COMP 208
Computers in Engineering

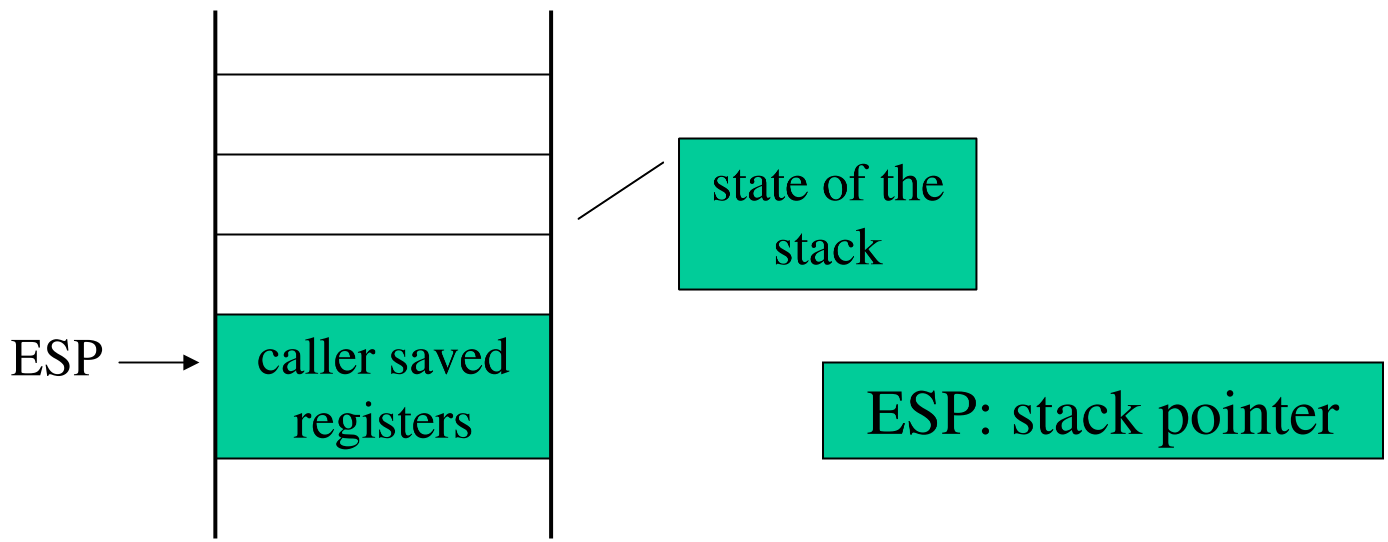
Lecture 19

Jun Wang
School of Computer Science
McGill University

Fall 2007

C function call mechanism (1/5)

- When calling a function (on Intel processors):
  1. Caller pushes some registers (caller-saved registers) on stack

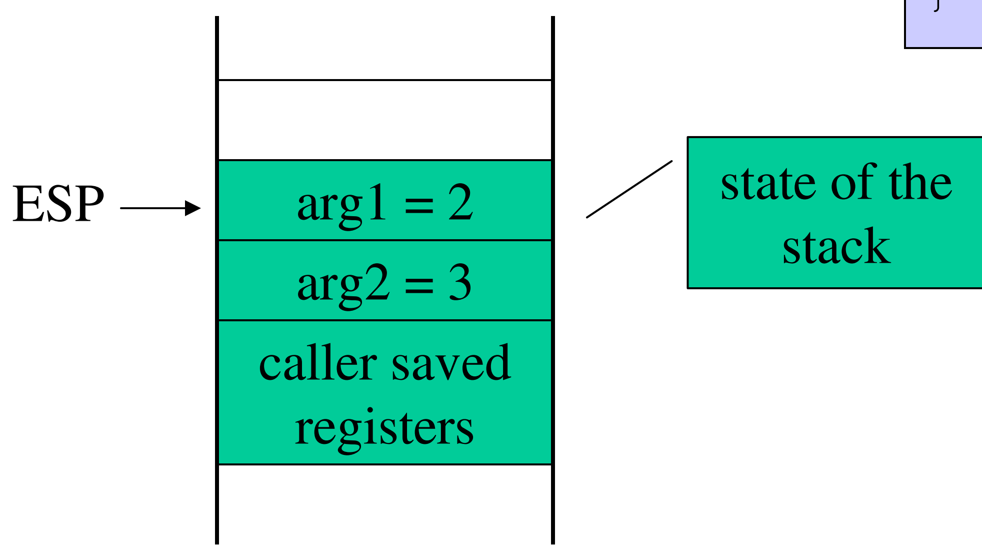


for more info:
www.cs.umbc.edu/~chang/cs313.s02/stack.shtml

C function call mechanism (2/5)

