CSCI 3104: Algorithms

Lecture 6: Recurrence Relations – Unrolling, Tree, and Master Methods

Rachel Cox

Department of Computer Science

What will we learn today?

- ❑ MergeSort
- ❑ Recurrence Relations
- ❑ Unrolling Method
- ❑ Recursion Tree Method
- ❑ Master Method

Intro to Algorithms, CLRS:
Sections 2.3, 4.3, 4.4, 4,5

Divide_and_Conquer

Divide-and-Conquer: Solve recursively. Apply each step below for each recursive call.

- **Divide** the problem into smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems to achieve the final solution to the original problem.

Study MergeSort - divide + conquer.

Solving Recurrences

A general form of recurrence relations is as follows:

Θ, Ω, Θ

Running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Four Methods for Solving Recurrences:

- The Substitution Method
- Unrolling Method
- The Recursion-Tree Method
- The Master Method

$T(n)$ is the running time for a problem of size n.

$D(n)$ time to divide the problem into subproblems

$C(n)$ time to combine solutions to get final solution.

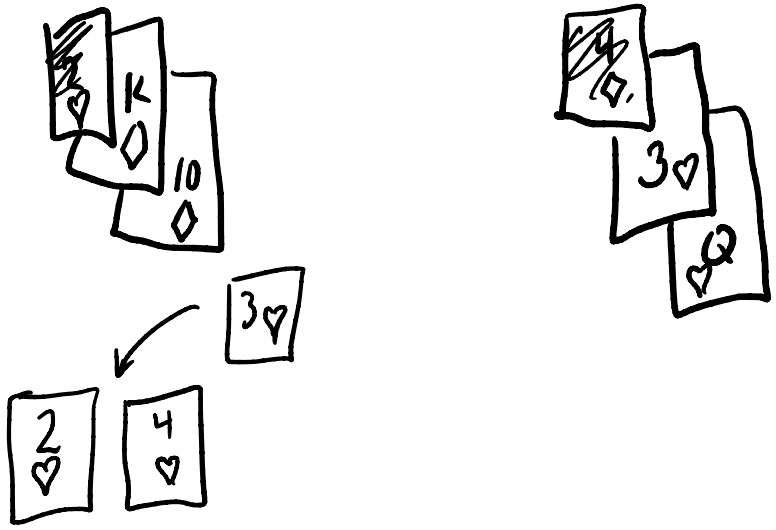
a Represents the number of divisions. (e.g. with binary search, $a = 2$)

n/b represents size of subproblem.

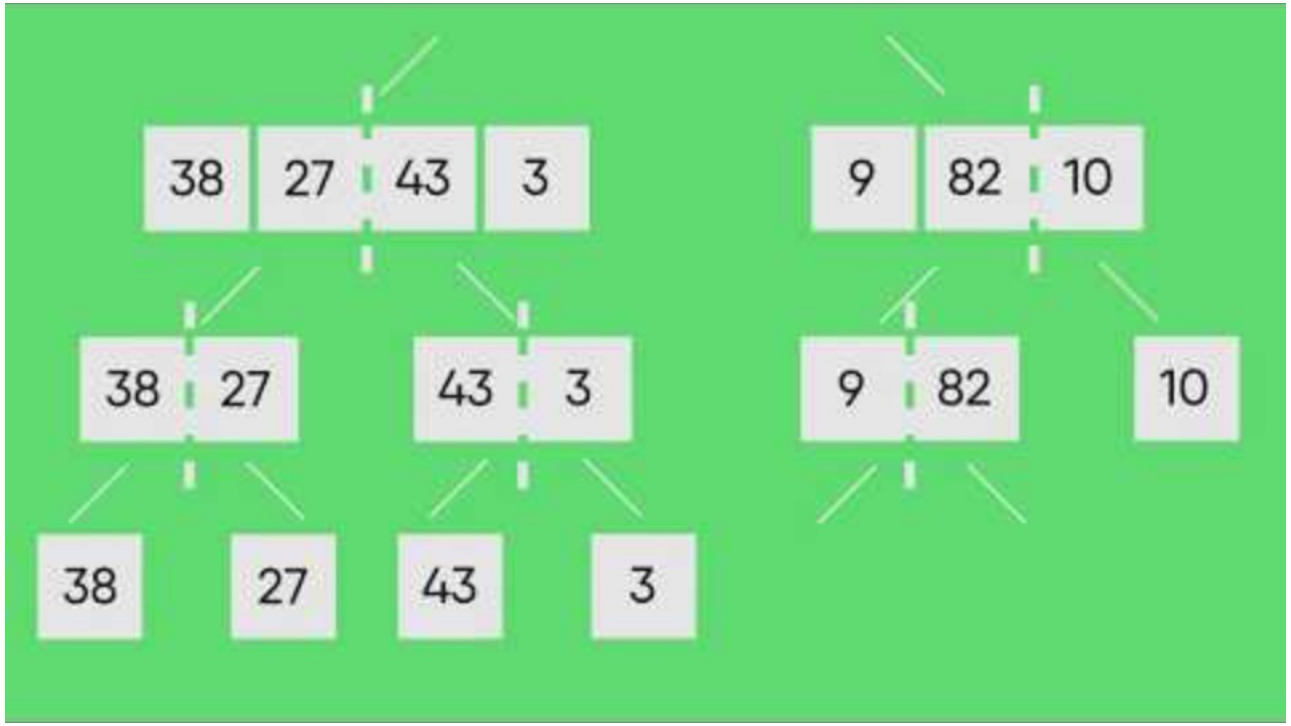
Merge Sort

Analogous to Sorting Cards:

- Imagine 2 piles of cards face up on a table.
- Each pile is sorted.
- We want to merge the two piles into a single sorted pile.



