Sorting (runtime, memory and stability considerations)

bubble insertion merge quick sort sorting stable sort

Original Article:

- Sorting (by timmac)

General Considerations

The top considerations for sorting are:-

- Runtime
- Memory
- Stability

Faster algorithms that require multiple recursive calls may need to make a copy of the data again and again, this may make the entire operation extremely memory intensive in cases where memory is at a premium (embedded systems).

Stability in sorting means that whether the algorithm preserves the relative order of elements that are comparitively the same. A stable sort algorithm is the one where the relative order is unchanged.

Bubble Sort

- O(n^2)
- In place sort (no additional memory)
- Stable sort

```
void bubble_sort(vector<int> &arr){
  int n = arr.size();
  for(int i = 0; i < n; ++i){
    for(int j = 0; j < n - i - 1; ++j){
      if(arr[j] > arr[j + 1])
        swap(arr[j], arr[j + 1]);
    }
  }
}
```

Insertion Sort

- works well on lists that are mostly sorted
- worst case O(n^2)
- in place sort
- stable sort
- easy to maintain a DS that gets added to frequently (if one wanted to maintain a sorted list and elements kept on adding to it)

```cpp
void insert_sort(vector<int> &arr){
  int n = arr.size();
  for(int i = 0; i < n; ++i){
    int k = i - 1;
    int temp = arr[i];
    while(k > 0){
      if(arr[k] <= temp)
        break;
      arr[k] = temp;
      k--;
    }
    arr[k + 1] = temp;
  }
}
```

Merge Sort

- O(nlogn)
- needs extra storage for the merge step
- stable sort
- application in counting the number of inversions (determining how "unsorted" a given list is)

```cpp
void merge_sort(vector<int> &arr){
  if(arr.size() == 0 || arr.size() == 1)
    return;
  int sz = arr.size() / 2;
  vector<int> A(arr.begin(), arr.begin() + sz);
  vector<int> B(arr.begin() + sz, arr.end());
  merge_sort(A);
  merge_sort(B);
  int i = 0, j = 0, k = 0;
  while(i < A.size() && j < B.size()){
    if(A[i] <= B[j]){
      arr[k] = A[i];
      i++;
      k++;
    }
    else{
      arr[k] = B[j];
      j++;
      k++;
    }
  }
  while(i < A.size()){
    arr[k] = A[i];
    i++;
    k++;
  }
  while(j < B.size()){
    arr[k] = B[j];
    j++;
    k++;
  }
}
```

Heap Sort

- O(nlogn) with low constants
- in place sort
- **not** a stable sort
- somewhat difficult to implement
- to sort in the ascending (descending) order first construct the max (min) heap and then repeatedly remove the top element from the heap. the heapify process has an amortized cost of O(n) and each deletion takes O(logn) time hence it takes O(nlogn) to delete all elements. since the deleted elements are replaced at the end of the heap you finally get a completely sorted list

Quick Sort

- best case O(nlogn), worst case O(n^2), average case randomized quick sort O(nlogn).
- in place sort
- **not** a stable sort
- partition step needs to be applied

```
int partition(int l, int r, vector<int> &arr){
  int p = l;
  for(int i = l + 1; i <= r; ++i){
    if(arr[i] >= arr[p])
      continue;
    swap(arr[i], arr[p]);
    swap(arr[i], arr[p + 1]);
    p++;
  }
  return p;
}

void quick_helper(int l, int r, vector<int> &arr){
  if(l >= r)
    return;
  int p = partition(l, r, arr);
  quick_helper(l, p - 1, arr);
  quick_helper(p + 1, l, arr);
}

void quick_sort(vector<int> &arr){
  quick_helper(0, arr.size() - 1, arr);
}
```

Radix Sort

- looks into the structure of the keys itself
- O(bn) where b is the number of bits in a key

Radix Exchange Sort

- start left to right from the MSB, in each step partition the array based on the bit under examination, then recurse on the sub arrays
- O(bn)
- in place sort
- **not** a stable sort (due to the partition step)

To partition in on a given bit:

repeat

- scan left to right to find key with 1 (at bit b)
- scan right to left to find key with 0 (at bit b)
- exchange indices
  until indices cross

```
int partition(int b, vector<int> &arr){
  int scan = 1 << (b - 1);
  int i = 0, j = arr.size() - 1;
  while(i < j){
    if(arr[i] & scan == 0){
      i++;
      continue;
    }
    if(arr[j] & scan == 1){
      j--;
      continue;
    }
    swap(arr[i], arr[j]);
    i++;
    j--;
  }
  return arr[i] & scan ? i : i + 1;
}
```

Straight Radix Sort

- start right to left from the LSB, in each step sort the array in a **stable** way based on the bit under examination, then keep on sorting with the next significant bit
- O(bn)
- out of place sort (need to use bucket sort for stable sorting)
- a stable sort

Lower Bound on comparison based sorting

Assume all possible arrays of n elements. Total possible such arrays is n!. Now make a tree of all the comparison decisions that need to be taken to sort the array. The leaves of the tree denote each possible permuation. Since the number of leaves is n!, the smallest height of the tree can be O(log(n!)) which is roughly equal to O(nlogn). This gives us the lower bound on comparison based sorting. Radix sort is not counted here because it is not a comparison based sort. It analyses the structure of the keys.