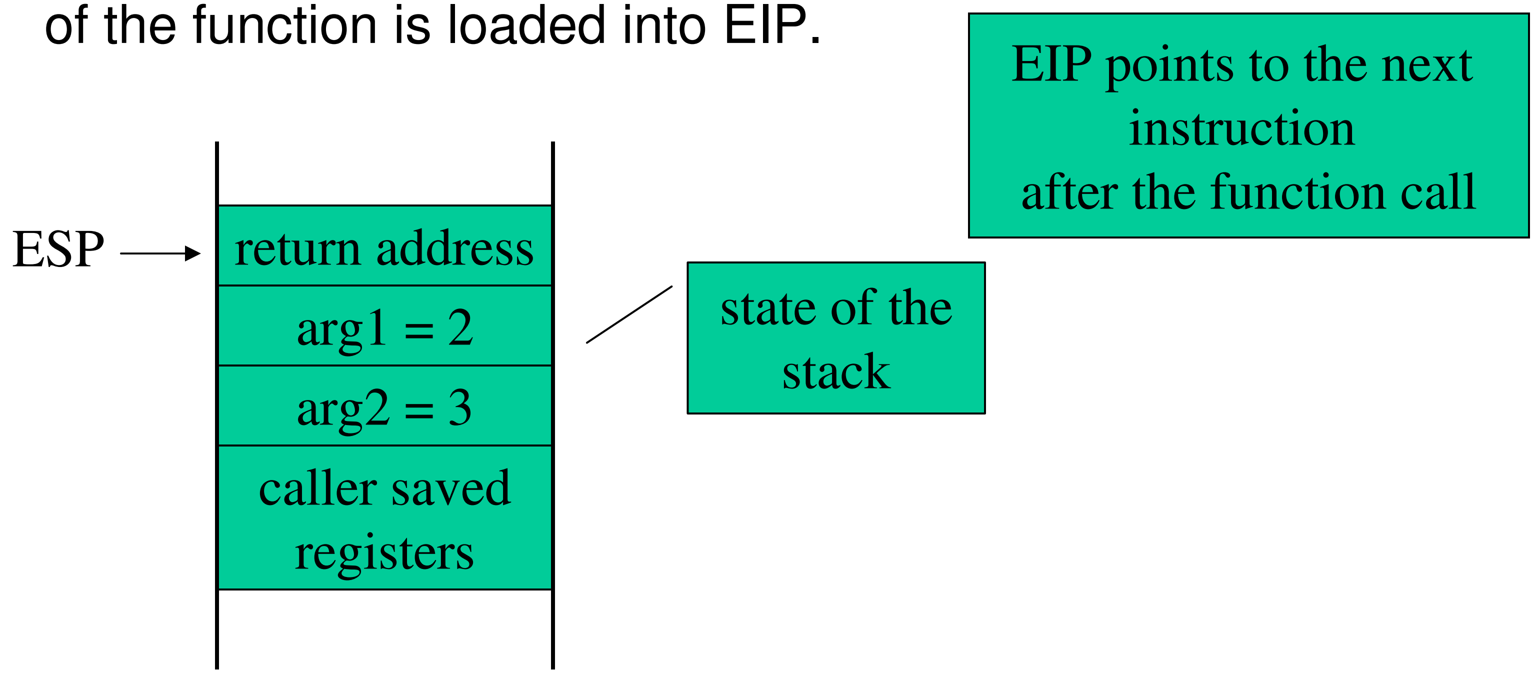
2. Caller pushes actual parameters on stack: last argument first

```
int sum(int a, int b)
{
  int result = 0;
  result = a+b;
  return result;
}
int main()
{
  int x;
  x = sum(2, 3);
  x = x+1;
}
```



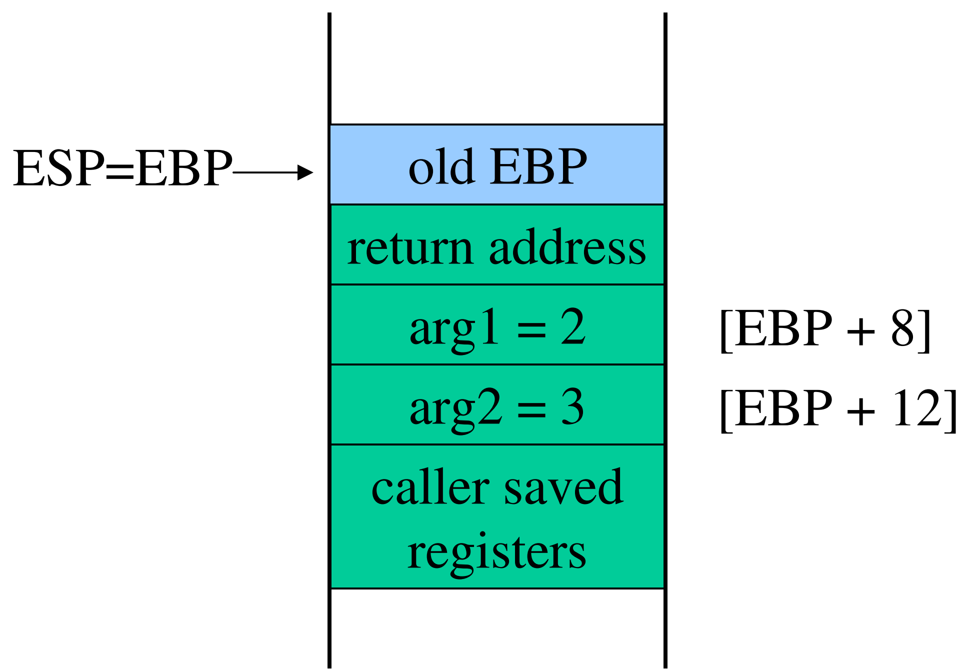
C function call mechanism (3/5)