Nice Video of How MergeSort works



Merge Sort

Combine Step

begin index, mid, end index

$MERGE(A, p, q, r)$

constant time:
$n_1 = q - p + 1$
$n_2 = r - q$
} computing length of the subarrays

$let\ L[1..n_1 + 1]\ and\ R[1..n_2 + 1]\ be\ new\ arrays$

n_1:
for $i = 1$ **to** n_1
  $L[i] = A[p + i - 1]$

n_2:
for $j = 1$ **to** n_2
  $R[j] = A[q + j]$

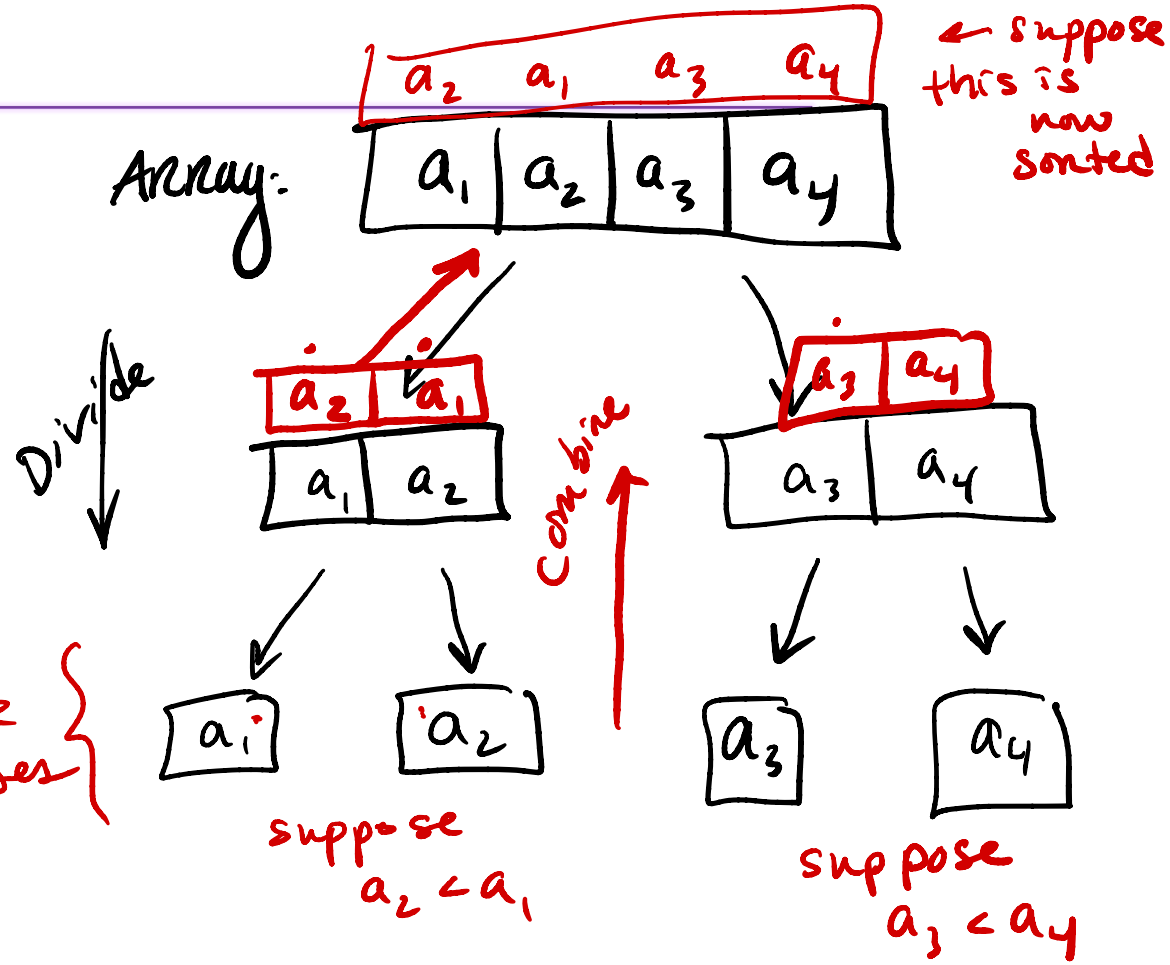
} copying A to the left subarray + right subarray

$\Theta(n_1 + n_2) = \Theta(n)$

$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
} sentinel value - meant to signal the end of the subarray

constant time:
$i = 1$
$j = 1$

n iterations - each in constant time:
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$
  else $A[k] = R[j]$
    $j = j + 1$



Runtime of Merge

Merge is $\Theta(n)$ time

very informal analysis.

