1. Programming Practice – Testing

Testing – an overview of crucial points

(i) Test as you write the code
   - Test code at its boundaries
   - Test pre- and post-conditions
   - Program defensively
   - Check error returns

(ii) Systematic testing
   - Test incrementally
   - Test simple parts first
   - Know what output to expect
   - Verify conservation properties
   - Compare independent implementations
   - Measure test coverage

Testing Binary Search Algorithm

Binary search algorithm takes an ordered array \( a \) as an input and searches it for an item \( v \), by iteratively splitting \( a \) into two segments \( a[l..m-1] \) and \( a[m+1..r] \).

```c
int search(int a[], int v, int l, int r)
{
   while(r>=l)
   {
      int m=(l+r)/2;
      if(v == a[m])
         return m;
      if(v<a[m])
         r=m-1;
      else
         l=m+1;
   }
   return -1;
}
```

Problem 1: Prepare C++ code for the following test:
   - Prompt the user for an input number \( n \).
   - Allocate a dynamic array of integer type of size \( n \) and place consecutive \( n \) odd integers into it.
   - Run a loop, which will check whether for all \( v=0,...,2*n \) the output of each call `search (a,v,0,n-1)` is correct.
   - Inform the user about the result of the test.
**Problem 2:** Prepare C++ code for the following test:
- Prompt the user for an input number \( n \).
- Allocate a dynamic array of integer type of size \( n \) and place \( n \) random numbers in range \([0..n-1]\) into it.
- Sort the array in non-decreasing order.
- Test whether the array has been sorted.
- Run a loop, which will check whether for all \( v=0,...,n-1 \) the output of each call \( \text{search}(a,v,0,n-1) \) is correct.
- Inform the user about the result of the test.

**2. Heap-Sort algorithm**

Heap-Sort algorithm in pseudo-code

(i) The correspondence of array and tree indices
- \( i \) – the array index
- \( l \) – the level inside the tree
- \( p \) – the position inside the level \( l \), \( p=0,...,2^l-1 \)
Take \( i \), the index of an array element and compute \( l \) and \( p \) from the formula \( i+1=2^l+p \)

(ii) Formulas for parents and children
- \( i \) – the array index
- \( [(i-1)/2] \) – the array index of the parent
- \( 2i+1 \) – the array index of the left child
- \( 2i+2 \) – the array index of the right child

(iii) Definition of a heap
A heap is a complete binary tree, which satisfies the property: numbers stored inside data fields of the parents are no smaller than numbers stored inside data fields of their children.

(iv) The operation of making a heap

```cpp
void make_heap(int a[], int n)
{
    int k,i;
    for(i=1; i<n; i++)
    {
        k=i;
        while(a[k] is bigger than its parent)
            swap a[k] with its parent and reset k to be the index of the parent;
    }
}
```
(v) The operation of restoring the heap

```c
void restore_heap(int a[], int n)
{
    int k=0;
    while(a[k] is smaller than one of its children inside a[0],…,a[n-1])
        swap a[k] with its largest child and reset k to be the index
        of this child;
}
```

(vi) Heap Sort algorithm

```c
void heapsort(int a[0], int n)
{
    int k;
    make_heap(a,n);
    for(k=n-1;k>0;k--)
    {
        swap a[0] and a[k];
        restore_heap(a,k);
    }
}
```

3. Programming Practice - Performance

**Spam filter story**

Spam filter: version 1

```c
for(i=0;i<npat; i++)
    if(strstr(mesg,pat[i])!=NULL)
        return 1;
```

Estimated work: strlen(mesg)*npat comparisons

Spam filter: version 2

pseudo-code:

```c
for(j=0; mesg[j] != '\0'; j++)
    if(some pattern matches starting at mesg[j])
        return 1;
```

Estimated work: strlen(mesg)*npat/52 comparisons
C code:
for(j=0; (c=mesg[j]) != ‘\0’; j++){  
   for(i=0; i<nstarting[c]; i++){  
      k=starting[c][i];  
      if(memcmp(mesg+j, pat[k], patlen[k]) == 0){  
         printf(...);  
         return 1;  
      }  
   }  
}

Data structure creation:
...
for(i=0;i<npat; i++){  
   c=pat[i][0];  
   starting[c][nstarting[c]++]=i;  
   patlen[i] = strlen(pat[i]);  
}
...

Timing and Profiling
• Perform timing measurements  
• Use a profiler  
• Concentrate on the hot spots  
• Draw pictures

Strategies for speed
• Use a better algorithm or a data structure  
• Enable compiler optimization  
• Tune the code  
• Don’t optimize what doesn’t matter

Tuning the code
• Collect common sub-expressions  
• Replace expensive operations by cheap ones  
• Unroll or eliminate loops  
• Cache frequently used values  
• Write special purpose allocator  
• Buffer input and output  
• Handle special cases separately  
• Precompute results  
• Use approximate values  
• Rewrite in lower level language
Space efficiency

- Save space by using the smallest possible data structure
- Don’t store what you can easily recompute