3. Caller executes the `CALL` instruction, which pushes the EIP (instruction pointer, holding the address of next instruction) register on stack. This is known as the **return address**. Then the address of the 1st instruction of the function is loaded into EIP.



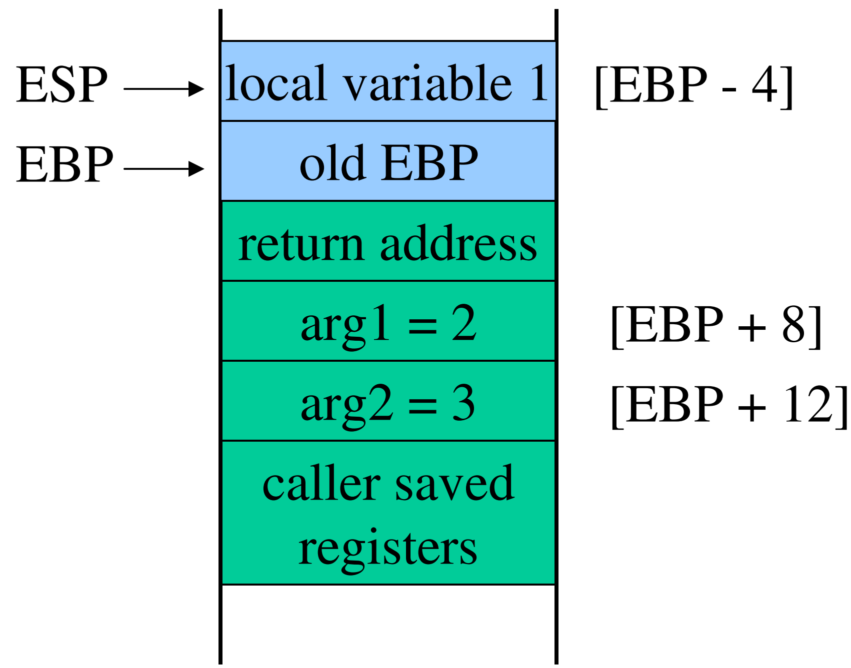
C function call mechanism (4/5)

- Now control is in callee:
  1. Callee pushes EBP on stack, and assigns ESP to EBP. Now, the 1st argument is at [EBP+8], the 2nd argument at [EBP+12], and so on.



C function call mechanism (5/5)

2. Callee allocates storage for local variables on stack: the 1st local variable is at [EBP-4], the 2nd at [EBP-8], and so on.



