Problem-solving

Many people "get stuck" when faced with a new problem

• where to start? ... what approach should I use?
• is there a known way to solve this problem?

Standard strategies and programming techniques

• recursion ... solve a problem in terms of a simpler version of itself
• divide-and-conquer ... partition, solve sub-problems, combine results
• higher-order functions ... package patterns of computation into generic tools
Recursive Functions

• problems can sometimes be expressed in terms of a simpler instance of the same problem

• Example: factorial
  
  • \( 1! = 1 \)
  • \( 2! = 1 \times 2 \)
  • \( 3! = 1 \times 2 \times 3 \)
  • \( \ldots \)
  • \( (N-1)! = 1 \times 2 \times 3 \times \ldots \times (N-1) \)
  • \( N! = 1 \times 2 \times 3 \times \ldots \times (N-1) \times N \)

\[ 2! = 1! \times 2 \]
\[ N! = (N-1)! \times N \]
Recursive Functions

- Solving problems recursively in a program involves
  - Developing a function that calls itself
  - Must include
    - **Base Case**: aka stopping case: so easy no recursive call is needed
    - **Recursive Case**: calls the function on a ‘smaller’ version of the problem
Iteration vs Recursion

• **Compute** $N! = 1 \times 2 \times 3 \times \ldots \times N$

```c
//An iterative solution
int factorial(int N){
    result = 1;
    for (i = 1; i <= N; i++)
        result = i * result;
    return result;
}
```

• **Alternative Solution:** factorial calls itself recursively

```c
int factorial (int N) {
    if (N == 1) {
        return 1; //base case
    } else {
        return N * factorial (N-1); //recursive case
    }
}
```
Bad Fibonacci

- Sometimes recursive code results in horribly in-efficient code that re-evaluates things over and over.
- \(2^n\) calls: \(O(k^n)\) - exponential
- Exponential functions can only be used in practice for very small values of \(n\)

```c
// Code to return the nth fibonacci number
// 0 1 1 2 3 5 8 13 21
int badFib(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;
    return badFib(n-1) + badFib(n-2);
}
```
Why badFib is bad

- Tracing calls on BadFib produces a tree of calls where intermediate results are recalculated again and again.

F(3) is evaluated three times independently
Linked Lists

A linked list can be described recursively

- A list is comprised of a
  - head (a node)
  - a tail (the rest of the list)

```c
typedef struct node * link;

struct node{
    int item;
    link next;
};
```
Recursive List Functions

• We can define some list operations as recursive functions:
  • **length**: return the length of a list
  • **sumOfElems**: return the length of a list
  • **printList**: print the list
  • **printListReverse**: print out the list in reverse order
• Recursive list operations are not useful for huge lists
  • The depth of recursion may be proportional to the length of the list
Recursive List Functions

```c
int length (link ls) {
    if (ls == NULL) {
        return 0; // base case
    }
    return 1 + length (ls->next); // recursive case
}

int sumOfElems (link ls) {
    if (ls == NULL) {
        return 0; // base case
    }
    return (ls->item + sumOfElems(ls->next)); // recursive case
}
```
Recursive List Functions

```c
void printList(link ls)
{
    if(ls != NULL)
    {
        printf("%d\n",ls->item);
        printList(ls->next);
    }
}

//To print in reverse change the
//order of the recursive call and
//the printf
void printListReverse(link ls)
{
    if(ls != NULL)
    {
        printListReverse(ls->next);
        printf("%d\n",ls->item);
    }
}
```
Divide and Conquer

Basic Idea:

• divide the input into two parts
• solve the problems recursively on both parts
• combine the results on the two halves into an overall solution
Divide and Conquer

Divide and Conquer Approach for finding maximum in an unsorted array:

• Divide array in two halves in each recursive step

  Base case
  • subarray with exactly one element: return it

  Recursive case
  • split array into two
  • find maximum of each half (recursively)
  • return maximum of the two sub-solutions
//iterative solution O(n)
int maximum(int a[], int n){
    int a[N];
    int max = a[0];
    int i;
    for (i=0; i < n; i++){
        if (a[i] > max){
            max = a[i];
        }
    }
    return max;
}
Divide and Conquer Solution

//Divide and conquer recursive solution
int max (int a[], int l, int r) {
    int m1, m2;
    int m = (l+r)/2;
    if (l==r) {
        return a[l];
    }
    //find max of left half
    m1 = max (a,l,m);
    //find max of right half
    m2 = max (a, m+1, r)
    //combine results to get max of both halves
    if (m1 < m2) {
        return m2;
    } else {
        return m1;
    }
}

Complexity Analysis

How many calls of max are necessary for the divide and conquer maximum algorithm?

• Length = 1
  \[ T_1 = 1 \]

• Length = \( N > 1 \)
  \[ T_N = T_{N/2} + T_{N/2} + 1 \]

• Overall, we have
  \[ T_N = N + 1 \]

In each recursive call, we have to do a fixed number of steps (independent of the size of the argument)

• \( O(N) \)
Recursive Binary Search

Maintain two indices, l and r, to denote leftmost and rightmost array index of current part of the array
• initially l=0 and r=N-1

Base cases:
• array is empty, element not found
• a[(l+r)/2] holds the element we’re looking for

Recursive cases: a[(l+r)/2] is
• larger than element, continue search on a[l]..a[(l+r)/2-1]
• smaller than element, continue search on a[(l+r)/2+1]..a[r]

O(log(n))