFOUND);
   return 0;
}

int findAge(const char* name, const person* p, int n)
{
   for(int j=0; j<n; j++){
      if(strcmp(name, p[j].name) == 0){
         return p[j].age;
      }
   }
   return NOTFOUND;
}
```

Sequential Search

- Sometimes our list of people may not be random.
- If, for instance, it is sorted, we can use strcmp() in a slightly cleverer manner.
- We can stop searching once the search key is alphabetically greater than the item at the current position in the list.
- This halves, on average, the number of comparisons required.

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>

#define NOTFOUND -1
#define NUMPEOPLE 6
typedef struct person{
        char* name; int age;
} person;

int findAge(const char* name, const person* p, int n);

int main(void)
{
   person ppl[NUMPEOPLE] = { {"Ackerby", 21}, {"Bloggs", 25},
              {"Chumley", 26}, {"Dalton",  25},
              {"Eggson", 22}, {"Fulton",  41} };

   assert(findAge("Eggson",   ppl, NUMPEOPLE)==22);
   assert(findAge("Campbell", ppl, NUMPEOPLE)==NOTFOUND);
   return 0;
}

int findAge(const char* name, const person* p, int n)
{
   for(int j=0; j<n; j++){
      int m = strcmp(name, p[j].name);
      if(m == 0) // Braces!
         return p[j].age;
      if(m < 0)
         return NOTFOUND;
   }
   return NOTFOUND;
}
```

Binary Search for *101*

- Searching small lists doesn't require much computation time.
- However, as lists get longer (e.g. phone directories), sequential searching becomes extremely inefficient.
- A binary search consists of examining the middle element of the array to see if it has the desired value. If not, then half the array may be discarded for the next search.

4	7	19	25	36	37	50	100	101	205	220	270	301	321
⇑						↑							⇑
							⇑			↑			⇑
							⇑	↑	⇑				

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#define NMBRS 1000000

int bin_it(int k, const int* a, int l, int r);

int main(void)
{
   int a[NMBRS];
   srand(time(NULL));

   // Put even numbers into array
   for(int i=0; i<NMBRS; i++){
      a[i] = 2*i;
   }

   // Do many searches for a random number
   for(int i=0; i<10*NMBRS; i++){
      int n = rand()%NMBRS;
      if((n%2) == 0){
         assert(bin_it(n, a, 0, NMBRS-1) == n/2);
      }
      else{ // No odd numbers in this list
         assert(bin_it(n, a, 0, NMBRS-1) < 0);
      }
   }
   return 0;
}
```

Iterative v. Recursion Binary Search

```
int bin_it(int k, const int* a, int l, int r)
{
    while(l <= r){
        int m = (l+r)/2;
        if(k == a[m]){
            return m;
        }
        else{
            if (k > a[m]){
                l = m + 1;
            }
            else{
                r = m - 1;
            }
        }
    }
    return -1;
}
```

```
int bin_rec(int k, const int* a, int l, int r)
{
    if(l > r) return -1;
    int m = (l+r)/2;
    if(k == a[m]){
        return m;
    }
    else{
        if (k > a[m]){
            return bin_rec(k, a, m+1, r);
        }
        else{
            return bin_rec(k, a, l, m-1);
        }
    }
}
```

Interpolation Search

- When we look for a word in a dictionary, we don't start in the middle. We make an educated guess as to where to start based on the 1st letter of the word being searched for.
- This idea led to the interpolation search.
- In binary searching, we simply used the middle of an ordered list as a best guess as to where to begin the search.
- Now we use an interpolation involving the key, the start of the list and the end.

$$i = (k - l[0])/(l[n-1] - l[0]) * n$$

- when searching for '15' :

$$\begin{array}{cccccccc} 0 & 4 & 5 & 9 & 10 & 12 & 15 & 20 \\ & & & & & & \Uparrow & \end{array}$$

```
int interp(int k, const int* a, int l, int r)
{

   int m;
   double md;

   while(l <= r){
      md = ((double)(k-a[l])/
             (double)(a[r]-a[l])*
             (double)(r-l)
            )
            +(double)(l);
      m  = 0.5 + md;
      if((m > r) || (m < l)){
         return -1;
      }
      if(k == a[m])
         return m;
      else{
         if (k > a[m]){
            l = m + 1;
         }
         else{
            r = m- 1;
         }
      }
   }
```

Algorithmic Complexity

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

#define CSEC (double)(CLOCKS_PER_SEC)
#define BIGLOOP 1000000000

int main(void)
{

    clock_t c1 = clock();
    for(int i=0; i<BIGLOOP; i++){
          int j = i * 2;
    }
    clock_t c2 = clock();
    printf("%f\n", (double)(c2-c1)/CSEC);
    return 0;

}
```

- This code on an old Dell laptop took:
    - 3.12 seconds using a non-optimzing compiler -O0
    - 0.00 seconds using an aggressive optimization -O3
- But "wall-clock" time is generally not the thing that excites Computer Scientists.

