Divide and Conquer Algorithms

CSE 101: Design and Analysis of Algorithms

Lecture 15

CSE 101: Design and analysis of algorithms

- Divide and conquer algorithms
  - Reading: Chapter 2
- Homework 6 is due Nov 20, 11:59 PM

Divide and conquer example

POWER OF 2

Power of 2

- Problem: Given n, compute the digits of 2^n in decimal
- Note: $\log_{10} 2^n = n \log_{10} 2 \approx 0.3n$ is $\theta(n)$, so 2^n has cn digits for some $c > 0$
- As such, we cannot treat multiplication as a single step, but we can use multiplyKS
- What sub-problem would be useful in computing 2^n?

Power of 2

- Problem: Given n, compute the digits of 2^n in decimal
- Note: $\log_{10} 2^n = n \log_{10} 2 \approx 0.3n$ is $\theta(n)$, so 2^n has cn digits for some $c > 0$
- As such, we cannot treat multiplication as a single step, but we can use multiplyKS
- What sub-problem would be useful in computing 2^n?
  - Compute $2^{n/2}$, then square it

Power of 2 algorithm

```
PoT (n)

   IF n=0 THEN return 1
   IF n=1 THEN return 2
   P : = PoT(⌊n/2⌋)
   P : = multiplyKS(P,P)
   IF n mod 2 = 1 THEN
       P:= Add(P,P)
  Return P
```

Power of 2 algorithm, correctness

PoT (n)

IF n=0 THEN return 1 $2^0 = 1$

IF n=1 THEN return 2 $2^1 = 2$

Base cases

P : = PoT($\left\lfloor \frac{n}{2} \right\rfloor$) By induction, P = $2^{\frac{n}{2}}$ if n is even, $2^{\frac{n-1}{2}}$ if n is odd

P : = multiplyKS(P,P) P= $(2^{n/2})(2^{n/2})$=2^n if n is even, 2^{n-1} if n is odd

IF n mod 2 = 1 THEN

P:= Add(P,P) P = 2^n

Return P

Power of 2 algorithm, runtime

PoT (n) $T(n)$

IF n=0 THEN return 1

IF n=1 THEN return 2

P : = PoT($\left\lfloor \frac{n}{2} \right\rfloor$) $T\left(\frac{n}{2}\right)$

P : = multiplyKS(P,P) $O(n^{\log_2 3})$

IF n mod 2 = 1 THEN

P:= Add(P,P) $O(n)$

Return P

$$T(n) = T\left(\frac{n}{2}\right) + O(n^{\log_2 3})$$

Master theorem applied

- The recursion for the runtime is
$$T(n) = T\left(\frac{n}{2}\right) + O(n^{\log_2 3})$$

$$T(n) = aT(n/b) + O(n^d)$$
$$T(n) \epsilon \begin{cases} O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^d \log n\right) & \text{if } a = b^d \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

- So, we have that $a = 1$, $b = 2$, and $d = \log_2 3$. In this case, $a < b^d$ so
$$T(n) \epsilon O(n^{\log_2 3}) \approx O(n^{1.58})$$

Top-heavy

Divide and conquer example

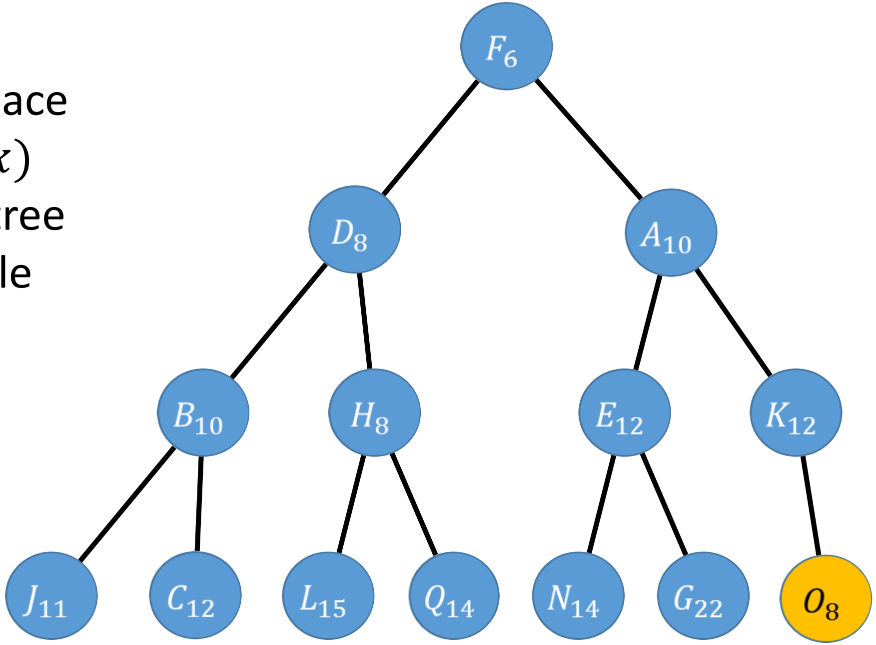
MAKING A BINARY HEAP

Making a binary heap

- How long do you expect it takes to make a binary heap from n objects each with a key value?
- Recall
  - A complete binary tree of objects (vertices) with the property that each key value of an object is less than the key value of its child
  - To **insert**$((o, k), H)$, place the new element (o, k) at the bottom of the tree H, (in the first available position) and let it "bubble up"

Making a binary heap

To **insert**$((o, k), H)$, place the new element (o, k) at the bottom of the tree H, (in the first available position) and let it "bubble up"

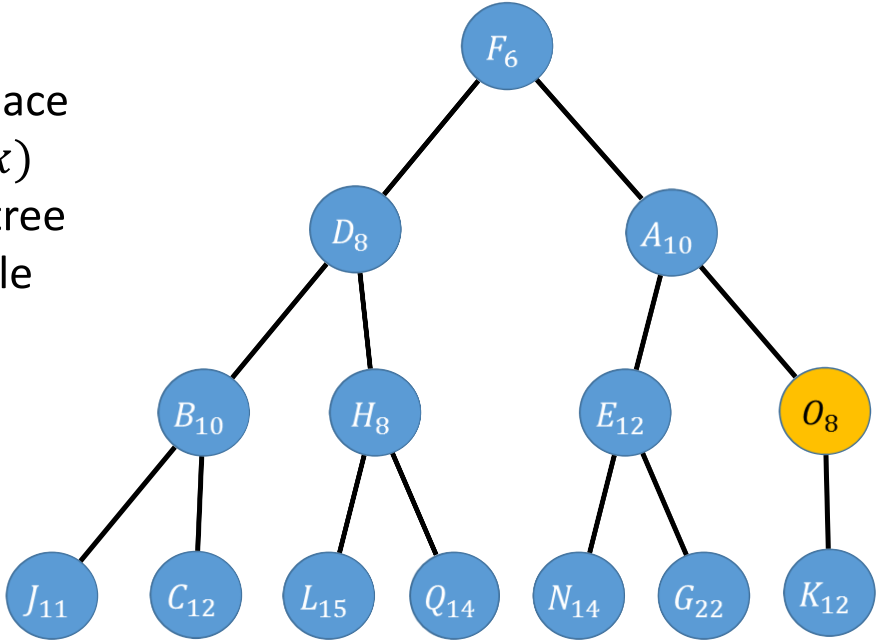


1 2 3 4 5 6 7 8 9 10 11 12 13 14

$[F_6, D_8, A_{10}, B_{10}, H_8, E_{12}, K_{12}, J_{11}, C_{12}, L_{15}, Q_{14}, N_{14}, G_{22}, O_8$

Making a binary heap

To **insert**$((o,k),H)$, place the new element (o,k) at the bottom of the tree H, (in the first available position) and let it "bubble up"

