Faculty of Computer Science, Dalhousie University *17-Oct-2018*

CSCI 2132 — Software Development

Lecture 18: Implementation of Recursive Algorithms

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- Function definitions
- Function declarations or prototypes
- Arguments and parameters
- Arguments passed by value
- Call stack
- Simple recursion example: power
- Mergesort algorithm

Slide notes:

MergeSort Algorithm: Overview

```
Algorithm MergeSort(A, lo, hi)
INPUT: A is an array of comparable elements,
  lo and hi (lo<=hi) are indices into A
OUTPUT: A[lo..hi] part of the array is sorted

1. if lo == hi then Return
2. split array into two subarrays lo-mid and (mid+1)-hi
3. MergeSort(A, lo, mid)
4. MergeSort(A, mid+1, hi)
5. Merge A[lo..mid] with A[mid+1..hi]
```

Now we will look at the implementation available in `~prof2132/public/mergesort.c-blanks`, which contains some blank parts that need to be filled:

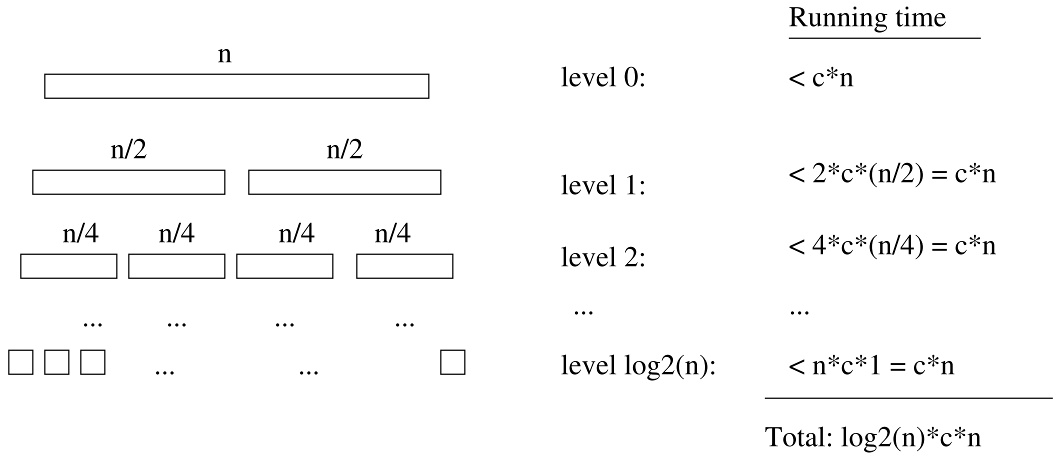
```
/* Program: mergesort.c */
#include <stdio.h>

void mergesort(int array[], int lower, int upper);
void merge(int array[], int lower, int middle, int upper);

int main() {
  int len, i;

  printf("Enter the length of the array: ");
  scanf("%d", &len);

  int array[len];

  printf("Enter %d integers:\n", len);
  for (i = 0; i < len; i++)
```

```
    scanf("%d", &array[i]);

  mergesort(array, 0, len-1);

  printf("The sorted array is: \n");
  for (i = 0; i < len; i++) {
    printf("%d", array[i]);
    if (i < len - 1) printf(" ");
  }
  printf("\n");

  return 0;
}

void mergesort(int array[], int lower, int upper) {
  if (lower < upper) {
    int middle = (lower + upper) / 2;
    mergesort(array, lower, middle);
    mergesort(array, middle+1, upper);
    merge(array, lower, middle, upper);
  }
}

/* middle element is a part of lower sub-array */
void merge(int array[], int lower, int middle, int upper) {
  int len_left = middle - lower + 1;
  int len_right = upper - middle;
  int left[len_left], right[len_right];
  int i, j, k;

  for (i = 0, j=lower; i < len_left; i++, j++)
    left[i] = array[j];

  for (i = 0, j = _____________ ; i < len_right; i++, j++)
    right[i] = array[j];

  i = j = 0; k = lower;
  while (i < len_left && j < len_right) {
    if (left[i] <= right[j]) array[k++] = left[ _______ ];
    else                     array[k++] = right _______ ];
  }

  while (i < len_left)  array[k++] = left [ _______ ];
  while (j < len_right) array[k++] = right[ _______ ];
}
```

Mergesort Running-Time Complexity

There are several simple algorithms for sorting arrays, such as Bubble Sort, Insertion Sort, and Selection Sort, which have the $O(n^2)$ running-time complexity, in the worst case. The Merge Sort is superior in the sense that it has a better running-time complexity of $O(n \log n)$, which becomes significant when sorting very large arrays.

Let us do a quick analysis to understand why the Merge Sort running time is $O(n \log n)$. If we put aside function calls, we can see that the `mergesort` function in the previous code takes constant time in other operations ($O(1)$) and that the `merge` operation takes linear time ($O(n)$) when making a merge of two sub-arrays whose total length is n. This means that there is a constant c such that the merge operation takes less than $c \cdot n$ steps, for all n. Now, we can analyze how the mergesort recursive calls work on smaller and smaller arrays in the following figure:



The figure shows all merge executions as rectangles for the parts of array that are merged. At the top-level, i.e., level 0, we merge the final array from two halves. The array is of length n, so the running time for the merge operation is less than $c \cdot n$, for the constant c that we chose earlier. At the level 1, we make two merges, each one for the resulting array of length $n/2$, so their total running time is less than $2 \cdot c \cdot (n/2) = c \cdot n$. Similarly, the total running time at the level 2 is less than $c \cdot n$, and so on, until we reach individual elements of the array, which happens at approximately '$\log_2(n)$'-th level. When we add up all running times of all merge operations, we see that the total running time is less than $c \cdot n \cdot \log_2(n)$ steps. This means that the total running time is $O(n \log_2 n)$, or, since the $\log_2$ is in a constant proportion to a logarithm of any base, we can express the running time as $O(n \log n)$ independently of logarithm base.

