Computer Programming: Skills & Concepts (CP)
Recursion (including `mergesort`)

Ajitha Rajan

Tuesday 14 November 2017

Today's lecture

- Recursion: functions that call themselves.
- Examples: `factorial` and `fibonacci`.
- The `long` integer type.
- MergeSort (recursive version).
- Allocating memory dynamically with `calloc`.

Computing Factorial

Task: write a function that computes factorial

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Factorial with `for` loop

```
long factorial(int n) {
  long fact = 1;
  int i;
  for(i=1; i<=n; i++ ) {
    fact = fact * i;
  }
  return fact;
}
```

We use `long` (meaning 'long int') rather than `int`, because as `n` increases, `factorial(n)` can get very large - for example 13! is too large for an `int` variable (on DICE).

On DICE, `long` uses 64 bits - can store values up to $\pm 9.22337204 \times 10^{18}$. System dependent: On old machines, `long` might be 32 bits only.

Factorial with recursion

```
long factorial(int n) {
  if (n<=1) {
    return 1;
  }
  return n * factorial(n-1);
}
```

The function `factorial` calls itself!

Factorial with recursion

```
long factorial(int n) {
  if (n<=1) {
    return 1;
  }
  return n * factorial(n-1);
}
```

The function `factorial` calls itself!
A function is a 'black box' to which you give arguments, and it returns a result. Recursive calls are no different from any other call!

Factorial with recursion

```
long factorial(int n) {
  if (n<=1) {
    return 1;
  }
  return n * factorial(n-1);
}
```

The function `factorial` calls itself!
A function is a 'black box' to which you give arguments, and it returns a result. Recursive calls are no different from any other call!
`factorial(n)` needs to know what $(n-1)!$ is, so it just calls the black box `factorial(n-1)`.

Factorial with recursion

```
long factorial(int n) {
  if (n<=1) {
    return 1;
  }
  return n * factorial(n-1);
}
```

The function `factorial` calls itself!

A function is a 'black box' to which you give arguments, and it returns a result. Recursive calls are no different from any other call!

`factorial(n)` needs to know what $(n-1)!$ is, so it just calls the black box `factorial(n-1)`.

`factorial(1)` doesn't need to call anything.

Execution of recursion

If you look at how the sequence of execution goes, it's like this:

```
factorial(5)
    return 5 * factorial(4);
               return 4 * factorial(3);
                          return 3 * factorial(2);
                                     return 2 * factorial(1);
                                                return 1;
                                     return 2 * 1;
                          return 3 * 2
               return 4 * 6
    return 5 * 24
120
```

but just think in terms of the black box.

Fibonacci numbers

The Fibonacci numbers are the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

$\frac{F(n+1)}{F(n)}$ converges to the golden ratio $\frac{1+\sqrt{5}}{2} = 1.618034$.