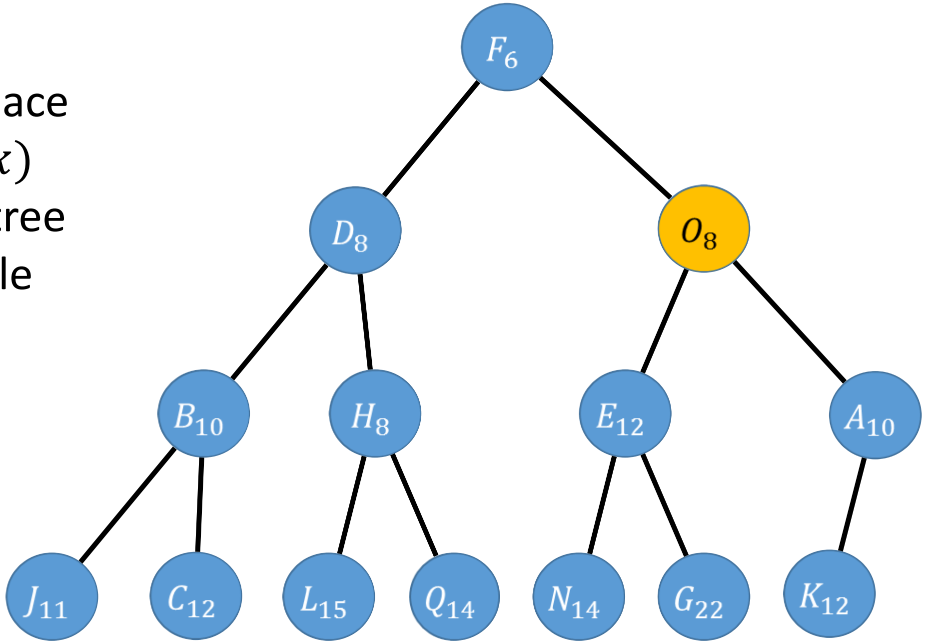


1 2 3 4 5 6 7 8 9 10 11 12 13 14

$[F_6, D_8, A_{10}, B_{10}, H_8, E_{12}, O_8, J_{11}, C_{12}, L_{15}, Q_{14}, N_{14}, G_{22}, K_{12}]$

Making a binary heap

To **insert**$((o, k), H)$, place the new element (o, k) at the bottom of the tree H, (in the first available position) and let it "bubble up"



1 2 3 4 5 6 7 8 9 10 11 12 13 14

$[F_6, D_8, O_8, B_{10}, H_8, E_{12}, A_{10}, J_{11}, C_{12}, L_{15}, Q_{14}, N_{14}, G_{22}, K_{12}]$

Making a binary heap

- How long do you expect it takes to make a binary heap from n objects each with a key value?
  - Insert n times
    - How much time for each insert?

Making a binary heap

- How long do you expect it takes to make a binary heap from n objects each with a key value?
  – Insert n times
    - Each insert is $O(\log n)$, so total time is $O(n \log n)$

Making a binary heap

- How long do you expect it takes to make a binary heap from n objects each with a key value?
- Start with your first object and repeatedly **insert** a new object into your heap

makeheap$((o_1, k_1), \ldots (o_n, k_n))$
If $n == 1$:
    return $[(o_1, k_1)]$
return insert$((o_n, k_n)$,**makeheap**$((o_1, k_1), \ldots (o_{n-1}, k_{n-1}))$

Making a binary heap

- How long do you expect it takes to make a binary heap from n objects each with a key value?
- Start with your first object and repeatedly **insert** a new object into your heap

makeheap$((o_1, k_1), \dots (o_n, k_n))$ $T(n)$

If $n == 1$:

  return $[(o_1, k_1)]$ $O(1)$

return insert$((o_n, k_n)$,**makeheap**$((o_1, k_1), \dots (o_{n-1}, k_{n-1}))$

$O(\log n)$ $T(n-1)$

Making a binary heap

- Can we improve on $O(n \log n)$?
- Let's try divide and conquer
  - How would that work?

Making a binary heap

- Can we improve on $O(n \log n)$?
- Let's try divide and conquer
  - How would that work?
    - Base case
    - Break the problem up
    - Recursively solve each problem
      - Assume the algorithm works for the subproblems
    - Combine the results

Making a binary heap

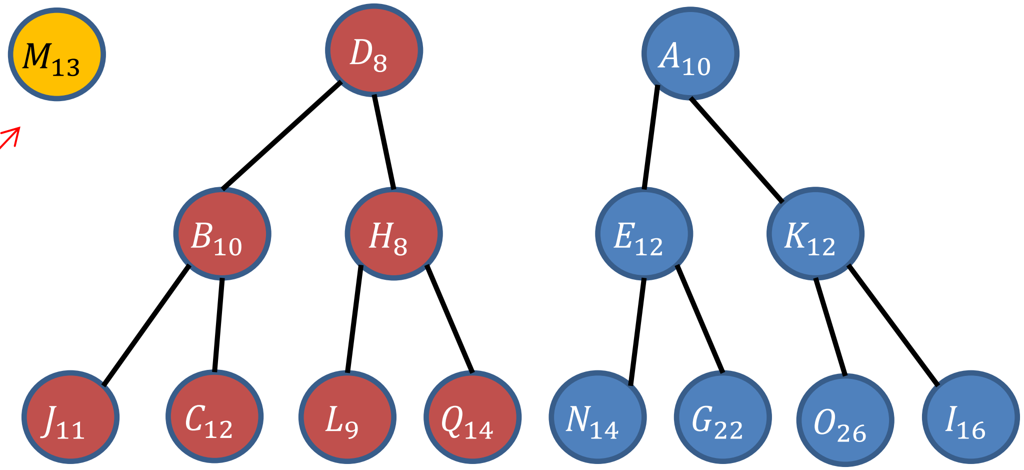
- Let's assume that $n = 2^k - 1$
- Goal: Make a min heap out of the list of objects $[(o_1, k_1), \ldots, (o_n, k_n)]$

Making a binary heap

- Let's assume that $n = 2^k - 1$
- Goal: Make a min heap out of the list of objects $[(o_1, k_1), \ldots, (o_n, k_n)]$
- Put (o_1, k_1) aside and break the remaining part into 2 each of size $2^{k-1} - 1$
  - Assume our algorithm works on the two subproblems
- This results in two binary heaps
- Then make (o_1, k_1) the root and let it trickle down

