Recursion and Iteration

Function calls and Call Stack

- Modularity vs. Efficiency

Tail Calls and Tail Call Elimination

- Tail Recursion

Divide-and-Conquer

- Recursive vs. Iterative Implementations

Space Complexity and Time Complexity

- Relation between *the time taken to run a program* and *the space used by* it is not simple.
  - Each program is allocated Virtual Memory space (i.e. set of logical addresses)
  - These addresses are mapped partially to RAM (primary memory) and partially to disk (secondary memory)
    - Typical ratio of raw access times: 50ns for RAM to 10ms for disk (i.e. $1 / 10^5$)
    - In an OS with efficient buffering, I/O scheduling, and disk caching typical ratio of access times can be made 1 : 1000
  - So if space usage crosses the RAM capacity (or available RAM capacity)
    - then increase in space will drastically increase time taken

Modularity vs. Efficiency

- We structure our program into procedures (or functions) to make it modular:
  - and modularity brings in ease of maintenance and reuse.
- Once a procedure is defined
  - it can be used – i.e. called – as many times as needed and
  - its implementation can be changed without affecting its use
- On the other hand, each procedure call has an implementation overhead
  - the cost of pushing and popping call frames on the call stack (or activation stack)
    - typically several instructions (5 to 10) in most modern platforms
    - in addition to store parameters and local variables on the call frame

PROCEDURE CALLS AND CALL STACK

Consider the following program:

```
int PI = 3;
int a(int n) { return PI*n*n; }
int t(int n) { return a(4*n) + a(2*n); }
void main() { int x = t(5); print(x); return 1; }
```

Order of Call / Return: ---> main ---> t ---> a_1

<---

---> a_2

<---

<---

<---

Returns are reverse order of calls – LIFO order

PROCEDURE CALLS AND CALL STACK

Consider the following program:

```
int PI = 3;
int a(int n) { return PI*n*n; }
int t(int n) { return a(4*n) + a(2*n); }
void main() { int x = t(5); print(x); return 1; }
```

For each call there is – possibly - work to do after return; this is called a continuation:

t in main	**assignment to x; print;**
a_1 in t	**multiplication; call a_2 ; addition;**
a_2 in t	**addition**

For each procedure local variables (including parameters) must be saved somewhere –

this space is known as a call frame

PROCEDURE CALLS AND CALL STACK

Combining the arguments from the two previous slides, we need a call stack (a.k.a activation stack)

- a call results in pushing of a call frame on stack

- a return results in popping of the top of stack frame

Exercise: Draw the call stack at the marked points in the following program.

```
int PI = 3;
int a(int n) {  return PI*n*n; }
int t(int n) { return a(4*n ) + a(2*n ); }
void main() { int x = t(5);  print(x); return 1; }
```

TAIL CALLS AND TAIL-CALL ELIMINATION

- A *tail call* is a procedure call with an empty continuation.
  - i.e. after the call returns the enclosing procedure can also return (immediately, without any further computation).
- What is the implication (for the call stack)?
  - Callee procedure can overwrite the frame of the caller
    - i.e. local variables & parameters of the caller can be overwritten.
- Once the stack operations are taken care of,
  - the call becomes a jump to the start of the callee procedure.
- The combination of these two steps is called tail-call elimination i.e.
  - replacing the call with a jump and
  - overwriting the caller's frame

Tail Calls and Tail-Call Elimination

- Compilers may do automatic tail-call elimination.
- Exercise:
  - Find out the command line option in ***gcc*** for tail call elimination. (Refer to the man pages).
    - See specifically under the "-f" option, that lists all optimizations related to function calls.
  - Write a program with tail calls and non-tail calls
    - Compile the program separately with and without the tail-call-elimination option
    - Compare the code. (Use ***gcc*** to generate assembly code in .s files).
      - Question: What is the ***gcc*** option for this?

TAIL RECURSION

- A recursive procedure call that is also a tail call is referred to as ***a tail recursive call***.
  - What is the implication for the call stack?
    - Caller and callee are the same –
      - so overwriting the frame is same as updating the local variables and parameters.
    - One single call frame is enough irrespective of the number of (tail) recursive calls.
- The following procedure should run forever (without stack overflow):
  - void eternity() { eternity(); }
- Exercise: Verify this using ***gcc***. Of course you need to use the appropriate option.

