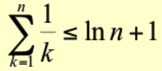
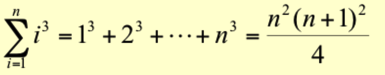
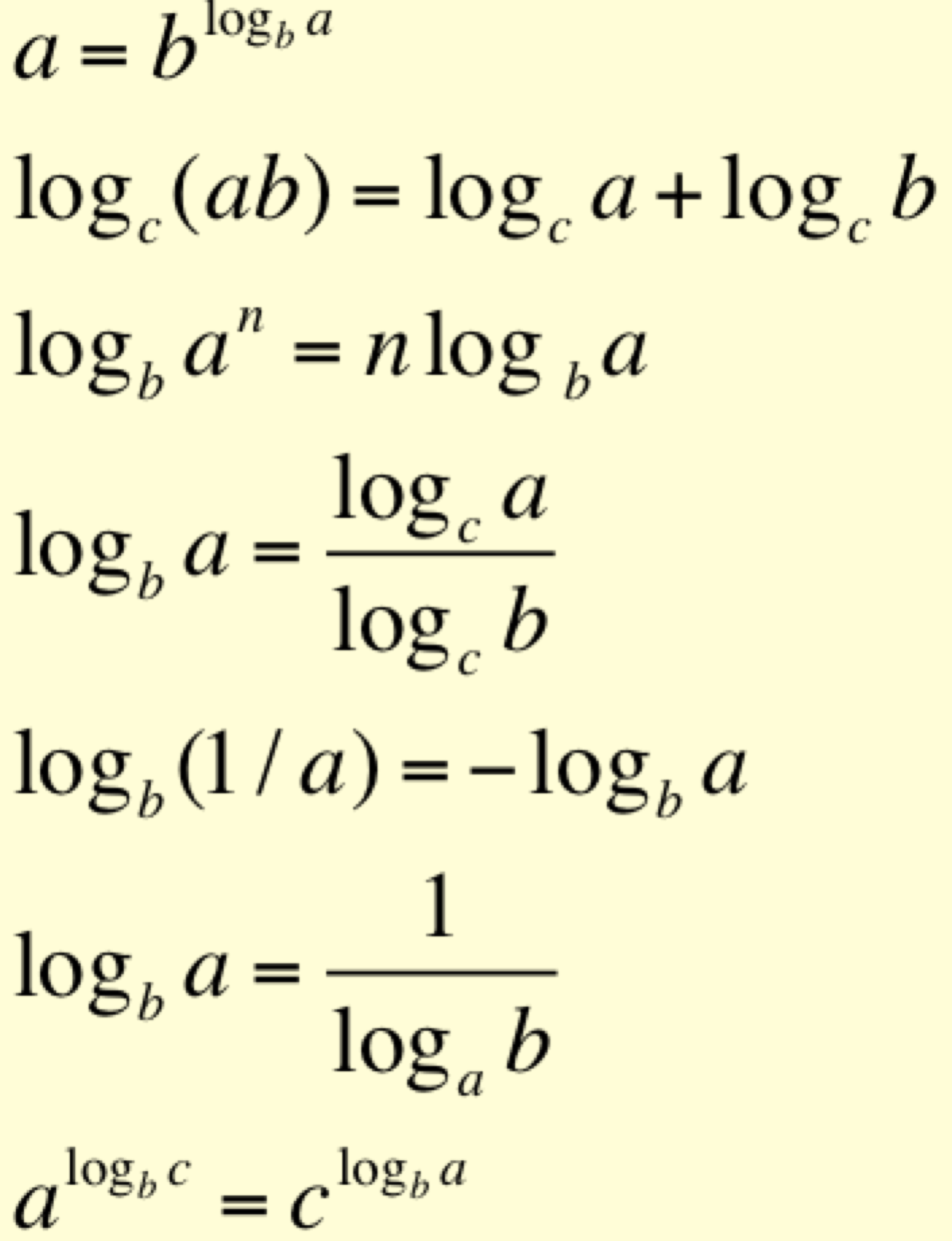
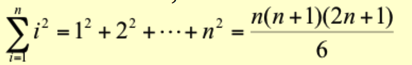
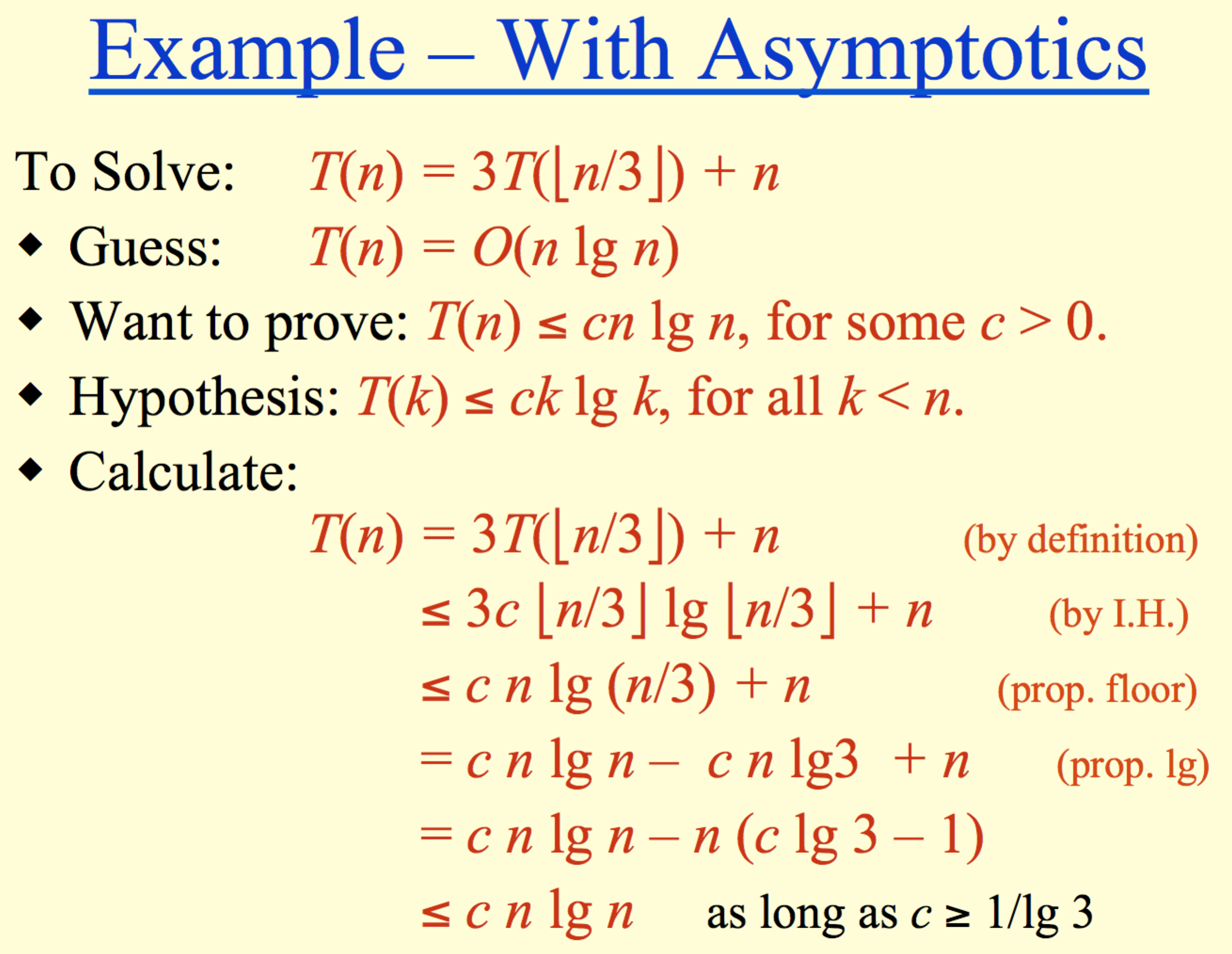
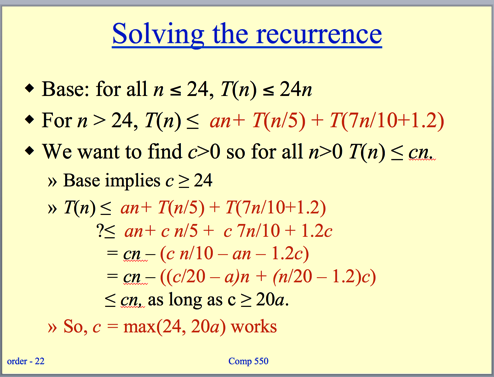
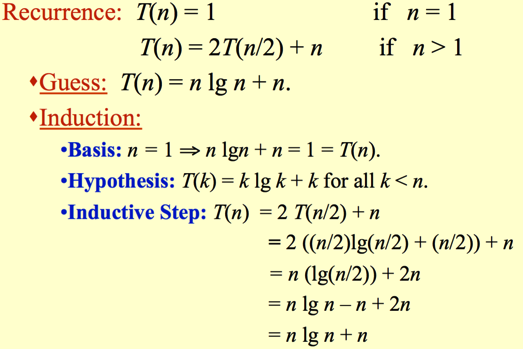
**Red black trees:** binary trees that are ensured balance. Take O(log n) in worst case. || You can build a **min/max heap** in O(n) time. || **Quick sort is worst** when array is already sorted or reverse sorted. **|| Quick sort is not very efficient** on small lists and is slow if there are many identical keys. || **Quick sort** is not stable. || **Stable sorting algorithm**: a sort in which equal items maintain the same relative order. || We can make **quicksort** run in Theta(n log n) using median finding. Can get ith largest element in theta(n) worst case using median. || **Randomized quick sort** is no different best/worse case than quicksort. ||