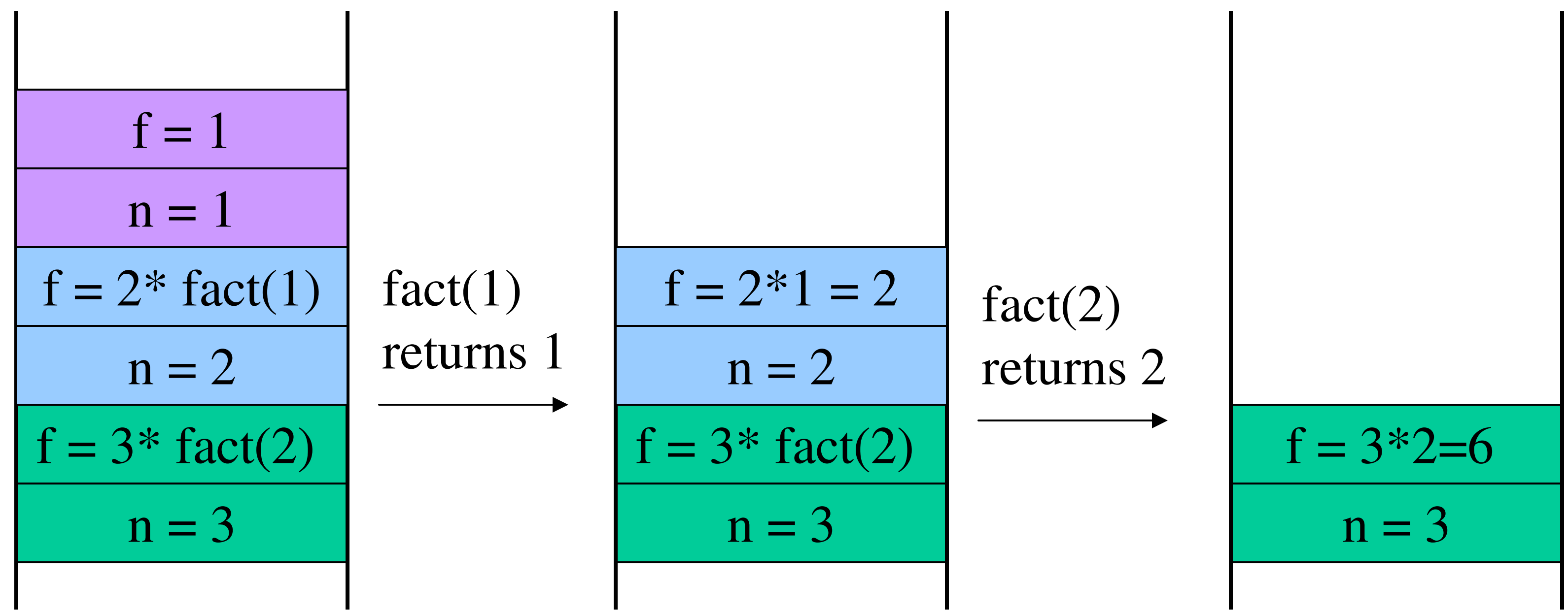
This is termed a stack frame or activation record.
For every function invocation, a stack frame is created.

Stack frames in recursive function calls

- The stack frames here are simplified. Only arguments and local variables are shown.
- Calling `fact(3)`

```
int fact(int n)
{
  int f;
  if(n <= 1)
    f = 1;
  else
    f = n * fact(n-1);

  return f;
}
```



The Merge Sort Shell Again

Now that we have seen the use of dynamic memory allocation, let's have another look at the mergesort shell.

```
void merge_sort(int arr[], int size){
  // Allocate the temporary array.
  int *temporary =
             (int *) malloc(size * sizeof (int));
  // Start the recursive sort.
  _merge_sort(arr, size, temporary);
  // Free the allocated array.
  free(temporary);
}
```

Merge Sort Itself

```
void _merge_sort(int arr[], int size, int temporary[]){

  int half = size / 2;
  int i;

  if(size <= 1) return; //base case

  _merge_sort(arr, half, temporary);
  _merge_sort(arr + half, size-half, temporary + half);
  merge(arr, half, arr + half, size - half, temporary);

  for (i=0;i<size;i++)
    arr[i] = temporary[i];
}
```

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781

Merging Two Sorted Lists

Sorted Array a:

781 **8641** 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 **6223**

Merging Two Sorted Lists

Sorted Array a:

781 **8641** 9819 14287 15229 21020 24044

Sorted Array b:

6223 **11311** 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 6223 **8641**

Merging Two Sorted Lists

Sorted Array a:

781 8641 **9819** 14287 15229 21020 24044

Sorted Array b:

6223 **11311** 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 6223 8641 **9819**

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 **14287** 15229 21020 24044

Sorted Array b:

6223 **11311** 14153 15751 17411 21626 28948 32560 32765

Merged Array:

781 6223 8641 9819 **11311**

Merging Two Sorted Lists

We continue in this way until one of the lists is exhausted.

Then just fill in the rest of the merged list with the remaining values.

Merging Two Sorted Lists

```
Sorted Array a:
   781 8641 9819 14287 15229 21020 24044
Sorted Array b:
   6223 11311 14153 15751 17411 21626 28948
     32560 32765
Merged Array:
   781 6223 8641 9819 11311 14153 14287
     15229 15751 17411 21020 21626 24044
```

Merging Two Sorted Lists

Sorted Array a:

781 8641 9819 14287 15229 21020 24044

Sorted Array b:

6223 11311 14153 15751 17411 21626 **28948 32560 32765**

Merged Array:

781 6223 8641 9819 11311 14153 14287 15229 15751 17411 21020 21626 24044

28948 32560 32765

Merge

```
void merge(int left[], int left_size, int right[],
           int right_size, int destination[]){

  int left_i = 0, right_i = 0, destination_i = 0;

  while((left_i < left_size) && (right_i < right_size))
     if(left[left_i] < right[right_i])
        destination[destination_i++] = left[left_i++];
     else
        destination[destination_i++] = right[right_i++];

  while(left_i < left_size)
     destination[destination_i++] = left[left_i++];
  while(right_i < right_size)
     destination[destination_i++] = right[right_i++];
}
```