CALLS WITH A CONTINUATION

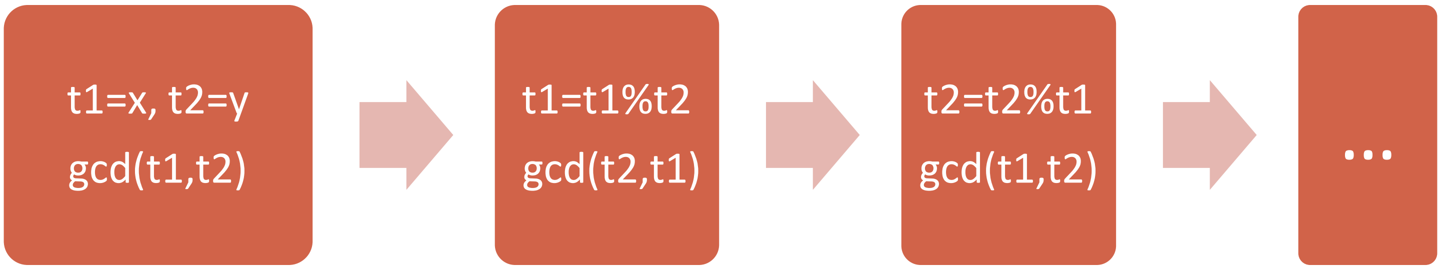
- Non-tail calls can be converted into tail calls:
- E.g. Consider the following code:
  - f(p1, p2) {   S1; g(e); S2; }
  - // S1 and S2 are statement w/o calls;
  - //S2 uses p2 but not p1 nor other local variables
  - g(p3) { S3; }
- This can be rewritten as
  - f(p1,p2) { S1; g1(e,p2); }
  - g1(p3,p2) { S3; S2; }
  - // You need to handle conflicts in local names
- There are a few other tricks! We'll see them soon!

RECURSIVE VS. ITERATIVE ALGORITHMS - EXAMPLE

- Problem: Compute the greatest common divisor (gcd) of two natural numbers.
- Known properties:
  - x|0 for any x, so, gcd(x,0) is x
  - gcd(x,y) is the same as gcd(y,x)
  - if d|x and d|y then d|(x-y)
    - By induction d|(x%y)
- Divide-and-Conquer strategy:
  - The sub problem for gcd(x,y) is
    - gcd(y, x%y)
  - The atomic subproblem is
    - gcd(x,0)

RECURSIVE VS. ITERATIVE ALGORITHMS - EXAMPLE

- Euclid's Algorithm - Divide and Conquer Design



Recursive Algorithm:

```
//Precondition:  x > y >= 0
//Postcondition:  returns z>0 such that z is gcd(x,y)
int gcd(int x, int y)
{
        if (y==0) return x; else return gcd(y, x%y);
}
```

Euclid's algorithm – Time Complexity

- When does the worst case happen?
  - Consider the case: $x \approx y$, then $(x \% y) \ll y$
  - Consider the case: $x \gg y$, then $(x\%y) \approx y \ll x$
    - i.e. either will lead to quick convergence of problem size to 0 i.e. they will not result in worst case behavior
- The worst case will happen when
  - neither of the above (conditions) is true for x and y
  - and that is also the case for the sub-problem inputs
    - i.e. y and x%y
- Fibonacci numbers fit the bill:
  - Consider $x==F_n$ and $y==F_{n-1}$, then the call sequence is:
  - $gcd(F_n, F_{n-1})$ --> $gcd(F_{n-1}, F_{n-2})$ --> $gcd(F_{n-2}, F_{n-3})$ --> …

Euclid's algorithm – Time Complexity

- Fibonacci numbers fit the bill:
  - Consider $x==F_n$ and $y==F_{n-1}$, then the call sequence is:
  - $gcd(F_n , F_{n-1})$ --> $gcd(F_{n-1} , F_{n-2})$ --> $gcd(F_{n-2} , F_{n-3})$ --> …
- So gcd(x,y) will terminate in k steps (worst case) where
  - $\mathbf{F_k <= x < F_{k+1}}$
- What is the value of k?
  - By solving the Fibonacci recurrence: k=ceil(log(x))
- Time Complexity of Euclid's algorithm is O(log(x))
  - This is true assuming the uniform cost model.
  - What is the complexity using the logarithmic cost model?
- Space Complexity (due to call stack) is O(log(x))