**Insertion sort:** Like one sorts cards. Consider looking through the cards one by one from left to right, and then any time you see something that should be lower you move all the cards up and put that card in the correct spot.

**Merge sort:** Recursive call to merge sort on two halves of the call, and at the end a call to merge which will actually do the work for us of combining sorts.

**Asymptotic bounds** refer to O, Omega, etc.



**Quick sort:** (again recursive) You can have non-random and random. In non-random, you choose the last element to be the pivot, and then you organize everything into what’s less than and what’s greater than the pivot and then once you’ve compared everything you put the pivot right between those two groups. Then, you call again on the two groups, those greater and those less than the pivot. The loop invariants are that everything below is in one area, above is in another, and pivot is certain index.

*Partition(A, p, r)*

*x, i := A[r], p – 1;*

*for j := p to r – 1 do*

*if A[j] ≤ x then*

*i := i + 1;*

*A[i] ↔ A[j]*

*fi*

*od;*

*A[i + 1] ↔ A[r];*

*return i + 1*

***Quicksort(A, p, r)***

***if p < r then***

*q := Partition(A, p, r);*

*Quicksort(A, p, q – 1);*

*Quicksort(A, q + 1, r)*

***fi***

A[p..i] — All entries in this region are ≤ pivot.

A[i+1..j – 1] — All entries in this region are > pivot.

A[r] = pivot.

**Fix 2:** Median-of-three Quicksort.

Use median of three fixed elements (say, the first, middle, and last) as the pivot.

To get O(*n*2) behavior, we must continually be unlucky to see that two out of the three elements examined are among the largest or smallest of their sets

Worst case for Quicksort is when the partition is T(n) = T(n-1) + T(0) + PartitionTime(N) = T(n-1) + c\*n. = O(n^2).

**Invariant**: *at the start of each* ***for*** *loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order; all other elements are unchanged*

**Termination:** the loop terminates, when *j=n+1*. Then the invariant states: *“A[1…n] consists of elements originally in A[1…n] but in sorted order.”*

**Maintenance**: the inner **while** loop finds the position *i* with *A*[*i*] *<= key,* and shifts *A*[*j-1*], *A*[*j-2*], …, *A*[*i+1*] right by one position. Then *key,* formerly known as *A*[*j*], is placed in position *i+1* so that *A*[*i*] **<=** *A*[*i+1*] **<** *A*[*i*+2].

*A*[*1…j-1*] sorted + *A*[*j*] -> *A*[*1…j*] sorted

**Initialization**: *j = 2,* the invariant trivially holds because *A*[1] is a sorted array. √

for j=2 to *length*(A)

do key=A[j]

