Princeton Algorithms Course Part 1

Fouad Elbakly

1 Elementary Sorts

1.1 Selection sort

- Algorithm scans from left to right.
- Entries left of index are fixed and sorted
- Entries right of index are not fixes and are not sorted

1.1.1 Java Implementation

```
for (int i = 0; i < N; i++) {
 int min = i;
 for (int j = i + 1; j < N; j++)
  if (a[j] < a[min])
   min = j;
 swap(arr[i], arr[min]);
}
```

1.1.2 Time Complexity

Takes $O(n^2)$ time due to worst case scenario, outer loop N times, and first inner loop iteration N times.

1.1.3 Remarks

- Is inplace
- N exchanges
- Not stable

1.2 Insertion Sort

- Scans from left to right
- Elements to the left of index are sorted
- Elements to the right of index have not been seen and are not sorted

1.2.1 Java Implementation

```
for (int i = 0; i < N; i++) {
 for (int j = i; j > 0; j--) {
  if (a[i] < a[j - 1])
   swap(a[j], a[j - 1])
  else
   break
 }
}
```

1.2.2 Time Complexity

Takes $O(n^2)$ time due to worst case scenario being a reversely ordered array. Thus, N iterations in outer loop and N iterations in inner loop.

1.2.3 Remarks

- Is inplace
- Is stable
- Use for small N or partially ordered

1.3 Shellsort

- Moves entries more than one position at a time by h-sorting an array.
- When h is large, the number of subarrays is small, and as h gets smaller the array is nearly in order.
- Use the Knuth increment sequence $h = 3h + 1, (1, 4, 13, 40, 121, 364, ...)$

1.3.1 Java Implementation

```
int h = 1;

while (h < N/3)
 h = 3*h - 1;

while (h >= 1) {
 for (int i = h; i < N; i++)
  for (int j = i; j >= h && a[j] < a[j-h]; j -= h)
   swap(a[j], a[j - h]);
 h /= 3;
}
```

1.3.2 Time Complexity

Worst case time complexity is $O(n^2)$, but the average time complexity is unknown.

1.3.3 Remarks

- Is inplace
- Tight code, subquadratic

1.4 Shuffle Sort (Knuth Shuffle)

- Loop from left to right
- in interation i pick a random integer r between 0 and i.
- swap a[i] and a[r]

1.4.1 Java Implementation

```
int N = a.length;
for (int i = 0; i < N; i++) {
 int r = Random.nextInt(i + 1);
 swap(a[i], a[r]);
}
```

1.4.2 Time Complexity

Takes $O(n)$ time due to the need to loop through the array of size n only once.

2 Advanced Sorts

2.1 Mergesort

- Divide array into two halves
- Recursively sort each half
- Merge the two halves

2.1.1 Java Implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);
    assert isSorted(a, mid+1, hi);

    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for(int k = lo, k <= hi; k++) {
        if (i > mid)
            a[k] = aux[j++];
        else if (j > hi)
            a[k] = aux[i++];
```

```
        else if (less(aux[j], aux[i]))
            a[k] = aux[j++];
        else
            a[j] = aux[i++];
    }
    assert isSorted(a, lo, hi);
}

public static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo)
        return;
    int mid = lo + (hi - lo)/2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

2.1.2 Time Complexity

$Nlog(N)$ time for all cases.

2.1.3 Remarks

- Is not inplace, therefore needs N extra space for auxiliary array.
- Is stable

2.2 Quicksort

- Shuffle the array
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j
- Sort each piece recursively

2.2.1 Java Implementation

```
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo; j = hi + 1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi)
                break;
        while (less(a[lo], a[--j]))
```

```
            if(j == lo)
                break;
        if (i >= j)
            break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}

public static void sort(Comparable[] a)
{
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
}

private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo)
        return;
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

2.2.2 Time Complexity

Work case is N^2, however it is more likely that you get struck by lightning than worst case scenario occuring since the array is shuffled at the start. Worst case scenario is when the suffle sorts the array.

2.2.3 Remarks

- Not stable
- $Nlog(N)$ probabilistic gaurantee fastest in practice
- If working with a lot of duplicate keys use median of 3 quicksort *(Read book for more information)*

2.3 Binary Heaps