Merge Sort

$MERGE-SORT(A, p, r)$

→ *if* $p < r$

$q = \lfloor (p+r)/2 \rfloor$

Divide { $MERGE-SORT(A, p, q)$ • recursive call on lower half

$MERGE-SORT(A, q+1, r)$ • recursive call on upper half

Combine { $MERGE(A, p, q, r)$ • then we merge.

we have A initially ; $A = \langle A[1], A[2], .. A[n] \rangle$

- we initially call MergeSort on the whole array.
- MergeSort (A , 1 , n) (p = 1, r = n)
- q - the midpoint index

e.g. $A = [4, 3, 2, 1]$

$p = 1 \quad r = 4$

if $p < r$

$q = \lfloor 5/2 \rfloor = 2$

MergeSort (A, 1, 2) ← first half

MergeSort (A, 3, 4) ← second half

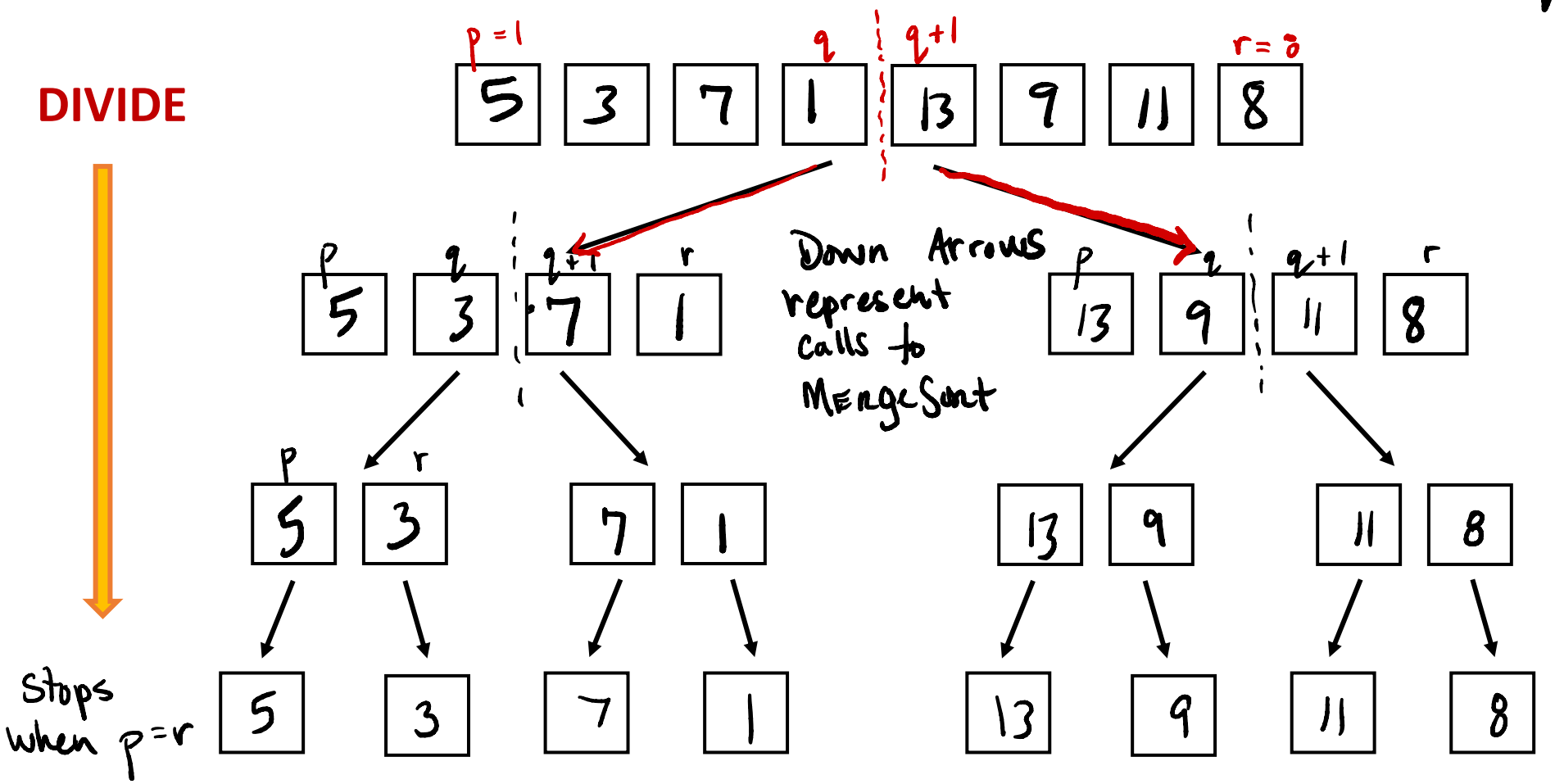
Merge (A, 1, 2, 4) ← combine step

Merge Sort

Example: Use Merge-Sort to sort the following array: $A = [5, 3, 7, 1, 13, 9, 11, 8]$

We divide first
→ we keep going until we reach the "base cases" aka lists of length 1