Binary heap

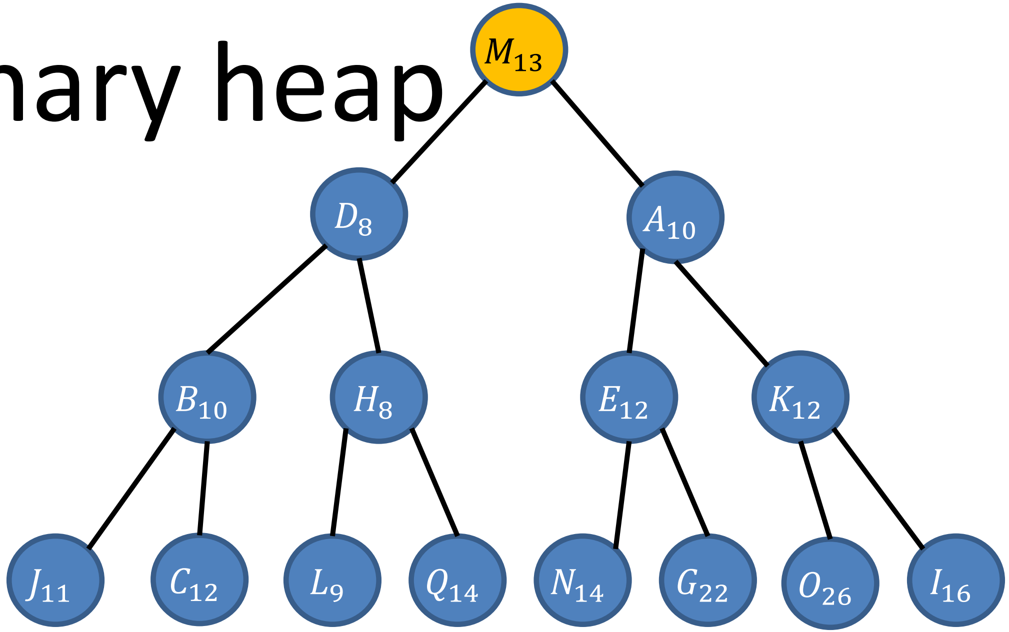
- Put (o_1, k_1) aside and break the remaining part into 2 each of size $2^{k-1} - 1$



$$[M_{13}, \underbrace{D_8, B_{10}, J_{11}, C_{12}, H_8, L_9, Q_{14}}, \underbrace{A_{10}, I_{16}, O_{26}, K_{12}, E_{12}, N_{14}, G_{22}}]$$

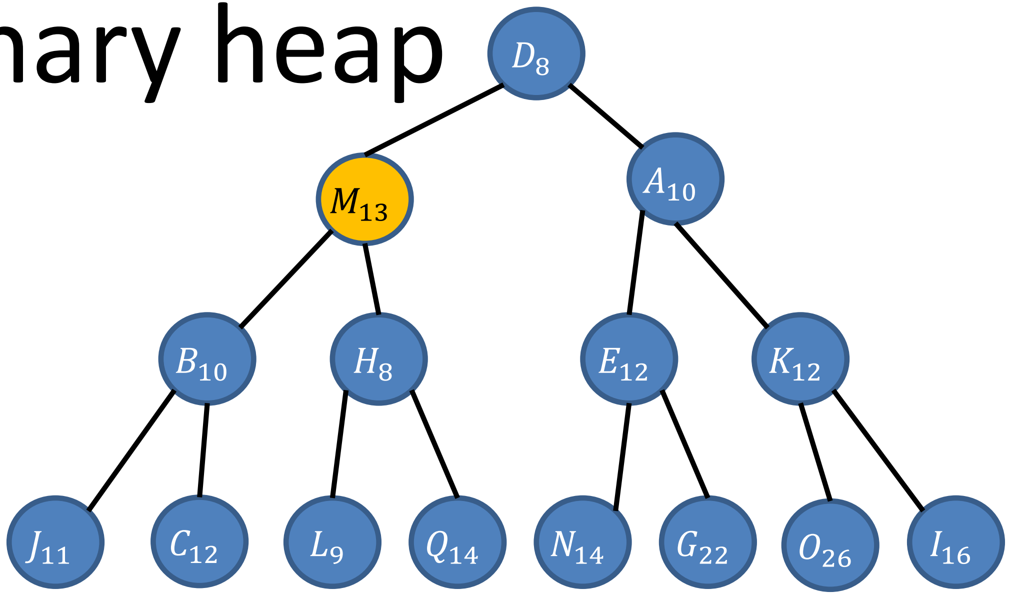
Binary heap

- Put the first object as the root of the two subtrees and let it trickle down



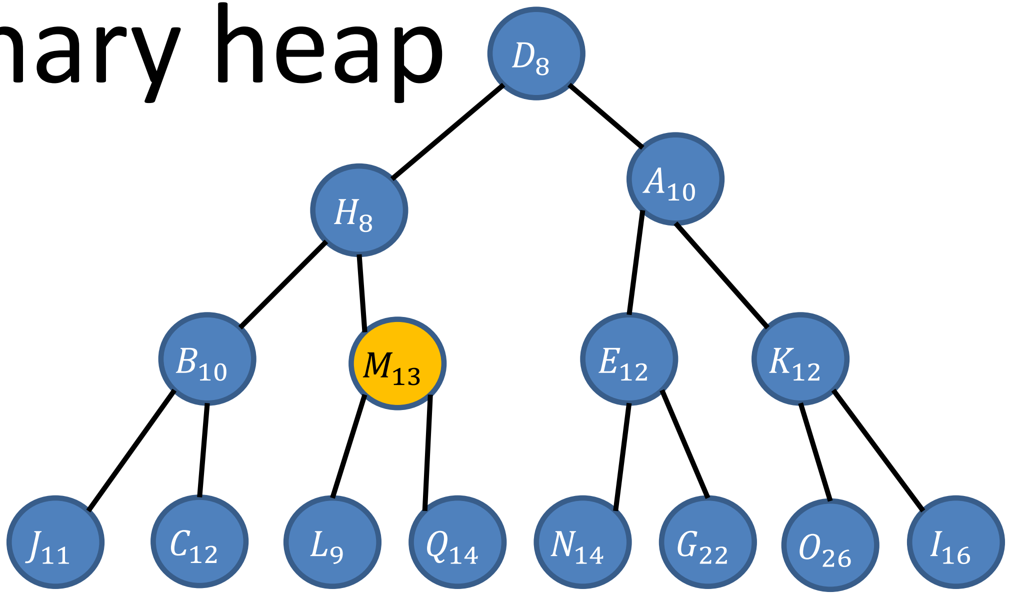
Binary heap

- Put the first object as the root of the two subtrees and let it trickle down



Binary heap

- Put the first object as the root of the two subtrees and let it trickle down



Divide and conquer make heap

makeheapDC$((o_1, k_1), \ldots (o_n, k_n))[n = 2^k - 1$ for some integer $k]$

If $n == 1$:

  return $[(o_1, k_1)]$

$H_1 =$**makeheapDC**$((o_2, k_2), \ldots (o_{\frac{n+1}{2}}, k_{\frac{n+1}{2}}))$