Recursive vs. Iterative Algorithms - Example

- Euclid's Algorithm - Divide and Conquer Design

t1=x, t2=y gcd(t1,t2) → t1=t1%t2 gcd(t2,t1) → t2=t2%t1 gcd(t1,t2) → …

Iterative Algorithm:

```
//Precondition:  x > y >= 0
//Postcondition:  returns z>0 such that z is gcd(x,y)
int gcd(int x, int y)
{
   int a=x, b=y, t;
   while (b!=0) {  // Loop Invariant: gcd(x,y) == gcd(a,b)
      t=a;    a=b;    b=t%b;
    }
   // Post-condition: gcd(x,y)=gcd(a,b)  and b=0
    return a;
 }
```

Time & Space Complexities?

Recursion and Iteration

Divide-and-Conquer

- Recursive vs. Iterative Implementations

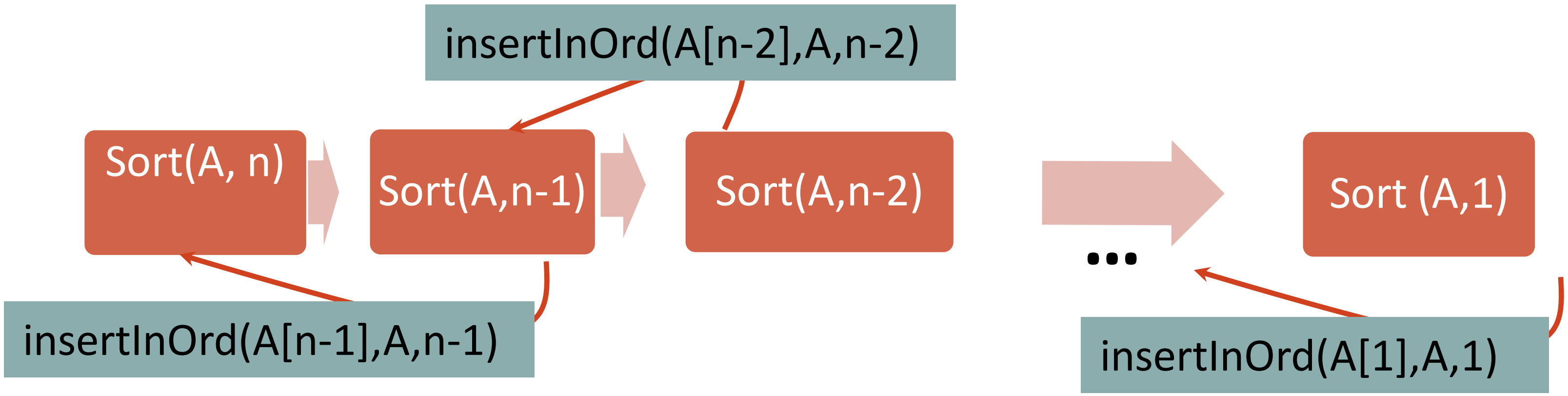
Recursive, Tail-Recursive and Iterative Procedures

QuickSort – Space Complexity,

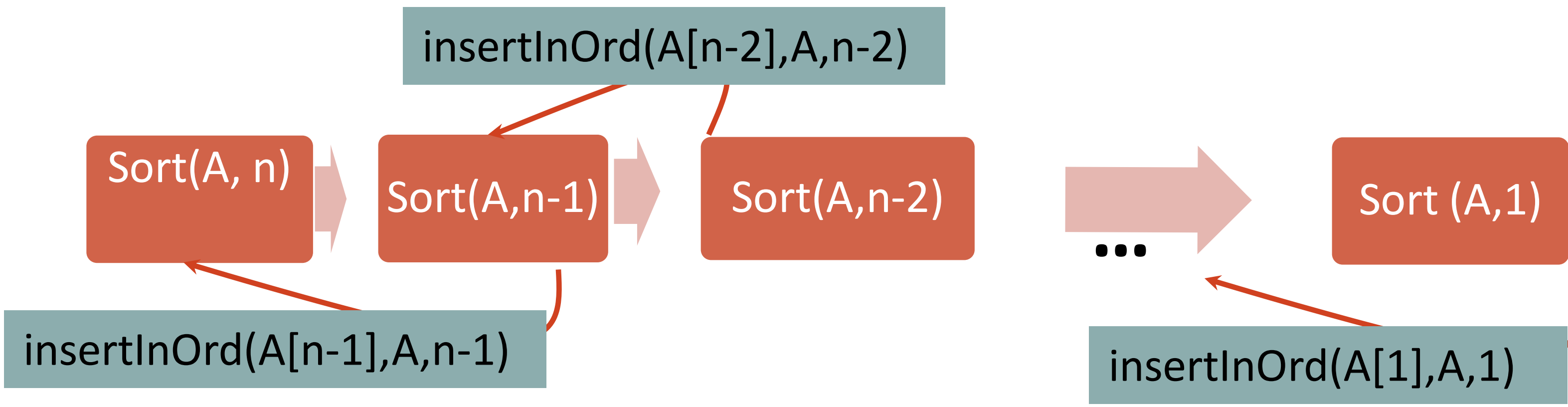
- Iterative Procedure with explicit stack
- Controlling Stack Size.

