Recursion

Problem decomposition

- Problem decomposition is a common technique for problem solving in programming – to reduce a large problem to smaller and more manageable problems, and solving the large problem by combining the solutions of a set of smaller problems.

- Example 0:
  - Problem: Sort an array of A[0..N]
  - Decompose to:
    - Subproblem1: Sort the array of A[1..N], *why is it a smaller problem?*
    - Subproblem2: insert A[0] to the sorted A[1..N]. *this is easier than sorting.*

Problem decomposition

- Example 1:
  - Problem (size = N): Compute $\sum_{i=1}^{N} i^3$
  - Decompose to:
    - Subproblem (size = N-1): Compute X = $\sum_{i=1}^{N-1} i^3$
    - Solution is X + N*N*N.
- Example 2:
  - Problem: find the sum of A[1..N]
  - Decompose to:
    - X = sum of A[2..N] (sum of an array of one element less)
    - Solution is X+A[1];

Problem decomposition and recursion

- When a large problem can be solved by solving smaller problems of the same nature -- recursion is the nature way of implementing such a solution.

- Example:
  - Problem: find the sum of A[1..N]
  - Depose to:
    - X = sum of A[2..N] (sum of an array of one element less)
    - Solution is X+A[1];

Writing recursive routines

- Key step number 1: understand how a problem can be decomposed into a set of smaller problems of the same nature; and how the solutions to the small problems can be used to form the solution of the original problem.

- Example:
  - Problem: find the sum of A[1..N]
  - Decompose to:
    - X = sum of A[2..N] (sum of an array of one element less)
    - Solution is X+A[1];

Writing recursive routines

- Key step number 2: formulate the solution of the problem into a routine with proper parameters. The key is to make sure that both the original problem and the smaller subproblems can both be formulated with the routine prototype.
- Example:
  - Problem: find the sum of A[1..N]
  - Generalize the problem to be finding the sum of A[beg..end]
  - Decompose to:
    - X = sum of A[beg+1..end]
    - Solution is X+A[beg];
- Formulate the problem with a routine
  - sum(A, beg, end) be the sum of A[beg..end] (original problem)
  - sum(A, beg+1, end) is the sum of A[beg+1..end] (subproblem)

Writing recursive routines

- Key step number 2: formulate the solution of the problem into a routine with proper parameters (routine prototype). The key to the make sure that both the original problem and the smaller subproblems can both be formulated.

- Formulate the problem with a routine
  - sum(A, beg, end) be the sum of A[beg..end] (original problem)
  - sum(A, beg+1, end) is the sum of A[beg+1..end] (subproblem)
- Recursive function prototype:
  - int sum(int A[], int beg, int end);

Writing recursive routines

- Key step number 3: define the base case. This is often the easy cases when the problem size is 0 or 1.

- int sum(int A[], int beg, int end)
  - Base case can either be when the sum of one element or the sum of 0 element.
  - Sum of 0 element (beg > end), the sum is 0.

  - Write it in C++:
    - If (beg > end) return 0; ← this is the base case for the recursion.

Writing recursive routines

- Key step number 4: define the recursive case – this is logic to combine the solutions for smaller subproblems to form solution for the original problem.

  - Decompose to:
    - X = sum of A[beg+1..end]
    - Solution is X+A[beg];

  - Recursive case:
    - X= sum(A, beg+1, end);
    - Return X + A[beg];

    - Or just return A[beg] + sum(A, beg+1, end);

Writing recursive routines

- Put the routine prototype, base case, and recursive case together to form a recursive routine (sample1.cpp)

```
int sum(int A[], int beg, int end)
{
   if (beg > end) return 0;
   return A[beg] + sum(A, beg+1, end);
}
```

Trace the recursive routine

```
int sum(int A[], int beg, int end)
{
   if (beg > end) return 0;
   return A[beg] + sum(A, beg+1, end);
}
```

- Let A = {1,2, 3,4,5};

```
sum(A, 0, 4)
     A[0] + Sum(A, 1, 4)
              A[1] + Sum(A, 2, 4)
                       A[2] + sum (A, 3, 4)
                                A[3] + sum(A, 4, 4)
                                         A[4] + sum(A, 5, 4) ← sum(A, 5, 4) returns 0
                                         A[4] + 0 = 5 ← sum(A, 4, 4) returns 5
                                A[3] + 5 = 9 ← sum(A, 3, 4) returns 9
                       A[2] + 9 = 12 ← sum(A, 2, 4) returns 12
               A[1]+12 = 14  ← sum(A, 1, 4) returns 14
     A[0] + 14 = 15       ← sum(A, 0, 4)  returns 15
```