$\mathrm{H}_2 =$ **makeheapDC**$\left(\left(o_{\frac{n+1}{2}+1}, k_{\frac{n+1}{2}+1}\right), \ldots (o_n, k_n)\right)$

Return combine$\left((o_1, k_1), H_1, H_2\right)$ If k = 2, n =3

Divide and conquer make heap, runtime

makeheapDC$((o_1, k_1), \ldots (o_n, k_n))[n = 2^k - 1$ for some integer $k]$ $T(n)$

If $n == 1$:

  return $[(o_1, k_1)]$

H_1 =**makeheapDC**$((o_2, k_2), \ldots (o_{\frac{n+1}{2}}, k_{\frac{n+1}{2}}))$ $T\left(\frac{n}{2}\right)$

H_2 = **makeheapDC**$\left(\left(o_{\frac{n+1}{2}+1}, k_{\frac{n+1}{2}+1}\right), \ldots (o_n, k_n)\right)$ $T\left(\frac{n}{2}\right)$

Return combine$\left((o_1, k_1), H_1, H_2\right)$ $O(\log n)$

$T(n) = 2T\left(\frac{n}{2}\right) + O(\log n)$

Divide and conquer make heap, runtime

- Problem: $T(n) = 2T(n/2) + O(\log n)$ not of the form for master theorem
- One solution: go back to tree

Divide and conquer make heap, runtime

- Problem: $T(n) = 2T(n/2) + O(\log n)$ not of the form for master theorem
- One solution: go back to tree
- Another solution: cheat

  $L(n) = 2L(n/2) + 1, L(n) < T(n)$

  $U(n) = 2U(n/2) + n^{1/2}, U(n) > T(n)$

  - Master theorem: $L(n), U(n)$ both bottom heavy (same a, b)
  - So both $U(n), L(n)$ are $\theta(n^{\log_2 2})$=θ(n)
  - Since $L(n) < T(n) < U(n)$
    - Same true for $T(n)$
    - $T(n) \in \theta(n)$

Uneven binary heap

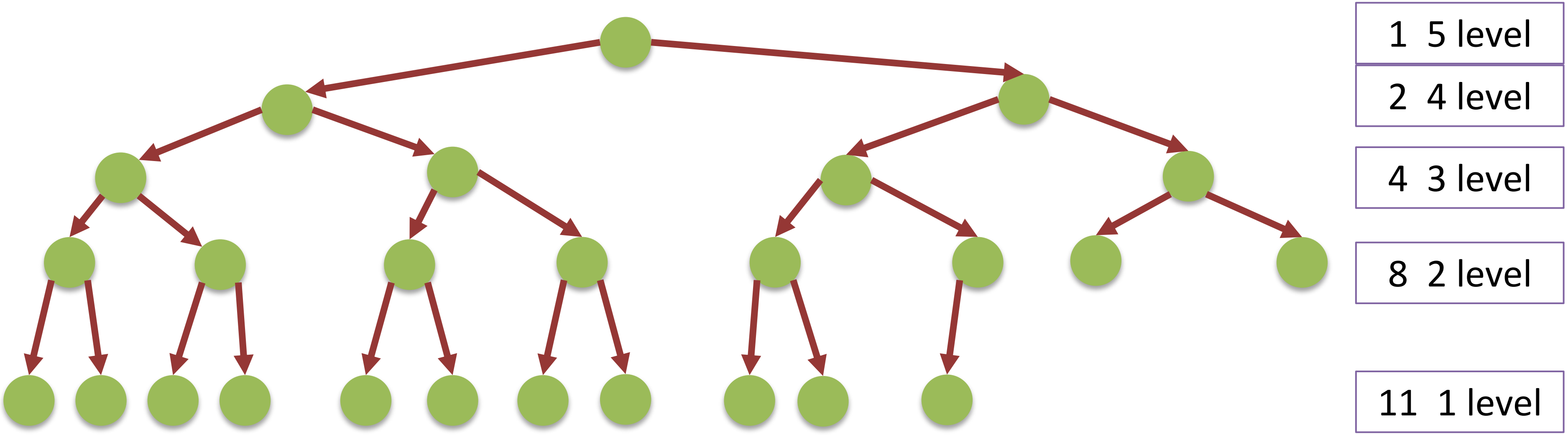
- What if the input size n is not $2^k - 1$?
- Then, let n be the input size and let m be the smallest integer greater than n such that $m = 2^k - 1$ for some k
- Then add in $m - n$ objects (o_i, ∞) and redo the original algorithm
- $m < 2n$ and the algorithm will run in $O(m)$ time so it will run in $O(n)$ time

Uneven binary heap

- What if the input size n is not $2^k - 1$?
- Another way is to use Floyd's buildheap algorithm: fill in the heap randomly and organize it from the bottom up
  - Percolate down from the bottom up

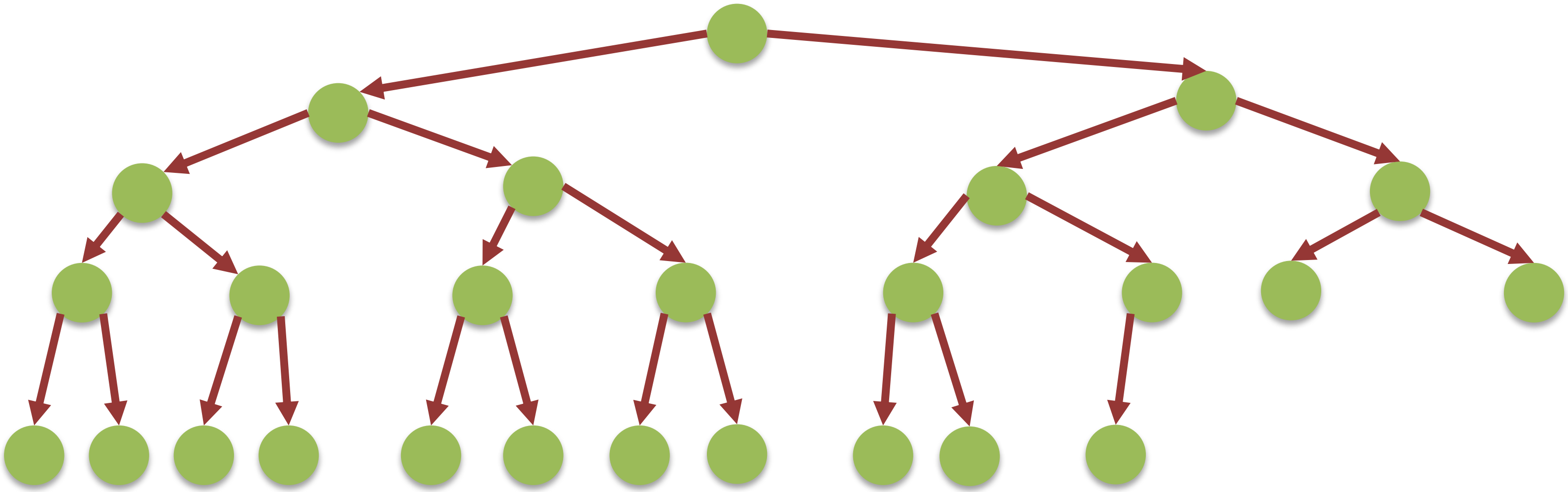
Uneven binary heap

- Floyd's buildheap algorithm: percolate down from the bottom up.