i=j-1

while i>0 and A[i]>key

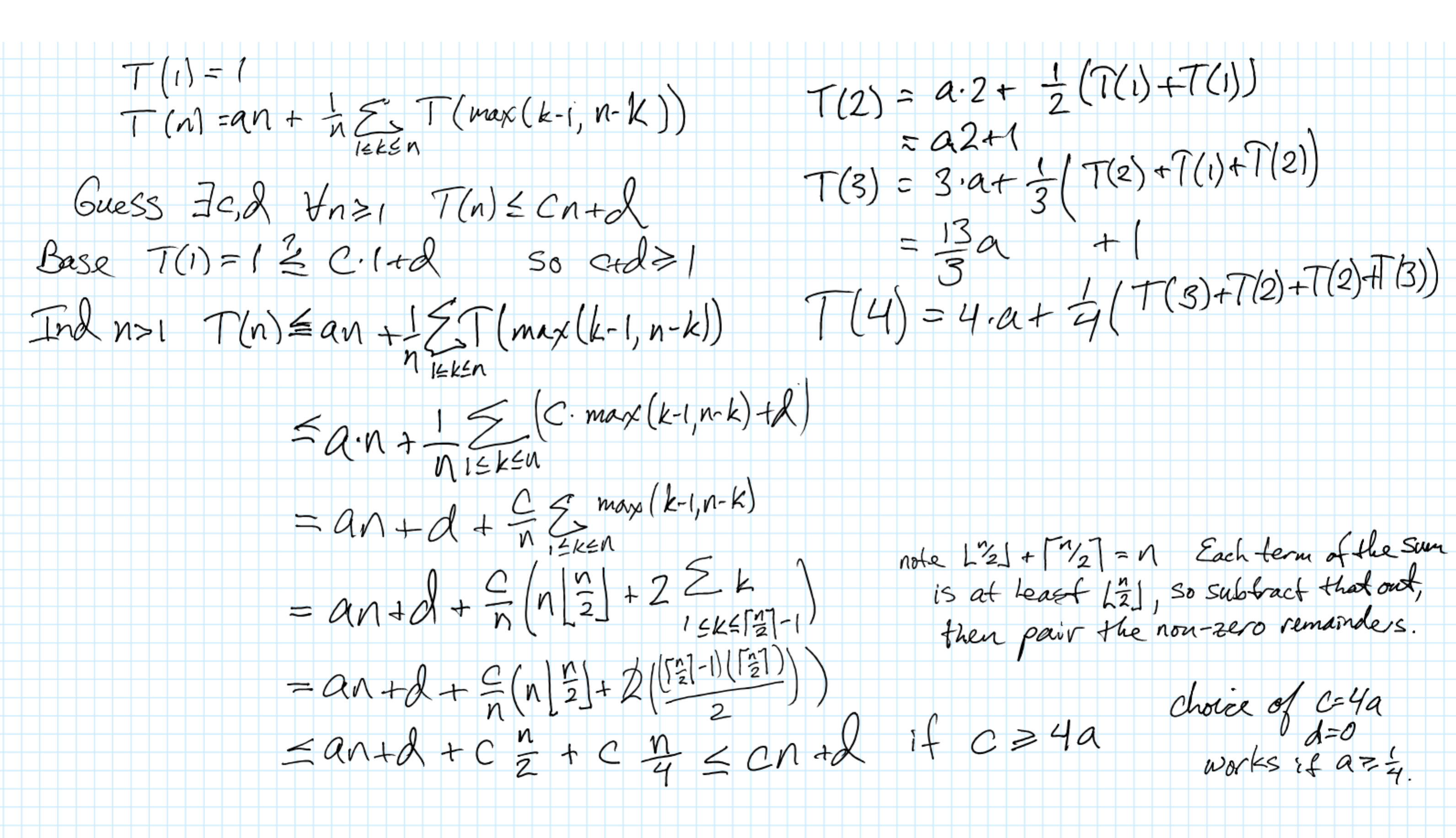
do A[i+1]=A[i]

i--

A[i+1]:=key

**Loop Invariant**

**Heap sort:** Put all the elements into a binary tree, and then pull them out one by one?



Pr{candidate *i* is hired}

*i* is hired only if *i* is better than 1, 2,…,*i*-1.

By assumption, candidates arrive in random order

Candidates 1, 2, …, *i* arrive in random order.

Each of the *i* candidates has an equal chance of being the best so far.

Pr{candidate *i* is the best so far} = 1/*i*.

E[*X*i] = 1/*i*. (By Lemma 5.1)

**Red black trees**

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

*f(n) ∈ O(g(n)) implies lg(f(n)) ∈ O(lg(g(n))),*

*where lg(g(n)) ≥ 1 and f(n) ≥ 1 for all sufficiently large n.*

we need to find c,n0 such that, for all n> n0, we have 0 ≤lg(f(n)) ≤ c lg(g(n)).

We know that f(n) = O(g(n)), so there exist d,n1 such that for all n>n1, we have f(n) ≤ d g(n).

We also know that there exists an n2 such that for all n>n2, we have lg(g(n)) ≥ 1 and f(n) ≥ 1.

Choose n0 = max(n0, n1, lg(d)). Then, for any n> n0, we know that 0 ≤ lg(f(n)), and that

lg(f(n)) ≤ lg(d g(n)), since lg is monotone increasing

= lg(d) + lg(g(n)) by property of lg

≤ (lg(d)+1) lg(g(n)), since lg(g(n)) ≥ 1

≤ c lg(g(n)), if we choose any c ≥ (lg(d)+1).

***MergeSort* (*A*, *p*, *r*) //** sort *A*[*p..r*] by divide & conquer

1. **if***p* < *r*
2. **then** *q* ← ⎣(*p*+*r*)/2⎦
3. *MergeSort* (*A*, *p*, *q*)
4. *MergeSort* (*A*, *q*+1, *r*)
5. *Merge* (*A*, *p*, *q*, *r*) // merges *A*[*p..q*] with *A*[*q+1..r*]

*Merge(A, p, q, r)*

*1 n1 ← q – p + 1*

*2 n2 ← r – q*

1. *for i ← 1 to n1*
2. *do L[i] ← A[p + i – 1]*
3. *for j ← 1 to n2*
4. *do R[j] ← A[q + j]*
5. *L[n1+1] ← ∞*
6. *R[n2+1] ← ∞*
7. *i ← 1*
8. *j ← 1*
9. *for k ←p to r*
10. *do if L[i] ≤ R[j]*
11. *then A[k] ← L[i]*
12. *i ← i + 1*
13. *else A[k] ← R[j]*
14. *j ← j + 1*