DIVIDE



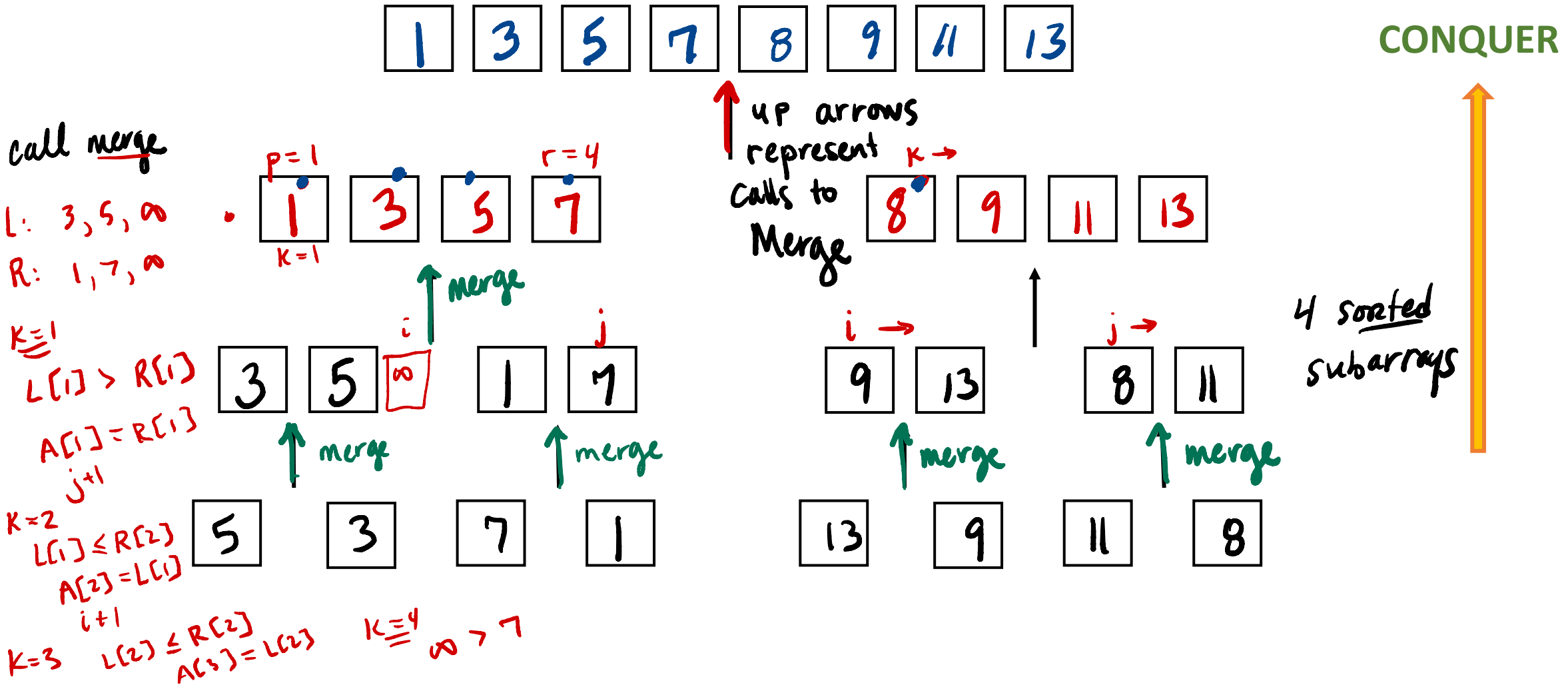
} Base cases

At this point, we have 8 sorted subarrays.

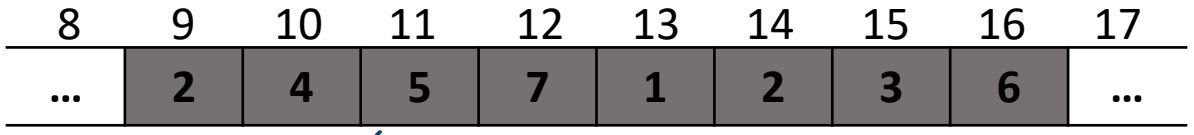
- our recursion "bottoms out" when $p = r$
- A list with one element is defined to be sorted!

Merge Sort

Example: Use Merge-Sort to sort the following array: $A = [5, 3, 7, 1, 13, 9, 11, 8]$