Recursive computation of Fibonacci numbers

```
long fibonacci(int n) {
  if (n==0)
    return 0;
  if (n==1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

- How many function calls does it roughly take to compute `fibonacci(10)` or `fibonacci(100)`?
- Running time?
  - Is this faster/slower than the `for` and `while` versions from lecture 6?
  - Interesting comparison using `clock()` :-).
    more interesting than comparing linear against binary search.

Merge

Idea:

> Suppose we have two arrays `a, b` of length `n`; and `m` respectively, and that these arrays **are already sorted**. Then the merge of `a` and `b` is the sorted array of length `n+m` we get by walking through both arrays jointly, taking the smallest item at each step.

a

2	7	18

↑ i=0

b

4	5	6

↑ j=0

c

↑ k=0

Merge

Idea:

> Suppose we have two arrays `a, b` of length `n;` and `m` respectively, and that these arrays **are already sorted**. Then the merge of `a` and `b` is the sorted array of length `n+m` we get by walking through both arrays jointly, taking the smallest item at each step.

a

2	7	18

$\uparrow$ i=0

b

4	5	6

$\uparrow$ j=0

c

$\uparrow$ k=0

2	7	18

$\Uparrow$ i=0

4	5	6

$\uparrow$ j=0

2					

$\Uparrow$ k=0

Merge

Idea:

Suppose we have two arrays `a, b` of length `n;` and `m` respectively, and that these arrays **are already sorted**. Then the merge of `a` and `b` is the sorted array of length `n+m` we get by walking through both arrays jointly, taking the smallest item at each step.

a

2	7	18

↑ i=0

2	7	18

⇑ i=0

2	7	18

↑ i=1

b

4	5	6

↑ j=0

4	5	6

↑ j=0

4	5	6

↑ j=0

c

↑ k=0

2					

⇑ k=0

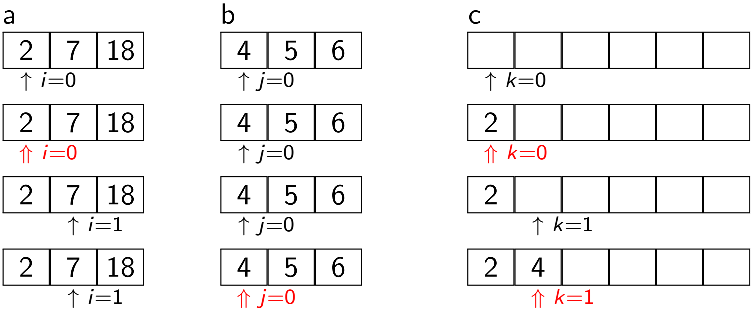
2					

↑ k=1

Merge

Idea:

Suppose we have two arrays `a, b` of length `n`; and `m` respectively, and that these arrays **are already sorted**. Then the merge of `a` and `b` is the sorted array of length `n+m` we get by walking through both arrays jointly, taking the smallest item at each step.



2	7	18
	↑ i=1	

4	5	6
	↑ j=1	

2	4				
		↑ k=2			

2	7	18
	↑ i=1	

4	5	6
	↑ j=1	

2	4				
		↑ k=2			

2	7	18
	↑ i=1	

4	5	6
	⇑ j=1	

2	4	5			
		⇑ k=2			

2	7	18
	↑ i=1	

4	5	6
	↑ j=1	

2	4				
		↑ k=2			

2	7	18
	↑ i=1	

4	5	6
	⇑ j=1	

2	4	5			
		⇑ k=2			

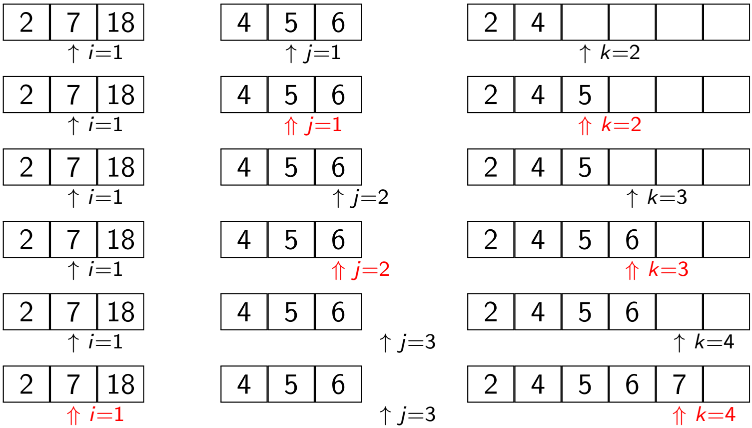
2	7	18
	↑ i=1	

4	5	6
		↑ j=2

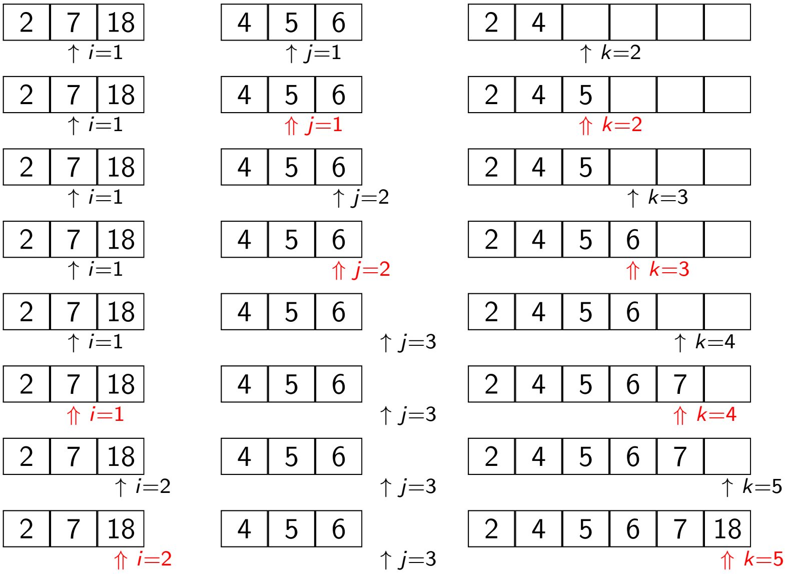
2	4	5			
			↑ k=3		

2 | 7 | 18	4 | 5 | 6	2 | 4 | | | |
↑ i=1 (under 7)	↑ j=1 (under 5)	↑ k=2
