
Computer Programming: Skills & Concepts (CP1)

MergeSort

2nd November 2009



Last Thursday (29th October)

- Review of Linear Search and Binary Search.
- Formal definition of (worst-case) running time.
- BubbleSort program/algorithm.
- $\Theta(n^2)$ running-time for BubbleSort

Today's lecture

- New sorting algorithm called MergeSort
- Implementation for arrays where length is a power-of-2.
- Analysis of running time.
- `calloc` for dynamically-sized arrays.

Merge

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.

example on board

merge

```
void merge(int a[], int b[], int c[], int m, int n) {
    int i=0, j=0, k=0;
    while (i < m && j < n) {
        if (a[i] <= b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while (i < m)
        c[k++] = a[i++];
    while (j < n)
        c[k++] = b[j++];
}
```

MergeSort - the idea

Given an array a of length n .

- (i) Sort all subarrays of length 2: $a[0..1]$, $a[2..3]$...
- (ii) Create sorted subarrays of length $2 * 2 = 4$ by *merging* pairs of the sorted length-2 subarrays ...
- (iii) Create sorted subarrays of length $2 * 4 = 8$ by *merging* pairs of the sorted length-4 subarrays ...

...

Iterative approach - build from “the bottom up”.

mergesort (for n a power of 2)

```
void mergesort(int key[], int n){
    int j, k, *w;
    w = calloc(n, sizeof(int));      /* Allocate space for the array */
    assert (w != NULL);              /* Check there was enough space */
    for (k = 1; k < n; k *= 2) {
        for (j = 0; j < n - k; j += 2*k)
            merge(key + j, key + j + k, w + j, k, k);
        for (j = 0; j < n; ++j)
            key[j] = w[j];
        for (j = 0; j < n; ++j)
            printf("%4d%s", key[j], ((j < n - 1) ? "" : "\n"));
    }
    free(w);                          /* Free up the memory pointed to by w */
}
```

```
void wrt(int key[], int sz) {  
    int i;  
    for (i = 0; i < sz; ++i)  
        printf("%4d%s", key[i], ((i < sz - 1) ? "" : "\n"));  
}
```

Trial run

```
int main(void) {
    int i, sz, key[] = {4, 3, 1, 67, 55, 8, 0, 4, -5, 37, 7, 4, 2, -1, 10, 199};
    sz = sizeof(key)/sizeof(int);
    for (i=1; i<sz; i *= 2) {};
    if (i == sz) {
        printf("Before mergesort: \n");
        wrt(key, sz);
        mergesort(key, sz);
        printf("After mergesort:\n");
        wrt(key, sz);
    } else
        printf("Need an even-length array for this implementation.\n");
    return EXIT_SUCCESS;
}
```

calloc

In previous applications we have always specified the length of the array as a fixed parameter defined in advance, directly in the program.

To define array size *dynamically*, use `calloc`:

- `calloc()` takes 2 arguments (of type `size_t`):
`calloc(n, el_size)`
- This allocates (if available) space an array of length `n` of type `el` (each cell using `el_size` bytes).
- `calloc ()` returns a pointer to the address of the start of the array in memory.
- Space created is initialized to all-bits-0.
- `malloc()` similar.

Running-time of mergesort

“power-of-2” length gives intuition

- Double the “merge-size” k (starting from 1) at each pass.
- Can do this ONLY $\log(n)$ times for $k < n$.
- Do a linear amount of “work” ($\Theta(n)$) across the array for each value of k .

\Rightarrow Roughly $\Theta(n \log(n))$ overall running-time.

Homework

- Sections 6.8 and 6.9 of Kelley and Pohl!
- (from tomorrow) Experiment with the code.

TOMORROW'S LECTURE

- revision on pointers, functions etc! Come prepared