**CS512 LECTURE NOTES - LECTURE 6**

**Quick Sort Worst Case**

The worst case is when in partition, the list is always divided into an empty list and another of size $n-1$, in this case the recurrence relation is

$$T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$$

# 1 Average case analysis of Quicksort

## 1.1 Randomized version

In order to compute the expected running time of quicksort we will provide a randomized version of the algorithm so that it performance will not depend on the actual distribution of a particular sample data given as input to the algorithm, but on the behavior of a random function. The only method that we will modify from our previous version is the partition, so that it is now randomized, but it will only assign the pivot randomly, and then call our original `partition` method.

```
randomizedPartition(A,first,last)
   r=random(first,last)
   exchange A[r] with A[last]
   p=partition(A,first,last)
   return p
```

We are interested in counting the number of comparisons of array elements. Notice that the only place where such a comparison takes place is in the `partition` algorithm where we do `if ([j] <= pivot)`, and in this comparison we always compare to the pivot, so we are interested ONLY in how many times throughout the entire algorithm pivot elements are compared!

### 1.2 Counting comparisons to pivot elements

**BIG idea:** We are not going to look at the actual data as it is being processed by the algorithm, it will make things more complicated than they should. What we will do is to look at the sorted version of the data:

$$z_1, z_2, z_3, \ldots, z_n$$

Notice that:

1. Array elements are ONLY compared to the pivot.
2. Once the pivot has been selected (placed in its proper position) it is never compared to any other value again.

Therefore, each pair $z_i$, $z_j$ of elements is compared at most once.

### 1.3 Counting comparisons (cont'd)

If we look at the elements form $z_i$ to $z_j$ we can have the following cases based on when an element between $z_i$ and $z_j$ is FIRST selected as a pivot, i.e. $z_i$ and $z_j$ belong to the same partition at this time:

1. Some element $x$ between $z_i$ and $z_j$ is selected as pivot but it is NOT $z_i$ or $z_j$.
   **Comparisons:** In this case $z_i$ will end up in a different partition than $z_j$ and therefore, will never be compared.

2. $z_i$ is selected as a pivot.
   **Comparisons:** In this case, since the pivot is compared once to each other element, then it will be compared to $z_j$

3. $z_j$ is selected as a pivot.
   **Comparisons:** In this case, since the pivot is compared once to each other element, then it will be compared to $z_i$

Since we are using a randomized algorithm we will assume that any element from the set $z_i, \ldots z_j$ is equally likely to be the pivot, and since there are $j - i + 1$ elements in that set, we have that the expected value of the number of comparisons made by the algorithm on $z_i$ and $z_j$ is

$$E_{ij} = 1\frac{1}{j-i+1} + 0\frac{1}{j-i+1} + \ldots + 0\frac{1}{j-i+1} + 1\frac{1}{j-i+1} = \frac{2}{j-i+1}$$

### 1.4 Expected value

What we want to compute is the expected value of the total number of comparisons ($Avg(n)$), i.e. the sum of the expected values for each pair:

$$\begin{aligned}
Avg(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left( \frac{2}{j-i+1} \right) \\
\text{(changing variable } k = j - i) &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} \\
&\leq 2 \sum_{i=1}^{n-1} \lg n \\
&= 2(n-1) \lg n
\end{aligned}$$

Therefore, the average number of comparisons made by the `randomized quicksort algorithm` is

$$Avg(n) \in O(n \lg n)$$

**Mergesort** uses the Divide and Conquer strategy:

- Divide the lists easily into two sublists (half of the elements go to one list, the other half goes to the second list)
- Recursive sort each list
- Carefully combine both lists using a merge algorithm

## 2 Mergesort

In this strategy we split the initial list in half and recursively sort each one of the sublists. The sorted lists are not easily combined, and need to be merged using a Merge algorithm.

In the following algorithm we are given two SORTED arrays `x[1...n]` and `y[1...m]`. The algorithm returns an array with the elements of `x` and `y` sorted.

IMPORTANT: this algorithm does not work if the input arrays are not sorted.

```
Algorithm Merge(a,first,middle,last)
i=first
j=middle+1
for k=first to last
   if (i<=middle) and (j<=last)
      if (a[i]<a[j])
         b[k]=a[i]
         i=i+1
      else
         b[k]=a[j]
         j=j+1
   else if i>middle then
      b[k]=a[j]
      j=j+1
   else
      b[k]=a[i]
      i=i+1
for k=first to last
   a[k]=b[k]
```

We can use this algorithm in a recursive version of our Mergesort algorithm:

```
Algorithm MergeSort(a,first,last) // array a[1..n]
if first<last
   middle = floor((first+last)/2)
   MergeSort(a,first,middle)
   MergeSort(a,middle+1,last)
   Merge(a,first,middle,last)
```

**Example**

5 2 4 7 1 3 2 6