Recursion

recursion

- A recursive definition is one which is defined in terms of itself.

Example 1

- factorial function:
- n! = 1 if (n =0) anchor (base case)
- = n* (n-1)! if (n>0) (recurrence)

```
unsigned int factorial(unsigned int)
{
   if (n == 0)
          return 1;
   else
          return n*factorial(n-1);
}
```

- 5! = 5*4!
- = 5*4*3!
- …
- = 5*4*3*2*1*1
- The repetitive steps eventually lead to the anchor (base case), the last step of recursive calls.

Example 2

- A phrase is a “palindrome” if the first and last letters are the same, and what’s inside is itself a palindrome (or empty)
- A phrase is a “palindrome” if it reads the same backwards as forwards (non-recursive definition)

Palindrome

- How do we write a function that reflects the recursive definition?

```
bool palindrome(const vector<char> w)
{           return palindrome(w, 0, w.size() - 1);         }

bool palindrome(const vector<char> w, int left, int right)
{
        //what goes here?

}
```

solution

```
bool palindrome(const vector<char> w)
   {          return palindrome(w, 0, w.size() - 1);          }

   bool palindrome(const vector<char> w, int left, int right)
   {          if (left >= right) return true;
              else if (w[left] != w[right]) return false;
              else return palindrome(w, left + 1, right – 1);
   }
```

What happened when the function is called?

- What information should be preserved when a function is called?

Calling function/Entering block

- Whenever a function is called (or {..} block is entered), a new "activation record" is created, containing:
  - A separate copy of all local variables and parameters
  - Control information such as where to return to
- The activation record is pushed onto the runtime stack when it is created

Activation record

1. Values for all parameters to the function, location of the 1^{st} cell if an array is passed or a variable is passed by reference, and copies of all other data items.
2. Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations.
3. Return address to resume control by the caller (the address of the caller’s instruction immediately after the call).
4. Dynamic link, a pointer to the caller’s activation record.
5. Returned value of the function is not declared as “void”.

Function returns/Block exits

- Activation record is alive until the function returns (or block exits)
- When a function returns/block exits, its activation record is removed (pop off) from the runtime stack.

The points reiterated …

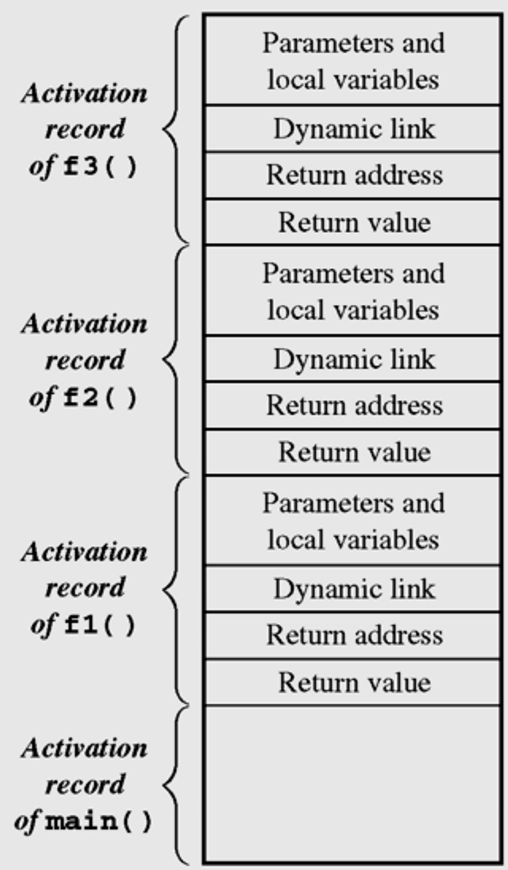
- Every time you call a function, you get a fresh copy of activation record
- If you call it recursively, you end up with more than one copy of the function active.
- When you exit a function, only that copy of it goes away.

Tracing function calls

- Create an activation record for each function call (activation), including local variables, parameters, return value, and return address. Push it onto the runtime stack.
- Pop off the activation record when function returns.

The stack of activation records

FIGURE 5.1 Contents of the run-time stack when `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`.



