Sorting

CS10003 PROGRAMMING AND DATA STRUCTURES
The Basic Problem

Given an array: $x[0], x[1], \ldots, x[size-1]$ reorder the elements so that

$x[0] \leq x[1] \leq \ldots \leq x[size-1]$ 

- That is, reorder entries so that the list is in increasing (non-decreasing) order.

We can also sort a list of elements in decreasing (non-increasing) order.

We prefer not to use additional arrays for the element rearrangement.
Example

Original list:

10, 30, 20, 80, 70, 10, 60, 40, 70

Sorted in non-decreasing order:

10, 10, 20, 30, 40, 60, 70, 70, 80

Sorted in non-increasing order:

80, 70, 70, 60, 40, 30, 20, 10, 10
Selection Sort
SELECTION SORT: The idea

General situation:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>k</th>
<th></th>
<th>size-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>smallest elements, sorted</td>
<td>remainder, unsorted</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Steps:

- Initialize $k = 0$.
- Find smallest element, $mval$, in the array segment $x[k...size-1]$.
- Swap smallest element with $x[k]$, then increase $k$.
Subproblem

/* Find index of smallest element in x[k...size-1] */

int min_loc (int x[ ], int k, int size)
{
    int j, pos;
    pos = k;
    for (j=k+1; j<size; j++)
        if (x[j] < x[pos])
            pos = j;
    return pos;
}
Selection Sort Function

```c
/* Sort x[0..size-1] in non-decreasing order */

int sel_sort (int x[], int size) {
  int k, m, temp;

  for (k = 0; k < size-1; k++) {
    m = min_loc (x, k, size);
    /* Swap x[k], x[m]*/
    temp = x[k];
    x[k] = x[m];
    x[m] = temp;
  }
}
```
Example

\[ x: \begin{array}{cccccccc}
3 & 12 & -5 & 6 & 142 & 21 & -17 & 45 \\
\hline
-17 & 12 & -5 & 6 & 142 & 21 & 3 & 45 \\
-17 & -5 & 12 & 6 & 142 & 21 & 3 & 45 \\
-17 & -5 & 3 & 6 & 142 & 21 & 12 & 45 \\
-17 & -5 & 3 & 6 & 142 & 21 & 12 & 45 \\
\end{array} \]
Bubble Sort
BUBBLE SORT: The idea

General situation:

In every pass, we go on comparing neighboring pairs, and swap them if out of order.

for j = 0 to k-1
  if (x[j] > x[j+1])
    interchange them.

At the end of this iteration, the ‘next largest’ element (among the unsorted part) will settle at x[k].

Lighter elements bubble up.
Heavier elements settle down.
void bubble_sort (int x[], int size) {
    int t;
    for (i = 0; i < size; i++)
        for (j = 0; j < size-i-1; j++)
            if (x[j] > x[j+1]) {
                // swap a[j] and a[j+1]
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
}

How do the passes proceed?
In pass 1, we consider index 0 to size-1
In pass 2, we consider index 0 to size-2
In pass 3, we consider index 0 to size-3
......
In pass size-1, we consider index 0 to 1.
A more efficient sorting method: Mergesort
A popular sorting algorithm based on the divide-and-conquer approach.

**Basic idea (divide-and-conquer method)**

```plaintext
sort (list)
{
    if the list has length greater than 1
    {
        Partition the list into lowlist and highlist;
        sort (lowlist);
        sort (highlist);
        combine (lowlist, highlist);
    }
}
```
Merge Sort

Input Array

Part-I

Split

Part-II

Merge

Sorted Arrays
void merge_sort (int *A, int n)
{
    int i, j, k, m;
    int *B, *C;

    if (n > 1)  {
        k = n/2;   m = n - k;
        B = (int *) malloc (k * sizeof(int));
        C = (int *) malloc (m * sizeof(int));
        for (i=0; i<k; i++)  B[i] = A[i];
        for (j=k; j<n; j++)  C[j-k] = A[j];
        // B contains first half of A
        // C contains second half of A
        merge_sort (B, k);
        merge_sort (C, m);
        merge (B, C, A, k, m); // destination array is A
        free(B); free(C);
    }
}
Merging two sorted arrays

Copy element from a (indexed by i) if its value is smaller than the element in b pointed by j; otherwise, copy the element from b (indexed by j).

If one of the arrays a or b get exhausted, simply copy the rest of the other array.
void merge (int *a, int *b, int *c, int m, int n)
   // c is the destination array
{
    int i=0, j=0, k=0, p;
    // loop as long as neither array a nor array b is completed
    while ((i<m) && (j<n)) {
        if (a[i] < b[j])
            { c[k] = a[i]; i++; }
        else
            { c[k] = b[j]; j++; }
        k++;
    }
    if (i == m) { // array a completed; copy rest of array b to array c
        for (p=j; p<n; p++)
            { c[k] = b[p]; k++; }
    } else {
        // array b completed; copy rest of array a to array c
        for (p=i; p<m; p++)
            { c[k] = a[p]; k++; }
    }
}
Example: showing the merge phase only

Initial array A contains 16 elements:

• 66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30, 47, 23

Pass 1 :: Merge each pair of elements

• (33, 66) (22, 40) (55, 88) (11, 60) (20, 80) (44, 50) (30, 70) (23, 47)

Pass 2 :: Merge each pair of pairs

• (22, 33) (40, 66) (11, 55) (60, 88) (20, 44) (50, 80) (23, 30) (47, 77)

Pass 3 :: Merge each pair of sorted quadruplets

• (11, 22, 33, 40, 55, 60, 66, 88) (20, 23, 30, 44, 47, 50, 77, 80)

Pass 4 :: Merge the two sorted subarrays to get the final list

• (11, 20, 22, 23, 30, 33, 40, 44, 47, 50, 55, 60, 66, 77, 80, 88)
void merge_sort (int *A, int n) 
{
    int i, j, k, m;
    int *B, *C;
    if (n > 1) {
        k = n/2;  m = n - k;
        B = (int *) malloc (k * sizeof(int));
        C = (int *) malloc (m * sizeof(int));
        for (i=0; i<k; i++)
            B[i] = A[i];
        for (j=k; j<n; j++)
            C[j-k] = A[j];
        // B contains first half of A
        // C contains second half of A
        merge_sort (B, k);
        merge_sort (C, m);
        merge (B, C, A, k, m); // dest A
        free(B); free(C);
    }
}
Time complexity of merge sort

If \( n \) denotes the number of elements to be sorted, then the number of comparisons required in merge sort is approximately proportional to \( n \log n \).

