Imperative programming - tutorial week 7

Arnold Meijster

Dept. computer science (university of Groningen)

October 15, 2017

8.1.1 Minimum and maximum of an array

Write a function `minmax` that computes the minimum and the maximum value of a (non-empty) sequence with length `len`. The function should not need more than `3*len/2` comparisons. Use the following protoype:

```
void minmax(int len, int arr[], int *min, int *max);
```

8.1.1 Minimum and maximum of an array

```
void minmax(int len, int arr[], int *min, int *max) {
  int minimum, maximum;
  *min = *max = arr[len-1];
  if (len % 2 == 1) {
    len--;
  }
  for (int i=0; i<len; i+=2) {  // len is even
    if (arr[i] < arr[i+1]) {  /* len/2 comparisons */
      minimum = arr[i];
      maximum = arr[i+1];
    } else {
      minimum = arr[i+1];
      maximum = arr[i];
    }
    // and another len comparisons
    *min = (minimum < *min ? minimum : *min);
    *max = (maximum > *max ? maximum : *max);
  }
}
```

8.1.2 Grid points on a disc

Write a code fragment that computes how many grid points are on a closed (i.e. the edge include) circular disc with center `(0,0)` and a positive integer radius `r`. The fragment mus have a time complexity that is linear in `r`.

8.1.2 Grid points on a disc

Write a code fragment that computes how many grid points are on a closed (i.e. the edge include) circular disc with center (0,0) and a positive integer radius r. The fragment mus have a time complexity that is linear in r.

```
int trivialCountGridPoints(int r) {
  int x, y, cnt = 0;
  for (x=1; x <= r; x++) {
    for (y=1; y <= r; y++) {
      if (x*x + y*y <= r*r) {
        cnt++;
      }
    }
  }
  return 4*(cnt + r) + 1;  // we missed axes + origin
}
```

8.1.2 Grid points on a disc

```
int smartCountGridPoints(int r) {
  int x, y=r, cnt=0;
  for (x=0; x <= r; x++) {
    while (x*x + y*y > r*r) {
      y--;
    }
    cnt += y;
  }
  return 4*cnt + 1;
}
```

8.1.3 Sieve of Eratosthenes

The Ancient Greeks already knew a method to compute the list of all primes smaller than a certain limit `n`. The same procedure is still used today, and is called the *sieve of Eratosthenes*.

The method works as follows. We start with the complete list of all the numbers from 2 to `n`. The first number of this list (i.e. 2) is prime, and is printed on the screen. Then we remove all multiples (including 2 itself) from the list. Now, the first element of the remaining list is prime again: it gets printed and all its multiples are removed. This procedure is repeated until the list is empty.

(a) Write a program fragment that prints all primes less than `n` on the screen.

(b) Write a program that returns the list of all primes less than `n`.

8.1.3(a) Sieve of Eratosthenes

```c
void sieveOfEratosthenes(int n) {
  n--;  // initial list. 0, 1 are not in it.
  int *sieve = safeMalloc(n*sizeof(int));
  for (int i=0; i < n; i++) {
    sieve[i] = i+2;
  }
  int length = 0;
  while (length < n) {
    int prime = sieve[length];  // head list is prime
    printf("%d ", prime);
    length++;
    int idx = length;
    for (i=length; i < n; i++) {
      if (sieve[i]%prime != 0) { // sieve[i] survives
        sieve[idx] = sieve[i];
        idx++;
      }
    }
    n = idx;
  }
  free(sieve);
}
```

8.1.3(b) Sieve of Eratosthenes

```
int *sieveOfEratosthenes(int n, int *len) {
  n--;  // initial list. 0, 1 are not in it.
  int *sieve = safeMalloc(n*sizeof(int));
  for (int i=0; i < n; i++) {
    sieve[i] = i+2;
  }
  int length = 0;
  while (length < n) {
    int prime = sieve[length];  // head list is prime
    length++;
    int idx = length;
    for (i=length; i < n; i++) {
      if (sieve[i]%prime != 0) { // sieve[i] survives
        sieve[idx] = sieve[i];
        idx++;
      }
    }
    n = idx;
  }
  *len = length;
  return realloc(sieve, length*sizeof(int));
}
```

8.1.3(b) Sieve of Eratosthenes

```
int main() {
  int len, n=0;
  do {
    printf("Type upperbound n (n>=2): ");
    scanf("%d", &n);
  } while (n < 2);

  int *primes = sieveOfEratosthenes(n, &len);

  /* print primes */
  printf("%d", primes[0]);
  for (n=1; n < len; n++) {
    printf(",%d", primes[n]);
  }
  printf("\n");

  /* free memory */
  free(primes);

  return 0;
}
```

8.1.4 Floyd-Warshall algorithm

We consider `N` villages: they are enumerated `0,1,..,N-1`. Some of these villages are directly connected by a road, while others are not. However, from each village any other village is reachable direclty or via other villages.
We have a distance matrix:

```
int dist[N][N];
```

The value `dist[i][j]` (which is equal to `dist[j][i]`) is non-zero if there exists a direct road from `i` to `j`. In that case, `dist[i][j]` denotes the distance between `i` and `j`. If `dist[i][j]==0`, then this means either `i==j` or there is no direct road between `i` and `j`.

Write a program fragment that computes the length of the shortest path between `i` and `j` for all pairs `(i,j)`. The time complexity of your algorithm should be $O(N^3)$.

Hint: Consider a shortest path from i to j via k. Then the subpath from i to k is also optimal. The same holds for the subpath from k to j.

8.1.4 Floyd-Warshall algorithm

```
void floydWarshall() {
  for (int k = 0; k < N; ++k) {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        /* If i and j are different nodes and if the
           paths between i and k and between k and j
           exist, then ... */
        if ((i!=j) && !dist[i][k] && !dist[k][j]) {
          /* See if you can get a shorter path between
               i and j by visiting k somewhere along
             the current path */
          if ((dist[i][j] == 0) ||
              (dist[i][k]+dist[k][j] < dist[i][j])) {
            dist[i][j] = dist[i][k] + dist[k][j];
          }
        }
      }
    }
  }
}
```