Merge Sort (Variation)

```
static void _merge_sort(int arr[], int size, int temporary[])
{
  int half = size / 2;

  if(size <= 1) return;

  _merge_sort(arr, half, temporary);

  _merge_sort(arr + half, size - half, temporary + half);

  merge(arr, half, arr + half, size - half, temporary);

  memcpy(arr, temporary, size * sizeof (int));

  return;
}
```

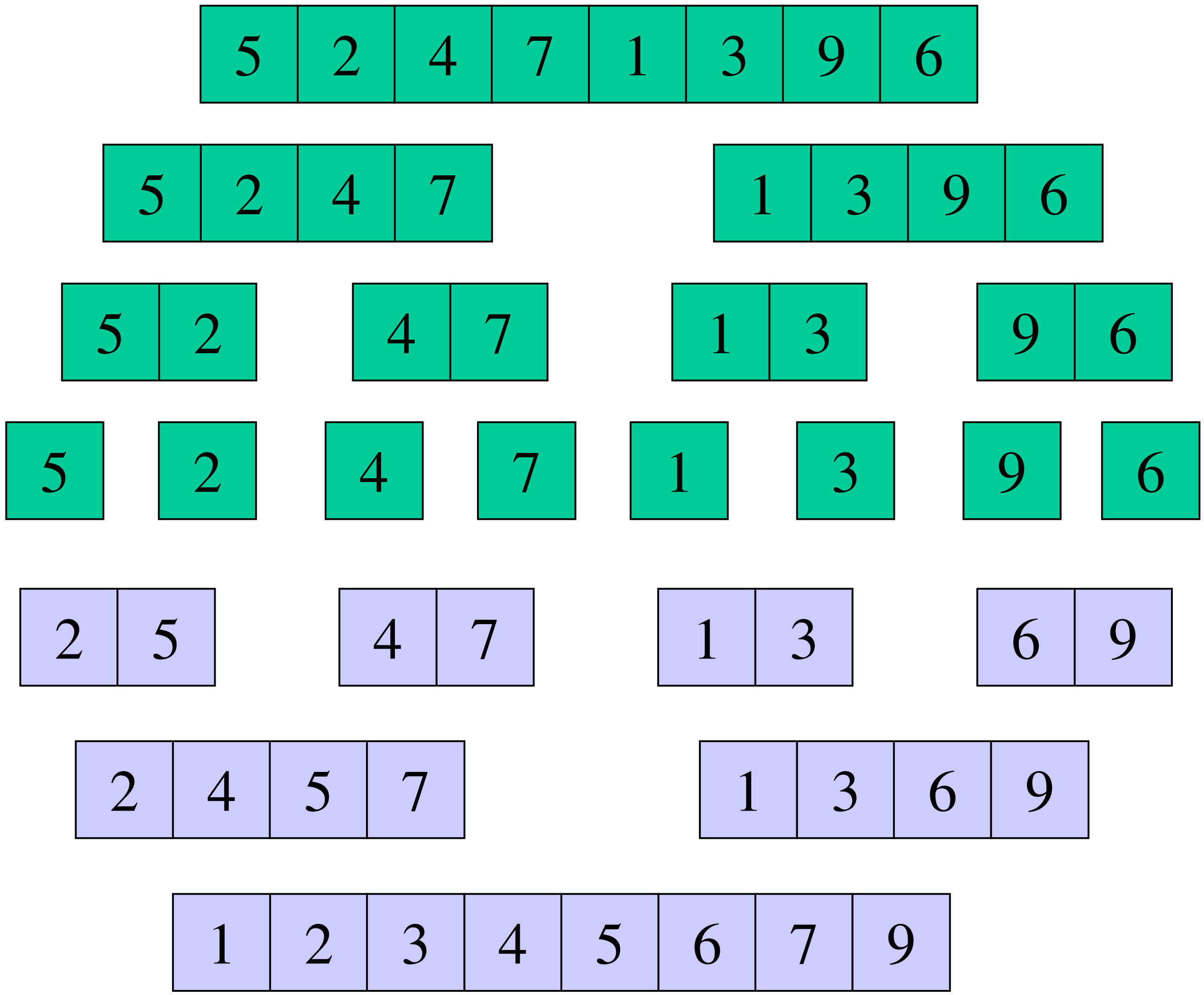
What's so great about mergesort?

- Insertion sort, Selection sort, Bubble sort all take time $O(n^2)$ to sort n values.
- The call tree for mergesort shows that it takes $O(n \log n)$ time.
- For large data sets that is a tremendous improvement
- Mergesort is one of a group of very efficient sorting algorithms that are used in most applications.

McGill

COMP 208 – Computers in Engineering

Merge sort example



Root Finding

Nathan Friedman

Fall, 2007

Root Finding

- Many applications involve finding the roots of a function $f(x)$.