We need extra storage space as we have to temporarily create space for the arrays B and C.
Practically best sorting method: Quicksort
Introduction to Quick Sort

- Merge sort is a theoretically best (optimal) sorting algorithm.
- Quick sort is the practically best general-purpose sorting algorithm.
- Problems of merge sort:
  - Extra space requirement
  - Merging step is difficult to carry out without extra arrays.
- Quick sort is another recursive sorting algorithm.
- Quick sort takes a divide-and-conquer approach.
- In merge sort, the main work (merging) is done after the recursive calls return.
- In quick sort, the main work (partitioning) is done before the recursive calls are made.
- Basic idea of quick sort
  - Choose an element $p$ of the array $A$ as the pivot.
  - Decompose the array in three parts: $L$ consisting the elements of $A$ less than (or equal to) $p$, $R$ consisting of the elements of $A$ larger than $p$, and the single element $p$.
  - Recursively sort $L$ and $R$.
  - Output $\text{sorted}(L)$ followed by $p$ followed by $\text{sorted}(R)$.
  - If partitioning is done in $A$ itself, then there is no task left after the recursive calls.
void quicksort ( int A[], int n )
{
    int pivotidx;
    if (n <= 1) return;
    pivotidx = partition (A, n);
    quicksort (A, pivotidx);
    quicksort (A+pivotidx+1, n-pivotidx-1);
}
Partitioning using extra arrays

```c
int partition ( int A[], int n )
{
    int *L, *R, p, i, j, l, r;

    if (n <= 1) return n-1;

    L = (int *)malloc(n * sizeof(int));
    R = (int *)malloc(n * sizeof(int));
    p = A[n-1]; // Choose the last element of A as pivot
    l = r = 0; // Initialize the sizes of L and R
    for (i=0; i<=n-2; ++i)
    for (i=0; i<l; ++i) A[i] = L[i]; // Copy L to A
    A[i++] = p; // Append p to A
    for (j=0; j<r; ++j) A[i++] = R[j]; // Append R to A
    free(L); free(R); // No further needs for L and R
    return l;
}
```
In-place partitioning

- Possibility of partitioning A without any extra arrays make quick sort attractive and efficient.
- There are many variants of the in-place partitioning algorithm.
- We follow the CLRS variant:
  

- We take \( p = A[n-1] \) as the pivot.
- The array A is always maintained as the concatenation LRUp, where
  - L consists of elements \( \leq p \)
  - R consists of elements \( > p \)
  - U is the unprocessed part (elements in U are not yet classified to go to L or R)
- Each iteration processes one element from U, and sends that element to L or R as appropriate.
- After \( n - 1 \) iterations, there are no unprocessed elements, so the array is of the form LRp.
- It is then converted to the form LpR.
- Blocks (L and R) are never fully shifted. Only element swaps are used.
- This may destroy the order of the (equal) keys in the partitioned array.
In-place partitioning

Case 1: $A[i] > p$

Case 2: $A[i] \leq p$

After end of loop
In-place partitioning: The code

```c
int partition ( int A[], int n )
{
    int lend = -1, i;
    int p, t;

    p = A[n-1]; // Last element of A is the pivot
    for (i=0; i<=n-2; ++i) {
        if (A[i] <= p) { // Region L grows
            ++lend;
        }
        // else Region R grows, ++i will do it
    }
    // i is the first index of Region R
    i = lend + 1;
    return i;
}
```
In-place partitioning: An example

lend i

```
| 10 | 21 | 8 | 3 | 15 | 27 | 9 | 16 | 5 | 15 |
-1  0  1  2  3  4  5  6  7  8  9
```

lend i

```
| 10 | 8 | 21 | 3 | 15 | 27 | 9 | 16 | 5 | 15 |
  0  1  2  3  4  5  6  7  8  9
```

lend i

```
| 10 | 8 | 3 | 21 | 15 | 27 | 9 | 16 | 5 | 15 |
  0  1  2  3  4  5  6  7  8  9
```

lend i

```
| 10 | 8 | 3 | 15 | 21 | 27 | 9 | 16 | 5 | 15 |
  0  1  2  3  4  5  6  7  8  9
```
Performance of quick sort

- Running times are specified as “*roughly proportional to a function of the input size.*”
- No (comparison-based) sorting algorithm can run faster than \( n \log n \) is the worst case.
- For merge sort:
  - All cases are the same. No specific best / worst / average case.
  - Each case has running time \( n \log n \) for merge sort.
- For quick sort:
  - Best case: Partitioning divides the array roughly into two equal halves
  - Worst case: Partitioning always gives one subarray of size one less than the array.
  - Average case: The pivot is any one element (smallest to largest) with equal probability.
- Example of worst case: The array is already sorted in ascending or descending order.
- Running time of quick sort:
  - Best and average case: \( n \log n \)
  - Worst case: \( n^2 \)
- Quick sort is not theoretically optimal.
- In practice, quick sort is considered the fastest sorting algorithm for “general” arrays.