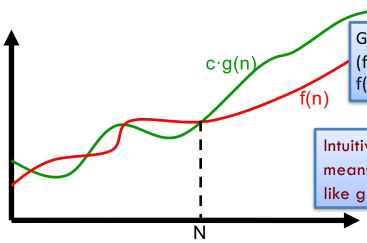
Analysis of Algorithms

Recitation 9

REVIEW OF BIG-O

"Big O" Notation

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$



Get out far enough (for $n \geq N$) f(n) is at most c·g(n)

Intuitively, f(n) is O(g(n)) means that f(n) grows like g(n) or slower

Prove that ($n^2 + n$) is O(n^2)

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Let f(n) and g(n) be two functions. f(n) >= 0 and g(n) >= 0.

Example:
f(n) = n + 6
g(n) = n
We show that n+6 is O(n)

n + 6 ---this is f(n)
<= <if 6 <= n, write as>
n + n
= <arith>
2*n
<choose c = 2>
= c*n ---this is c * g(n)

So choose c = 2 and N = 6

Prove that ($n^2 + n$) is O(n^2)

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

Let f(n) and g(n) be two functions. f(n) >= 0 and g(n) >= 0.

We showed that n+6 is O(n). In fact, you can change the 6 to any constant c you want and show that n+c is O(n)

An algorithm that executes O(n) steps on input of size n is called a linear algorithm

It means that as n gets larger and larger, any constant c that you use becomes meaningless in relation to n, so throw it away.

The difference between executing 1,000,000 steps and 1,000,0006 is insignificant

Oft-used execution orders

In the same way, we can prove these kinds of things:

1.	log(n) + 20	is	O(log(n))	(logarithmic)
2.	n + log(n)	is	O(n)	(linear)
3.	n/2 and 3*n	are	O(n)	
4.	n * log(n) + n	is	n * log(n)	
5.	$n^2 + 2*n + 6$	is	O(n^2)	(quadratic)
6.	$n^3 + n^2$	is	O(n^3)	(cubic)
7.	$2^n + 5n$	is	O(2^n)	(exponential)

Understand? Then use informally

1. log(n) + 20	is	O(log(n))	(logarithmic)
2. n + log(n)	is	O(n)	(linear)
3. n/2 and 3*n	are	O(n)	
4. n * log(n) + n	is	n * log(n)	
5. n^2 + 2*n + 6	is	$O(n^2)$	(quadratic)
6. $n^3 + n^2$	is	$O(n^3)$	(cubic)
7. 2^n + 5n	is	$O(2^n)$	(exponential)

Once you fully understand the concept, you can use it informally. Example:

An algorithm executes (7*n + 6) / 3 + log(n) steps. It's obviously linear, i.e. O(n)

Some Notes on O()

- Why don't logarithm bases matter?
  - For constants x, y: $O(\log_x n) = O((\log_x y)(\log_y n))$
  - Since $(\log_x y)$ is a constant, $O(\log_x n) = O(\log_y n)$
- Usually: O(f(n)) × O(g(n)) = O(f(n) × g(n))
  - Such as if something that takes g(n) time for each of f(n) repetitions . . . (loop within a loop)
- Usually: O(f(n)) + O(g(n)) = O(max(f(n), g(n)))
  - "max" is whatever's dominant as n approaches infinity
  - Example: $O((n^2-n)/2) = O((1/2)n^2 + (-1/2)n) = O((1/2)n^2)$ $= O(n^2)$

ANALYZING AN ALGORITHM

runtimeof MergeSort

```
/** Sort b[h..k]. */
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;

    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}
```

Throughout, we use mS for mergeSort, to make slides easier to read

We will count the number of comparisons mS makes

Use T(n) for the number of array element comparisons that mS makes on an array of size n

Runtime

```
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;

    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}
```

T(0) = 0
T(1) = 0

Use T(n) for the number of array element comparisons that mS makes on an array of size n

Runtime

```
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;

    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}
```

Recursion: T(n) = 2 * T(n/2) + comparisons made in merge

Simplify calculations: assume n is a power of 2

```
/** Sort b[h..k].
    Pre: b[h..e] and b[e+1..k] are sorted.*/
public static void merge (Comparable b[], int h, int e, int k) {
    Comparable[] c= copy(b, h, e);
    int i= h; int j= e+1; int m= 0;
    /* inv: b[h..i-1] contains its final, sorted values
            b[j..k] remains to be transferred
            c[m..e-h] remains to be transferred */
    for (i= h;  i != k+1;  i++) {
        if (j <= k && (m > e-h || b[j].compareTo(c[m]) <= 0)) {
            b[i]= b[j]; j++;
        }
        else {
            b[i]= c[m]; m++;
        }
    }
}
```