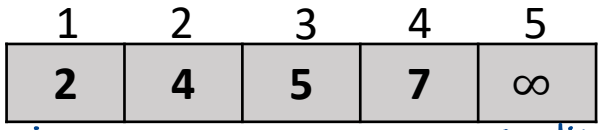


Merge Sort



Merge : $L[n_1+1]$ $R[n_2+1]$

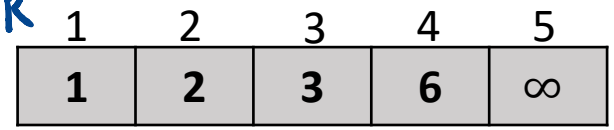
L



i=1

sentinel values

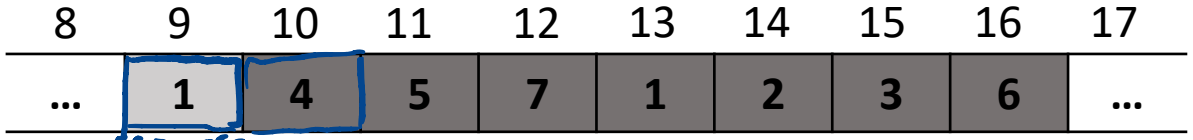
R



j=1

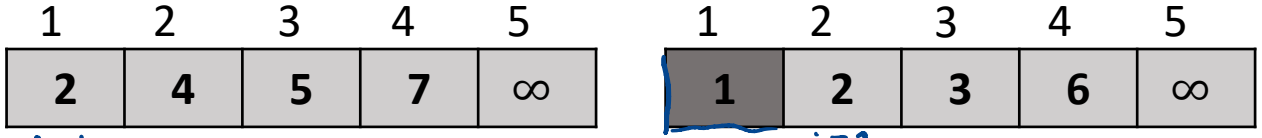
sentinel values

- Compare L[1] and R[1]



k index

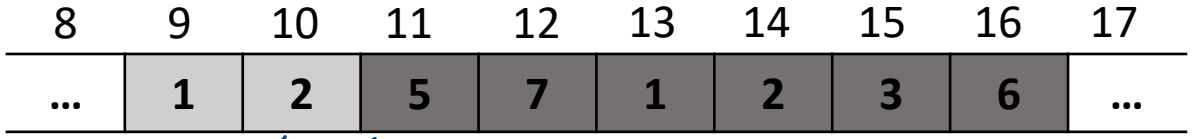
A[1]=R[1]



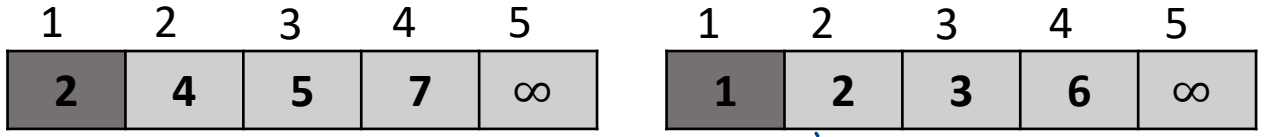
i=1

j=2

- Compare L[1] and R[2]



A[2]=L[1]



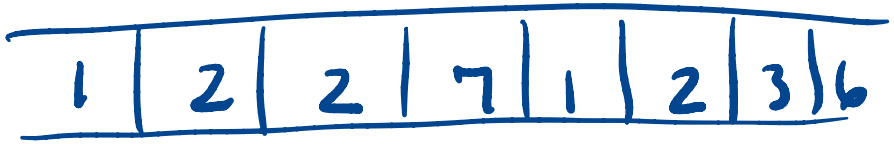
i=2

j=2

Now k=3

compare L[2] and R[2]

A[3]=R[2]



Now i=2, j=3

Merge Sort

Example: Set up and solve the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

procedure $\boldsymbol{MERGESORT(A, p, r)}$

$n = len(A)$ } Base case with an array of a single element. · executed in constant time $\Rightarrow \Theta(1)$

$if\ n \leq 1$

$\quad return\ A$

$else$ ← q ← computation is some constant time.

$\quad H_1 = MERGESORT(A, p, q)$ $T(\lceil n/2 \rceil)$

$\quad H_2 = MERGESORT(A, q+1, r)$ $T(\lfloor n/2 \rfloor)$

$\quad return\ MERGE(H_1, H_2)$ — $\Theta(n)$

· if the entire Mergesort call takes $T(n)$ then calling mergesort for half the list takes $T(n/2)$

Combine step: Merge $C(n) = \Theta(n)$

· letting $C(n)$ be the number of operations it takes Merge to actually merge two lists of length $n/2$

Overhead costs: check base case, break instance into two pieces, call Merge ..

$D(n) = \Theta(1)$

Merge Sort

Example: Set up and solve the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

procedure $\boldsymbol{MERGESORT}(\boldsymbol{A}, \boldsymbol{p}, \boldsymbol{r})$

```
n = len(A)
if n ≤ 1
    return A
else
    H1 = MERGESORT(A, p, q)
    H2 = MERGESORT(A, q + 1, r)
    return MERGE(H1, H2)
```

- Putting this all together:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + C(n) + D(n)$$

$$= 2T(n/2) + \underline{\Theta(n) + \Theta(1)}$$

- ignoring $\lceil n/2 \rceil$ or $\lfloor n/2 \rfloor$ because it will be absorbed by asymptotics

Note: $C(n) = \Theta(n)$

$\Rightarrow c_1 n \le C(n) \le c_2 n$

$D(n) = \Theta(1)$

$c_3 \cdot 1 \le D(n) \le c_4 \cdot 1$

$c_1 n + c_3 \le C(n) + D(n) \le c_2 n + c_4$

$c_1 n \le C(n) + D(n) \le (c_2 + c_4) n \Rightarrow C(n) + D(n)$ is $\Theta(n)$

Merge Sort

Example: Set up and solve the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

procedure $\boldsymbol{MERGESORT(A, p, r)}$

$n = len(A)$

$if\ n \leq 1$

$\quad return\ A$

$else$

$\quad H_1 = MERGESORT(A, p, q)$

$\quad H_2 = MERGESORT(A, q+1, r)$

$\quad return\ MERGE(H_1, H_2)$

Recurrence for Mergesort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Unrolling: Idea is to plug in the recursive definition of $T(n)$ a few times and try to find a pattern.