RECURSIVE VS. ITERATIVE ALGORITHMS – EXAMPLE 2

- Problem: Sort, in-place, a list of N elements.
  - Assume list is stored as an array A[0], A[1], … A[n-1]
- Design : Divide-And-Conquer
  - Sub-problem: Sort a list of size N-1 (A[0], A[1],…A[n-2])
  - Combination: Insert A[n-1] in order (i.e. in the right position)
  - Termination: Stop when size is <= 1.



RECURSIVE VS. ITERATIVE ALGORITHMS – EXAMPLE 2

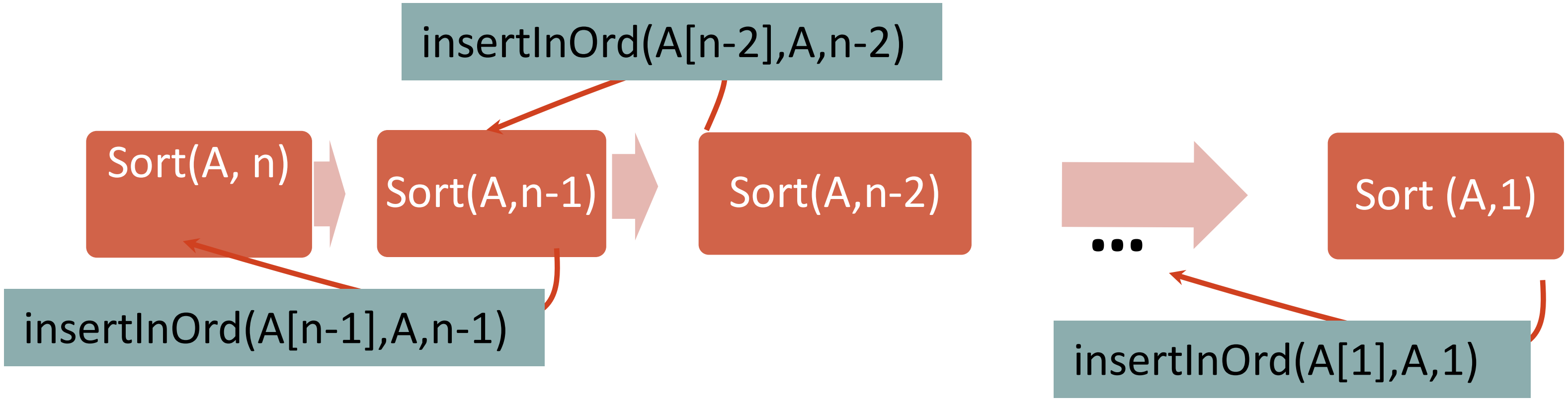


- Recursive Algorithm
  - // Precondition: A is an array of size n
  - // Postcondition: A is ordered in place

```
insertionSort(A, n) {
    if (n>1) { insertionSort(A,n-1);
               insertInOrder(A[n-1], A, n-1); }
}
```

Space Complexity?

RECURSIVE VS. ITERATIVE ALGORITHMS – EXAMPLE 2



- Iterative Algorithm

```
insertionSort(A,n) {
  for  j :=  1  to n-1    insertInOrder(A[j],A,j)       ;
}
```

Can be inlined. Find out option in ***gcc***.

```
// Pre-condition size of A is > last and A[last] can be overwritten
insertInOrder(v,A,last) {
   j := last;
   while (j>0) {  if (v<A[j]) { A[j] := A[j-1]; } else { A[j] := v;  return; }}
   A[0]  := v;
 }
```

DIVIDE-AND-CONQUER

- Recursive vs. Iterative Implementations
  - Is it possible / straight-forward to derive an iterative implementation from a divide-and-conquer design?
    - Under what conditions is this so?