- Searching and sorting algorithms have a complexity associated with them, called big-O.
- This complexity indicates how, for *n* numbers, performance deteriorates when *n* changes.
- Sequential Search : **O(n)**
- Binary Search : **O(log n)**
- Interpolation Search : **O(log log n)**
- We'll discuss the dream of a **O(1)** search later in "Hashing".

Binary vs. Interpolation Timing

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

int bin_it(int k, const int *a, int l, int r);
int bin_rec(int k, const int *a, int l, int r);
int interp(int k, const int *a, int l, int r);
int* parse_args(int argc, char* argv[], int* n, int* srch);

int main(int argc, char* argv[])
{

    int i, n, srch;
    int* a;
    int (*p[3])(int k, const int*a, int l, int r) =
        {bin_it, bin_rec, interp};

    a = parse_args(argc, argv, &n, &srch);

    srand(time(NULL));
    for(i=0; i<n; i++){
       a[i] = 2*i;
    }
    for(i=0; i<5000000; i++){
       assert((*p[srch])(a[rand()%n], a, 0, n-1) >= 0);
    }

    free(a);
    return 0;

}
```

Execution :

```
Binary Search : Iterative
n =    100000 = 0.57
n =    800000 = 0.84
n =   6400000 = 2.20
n =  51200000 = 3.87
Binary Search : Recursive
n =    100000 = 1.23
n =    800000 = 1.79
n =   6400000 = 3.20
n =  51200000 = 4.85
Interpolation
n =    100000 = 0.20
n =    800000 = 0.28
n =   6400000 = 0.50
n =  51200000 = 0.70
```

Table of Contents

Linked Data Structures

- Linked data representations are useful when:
  - It is difficult to predict the size and the shape of the data structures in advance.
  - We need to efficiently insert and delete elements.
- To create linked data representations we use pointers to connect separate blocks of storage together. If a given block contains a pointer to a second block, we can follow this pointer there.
- By following pointers one after another, we can travel right along the structure.

```c
#include <stdio.h>
#include <stdlib.h>
#include "general.h"

typedef struct data{
    int i;
    struct data* next;
} Data;

Data* allocateData(int i);
void printList(Data* l);

int main(void)
{
    int i;
    Data* start, *current;
    start = current = NULL;
    printf("Enter the first number: ");
    if(scanf("%i", &i) == 1){
        start = current = allocateData(i);
    }
    else{
        on_error("Couldn't read an int");
    }

    printf("Enter more numbers: ");
    while(scanf("%i", &i) == 1){
        current->next = allocateData(i);
        current = current->next;
    }
    printList(start);
    // Should Free List
    return 0;
}
```

Linked Lists

```c
Data* allocateData(int i)
{
    Data* p;
    p = (Data*) ncalloc(1, sizeof(Data));
    p->i = i;
    // Not really required
    p->next = NULL;
    return p;
}

void printList(Data* l)
{
    printf("\n");
    do{
        printf("Number : %i\n", l->i);
        l = l->next;
    }while(l != NULL);
    printf("END\n");
}
```

Searching and Recursive printing:

```c
Data* inList(Data* n, int i)
{
    do{
        if(n->i==i){
            return n;
        }
        n = n->next;
    }while(n != NULL);
    return NULL;
}

void printList_r(Data* l)
{
    // Recursive Base-Case
    if(l == NULL) return;

    printf("Number: %i\n", l->i);
    printList_r(l->next);
}
```

Abstract Data Types

- But would we really code something like this **every** time we need flexible data storage ?
- This would be horribly error-prone.
- Build something once, and test it well.
- One example of this is an **Abstract Data Type (ADT)**.
- Each ADT exposes its functionality via an *interface*.
- The user only accesses the data via this interface.
- The user of the ADT doesn't need to understand how the data is being stored (e.g. array vs. linked lists etc.)

Table of Contents

Collections

- One of the simplest ADTs is the **Collection**.
- This is just a simple place to search for/add/delete data elements.
- Some collections allow duplicate elements and others do not (e.g. Sets).
- Some are ordered (for faster searching) and others unordered.
- Our Collection will be unsorted and will allow duplicates.

```c
#include "../General/general.h"

typedef int colltype;

typedef struct coll coll;

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// Create an empty coll
coll* coll_init(void);
// Add element onto top
void coll_add(coll* c, colltype i);
// Take element out
bool coll_remove(coll* c, colltype d);
// Does this exist ?
bool coll_isin(coll* c, colltype i);
// Return size of coll
int coll_size(coll* c);
// Clears all space used
bool coll_free(coll* c);
```

Collection ADT

- Note that the interface gives you no hints as to the actual underlying implementation of the ADT.
- A user of the ADT doesn't really need to know how it's implemented - ideally.
- The ADT developer could have several **different** implementations.
- Here we'll see *Collection* implemented using:
  - A fixed-size array
  - A dynamic array
  - A linked-list

Fixed/specific.h:

```c
#pragma once

#define COLLTYPE "Fixed"

#define FIXEDSIZE 5000
struct coll {
   // Underlying array
   colltype a[FIXEDSIZE];
   int size;
};
```

Collection ADT using a Fixed-size Array

Fixed/fixed.c:

```c
#include "../coll.h"
#include "specific.h"

coll* coll_init(void)
{
   coll* c = (coll*) ncalloc(sizeof(coll), 1);
   c->size = 0;
   return c;
}

int coll_size(coll* c)
{
   if(c==NULL){
      return 0;
   }
   return c->size;
}

bool coll_isin(coll* c, colltype d)
{
   for(int i=0; i<coll_size(c); i++){
      if(c->a[i] == d){
          return true;
      }
   }
   return false;
}

void coll_add(coll* c, colltype d)
{
   if(c){
      c->a[c->size] = d;
      c->size = c->size + 1;
      if(c->size >= FIXEDSIZE){
         on_error("Collection overflow");
      }
   }
}

bool coll_remove(coll* c, colltype d)
{
   for(int i=0; i<coll_size(c); i++){
      if(c->a[i] == d){
         // Shuffle end of array left one
         for(int j=i; j<coll_size(c); j++){
            c->a[j] = c->a[j+1];
         }
         c->size = c->size - 1;
         return true;
      }
   }
   return false;
}

bool coll_free(coll* c)
{
   free(c);
   return true;
}
```

Collection ADT via a Dynamic (Realloc) Array

Realloc/specific.h:

```c
#pragma once

#define COLLTYPE "Realloc"

#define FIXEDSIZE 16
#define SCALEFACTOR 2
struct coll {
   // Underlying array
   colltype* a;
   int size;
   int capacity;
};
```

Realloc/realloc.c:

```c
#include "../coll.h"
#include "specific.h"

coll* coll_init(void)
{
   coll* c = (coll*) ncalloc(sizeof(coll), 1);
   c->a = (colltype*) ncalloc(sizeof(colltype), FIXEDSIZE);
   c->size = 0;
   c->capacity= FIXEDSIZE;
   return c;
}

void coll_add(coll* c, colltype d)
{
   if(c){
      c->a[c->size] = d;
      c->size = c->size + 1;
      if(c->size >= c->capacity){
         c->a = (colltype*) nremalloc(c->a,
                  sizeof(colltype)*c->capacity*SCALEFACTOR);
         c->capacity = c->capacity*SCALEFACTOR;
      }
   }
```

Collection ADT via a Linked List

Linked/specific.h:

```c
#pragma once

#define COLLTYPE "Linked"

struct dataframe {
   colltype i;
   struct dataframe* next;
};
typedef struct dataframe dataframe;

struct coll {
   // Underlying array
   dataframe* start;
   int size;
};
```

Linked/linked.c:

```c
#include "../coll.h"
#include "specific.h"

coll* coll_init(void)
{
   coll* c = (coll*) ncalloc(sizeof(coll), 1);
   return c;
}

int coll_size(coll* c)
{
   if(c==NULL){
      return 0;
   }
   return c->size;
}

bool coll_isin(coll* c, colltype d)
{
   if(c == NULL || c->start==NULL){
      return false;
   }
   dataframe* f = c->start;
   do{
      if(f->i == d){
         return true;
      }
      f = f->next;
   }while(f != NULL);
   return false;
}
```

Collection ADT via a Linked List II

```c
void coll_add(coll* c, colltype d)
{
   if(c){
      dataframe* f = ncalloc(sizeof(dataframe), 1);
      f->i = d;
      f->next = c->start;
      c->start = f;
      c->size = c->size + 1;
   }

}

bool coll_free(coll* c)
{
   if(c){
      dataframe* tmp;
      dataframe* p = c->start;
      while(p!=NULL){
         tmp = p->next;
         free(p);
         p = tmp;
      }
      free(c);
   }
   return true;
}
```

```c
bool coll_remove(coll* c, colltype d)
{
   dataframe* f1, *f2;
   if((c==NULL) || (c->start==NULL)){
      return false;
   }

   // If Front
   if(c->start->i == d){
      f1 = c->start->next;
      free(c->start);
      c->start = f1;
      c->size = c->size - 1;
      return true;
   }

   f1 = c->start;
   f2 = c->start->next;
   do{
      if(f2->i == d){
         f1->next = f2->next;
         free(f2);
         c->size = c->size - 1;
         return true;
      }
      f1 = f2;
      f2 = f1->next;
   }while(f2 != NULL);
   return false;
}
```

Collection Summary

- Any code using the ADT can be compiled against any of the implementations, e.g. the test (testcoll.c) code.
- The *Collection* interface (coll.h) is never changed.
- There are pros and cons of each implementation:
  - Fixed Array : Simple to implement - can't avoid the problems of it being a fixed-size. Deletion expensive.
  - Dynamic Array : Implementation fairly simple. Deletion expensive. Every realloc() is very **expensive**. Need to tune SCALEFACTOR.
  - Linked : Slightly fiddly implementation - fast to delete an element.

That Linked List code from the previous Chapter again:

```c
#include "coll.h"
#include "Fixed/specific.h"

int main(void)
{
    coll* c;
    int i;

    printf("Please type some numbers :");
    c = coll_init();
    while(scanf("%i", &i) == 1){
       coll_add(c, i);
    }
    // Do print etc.
    coll_free(c);
    return 0;
}
```