$T(n) = 2T(n/2) + cn$

$= 2[2T(n/4) + cn/2] + cn$

$= 4T(n/4) + cn + cn$

$= 4[2T(n/8) + cn/4] + cn + cn$

$= 8T(n/8) + cn + cn + cn$

$= 8[2T(n/16) + cn/8] + 3cn$

$= 16T(n/16) + 4cn$

$\vdots$

Note: $T(n/2) = 2T(n/4) + cn/2$

$T(n/4) = 2T(n/8) + cn/4$

$T(n/8) = 2T(n/16) + cn/8$

Note: k^{th} plugging in: $T(n) = 2^k T(n/2^k) + kcn$

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Unrolling: How many unrollings must we do? - Keep going until the recursion bottoms out.

→ Recursion bottoms out when

$$\left\lceil \frac{n}{2^k} \right\rceil = 1$$

Solve for k:

$$2^k = n$$

$$k = \log_2 n$$

This is the number of unrollings to get to the base case, $T(1)$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn$$

$$= 2^{\log_2 n} T(1) + (\log_2 n) \cdot cn = nT(1) + cn \log_2 n \Rightarrow$$

$T(n)$ is $\Theta(n \log n)$

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Unrolling:

So Mergesort (worst case) is $\Theta(n \log n)$

Solving Recurrences

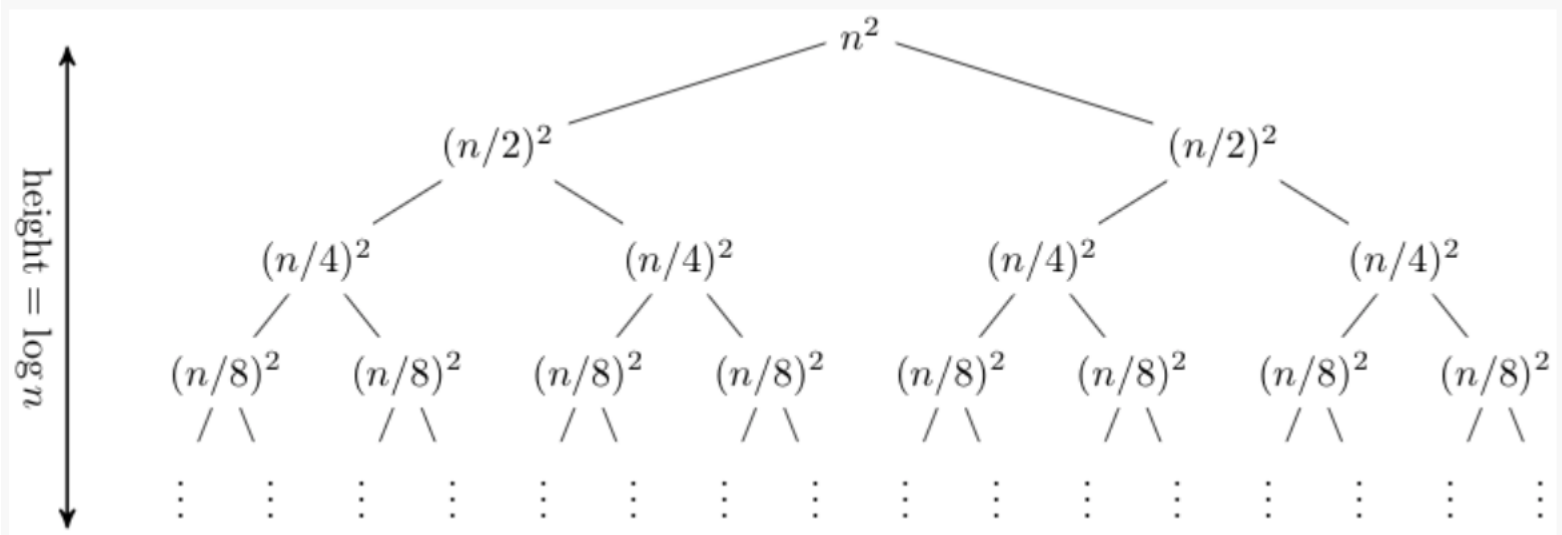
In a **recursion tree**, each node represents the cost of a single subproblem.

1. Sum the costs within each level of the tree.
2. Find the total cost of all levels

- Good way to generate a good guess as to what your solution is.

- You can verify your guess with the substitution method.