Uneven binary heap

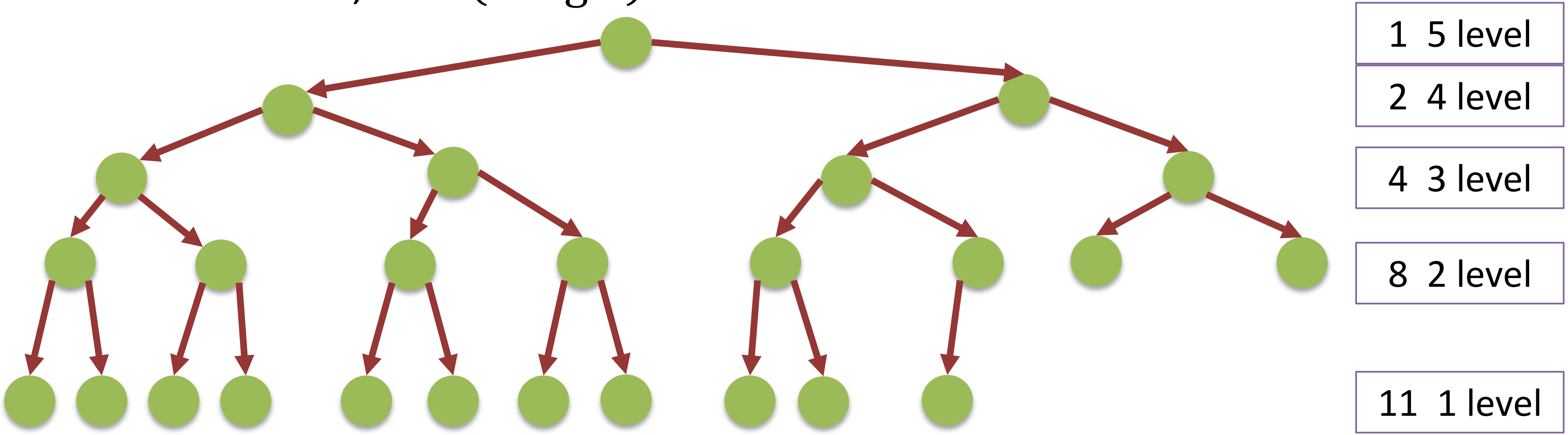
- Floyd's buildheap algorithm: percolate down from the bottom up.
- How long does this take?



1	5 level
2	4 level
4	3 level
8	2 level
11	1 level

Uneven binary heap

- Floyd's buildheap algorithm: percolate down from the bottom up.
- How long does this take? In the worst case $O(\log n)$ for each of the n elements, so $O(n \log n)$



Uneven binary heap

- The amount of work needed is:

$$\frac{n}{2} * 1 + \frac{n}{2^2} * 2 + \frac{n}{2^3} * 3 + \cdots$$

$$= n \sum_{i=1}^{\log n} i/2^i$$

$< c$

$T(n) < cn = O(n)$

Uneven divide and conquer

- Sometimes, sizes of sub-parts not identical or depends on the input
- Example: given a pointer to a binary tree, compute its depth

Depth(r: node) $T(n)$

If lc.r ≠NIL then

    d:= Depth(lc.r) $T(L)$

else

    d:=0

If rc.r ≠ NIL then

    d:= max(d, Depth(rc.r)) $T(R)$

Return d + 1

Time analysis

- If tree is balanced, the recurrence is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

$$T(n) = O\left(n^{\log_2 2}\right) = O(n)$$ Master theorem

$$T(n) = aT(n/b) + O(n^d)$$

$$T(n) \epsilon \begin{cases} O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^d \log n\right) & \text{if } a = b^d \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

- If tree is totally unbalanced, the recurrence is

$$T(n) = T(n-1) + O(1) = O(n)$$

Either case

- $T(n) = T(L) + T(R) + c$
  - $T(0)=0$, $T(1)=c$, to make things fit
- $n = L + R + 1$

Either case

- T(n) = T(L) + T(R) + c
  - T(0)=0, T(1)=c, to make things fit
- n = L + R + 1

- Then, by strong induction for some n, we claim $T(k) \leq ck$ for all $k < n$
- T(L) $\leq cL$
- T(R) ≤ cR
- T(n) ≤ cL+cR+c = c (L+R+1) =cn

Divide and conquer example

GREATEST OVERLAP

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- An interval [a,b] is a set of integers starting at a and ending at b. For example: [16,23] = {16,17,18,19,20,21,22,23}
- An overlap between two intervals [a,b] and [c,d] is their intersection
- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

Greatest overlap

- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

procedure overlap([a,b],[c,d]) [assume that $a \leq c$]

```
if b < c:                [a          b]
    return ?                        [c          d]
else:
    if b ≤ d:            [a          b]
        return ?                [c          d]
    if b > d:            [a          b]
        return ?             [c   d]
```

Greatest overlap

- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

```
procedure overlap([a,b],[c,d]) [assume that a ≤ c]
    if b < c:
        return 0
    else:
        if b ≤ d:
            return ?
        if b > d:
            return ?
```

[a b]
[c d]

[a b]
[c d]

[a b]
[c d]

Greatest overlap

- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

procedure overlap([a,b],[c,d]) [assume that $a \leq c$]

if $b < c$:

  return 0

[a   b]
        [c   d]

else:

  if $b \leq d$:

    return $b - c + 1$

[a   b]
    [c   d]

  if $b > d$:

    return ?