- That is, we want to find a value or values for x such that $f(x)=0$

Roots of a Quadratic

We have already seen an algorithm for finding the roots of a quadratic

We had a closed form for the solution, given by an explicit formula

There are a limited number of problems for which we have such explicit solutions

Root Finding

- What if we don't have a closed form for the roots?

- We try to generate a sequence of approximations $x_1, x_2, \ldots, x_n$ until we (hopefully) obtain a value very close to the root

Example: Firing a Projectile

Find the angle at which to fire a projectile at a target

Given:

- the velocity, v
- the distance to the base of the target, x
- the height of the target, h

Find: the angle at which to aim, a

Example: Firing a Projectile

The physics of the problem tells us that

```
h = v sin a t – ½ g t²
t = x / (v cos a)
```

where g is the gravitational constant.

Example: Firing a Projectile

Taking the equations:

$h = v*\sin(a)*t - \frac{1}{2} g*t^2$

$t = x / (v*\cos(a))$

By substituting, we have

$h = x*\tan(a) - 0.5*g*(x^2/(v^2 \cos^2(a))$

The angle a is a root of

$f(a) = x*\tan(a)-0.5*g*(x^2/(v^2*\cos^2(a))-h$

The Bisection Method

- We start with an interval that contains exactly one root of the function
- The function must change signs on that interval.
- (If the function changes signs, in fact there must be an odd number of roots in the interval)

The Bisection Method

- To get started, we must bracket a root
- How do we bracket the root?
  - From our knowledge of the function
  - By searching along axis at fixed increments until we find that the sign of $f(x)$ changes

The Bisection Method

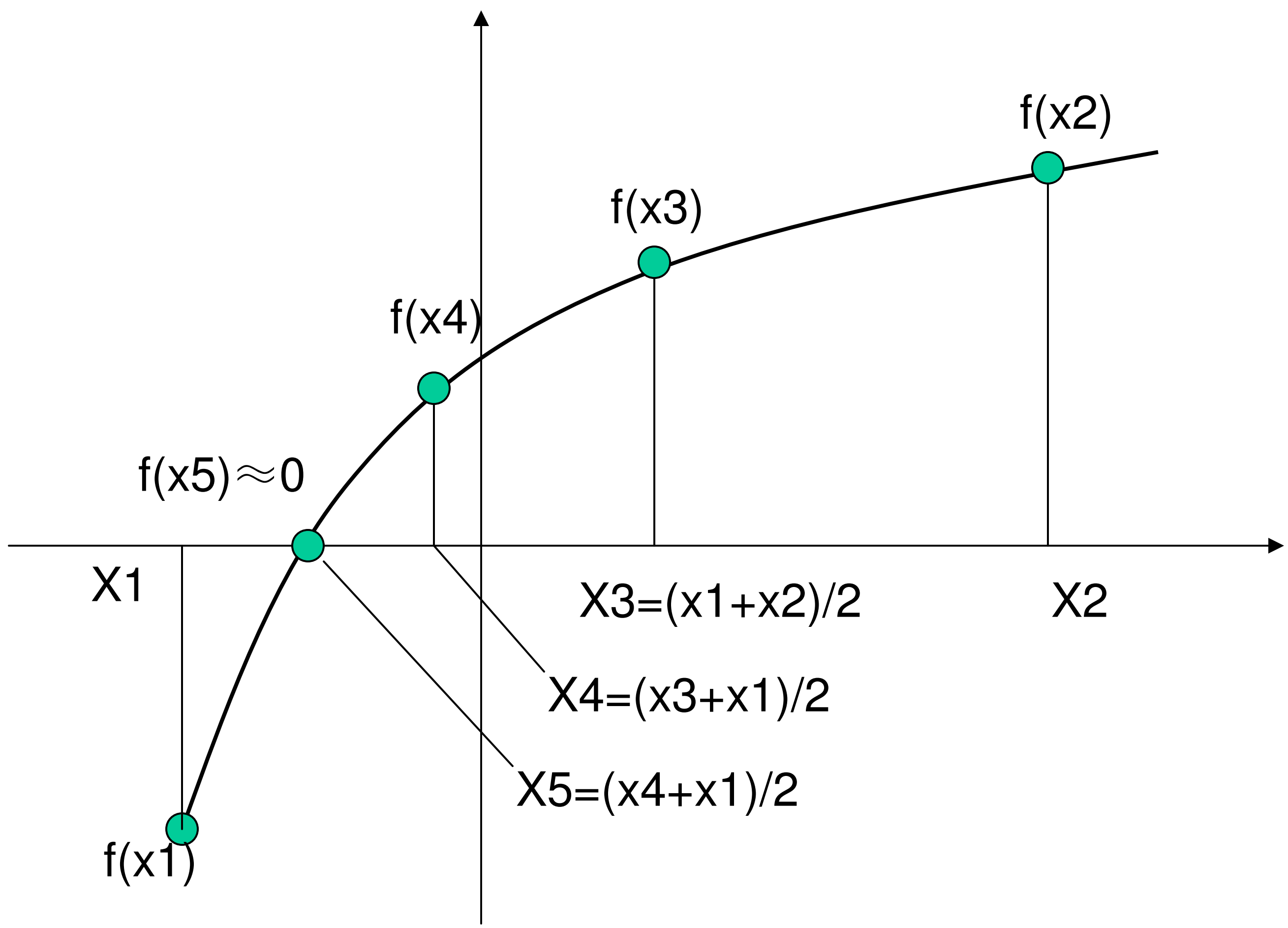
- Once we have an interval containing the root(s), we narrow down the search
- Similar to binary search: divide the interval in half and look for a sign change in one of the two subintervals half of the interval.
- From the **Intermediate Value Theorem**, if there is a sign change in an interval, there must be a root in that interval