Trace the recursive routine

sum(A, 0, 4)
A[0] + Sum(A, 1, 4)
A[1] + Sum(A, 2, 4)
A[2] + sum (A, 3, 4)
A[3] + sum(A, 4, 4)
A[4] + sum(A, 5, 4) ← sum(A, 5, 4) returns 0
A[4] + 0 = 5 ← sum(A, 4, 4) returns 5
A[3] + 5 = 9 ← sum(A, 3, 4) returns 9
A[2] + 9 = 12 ← sum(A, 2, 4) returns 12
A[1]+12 = 14 ← sum(A, 1, 4) returns 14
A[0] + 14 = 15 ← sum(A, 0, 4) returns 15

Every recursive step, the program is one step closer to the base case→ it will eventually reach the base case, and the build on that solutions for larger problems are formed.

Recursion example 2

- Problem: Sort an array of A[beg..end]
- Decompose to:
  - Subproblem1: Sort the array of A[beg+1..end]
  - Subproblem2: insert A[beg] to the sorted A[beg+1..end]

- Function prototype:
  - void sort(A, beg, end); // sort the array from index beg to end
  - How to solve a subproblem: sort(A, beg+1, end)

  - int sort(int A[], int beg, int end);

Recursion example 2

- int sort(int A[], int beg, int end);
  - When the array has no items it is sorted (beg > end);
  - When the array has one item, it is sorted (beg==end);

  - Base case: if (beg>= end) return;
  - Recursive case, array has more than one item (beg < end)
    - Subproblem1: Sort the array of A[beg+1..end], how?

      sort(A, beg+1, end)
    - Subproblem2: insert A[beg] to the sorted A[beg+1..end]

      tmp = A[beg];

      for (i=beg+1; i<=end; i++) if (tmp > A[i]) A[i-1] = A[i];

      A[i-1] = tmp;

Recursion example 2, put it all together (sample2.cpp)

- void sort(int A[], int beg, int end) {

```
    if (beg>= end) return;
    sort(A, beg+1, end)
    tmp = A[beg];
    for (i=beg+1; i<=end; i++)
        if (tmp > A[i]) A[i-1] = A[i];
        else break;
    A[i-1] = tmp;
}
```

Recursion example 3

- Example 1:
  - Problem (size = N): Compute $\sum_{i=1}^{N} i^3$
  - Depose to:
    - Subproblem (size = N-1): Compute X = $\sum_{i=1}^{N-1} i^3$
    - Solution is X + N*N*N.
- Function prototype:

  int sumofcube(int N);

  Base case: if (N=1) return 1;

  Recursive case: return N*N*N + sumofcube(N-1);

Recursion example 3 put it together

- Example 1:
  - Problem (size = N): Compute $\sum_{i=1}^{N} i^3$
  - Depose to:
    - Subproblem (size = N-1): Compute X = $\sum_{i=1}^{N-1} i^3$
    - Solution is X + N*N*N.

```
int sumofcube(int N) {
  if (N=1) return 1;
  return N*N*N + sumofcube(N-1);
}
```

Thinking in recursion

- Establish the base case (degenerated case) it usually trivial.
- The focus: if we can solve the problem of size N-1, can we use that to solve the problem of a size N?
  - This is basically the recursive case.
  - If yes:
    - Find the right routine prototype
    - Base case
    - Recursive case

Recursion and mathematic induction

- Mathematic induction (useful tool for theorem proofing)
  - First prove a ***base case (N=1)***
    - Show the theorem is true for some small degenerate values
  - Next assume an inductive hypothesis
    - Assume the theorem is true for all cases up to some limit (N=***k)***
  - Then prove that the theorem holds for the next value ***(N=k+1)***
  - ***E.g.***

$$\sum_{i=1}^{N} i = N(N+1)/2$$

- Recursion
  - Base case: we know how to solve the problem for the base case (N=0 or 1).
  - Recursive case: Assume that we can solve the problem for N=k-1, we can solve the problem for N=k.
- Recursion is basically applying induction in problem solving!!

Recursion – more examples

- void strcpy(char *dst, char *src)
  - Copy a string src to dst.

  - Base case: if (*src == '\0') *dst = *src; // and we are done
  - Recursive case:
    - If we know how to copy a string of one less character, can we use that to copy the whole string?

      - Copy one character (*dst = *src)
      - Copy the rest of the string ← a strcpy subproblem? How to do it?

Recursion – more examples (sample3.cpp)

```
void strcpy(char *dst, char *src) {
 if (*src == '\0') {*dst = *src; return;}
 else {
   *dst = *src;
   strcpy(dst+1, src+1);
 }
}
```

Recursion – more examples

void strlen(char *str)

If we know how to count the length of the string with one less character, can we use that the count the length of the whole string?