[a   b]
  [c d]

Greatest overlap

- Given two intervals [a,b] and [c,d], how would you compute the length of their overlap?

procedure overlap([a,b],[c,d]) [assume that $a \leq c$]

```
if b < c:
    return 0
else:
    if b ≤ d:
        return b – c + 1
    if b > d:
        return d – c + 1
```

[a   b]
              [c   d]

[a   b]
         [c   d]

[a   b]
     [c   d]

Greatest overlap

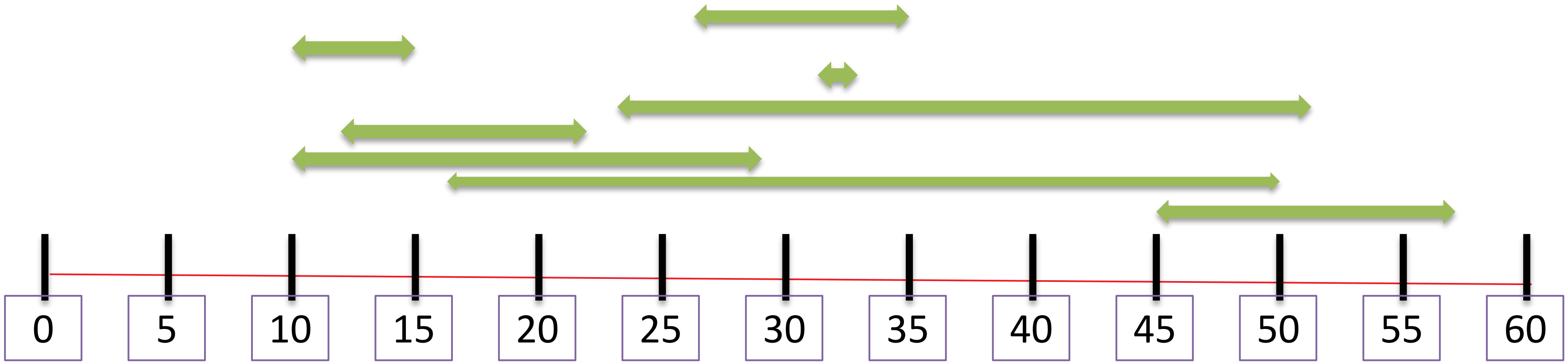
- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Example: What is the greatest overlap of the intervals

  [45,57],[17,50],[10,29],[12,22],[23,51],[31,32],[10,15],[23,35]

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Example: What is the greatest overlap of the intervals

  [45,57],[17,50],[10,29],[12,22],[23,51],[31,32],[10,15],[23,35]



Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Simple solution?

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Simple solution: Compute all overlap pairs and find maximum

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Simple solution

```
olap:=0
for i from 1 to n-1
    for j from i+1 to n
        if overlap([a_i,b_i],[a_j,b_j])>olap then
            olap:=overlap([a_i,b_i],[a_j,b_j])
return olap
```

Runtime?

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
- Simple solution

```
olap:=0
for i from 1 to n-1
    for j from i+1 to n
        if overlap([a_i,b_i],[a_j,b_j])>olap then
            olap:=overlap([a_i,b_i],[a_j,b_j])
return olap
```

$O(n^2)$ Can we do better?

Greatest overlap

- Given a list of intervals [a_1,b_1],…[a_n,b_n] write pseudocode for a divide and conquer algorithm that outputs the length of the greatest overlap between two intervals
  - Compose your base case
  - Break the problem into smaller pieces
  - Recursively call the algorithm on the smaller pieces
  - Combine the results

Greatest overlap

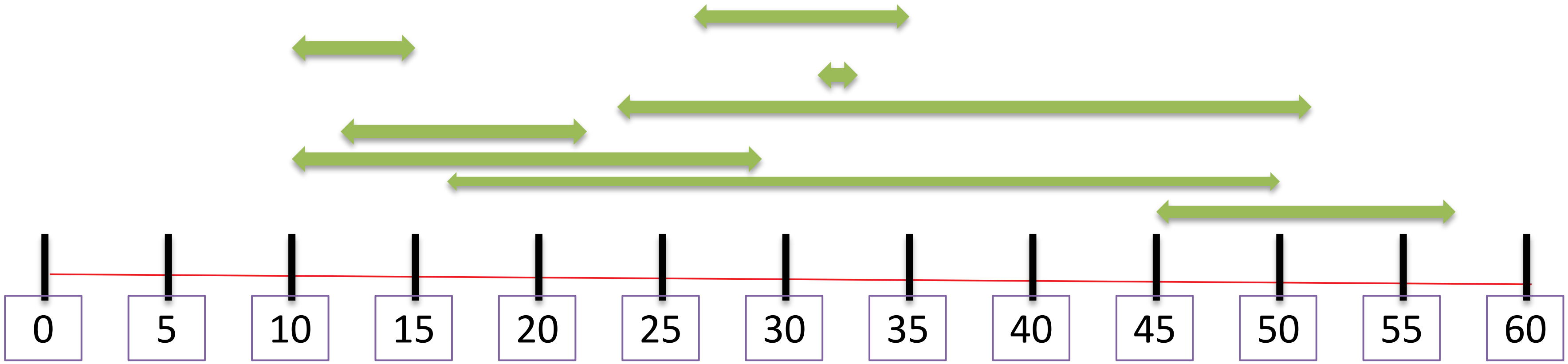
- Compose your base case
  - What happens if there is only one interval?

Greatest overlap

- Compose your base case
  - What happens if there is only one interval?
    - If n=1, then return 0

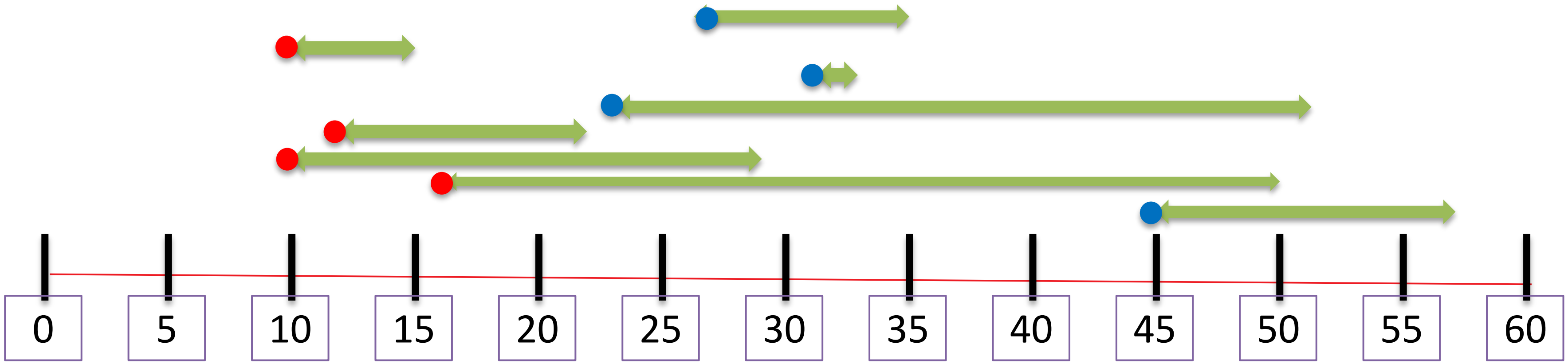
Greatest overlap

- Break the problem into smaller pieces
  - Would knowing the result on smaller problems help with knowing the solution on the original problem?
  - In this stage, let’s keep the combine part in mind
  - How would you break the problem into smaller pieces?