Quicksort Algorithm

– Before we compare quicksort and mergesort, let us go through a reminder of the Quicksort algorithm

```
Algorithm Quicksort(A, lo, hi)
INPUT: A is an array of comparable elements,
  lo and hi (lo<=hi) are indices into A
OUTPUT: A[lo..hi] part of the array is sorted

1: if lo < hi then
2:   p = partition(A, lo, hi)
3:   quicksort(A, lo, p)
4:   quicksort(A, p+1, hi)
```

```
Algorithm Partition(A, lo, hi)
INPUT: A is an array of comparable elements,
  lo and hi (lo<=hi) are indices into A
OUTPUT: Index p (lo<=p<=hi) such that
  A[lo..p]<=A[p+1..hi] (each element of left sub-array
  is <= than each element of right sub-array)
```

```
pivot = A[(lo+hi)/2];   // other choices of pivot...
i = lo - 1; j = hi + 1;
while (true) do
  do i++ while A[i] < pivot
  do j-- while A[j] > pivot
  if i >= j then return j
  swap A[i] with A[j]
```

There are different variations of the QuickSort algorithm. The above one is based on the original version of the inventor of the algorithm, C.A.R. Hoare. It is known to be tricky to implement since some seemingly benign changes can make it invalid. There are different versions, which are not as compact, but more clear.

Mergesort vs. Quicksort

- Quicksort tends to be faster in practice for array sorting
- Some advantages of Mergesort:
  - Mergesort is faster for linked lists
  - Margesort can be made I/O efficient more easily for large data
  - Mergesort is easier to parallelize
  - Mergesort has better worst-case analysis ($O(n \log(n))$ vs. $O(n^2)$)

You probably have already learned about the Mergesort and Quicksort algorithms, so we will just make some brief comments here about which one to use and when in practice. Quicksort tends to be faster than mergesort when we sort an array (which supports random access to any element). On the other hand, Mergesort tends to be faster than quicksort when we sort a linked list. There are other design considerations. If the data is so large that it does not fit in memory, mergesort is easy to be made I/O-efficient, since we can load part of the sequence into memory and sort it, before we merge sorted sublists. It is also easy to make mergesort a parallel algorithm to make use of computers with multiple CPUs or a CPU with multiple cores, as we can use each core to sort part of the sequence.

16.5 Generating Permutations

Slide notes:

Example: Generating Permutations

- For a given positive integer n, print out all permutations of numbers $\{1, 2, \ldots, n\}$
- For example, for $n = 3$, print:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 2 1
3 1 2
```

- This is a non-obvious problem and requires some thinking and algorithm design

We consider here the problem of generating all permutations of a set of elements. We use an array that stores the numbers 1, 2, ..., n, and it can then be easily modified for arbitrary array of n elements. For example, for $n = 3$, we want to print:

```
1 2 3
1 3 2
```

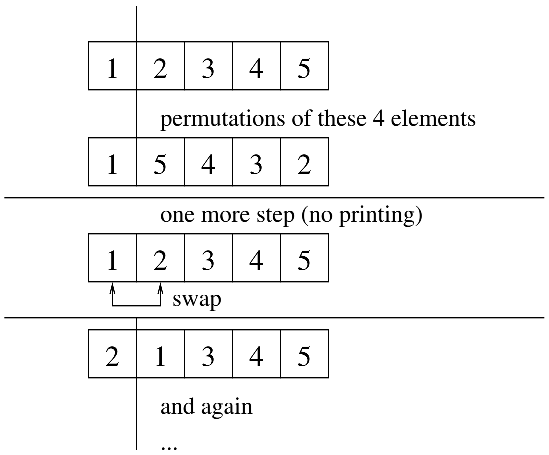
```
2 1 3
2 3 1
3 2 1
3 1 2
```

The solution to this problem is not obvious. We can not simply come up with a loop or even a few nested loops that print all permutations. This is a typical interview questions for a software developer. This kind of questions requires design before implementation, which is important in software development.

To attack this kind of problems, we can try different problem solving techniques until we find a promising idea. Let us try recursion.

To design a recursive solution, the key is to break the task to be performed into similar subtasks. We could for example try to break up this task in two half-arrays, like in the previous example, but we would soon realize that there is no sufficiently simple way, if any, of combining permutations of two sub-arrays into permutations of the whole subarray.

We can follow intuition of how we would manually generate permutations: We first choose the first element to be any of the elements, and then go through all permutations of other elements. We can look at the following example of 5 element permutations. First, we choose the first element to be 1 and then combine it with all permutations of the remaining elements, then we swap it with the second element 2, and combine it with permutations of the remaining elements, and so on.



If we consider how we should generate permutations of the remaining four elements, and so on, we see that the task of permuting elements of a whole array is broken into subtasks, where in each subtask we keep a number of elements at the beginning fixed and we permute the rest of the elements of the array. Before we implement the algorithm in C, let us use pseudocode first to plan the algorithm.

First, let `A` denote the array, n denote the length of the array, and k be the parameter determining the k first elements to be fixed in an invocation of the recursive algorithm. Initially, we do not keep any elements fixed, so the first invocation of the algorithm will be `Permute(A, 0, n)`.