Recursive and Tail-recursive Procedures – example

Problem: Find the length of an acyclic linked list.

```
// Pre-condition:
// ls is the head of an
// acyclic, singly linked list
len(ls) {
  if (ls == null) return 0;
  else return
       1 + len(ls->next);
 }
```

(Non-tail) Recursive Call

```
//Tail Recursive version of len
len(ls) {
  return len_tl(ls, 0);
}
// Pre-condition:
// acc == # nodes before ls
len_tl(ls, acc) {
   if (ls==null) return acc;
   else return
       len_tl(ls->next, acc+1);
}
```

Tail-Recursive Call

Tail Recursion Elimination - Example

```
len_tl(ls, acc) {
  if (ls==null) return acc;
  else return
    len_tl(ls->next, acc+1)
```

```
len_tl(ls, acc) {
  B:  if (ls==null) return acc;
  else {  // ls != NULL
      ls = ls->next;  acc = acc+1;
      goto B;
  }
}
```

```
len_tl(ls, acc) {
  while (ls!=null)  {
      ls = ls->next;
      acc = acc+1;
  }
  return acc;
}
```

Exercise: Argue that these 3 procedures are equivalent. End of Exercise.

[Hint: Use induction for formal proof. Use a flowchart for an informal argument. End of Hint.]

QuickSort - Call Stack Overhead

- QuickSort (recursive version)

```
void  qSort(Element ls[],
                int st, int en)
{
  if (st<en) {
    p = pivot(ls,en+1-st);
    pPos=part(ls, p, st, en);
    qSort(ls, st, pPos-1);
    qSort(ls, pPos+1, en);
  }
}
```

- Tail call elimination

```
void qSort(Element ls[],
           int  st, int en)
{
    while (st<en) {
      p=pivot(ls, en+1-st);
      pPos=part(ls, p,st, en);
      qSort(ls, st, pPos-1);
      st = pPos+1;
    }
}
```

QuickSort – Recursion Elimination

- QuicSort w/o tail call

```
void qSort(Element ls[],
              int  st, int en)
{
     while (st<en) {
       p=pivot(ls, en+1-st),
       pPos=part(ls, p,st, en);
       qSort(ls, st, pPos-1);
       st = pPos+1;
       }
    }
```

- QuickSort with Explicit Stack

```
void qSort(Element ls[], int st, int en)
    {
       //?????? What goes here?????
       while (st<en) {
            p = pivot(ls, en+1-st);
            pPos=part( ls, p,start, end);
            s = push(s, (st, pPos-1));
            st = pPos+1;
          }

    }
```

QuickSort – Recursion Elimination

- QuicSort w/o tail call

```
void qSort(Element ls[],
               int  st, int en)
{
     while (st<en) {
       p=pivot(ls, en+1-st),
       pPos=part(ls, p,st, en);
       qSort(ls, st, pPos-1);
       st = pPos+1;
       }
     }
```

- QuickSort with Explicit Stack

```
void qSort(Element ls[], int st, int en)
    {
     s = newStack();
     s = push(s, (st,en));
     while (!isEmptyStack(s)) {
       (st,en)=top(s);   s=pop(s);
       while (st<en) {
           p = pivot(ls, en+1-st);
           pPos=part( ls, p,start, end);
           s = push(s, (st, pPos-1));
           st = pPos+1;
         }
       }
    }
```

QuickSort – Space Complexity

- 1. Avoid putting trivial lists on stack:
  - i.e. push only if *start* < *end*
- 2. Put the smaller list on top of the larger list after each partitioning
  - Every list is above a list that is (at least) twice as large
    - i.e. at most logN items on stack at any time if N is initial size.

QuickSort – Controlling Stack Size

- QuickSort – Small Lists on Top

```
void qSort(Element ls[], int st, int en)
{
   s = newStack();  s = push(s, (st, en));
   while (!isEmptyStack(s)) {
      (st,en) = top(s); s = pop(s);
       while (st<en) {  p = pivot(ls, en+1-st);
        pPos = part(ls, p, st, en);
        if (pPos-st > en-pPos)      {
                 s = push(s, (st, pPos-1));
                 st = pPos+1;  // end = end;
         } else {
                s = push(s, (pPos+1, en));
                en = pPos-1;   // start = start;
  }}}}
```