# Recitation 11

Analysis of Algorithms and inductive proofs





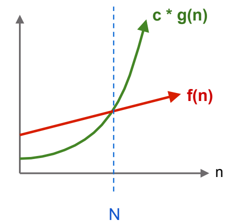
## Review: Big O definition

f(n) is O(g(n))

iff

There exists c > 0 and N > 0 such that:

**f(n) ≤ c * g(n)** for n ≥ N






## Example: 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

f(n) is O(g(n)): There exist c > 0, N > 0 such that:
**f(n) ≤ c * g(n)** for n ≥ N





## Review: Big O

Is used to classify algorithms by how they respond to changes in input size n.

**Important vocabulary:**

- Constant time: O(1)
- Logarithmic time: O(log n)
- Linear time: O(n)
- Quadratic time: $O(n^2)$
- Exponential time: $O(2^n)$





## Review: Big O

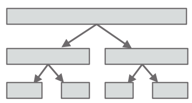
| | | | |
|---|---|---|---|
| 1. log(n) + 20 | is | O(log(n)) | (logarithmic) |
| 2. n + log(n) | is | O(n) | (linear) |
| 3. n/2 and 3*n | are | O(n) | |
| 4. n * log(n) + n | is | n * log(n) | |
| 5. $n^2$ + 2*n + 6 | is | $O(n^2)$ | (quadratic) |
| 6. $n^3$ + $n^2$ | is | $O(n^3)$ | (cubic) |
| 7. $2^n$ + n5 | is | $O(2^n)$ | (exponential) |



# Merge Sort

## Runtime of merge sort

```
/** 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);
}
```



mS is mergeSort for readability





## Runtime of merge sort

```
/** 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);
}
```

- 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 segment of size *n*

mS is mergeSort for readability





## Runtime of merge sort

```
/** 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);
}
```

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

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





## Runtime of merge sort

```
/** 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);
}
```

**Recursive Case:**
**T(n) = 2 * T(n/2) + O(n)**

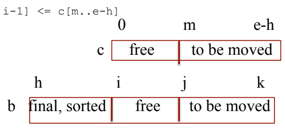
comparisons made in merge

Simplify calculations: assume *n* is a power of 2



## Runtime of merge

```
/** 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;
  // inv: b[h..i-1] contains final, sorted values
  //   b[j..k] remains to be transferred
  //   c[m..e-h] remains to be transferred
  //   b[0..i-1] <= b[j..k], b[h..i-1] <= c[m..e-h]
  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;
  }
}
```






## Runtime of merge

```
/** 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)** (for `Comparable[] c= copy(b, h, e);`)

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

**Overall: O(k-h)**

## Runtime of merge sort

```
/** 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);          T(e+1-h) comparisons = T(n/2)
    mS(b, e+1, k);        T(k-e)   comparisons = T(n/2)
    merge(b, h, e, k);    T(k+1-h) comparisons = n
}

     Thus: T(n) < 2 * T(n/2) + n, with 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

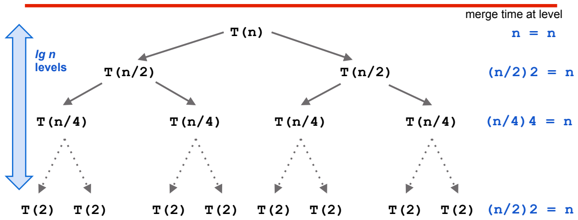
T(n) = n lg n

lg n means $\log_2 n$





## Proof by recursion tree




lg n levels * n comparisons is O(n log n)





## Inductive proof

Definition: T(n) by: T(1) = 0
T(n) = 2T(n/2) + n

Theorem: For n a power of 2, P(n) holds, where:
P(n): T(n) = n lg n

Proof by induction:

Base case: n = 1: P(1) is T(1) = 1 lg 1
T(1) = 0, by definition.
1 = 2^0, so 1 lg 1 = 0.

lg n means $\log_2 n$





## Inductive proof

Inductive case: Assume P(k), where k is a power of 2, and prove P(2k)

T(1) = 0
T(n) = 2T(n/2) + n

P(n): T(n) = n lg n

T(2k)
= <def of T>
2 k lg k + 2k
= <algebra>
2 k (lg(2k) – 1) + 2k
= <algebra>
2 k lg(2k)

Why is lg n = lg(2n) -1 ?
Rests on properties of lg.
See next slids





## Proof: lg(n) = lg(2n) - 1

Since $n = 2^k$ for some k:

lg(2n) – 1
= <definition of n>
$\lg(2*2^k) – 1$
= <arith>
$\lg(2^1 2^k) – 1$
= <property of lg>
$\lg(2^1) + \lg(2^k) – 1$
= <arith>
$1 + \lg(2^k) – 1$
= <arith, definition of n>
lg n

lg n means $\log_2 n$

Thus, if $n = 2^k$
lg n = k

## Merge sort vs Quicksort

- Covered QuickSort in Lecture
- MergeSort requires extra space in memory
  - It requires an extra array c, whose size is ½ the initial array size
  - QuickSort is an "in place" algorithm but requires space proportional to the depth of recursion
- 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!

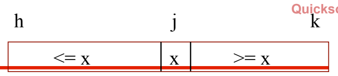


## Quicksort





## Quicksort




- 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 the array segment to the left of j, and on the array segment to the right of j





## 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 (h ≥ k) 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.

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





## Runtime of Quicksort

Look at case where pivot is always the smallest (or largest) element. Be careful about how many comparisons the partition algorithm makes.

To partition an array of n elements takes n-1 comparisons (not n).

If the pivot is always the smallest, then one of b[h..i-1] and b[j+1..k] is empty and the other has n-1 elements.

Recurrence relation for number of comparisons shown to the right:

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

T(0) = 0
T(1) = 0
T(n) = n-1 + T(n-1) for n > 1

# Worst-case Quicksort

T(0) = 0, T(1) = 1
T(n) = n-1 + T(n-1) for n > 1

**Theorem**: For n >= 0, P(n) holds, where

P(n): $T(n) = (n^2 - n) / 2$

Proof:

Base Cases: T(0) and T(1) are easy to see, by def of T(0) and T(1).

**Inductive case**: Assume P(k-1).

Proof of P(k) shown at right, starting with the definition of T(k)

k-1 + T(k-1)
= <Assumption P(k-1)>
$k-1 + ((k-1)^2 - (k-1)) / 2$
= <arithmetic –divide/multiply first term by 2 and add terms >
$((k-1)^2 + (k-1)) / 2$
= <factor out k-1>
((k-1) (k-1+1)) / 2
= <-1+1 = 0>
((k-1) (k)) / 2
= <arithmetic>
$(k^2 - k) / 2$





# Runtime of Quicksort

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 sorts the smaller of the 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 (h ≥ k) return;
   int j=  partition(b, h, k);
   QS(b, h, j-1);
   QS(b, j+1, k);
}
```