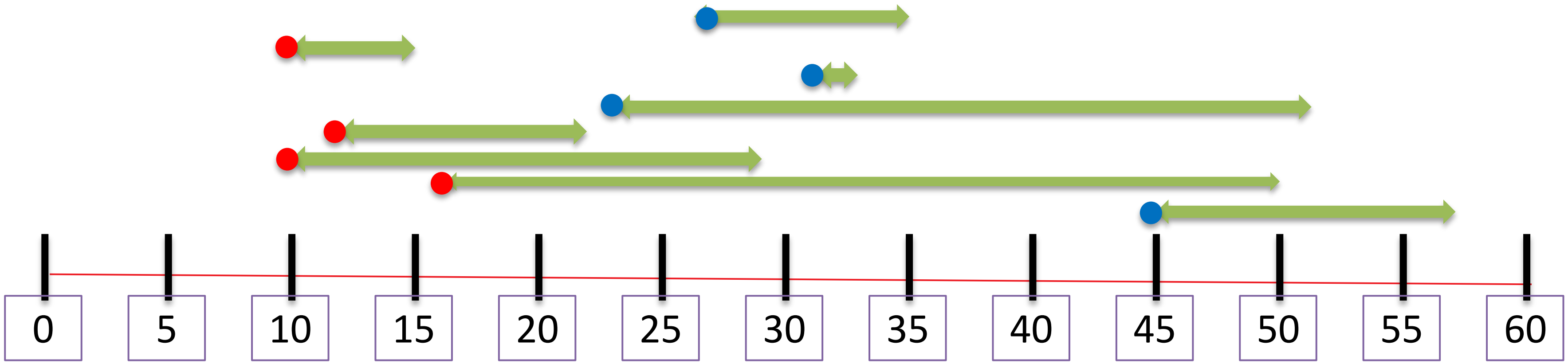
Greatest overlap

- Break the problem into smaller pieces
  - Would it be helpful to break the problem into two depending on the starting value?



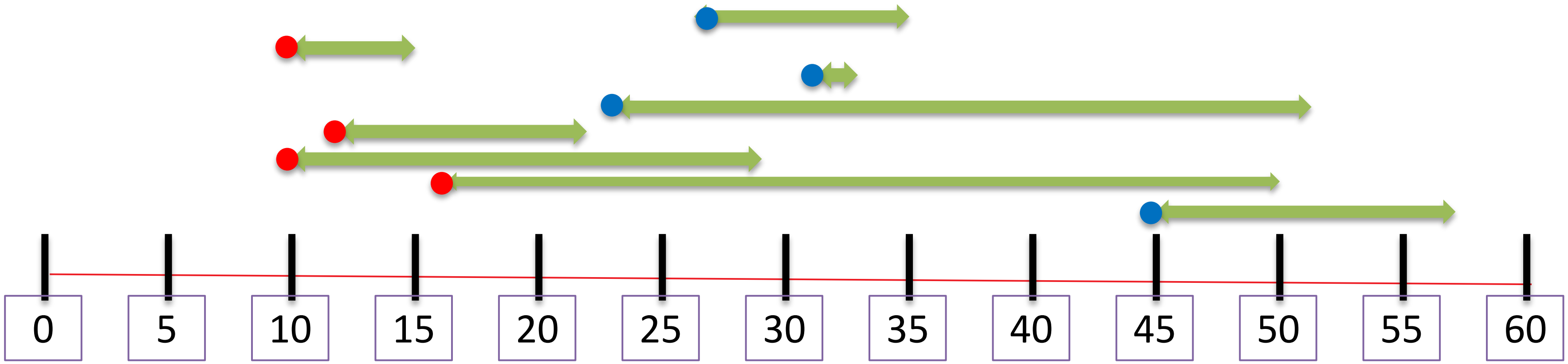
Greatest overlap

- Break the problem into smaller pieces
  - Sort the list and break it into lists each of size n/2.
    - [10,15],[10,29],[12,22],[17,50],[23,51],[27,35],[31,32],[45,57]



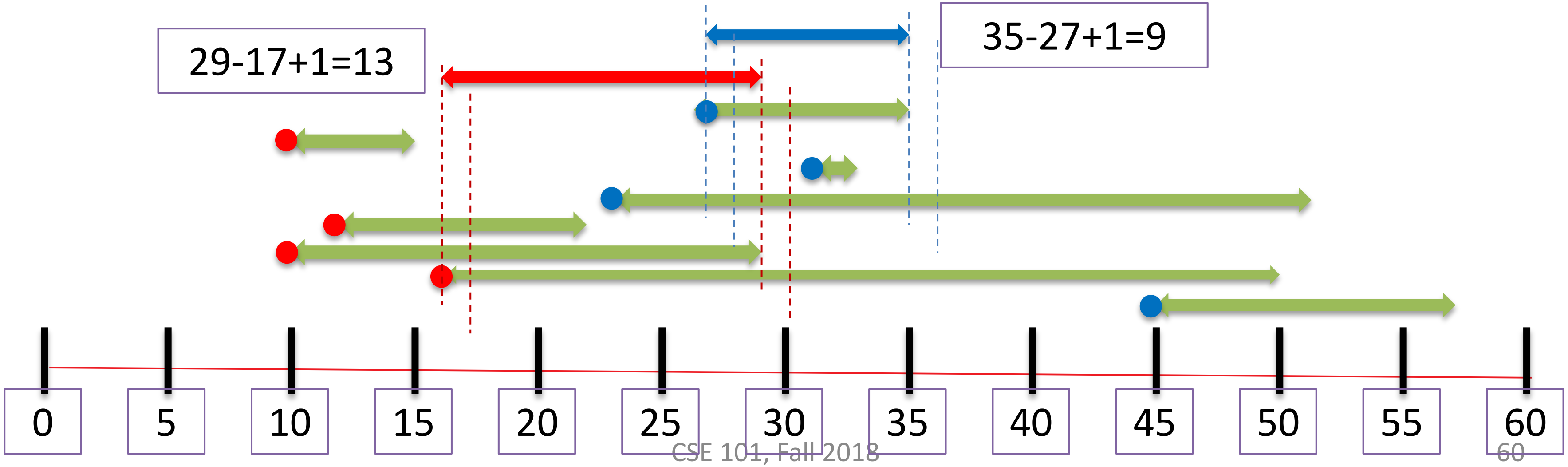
Greatest overlap

- Break the problem into smaller pieces
  - Sort the list and break it into lists each of size n/2.
    - [10,15],[10,29],[12,22],[17,50],[23,51],[27,35],[31,32],[45,57]
  - Let's assume we can get a divide and conquer algorithm to work. Then what information would it give us to recursively call each subproblem?