The Bisection Method

- We check the first subinterval for a change in sign.
- If there is one, that interval must have a root.
- If there is no change in sign, a root must be in the other half

- When do we stop?
  - If the length of the interval is very small, we must be close to the root.
  - Just take the midpoint as the approximation

Bisection example



Convergence Condition

- Root finding algorithms compute a sequence of approximations to the root, r, of f:

$x_1, x_2, \ldots, x_i, \ldots$

- When does the bisection method stop?
- We know the root must be between x_i and x_j.
- When these values are very close, we must be close to the root.

Function Arguments

- We want to write a bisection function that takes a function as an argument and returns a root of the function
- How can a function be an argument?
- We have to go back to first principals

Function Arguments

- How can a function be an argument?
  - The code defining a function is stored in memory, just like data
  - It has an address just like any block of memory cells
  - We can pass the address of that code
  - We just have to be careful about the type of the pointer

Bisection Header

- To define the bisection function we can use the header:

```
double bisection_rf(double (*f) (double),
          double x0, double x1, double tol)
```

```
double (*f) (double)
```

defines `f` as a pointer to a function that takes one double parameter and returns a double value.

Function pointer

- Just like `int*` is a pointer to an integer variable, a function pointer points to a certain type of functions.

```
void foo(int x)
{ printf("%d\n", x); }

int bar(int x)
{ return 2*x; }
```

```
int sum(int a, int b)
{ return a+b; }

int sub(int a, int b)
{ return a-b; }
```

```
int main() {
  void (*f) (int);
  // f is a pointer to a function that takes 1 int parameter and has no return value.
  // Note that *f must be inside ().
  // void *f (int); // this is a function prototype!

  int (*g) (int, int);
  // g is a pointer to a function that takes 2 int parameters and returns int value.

  f = foo;  //ok
  f = bar;  // error! type not match!
  g = sum; // ok!
  printf("%d\n", g(2,3));  // prints 5
  g = sub;
  printf("%d\n", g(2,3);  // prints -1
}
```

Function pointer

- A function pointer can only points to certain type of functions, just like int* and char* are 2 different types.

- When assign a function's address to a function pointer, we **don't have to** use the address-of operator &.

```
//in previous slide, the following are the same
g = sum;
g = &sum;
```

- A function can be called through a pointer that points to that function. And we **don't have to** use the de-referencing operator *.

```
g = sum;
x = g(2, 3);
x = (*g)(2, 3); //same as above
x = *g(2, 3); //error! () has
        //higher precedence than *
```

Bisection Header

C provides a `typedef` declaration that can simplify this code:

```
typedef double (*DfD) (double);

double bisection_rf(DfD f, double x0,
  double x1, double tol)
```

The Bisection Method

```
typedef double (*DfD) (double);

double bisection_rf(DfD f, double x0, double x1,
                    double tol){
   double middle = (x0 + x1) / 2.0;

   if ((middle - x0) < tol)  //tol is tolerance
     return middle;
   else if (f(middle) * f(x0) < 0.0)
     return bisection_rf(f, x0, middle, tol);
   else
     return bisection_rf(f, middle, x1, tol);
   }
```

Other Convergence Conditions

There are typically three ways of determining when to stop

1. $f(x_i)$ is close to zero
2. $f(x_i)$ is close to r
3. x_i is close to x_{i+1} so it doesn't pay to continue

The Secant Method

- We also begin with two initial approximations
- However they do not have to bracket the root
- We essentially approximate the function using straight lines forming the secant at the two points
- This is probably the most popular method
- It is not guaranteed to converge to the root

The Secant Method

- Start with two values, x_0 and x_1 that don't necessarily bracket the root
- Compute a new approximation, the point at which the line drawn between $f(x_0)$ and $f(x_1)$ intersects the x-axis