Infinite Recursion

- What will be the output of the following program?

```
int what(int n)
{
  return n * what(n-1);
}

int main()
{
  cout << what(3);
}
```

Infinite Recursion

- Must always have some way to make recursion to stop
- Otherwise it runs forever
  - Is the second bullet really accurate?

Using Recursion Properly

- For recursion to work properly, it needs two parts:
  - One or more base cases that are not recursive
  - One or more recursive cases that operate on smaller problems that get closer to a base case.
- The base case(s) should always be checked before the recursive calls .

Example 3

- Binary Search (Recursive Implementation)

Linear Search

- Given an array a of n integers, search for an element with value x.

```
int find(int a[], int n, int x)
{
   for(int i = 0; i < n; i++)
      if (a[i] == x)
              return i;

  return –1;
}
```

- How efficient does it run?

Binary Search

- If array is sorted, we can search faster
  - Start search in middle of array
    - If x is right there in the middle, done.
  - If x is less than middle element, need to search only in lower half
  - If x is greater than middle element, need to search only in upper half
- How efficient does it run? (Will prove it later.)

Binary Search

Find 26 in the following sorted array:

1 3 4 7 9 11 15 19 22 24 26 31 35 50 61

22 24 26 31 35 50 61

22 24 26

1 26

Binary Search (with helper function)

```
int find(int a[], int n, int x) {
        return findInRange(a, x, 0, n-1);
}
int findInRange(int a[], int x, int lo, int hi) {
        if (lo > hi) return –1;
        int mid = (lo+hi)/2;
        if (x == a[mid]) return mid;
        else if (x < a[mid])
                return findInRange(a, x, lo, mid-1);
        else return findInRange(a, x, mid+1, hi);
}
```

Kick-off and Helper Function

- Top-level “Kick-off” function
  - Not itself recursive
  - Calls the recursive helper function
  - Returns the ultimate answer
- Helper function
  - Contains the actual recursion
  - May require additional parameters to keep track of the recursion
- Client programs only need to call the kick-off function.

Recursion is not always good -- Excessive recursion

- Fibnocci:
  - Fib(n) = n if n <2
  - = Fib(n-2) + Fib(n-1) otherwise
  - Bad example to use recursion due to low efficiency.
    - 25 times to find the 7^{th} Fibnocci number
    - 1 quarter of a million calls to find the 26^{th} Fibnocci number;
    - 3 million calls to find 31^{st}
    - WHY?
      - No reuse of previous results.
      - Iterative algorithm is much better in this case.

Recursion vs. Iteration

- When to use recursion? – logical simplicity and readability
  - Processing recursive data structures
    - Array, linked list, trees, etc
  - Solving recursively defined problem
    - Palindrome, factorial, etc
  - Divide and Conquer algorithm
- When to use iteration instead?
  - When no obvious recursive solution exists
  - When iterative is a lot more efficient.

Recursion vs. Iteration

- In theory, any iteration can be written using recursion, and vice-versa
- Iteration is generally more efficient
  - Somewhat faster
  - Take less memory
- Recursion is generally more elegant

Pitfalls

- It is normally wrong to use loop (while, for statements) to control the recursion
- It is normally wrong to use static variables to control the recursion

Writing Recursion Function

- The General Strategy of Writing Recursive Functions
  - Decide if the base (stopping) case(s) is reached
  - If reached, solve it and you are done.
  - Otherwise, break the problem into smaller ones and solve them recursively, and then combine the solutions into the solution to the original problem.

Exercises

- Complete the following functions using both iteration and recursion:
  - int max(int *a, const int size): returns the index of the maximum integers in array a. size is the number of elements in the array.
  - int sum(int *a, const int size): returns the sum of elements in array a.
  - int search(int *a, const int size, int key) returns the index of the key if found, else returns –1.

Exercise

- Iteration and Recursion:
- int max(int *a, const int size)
- {
- }

Pseudocode for recursive version of max()

- If there is single element, return it.
- Else return the maximum of following two:
  - a) Last Element
  - b) Value returned by recursive call for n-1 elements.