Here we have a recursion tree for the recursion $T(n) = 2T(n/2) + n^2$



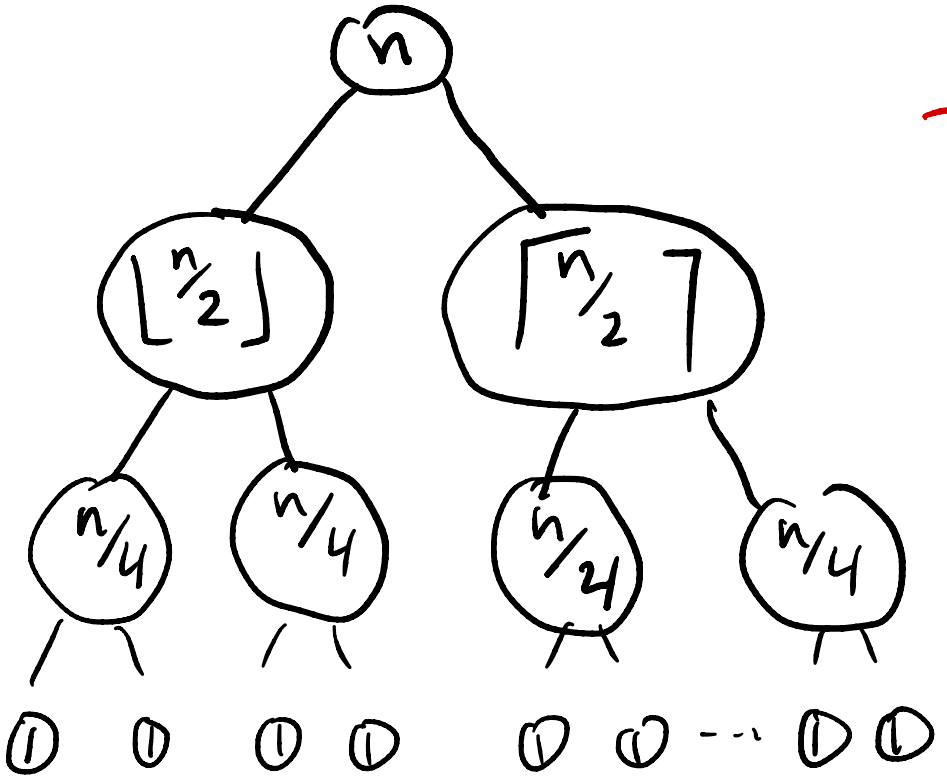
Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

Tree Method:

- The root of the tree is the top-level call
- Each node corresponds to a recursive call

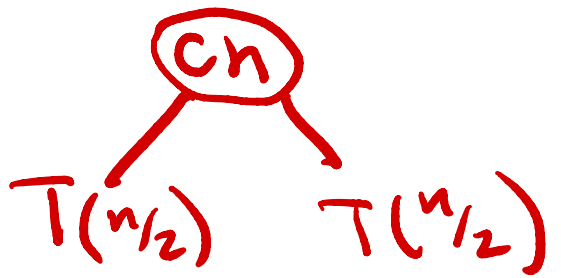


- Trying to write down the work done inside each call to MERGESORT

T(n)

} Base cases

Note $T(n) = 2T(n/2) + cn$



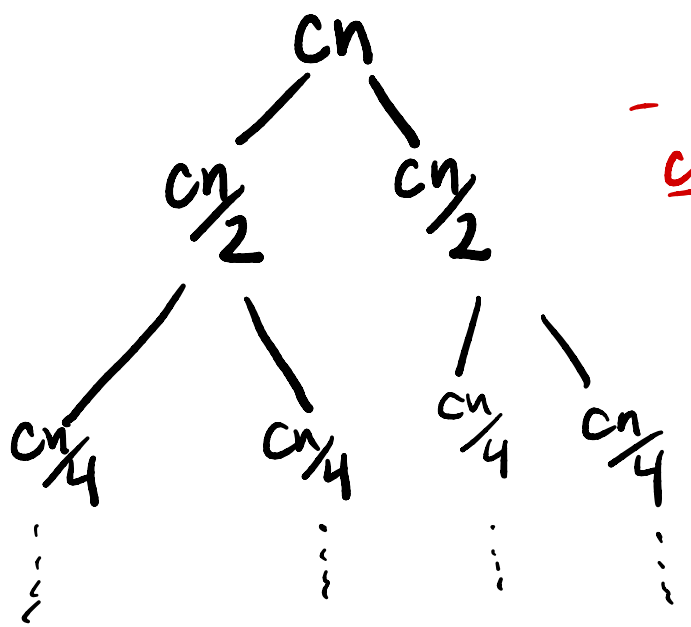
} T(n)

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Tree Method:

$T(n) = 2T(n/2) + cn$, Note: $T(n/2) = 2T(n/4) + cn/2$



Total Runtime $T(n)$

cn

- Add the costs across each level of the tree.

$\frac{cn}{2} + \frac{cn}{2} = cn$

$\frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} = cn$

c c c c ---- c c cn

- We expand each node in the tree until the problem size gets down to 1. $\Rightarrow$ n leaves at bottom level

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Tree Method:

- In general, the level i below the top has 2^i nodes, each contributes a cost of $c\left(\frac{n}{2^i}\right)$

→ so the ith level has a cost: $\left(2^i\right)\left(\frac{cn}{2^i}\right) = cn$ cost per level

How many levels?

$$\left\lceil \frac{n}{2^k} \right\rceil = 1 \quad \Rightarrow \quad n = 2^k$$
$$\log_2 n = k$$

Consider our base case where $n=1$. - there would be one level to our tree.