Recursion – more examples (sample4.cpp)

```
void strlen(char *str) {
  if (*str == '\0') return 0;
  return 1+ strlen(str+1);
}
```

Replace all 'X' in a string with 'Y'?

The treasure island problem (Assignment 6)

- N items, each has a weight and a value.
- Items cannot be splitted – you either take an item or not.
- Given the total weight that you can carry, how to maximize the total value that you take?
- Example
  - 6 items, 10 pounds at most to carry, what is the value?
    - Item 0: Weight=3 lbs, value = $9
    - Item 1: weight= 2lbs, value = $5
    - Item 2: weight = 2lbs, value = $5
    - Item 3: weight = 10 lbs, value = $20
    - Item 4: weight = 8 lbs, value = $16
    - Item 5: weight = 7 lbs, value = $11

The treasure island problem

- Thinking recursion:
  - If we know how to find the maximum value for any given weight for N-1 items, can we use the solution to get the solution for N items?

The treasure island problem

- Thinking recursion:
  - If we know how to find the maximum value for any given weight for N-1 items, can we use the solution to get the solution for N items?
    - We can look at the first item, there are two choices: take it or not take it.
      - If we take it, we can determine the maximum value that we can get by solving the N-1 item subproblem (we will use totalweight-item1's weight for the N-1 items)
        - item1.value + maxvalue(N-1, weight-item1.weight)
      - If we don't take it, we can determine the maximum value that we can get by solving the N-1 item subproblem (we will use totalweight on the N-1 items).
        - maxvalue(N-1, weight)
      - Compare the two results and decide which gives more value.

The treasure island problem

- If we know how to find the maximum value for any given weight for N-1 items, can we use the solution to get the solution for N items?

- Routine prototype:

  - int maxvalue(int W[], int V[], int totalweight, int beg, int end)
- Base case: beg > end, no item, return 0;
- Recursive case:
  - Two situations: totalweight < item1.weight, can't take the first item
  - Otherwise, two cases (take first item or not take first item), make a choice.

The treasure island problem

- How to record which item is taken in the best solution?
  - Use a flag array to record choices – this array needs to be local to make it easy to keep track.
    - Using global array would be very difficult, because of the number of recursions.

- Routine prototype:

- int maxvalue(int W[], int V[], int totalweight, int beg, int end, int flag[])
- int flag[] is set in the subroutine,
- Inside this routine, you needs to declare two flag arrays for the two choices
- You should then copy and return the right flag array and set the right flag value for the choice your make for the first item.

The treasure island problem

- int maxvalue(int W[], int V[], int totalweight, int beg, int end, int flag[]) {

```
int flag_for_choice1[42];
int flag_for_choice2[42];
….
….   maxvalue(W,V, totalweight-W[beg], beg+1, end, flag_for_choice1)
….
// copy one of flag_for_choice to flag
}
```

The number puzzle problem

- You are given N numbers, you want to find whether these N numbers can form an expression using +, -, *, / operators that evaluates to result.

- Thinking recursion:
  - Base case, when N=1, it is easy.
  - Recursive case: If given N-1 numbers, we know how to decide whether these N-1 numbers can form the expression that evaluates to result, can we solve the problem for N number?
    - We can reduce the N numbers problem to N-1numbers problem by picking two numbers and applying +, -, *, / on the two numbers (to make one number) and keep the rest N-2 numbers.

The number puzzle problem

- Recursive case: If given N-1 numbers, we know how to decide whether these N-1 numbers can form the expression that evaluates to result, Can we solve the problem for N number?
  - We can reduce the N numbers problem to N-1numbers problem by picking two numbers and applying +, -, *, / on the two numbers (to make one number) and keep the rest N-2 numbers.

```
for(i=0; i<N; i++)
  for (j=0; j<N; j++) {
    if (i==j) continue;
    // you pick num[i] and num[j] out
    // if (N-1 numbers) num[i]+num[j], and all num[x], x!=i, j can form the solution, done
           (return true)
    // if num[i]-num[j], and all num[x], x!=i, j can form the solution, done
   // if num[i]*num[j], and all num[x], x!=i, j can form the solution, done
    // if num[i]/num[j], and all num[x], x!=i, j can form the solution, done
  }
 return false.
```

The number puzzle problem

- To print out the expression
- You can associate an expression (string type) with each number (the expression evalutes to the number). You can print the expression in the base case, when the solution is found.
- The expression is in the parameter to the recursive function.

- Potential function prototype

```
bool computeexp(int n, int v[], string e[], int res)
```