2 | 7 | 18	4 | 5 | 6	2 | 4 | 5 | | |
↑ i=1	⇑ j=1	⇑ k=2
2 | 7 | 18	4 | 5 | 6	2 | 4 | 5 | | |
↑ i=1	↑ j=2	↑ k=3
2 | 7 | 18	4 | 5 | 6	2 | 4 | 5 | 6 | |
↑ i=1	⇑ j=2	⇑ k=3

2	7	18	4	5	6		2	4				
	$\uparrow$ i=1			$\uparrow$ j=1					$\uparrow$ k=2			
2	7	18	4	5	6		2	4	5			
	$\uparrow$ i=1			$\Uparrow$ j=1					$\Uparrow$ k=2			
2	7	18	4	5	6		2	4	5			
	$\uparrow$ i=1				$\uparrow$ j=2					$\uparrow$ k=3		
2	7	18	4	5	6		2	4	5	6		
	$\uparrow$ i=1				$\Uparrow$ j=2					$\Uparrow$ k=3		
2	7	18	4	5	6		2	4	5	6		
	$\uparrow$ i=1					$\uparrow$ j=3					$\uparrow$ k=4	


2 7 18
↑ i=1
4 5 6
↑ j=1
2 4
↑ k=2
2 7 18
↑ i=1
4 5 6
⇑ j=1
2 4 5
⇑ k=2
2 7 18
↑ i=1
4 5 6
↑ j=2
2 4 5
↑ k=3
2 7 18
↑ i=1
4 5 6
⇑ j=2
2 4 5 6
⇑ k=3
2 7 18
↑ i=1
4 5 6
↑ j=3
2 4 5 6
↑ k=4
2 7 18
⇑ i=1
4 5 6
↑ j=3
2 4 5 6 7
⇑ k=4

2 | 7 | 18	↑ i=1	4 | 5 | 6	↑ j=1	2 | 4 | | | |	↑ k=2
2 | 7 | 18	↑ i=1	4 | 5 | 6	⇑ j=1	2 | 4 | 5 | | |	⇑ k=2
2 | 7 | 18	↑ i=1	4 | 5 | 6	↑ j=2	2 | 4 | 5 | | |	↑ k=3
2 | 7 | 18	↑ i=1	4 | 5 | 6	⇑ j=2	2 | 4 | 5 | 6 | |	⇑ k=3
2 | 7 | 18	↑ i=1	4 | 5 | 6	↑ j=3	2 | 4 | 5 | 6 | |	↑ k=4
2 | 7 | 18	⇑ i=1	4 | 5 | 6	↑ j=3	2 | 4 | 5 | 6 | 7 |	⇑ k=4
2 | 7 | 18	↑ i=2	4 | 5 | 6	↑ j=3	2 | 4 | 5 | 6 | 7 |	↑ k=5


2 7 18 ↑ i=1 | 4 5 6 ↑ j=1 | 2 4 ↑ k=2
2 7 18 ↑ i=1 | 4 5 6 ⇑ j=1 | 2 4 5 ⇑ k=2
2 7 18 ↑ i=1 | 4 5 6 ↑ j=2 | 2 4 5 ↑ k=3
2 7 18 ↑ i=1 | 4 5 6 ⇑ j=2 | 2 4 5 6 ⇑ k=3
2 7 18 ↑ i=1 | 4 5 6 ↑ j=3 | 2 4 5 6 ↑ k=4
2 7 18 ⇑ i=1 | 4 5 6 ↑ j=3 | 2 4 5 6 7 ⇑ k=4
2 7 18 ↑ i=2 | 4 5 6 ↑ j=3 | 2 4 5 6 7 ↑ k=5
2 7 18 ⇑ i=2 | 4 5 6 ↑ j=3 | 2 4 5 6 7 18 ⇑ k=5

[2 | 7 | 18]	$\uparrow\ i=1$	[4 | 5 | 6]	$\uparrow\ j=1$	[2 | 4 | | | |]	$\uparrow\ k=2$
[2 | 7 | 18]	$\uparrow\ i=1$	[4 | 5 | 6]	$\Uparrow\ j=1$	[2 | 4 | 5 | | |]	$\Uparrow\ k=2$
[2 | 7 | 18]	$\uparrow\ i=1$	[4 | 5 | 6]	$\uparrow\ j=2$	[2 | 4 | 5 | | |]	$\uparrow\ k=3$
[2 | 7 | 18]	$\uparrow\ i=1$	[4 | 5 | 6]	$\Uparrow\ j=2$	[2 | 4 | 5 | 6 | |]	$\Uparrow\ k=3$
[2 | 7 | 18]	$\uparrow\ i=1$	[4 | 5 | 6]	$\uparrow\ j=3$	[2 | 4 | 5 | 6 | |]	$\uparrow\ k=4$
[2 | 7 | 18]	$\Uparrow\ i=1$	[4 | 5 | 6]	$\uparrow\ j=3$	[2 | 4 | 5 | 6 | 7 |]	$\Uparrow\ k=4$
[2 | 7 | 18]	$\uparrow\ i=2$	[4 | 5 | 6]	$\uparrow\ j=3$	[2 | 4 | 5 | 6 | 7 |]	$\uparrow\ k=5$
[2 | 7 | 18]	$\Uparrow\ i=2$	[4 | 5 | 6]	$\uparrow\ j=3$	[2 | 4 | 5 | 6 | 7 | 18]	$\Uparrow\ k=5$
[2 | 7 | 18]	$\uparrow\ i=3$	[4 | 5 | 6]	$\uparrow\ j=3$	[2 | 4 | 5 | 6 | 7 | 18]	$\uparrow\ k=6$

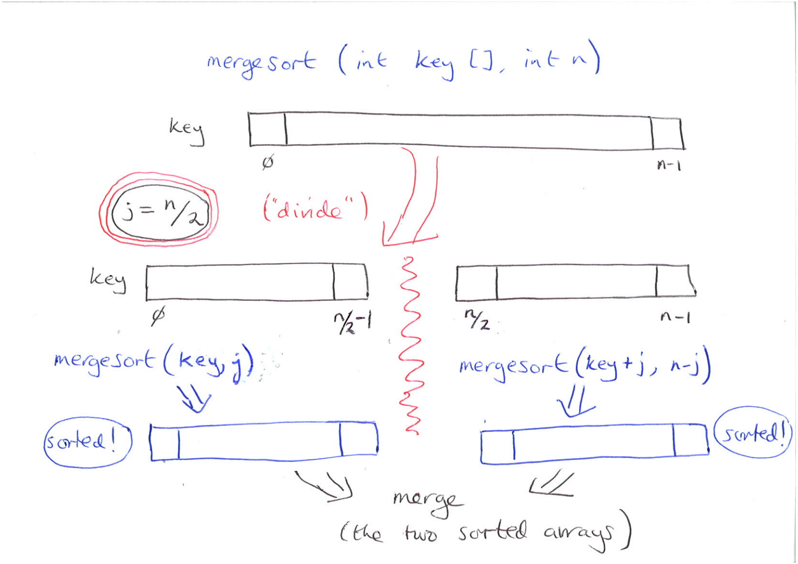
merge