Solution:

```
int max(int *a, const int size)
{
   if (size == 1) return a[0];
   int sub_max = max(a+1, size-1);
   if (a[0] > sub_max)
          return a[0];
   else
          return sub_max;
}
```

Complexity Analysis of Recursive Binary Search (using a version w/o kick-off/helper)

```
int bfind(int x, int a[], int n)
{   m = n / 2;
    if (n <= 1) return -1;
    if (x == a[m]) return m;
    if (x < a[m])
        return bfind(x, a, m);
    else
        return bfind(x, &a[m+1], n-m-1);
}
```

- Best case? 1
- Worst Case? **log(n)**, see next slides.
- Average Case? log(n). Analysis is not required. See details in the textbook.

Binary Search Analysis (using recurrent relation)

- One sub-problem, half as large
- Equation: **`(recurrence relation)`**
  - $T(1) \le b$ b is a constant
  - $T(n) \le T(n/2) + c$ for n>1

Solving Recursive Equations by Telescoping

$T(n)$	$= T(n/2) + c$		initial equation
$T(n/2) =$		$T(n/4) + c$	so this holds
$T(n/4) =$		$T(n/8) + c$	and this …
$T(n/8) =$		$T(n/16) + c$	and this …
…			
$T(4)$	$=$	$T(2) + c$	eventually …
$T(2)$	$=$	$T(1) + c$	and this …
$T(n)$	$=$	$T(1) + c\log n$	sum equations, canceling the terms appearing on both sides
$T(n)$	$=$	$\theta(\log n)$	

($T(n) = O(\log n)$, θ means tight bound, i.e. both upper and lower bound.)

(Optional) Solving Recursive Equations by Repeated Substitution

$$
\begin{aligned}
T(n) &= T(n/2) + c && \text{substitute for } T(n/2) \\
&= T(n/4) + c + c && \text{substitute for } T(n/4) \\
&= T(n/8) + c + c + c \\
&= T(n/2^3) + 3c && \text{in more compact form} \\
&= \ldots \\
&= T(n/2^k) + kc && \text{“inductive leap”}
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= T(n/2^{logn}) + c\,logn && \text{“}k = logn\text{: the steps that h} \\
&= T(n/n) + c\,logn \\
&= T(1) + c\log n = b + c\log n = \theta(\log n)
\end{aligned}
$$

Optional --- Revisit: Max Subsequence Sum

- Divide and conquer solution … O(nlogn).

Algorithm 3: Divide and Conquer

- Divide Part:
  - Split the problem into two roughly equal subproblems, which are then solved recursively.
- Conquer Part:
  - Patching together the two solutions of the subproblems with small amount of additional work.

Algorithm 3: Divide and Conquer

- Maximum subsequence sum problem
  - Divide the number sequence into two equal parts
  - The maximum subsequence sum can be found in one of three places:
    - Entirely in the left half of the input
    - Entirely in the right half of the input
    - Crosses the middle and in both halves

Algorithm 3: Divide and Conquer

- In first half: 6

First Half				Second Half			
4	-3	5	-2	-1	2	6	-2

Algorithm 3: Divide and Conquer

- An O(Nl*og*N) algorithm

Kick-off function

- int maxSubSum3(const vector<int> &a)
- {
-    return maxSumRec(a, 0, a.size()-1);
- }

```
int maxSumRec(const vector<int> &a, int left, int right)
{
   if (left == right)
            if (a[left] > 0)
                return a[left];
            else
                return 0;
  int center = (left + right)/2;
  int maxLeftSum = maxSumRec(a, left, center);
  int maxRightSum = maxSumRec(a, center+1, right);

 int maxLeftBorderSum = 0, leftBorderSum = 0;
 for (int I = center; I <= left; i--)
 {
    leftBorderSum += a[i];
    if (leftBorderSum > maxLeftBorderSum)
          maxLeftBorderSum = leftBorderSum;
 }

Int maxRightBorderSum = 0, rightBorderSum =0;
for (int j = center +1; j <= right; j++)
{
    rightBorderSum += a[j];
    if (rightBorderSum > maxRightBorderSUm)
              maxRightBorderSum = rightBorderSum;
}

return max3(maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSUm);
}
```

Total time

- $T(1)=1$
- $T(N) = 2T(N/2) + O(N)$