Recitation 8. Analysis of Mergesort and Quicksort

In this recition, we analyze mergesort and quicksort, deriving their Order of execution time. We also look carefully at Quicksort and show how to minimize the space it needs.

Order of execution time of mergesort

```
// sort b[h..k]
public static void mergesort(int[] b, int h, int k) {
    if (k+1-h <= 1)
        return;
    int e= (h+k+1)/2;
    mergesort(b, h, e-1);
    mergesort(b, e, k);
    merge(b, h, e, k);
}
```

We know that merge(b, h, e, k) takes time proportional to the number of elements in b[h..k], i.e. it is an O(k+1-h) algorithm. Suppose it takes s*(k+1-h) steps. Based on this assumption, we figure out the order of execution of mergesort.

Define T(n) to be the number of steps it takes to mergessort an array of size n.

We have:

2^0: T(1) = 1 (assume 1 unit of time for executing the base case)

2^1: T(2)
= <look at the function body>
2*T(1) + 2s
= <use T(1)>
2*1 + 2s Note that $2 = 2^1 * 1$

2^2: T(4)
= <look at the function body>
2*T(2) + 4s
= <use T(2)>
2*(2+2s) + 4s
= <arithmetic>
4 + 8s Note that $8 = 2^2 * 2$

2^3: T(8)
= <look at the function body>
2*T(4) + 8s
= <use T(4)>
2*(4+8s) + 8s
= <arithmetic>
8 + 24s Note that $16 = 2^3 * 3$

We see a pattern here: $T(2^p) = 2^p + 2^p * p * s$
or: $T(n) = n + n*\log(n)*s$

And indeed, we can prove this formula using INDUCTION. Proving a theorem by induction is akin to understanding a recursive function. We prove the theorem for a base case. Then, we prove it for the recursive (inductive) case under the assumption that it holds for smaller cases.

Theorem: $T(2^p) = 2^p + 2^p * p * s$

Proof of base case: $T(2^0) = T(1) = 1$ (earlier analysis)

Proof of recursive case: Assume p>0. We prove the theorem under the assumption that

$T(2^k) = 2^k + 2^k * k * s$

holds for 0<=k<p:

$T(2^p)$
= <look at the function body>
$2*T(2^{p-1}) + 2^p s$
= <use assumption, with k=p-1>
$2*(2^{p-1} + 2^{p-1} * (p-1)*s) + 2^p s$
= <arithmetic>
$2^p + 2^p * (p-1)*s + 2^p s$
= <arithmetic>
$2^p + 2^p * p * s$

Isn't that simple?

From the theorem, we see that mergesort takes time O(n log n) to sort an array of size n.

Discovering the theorem. Take a look at how we discovered the theorem. We calculated T(1), T(2), T(4), T(8), etc. until we saw a pattern. Then, we just formulated the theorem as that pattern.

A more general theorem

We have proved that mergesort takes time O(n log n) to sort an array of size n, when n is a power of 2. It is easy to show that it then takes O(n log n) time for an array of any size.

But we can do more. The analysis that we did holds whenever the recursive method satisfies certain assumptions, which we describe in this theorem:

Theorem: Suppose a recursive method that processes an array takes 1 step for an array of size 0 or 1 and, for an array of size n>1 does two things (in any order):

- Performs O(n) steps, processing the array.
- Recursively calls itself twice to process the first half and the last half of the array, of the same size.

Then the algorithm takes time O(n log n) to process an array of size n.

```
// sort b[h..k]
public static void quicksort(int b, int h, int k) {
    if (k+1-h <= 1)
        { return; }
    int j= partition(b,h.k);
    // {b[h..j-1] <= b[j] <= b[j+1..k]}
    quicksort(b,h,j-1);
    quicksort(b,j+1,k);
}
```

Look at quicksort. In the best case, each call on partition partitions the array into two equal halves -- at least, each contains no more than 1/2 of the array. Quicksort then partitions these two halves. Further, method partition takes time proportional to the size of the array segment that it is partitioning. Therefore, in the best case, quicksort takes time O(n log n) to sort an array of size n.

It has been shown that the expected or average case time for quicksort is also O(n log n), but the analysis is beyond the scope of CS211.

In the worst case, quicksort is $O(n^2)$.

Fixing Quicksort so that it takes at most O(log n) space to sort an array of size n and so that it is more efficient. But the original quicksort has problems..

1. The pivot value b[h] may be the smallest element of the array, and if so, partition creates one segment, b[h..j-1], that is empty and one segment, b[j+1..k] that contains all but one element. If this happened at each iteration, the depth of recursion would be k-h, so O(k-h), that is, linear space and would take $O(k-h)^2)$ time.

We can't solve this problem completely. But we can help it a bit by making b[h] be the median of three of the array values before doing the partition. This give more chance that partition will produce segments of nearly equal size.

2. Quicksort is particularly inefficient on small arrays, because of all the method calls. By experiment, it has been determined that insertion sort does better on arrays of about 10 or fewer elements. So, we change the base case from an array of size 1 or less to an array of size 10 or less and use insertion sort to sort it.

3. We have not solved the problem that in some cases the depth of recursion will be linear in the size of the array, so that the method will take time proportional to the size of the array.

If we can make the depth of recursion at most logarithmic in the size of the array, then space will also be logarithmic. At each call, we have two segments to sort: b[h..j-1] and b[j+1..k]. We solve our problem by sorting only the smallest one recursively and using iteration for the other. This works because the smaller one is smaller than half the array size, so that at each recursive call, the array size is at least halved, leading to logarithmic recursion depth.

```
// Sort b[h1..k1]
public static void quicksort (int[] b, int h1, int k1) {
    int h= h1; int k= k1;
    // inv:   b[h1..k1] is in ascending order
    //        if and only if b[h..k] is in ascending order
    // bound function: size of segment b[h..k]
    while (true) {
        if (k+1-h <= 10) {
            insertionSort(b,h,k);
            return;
        }

        medianOf3(b,h,k);
        // {b[h] is between b[(j+k/2)] and b[k]}

        int j= partition(b,h,k);
        // {b[h..j-1] <= b[j] <= b[j+1..k]}

        if (j-h <= k-j) {
            quicksort(b,h,j-1); // sort the smaller segment
            // {b[h1..k1] is sorted if b[j+1..k] is}
            h= j+1;
            }
        else {
            quicksort(b,j+1,k); // sort the smaller segment
            // {b[h1..k1] is sorted if b[h..j-1] is}
            k= j-1;
        }
    }
}

// Permute b[h], b[k], and b[(h+k)/2] to store their
// median in b[h]
public static void medianOf3(int b, int h, int k)
```

Partition algorithm (used in quicksort)

```
/** b[h..k] contains at least one value. Use "x"
    to name the value in b[h]. Permute b[h..k]
    and return a value j so that:
       b[h..j-1]  <=  x
       b[j]  =  x
       b[j+1]  >=  x
public static int partition(int[[] b, int h, int k) {
   int i= h+1; int j= k;
   /* invariant: to the right */
  while ( i < j) {
     if (b[i] <= x)  i= i+1;
     else  {Swap b[i] and b[j];  j= j-1; }
  }
}
```