```c
void merge(int a[], int b[], int c[], int m, int n) {
  int i=0, j=0, k=0;
  while (i < m || j < n) {
    /* In the following condition, if j >= n, C knows the
       condition is true, and it *doesn't* check the rest,
       so it doesn't matter that b[j] is off the end of the
       array. So the order of these is important */
    if (j >= n || (i < m && a[i] <= b[j])) {
      /* either run out of b, or the smallest elt is in a */
      c[k++] = a[i++];
    } else {
      /* either run out of a, or the smallest elt is in b */
      c[k++] = b[j++];
    }
  }
}
```

sorting using `merge`

- `merge` can create an *overall sort* from two smaller arrays which were individually sorted.
- Could we use `merge` as a helper function to perform the task of sorting a given array (*where the initial arrays is not at all sorted*)?
  - Divide-and-Conquer is the process of solving a big problem, by utilizing the solutions to smaller versions of that problem.
  - For sorting, we could divide our (unsorted) input array in two pieces, then sort those two smaller subarrays individually (recursion) and finally get the overall sort using merge.

sorting using `merge`



MergeSort through recursion

- Typical implementation of MergeSort is *recursive*:
  - *The sort of the array* `key` *is the result of sorting each half of* `key` *and then* `merge`-ing *those two sorted subarrays*
  - `merge` function takes two *sorted* arrays and creates the 'merge' of those arrays
- C issues:
  We will need to create a 'scratch array' to pass to `merge` (as the `c` parameter) to write the result of 'merging' the two smaller (already sorted) subarrays.
  - 'scratch' array needs same length as input array.
  - Need to *dynamically* allocate space.
  - In C, use `calloc` to get space of a 'dynamic' (not fixed) amount.
    `void *calloc(size_t num, size_t size)`
  - Returns a '*pointer to void*' ... just means a pointer/address of *no particular type*.
    The pointer will be `NULL` if there was not enough available space.

recursive mergesort

