Monday's lecture

- Arguing a program is correct
- Linear Search of an array
- Binary search of an array
- (Theoretical) measurement of running time
- Timing your code on DICE
- I never got to cover the slides on BubbleSort

NOTE In the tests in search.c, I did NOT initialise the test array to be sorted (as required by BinarySearch)
... does not matter as the key -1 is not in the array at all

Today's lecture

- BubbleSort algorithm (from slides18.pdf).
- New sorting algorithm called MergeSort
- Analysis of running time.
- calloc for dynamically-sized arrays.

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.

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) /* copying the 'rest' into c */
        c[k++] = a[i++]; /* (if b got finished first) */
    while (j < n) /* copying the 'rest' into c */
        c[k++] = b[j++]; /* (if a got finished first) */
}

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".

At each step we double the size of our "windows of interest"

cHECKING OUTPUT

* Function to write-out the contents of key[]. */
void wrt(int key[], int sz) {
    int i;
    for (i = 0; i < sz; ++i)
        printf("%4d%s", key[i], ((i < sz -1) ? "" : "\n"));
}

mergesort

void mergesort(int key[], int n){
    int j, k, *w;
    w = calloc(n, sizeof(int)); /* Allocate space for the array */
    assert (w != NULL); /* If not enough space, stop! */
    if ((n % 2) == 1)
        w[n-1] = key[n-1];
    for (k = 1; k < n; k *= 2) {
        for (j = 0; j < n - 2*k; j += 2*k)
            merge(key + j, key + j + k, w + j, k, k);
        if (n-j > k) /* k, n-j-k different => more work. */
            merge(key + j, key + j + k, w + j, k, (n-j)-k);
    }
    for (j = 0; j < n; ++j) /* copy sorted array into 'key' */
        key[j] = w[j];
    free(w); /* Free-up memory pointed to by w */
}
int main(void) {
    int i, sz, key[] = {4, 3, 1, 67, 0, 4, -5, 37, 7, 2, -1, 199};
    sz = sizeof(key)/sizeof(int);
    printf("Before mergesort: \n");
    wrt(key, sz);
    mergesort(key, sz);
    printf("After mergesort:\n");
    wrt(key, sz);
    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
(a) We double the “merge-size” k (starting from 1) at each pass.
(b) Can do this ONLY $2 \log(n)$ times for $k < n$.
(c) 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.

Not quite as obvious that (b) is true when the array-length is not a power-of-2 . . . still true though!

Big difference in speed from BubbleSort. EXPERIMENT

A more dramatic example
Sometimes the gap between a good and bad algorithm can be dramatic. Consider the problem of testing whether an n-bit number is prime.

▶ The obvious brute force method requires $2^{n/2}$ integer divisions
    - why?
    - This is completely infeasible if $n = 200$ (say).
▶ On the other hand, a (non-obvious) algorithm for primality testing which take time polynomial in $n$ was discovered in 2002
    (Agrawal-Kayal-Saxena)

(needed for RSA public-key cryptosystem.)
Homework

- Sections 6.8 and 6.9 of Kelley and Pohl!
- Experiment with the code.