$\log_2(1) = 0 \Rightarrow$ $K = \log_2 n + 1$ number of levels

# of levels +1 for the root

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Tree Method:

Total cost = (number of levels) * (cost per level)

$= (\log_2(n) + 1) * cn$

$= cn\log_2(n) + cn$

$\rightarrow \Theta(n \log n)$

Solving Recurrences

❖ The **master method** provides a "cookbook" method for solving recurrences of the form:

$$T(n) = aT(n/b) + f(n), \text{where } a \geq 1 \text{ and } b > 1.$$

a - number of subproblems
n/b - size of input to recursive calls
$f(n)$ - work at each node for divide + combine steps.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg(n)\right)$.
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\left(f(n)\right)$.

Solving Recurrences

- $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. Work done at leaves dominates.
   $T(n) = aT(n/b) + f(n)$
   suppose $f(n) = \Theta(n^c)$ where $c < \log_b a$
   $\Rightarrow T(n) = \Theta(n^{\log_b a})$
2. Cost is same at every level.
3. Cost is dominated by the root.

e.g. $T(n) = 8T(n/2) + n^2$ ($f(n)$) $a = 8$, $b = 2$

- The 8 signifies 8 recursive calls!

How many leaves? 8^k

8^k is way bigger than 8^{k-1}

$f(n)$ is $\Theta(n^2)$

$n^{\log_b a} = n^{\log_2 8} = n^3$

$\Rightarrow f(n)$ is $\Theta(n^{3-1})$ $\varepsilon = 1$

$\Rightarrow T(n)$ is $\Theta(n^{\log_2 8}) = \Theta(n^3)$

Solving Recurrences

$$T(n) = aT(n/b) + f(n), \text{where } a \geq 1 \text{ and } b > 1.$$

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. Work done at leaves dominates.
2. Cost is same at every level.
3. Cost is dominated by the root.

If $f(n) = \Theta(n^c)$ where $c = \log_b a$
then $T(n) = \Theta(n^c \log n)$

Example $T(n) = 2T(n/2) + n$ $a=2$ $b=2$

$f(n)$ is $\Theta(n)$

$n^{\log_b a} = n^{\log_2 2} = n$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$

Solving Recurrences

$$T(n) = aT(n/b) + f(n), \text{where } a \geq 1 \text{ and } b > 1.$$

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. Work done at leaves dominates.
2. Cost is same at every level.
3. Cost is dominated by the root.

If $f(n) = \Theta(n^c)$ $\quad c > \log_b a$

then $T(n) = f(n)$

Example $T(n) = 2T(n/2) + n^2$

$n^{\log_2 2} = n$

$f(n) = n^{\log_2 2 + 1} = n^{1+1} = n^2$

$\Rightarrow T(n) = \Theta(n^2)$

Merge Sort

Example: Set up and **solve** the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers.

Master Method:

MergeSort: $T(n) = 2T(n/2) + cn$

$a = 2$

$b = 2$

$f(n) = cn$ for $c > 0$

$\log_2 2 = 1$ so $n^{\log_b a} = n$

$\lim_{n \to \infty} \frac{n}{cn} = \frac{1}{c} > 0 \Rightarrow f(n)$ is $\Theta(n) = \Theta(n^{\log_2 2})$

$\rightarrow$ Case 2 of the Master Theorem.

$\Rightarrow T(n) = \Theta(n^{\log_2 2} \log(n))$

$= \Theta(n \log n)$

Merge Sort

The worst-case running time $T(n)$ of the Merge-Sort procedure described by the recurrence

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

has the solution $T(n) = \Theta(n \lg n)$.

Next Time

- Quicksort
- Randomness
- Probability Review

Extra Practice

Example: Prove that there are $\lg(n) + 1$ levels in the recursion tree for MERGESORT.

Assume as an inductive hypothesis that the number of levels of a recursion tree with 2^i leaves is $\lg(2^i) + 1 = i + 1$. We also assume that the input size is a power of 2. Therefore, the next input size to consider is 2^{i+1}.

A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves. So the total number of levels is $(i + 1) + 1 = \lg(2^{i+1}) + 1$.

Extra Practice

Example: Prove the correctness of the Merge algorithm.

Merge Loop Invariant: At the start of each iteration of the for loop where we fill in A, the subarray $A[p \ldots k-1]$ contains the k-p smallest elements of $L[1 \ldots n_1+1]$ and $R[1 \ldots n_2+1]$ in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Initialization: Prior to the first iteration of the loop, we have k=p, so that the subarray $A[p \ldots k-1]$ is empty. This empty subarray contains the k-p=0 smallest elements of L and R, and since $i = j = 1$, both L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Maintenance: To see that each iteration maintains the loop invariant, let us first suppose that $L[i] \leq R[j]$. Then L[i] is the smallest element not yet copied back into A. Because $A[p \ldots k-1]$ contains the k-p smallest elements, after line "$A[k] = L[i]$" copies L[i] into A[k], the subarray $A[p \ldots k]$ will contain the k-p+1 smallest elements. Incrementing k and i reestablishes the loop invariant for the next iteration. If instead, $L[i] > R[j]$, then lines "$else\ A[k] = R[j], j = j+1$" maintain the loop invariant.

Termination: At termination, k = r+1. By the loop invariant, the subarray $A[p \ldots k-1]$, which is $A[p \ldots r]$ contains the k-p = r-p+1 smallest elements of $L[1 \ldots n_1+1]\ and\ R[1 \ldots n_2+1]$ in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

Proof taken from CLRS, section 2.3