```
int mergesort(int key[], int n){
  int j, *w;
  if (n <= 1) { return 1; } /* base case, sorted */
  w = calloc(n, sizeof(int));    /* space for temporary array */
  if (w == NULL) { return 0; } /* calloc failed */
  j = n/2;
  /* do the subcalls and check they succeed */
  if ( mergesort(key, j)
       && mergesort(key+j, n-j) ) {
    merge(key, key+j, w, j, n-j);
    for (j = 0; j < n; ++j)
    key[j] = w[j];
    free(w);    /* Free up the dynamic memory no longer in use. */
    return 1;
  } else { /* a subcall failed */
    free(w);
    return 0;
  }
}
```

Wrap-up and Reading

- Running-time of `mergesort` is proportional to $n \lg(n)$.
  Compares favourably with `BubbleSort` (n^2).
- Read more about *recursion* in Sections 5.14, 5.15 of Kelley & Pohl.
- Read more about `calloc` in Section 6.8 of Kelley & Pohl.
- Our implementation of `mergesort` is on the course webpage.

  `mergerec.c` also has a `wrt` function for printing out small arrays, and a `main` for testing/timing on arrays of various sizes.
- Kelley & Pohl have a 'bottom-up' version of MergeSort for *array lengths* a power-of-2.
  - More troublesome/fiddly than the recursive version
  - Can adapt their 'bottom-up' version of MergeSort to work for general array lengths. *If you want a challenge try this*
    (but test it rigorously)