	0		m		e-h
c	free		to be moved		

	h		i		j		k
b	final, sorted		free		to be moved		

```
/** Sort b[h..k].   Pre: b[h..e] and b[e+1..k] are already sorted.*/
public static void merge (Comparable b[], int h, int e, int k) {
    Comparable[] c= copy(b, h, e);
    int i= h; int j= e+1; int m= 0;
    for (i= h;  i != k+1;  i= i+1) {
        if (j <= k && (m > e-h || b[j].compareTo(c[m]) <= 0)) {
            b[i]= b[j]; j= j+1;
        }
        else {
            b[i]= c[m]; m= m+1;
        }
    }
}
```

O(e+1-h)

Loop body: O(1).
Executed k+1-h times.

Number of array element comparisons is the size of the array segment – 1.
Simplify: use the size of the array segment O(k-h) time

Runtime

We show how to do an analysis, assuming n is a power of 2 (just to simplify the calculations)

Use T(n) for number of array element comparisons to mergesort an array segment of size n

```
public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    mS(b, h, e);        T(e+1-h) comparisons
    mS(b, e+1, k);      T(k-e) comparisons
    merge(b, h, e, k);  (k+1-h) comparisons
}
```

Thus: $T(n) < 2\ T(n/2) + n$, with $T(0) = 0$, $T(1) = 0$

Runtime

Thus, for any n a power of 2, we have

$T(1) = 0$

$T(n) = 2*T(n/2) + n$ for $n > 1$

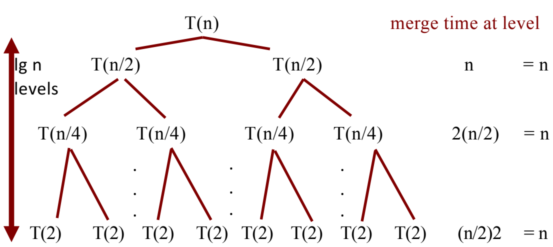
We can prove that

$T(n) = n \lg n$

lg n means $\log_2 n$

Proof by recursion tree of T(n) = n lg n

$T(n) = 2*T(n/2) + n$, for $n > 1$, a power of 2, and $T(1) = 0$



Each level requires n comparisons to merge. lg n levels.
Therefore T(n) = n lg n   mergeSort has time O(n lg n)

MergeSort vs QuickSort

- Covered QuickSort in Lecture
- MergeSort requires extra space in memory
  - The way we've coded it, it needs that extra array
  - QuickSort is an "in place" or "in situ" algorithm. No extra array. But it does require space for stack frame for re-cursive calls. Naïve algorithm: O(n), but can make O(log n)
- Both have "average case" O(n lg n) runtime
  - MergeSort always has O(n lg n) runtime
  - Quicksort has "worst case" $O(n^2)$ runtime
    - Let's prove it!

Quicksort

h		j		k
	<= x	x	>= x	

- Pick some "pivot" value in the array
- Partition the array:
  - Finish with the pivot value at some index j
  - everything to the left of j ≤ the pivot
  - everything to the right of j ≥ the pivot
- Run QuickSort on b[h..j-1] and b[j+1..k]

Runtime of Quicksort

- Base case: array segment of 0 or 1 elements takes no comparisons
  T(0) = T(1) = 0
- Recursion:
  - partitioning an array segment of n elements takes n comparisons to some pivot
  - Partition creates length m and r segments (where m + r = n-1)
  - T(n) = n + T(m) + T(r)

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

Runtime of Quicksort

- T(n) = n + T(m) + T(r)
  - Look familiar?
- If m and r are balanced (m ≈ r ≈ (n-1)/2), we know T(n) = n lg n.
- Other extreme:
  - m=n-1, r=0
  - T(n) = n + T(n-1) + T(0)


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


Worst Case Runtime of Quicksort

- When T(n) = n + T(n-1) + T(0)
- Hypothesis: T(n) = $(n^2 – n)/2$
- Base Case: T(1) = $(1^2 –1)/2=0$
- Inductive Hypothesis:
  assume $T(k)=(k^2-k)/2$
  $T(k+1) = k + (k^2-k)/2 + 0$
  $= (k^2+k)/2$
  $= ((k+1)^2 –(k+1))/2$
- Therefore, for all n ≥ 1:
  $T(n) = (n^2 – n)/2 = O(n^2)$


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


Worst Case Space of Quicksort

You can see that in the worst case, the depth of recursion is O(n). Since each recursive call involves creating a new stack frame, which takes space, in the worst case, Quicksort takes space O(n).

That is not good!

To get around this, rewrite QuickSort so that it is iterative but it sorts the smaller of two segments recursively. It is easy to do. The implementation in the java class that is on the website shows this.


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