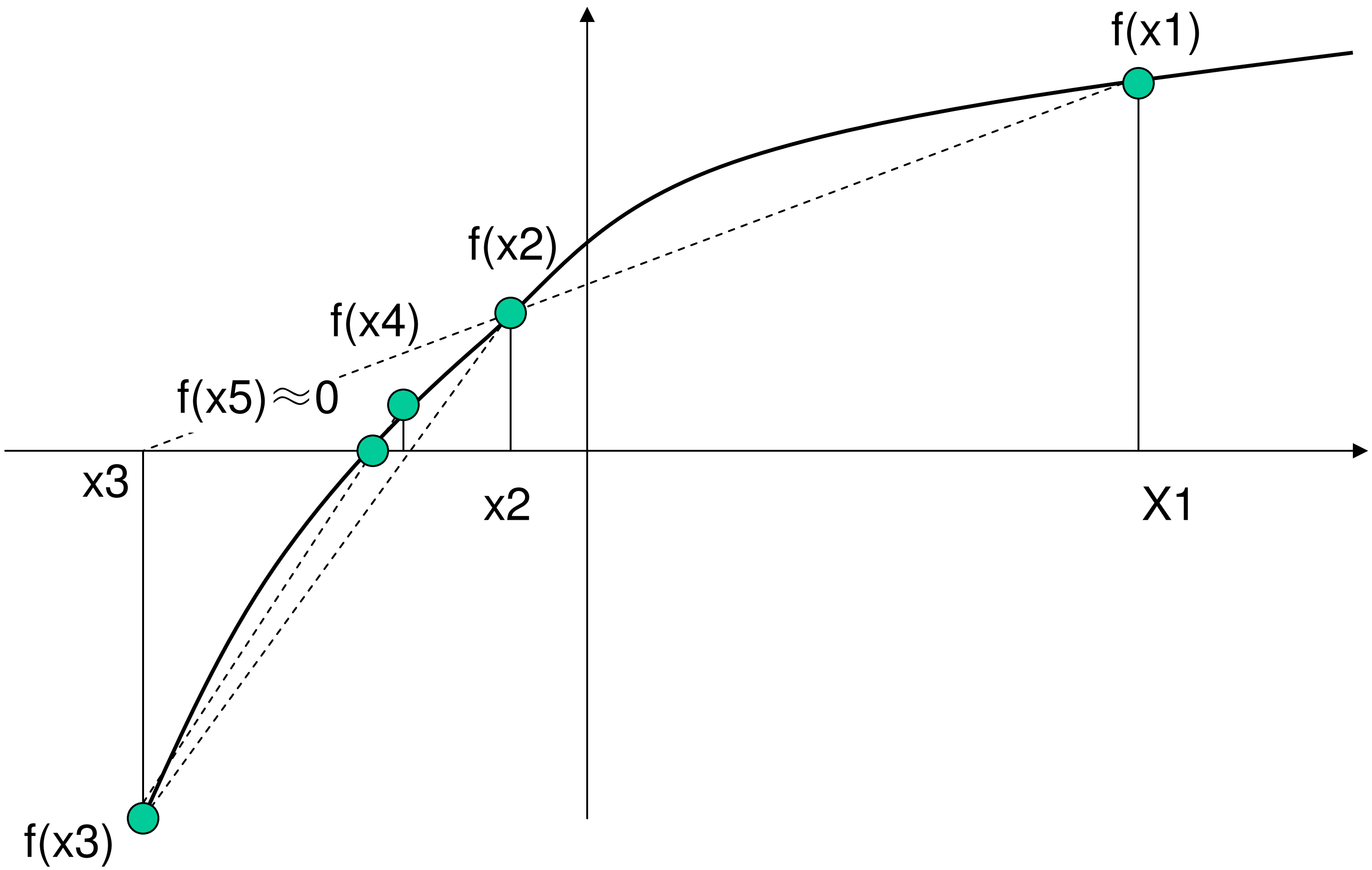
Computing the Approximation

The approximation is given by:

```
x2 = f(x0)*(x1-x0)/(f(x0)-f(x1))+x0
```

Iterate this process using x_1 and x_2 as the new pair of points

Secant method



Convergence Criteria

- When do we stop this process?
- We use the first of the criteria we described
- That is, we stop when the value of $f(x_i)$ is close to zero
- We then say that x_i is an approximate root

The Secant Method

- This method is one of the most popular ones in use
- It may not converge because successive intervals become larger or because it oscillates
- Therefore we terminate the algorithm after a specified number of steps if it has not converged

The Secant Method

```
double secant_rf(DfD f, double x1, double x2,
                        double tol, int count){
   double f1 = f(x1), f2 = f(x2),
          slope = (f2 - f1) / (x2 - x1),
          distance = -f2 / slope,
          point = x2 + distance;
   if(!count)
       return point;
   if(fabs(f(point)) < tol)
       return point;
   return secant_rf(f, x2, point, tol, count - 1);
}
```