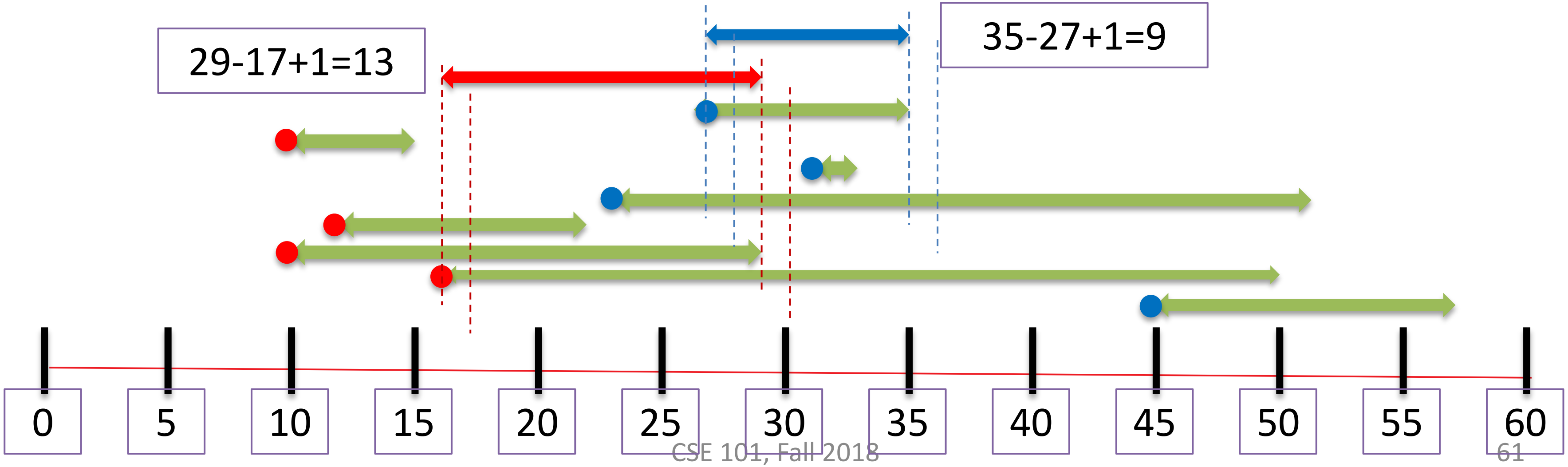
Greatest overlap

- Break the problem into smaller pieces
  - Let's assume we can get a divide and conquer algorithm to work. Then what information would it give us to recursively call each subproblem?
    - overlapDC([10,15],[10,29],[12,22],[17,50])=13
    - overlapDC([23,51],[27,35],[31,32],[45,57])=9



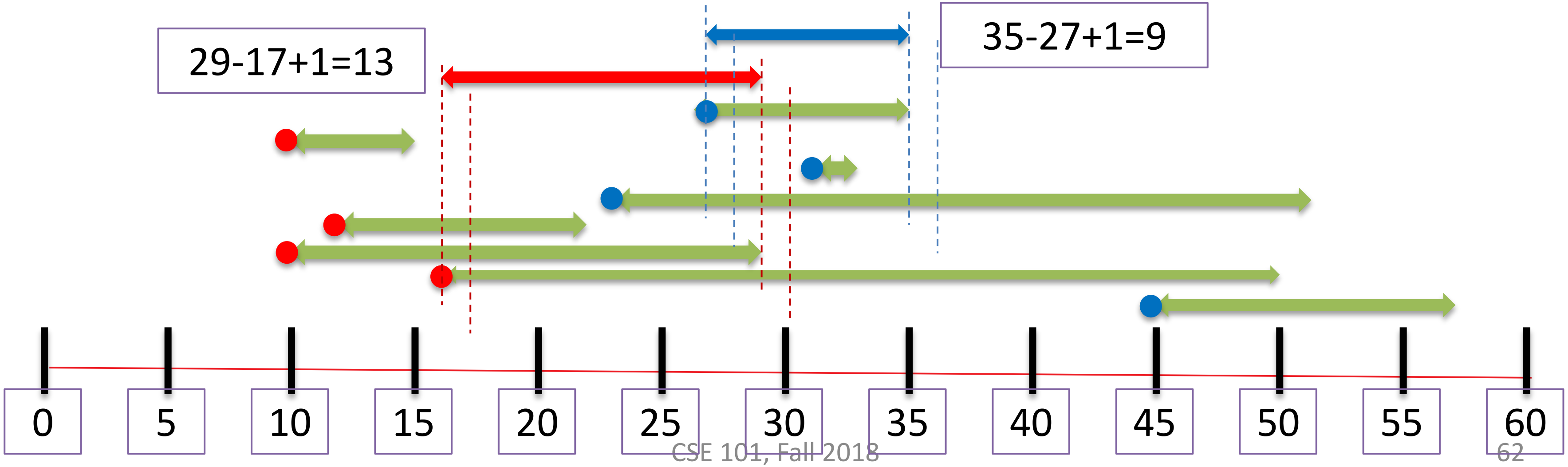
Greatest overlap

- Break the problem into smaller pieces
  - overlapDC([10,15],[10,29],[12,22],[17,50])=13
  - overlapDC([23,51],[27,35],[31,32],[45,57])=9
  - Is this enough information to solve the problem? What else must we consider?



Greatest overlap

- Break the problem into smaller pieces
    - overlapDC([10,15],[10,29],[12,22],[17,50])=13
    - overlapDC([23,51],[27,35],[31,32],[45,57])=9
  - The greatest overlap overall may be contained entirely in one sublist or it may be an overlap of one interval from either side



Greatest overlap

- Combine the results
  - So far, we have split up the set of intervals and recursively called the algorithm on both sides. The runtime of this algorithm satisfies a recurrence that looks something like

$$T(n) = 2T\left(\frac{n}{2}\right) + O(???)$$

  - What goes into the $O(???)$?

Greatest overlap

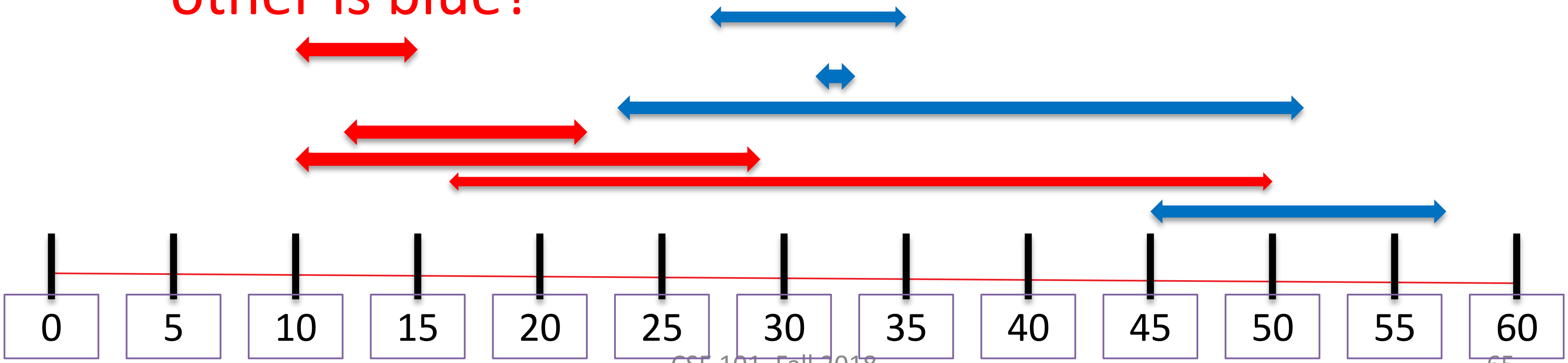
- Combine the results
  - So far, we have split up the set of intervals and recursively called the algorithm on both sides. The runtime of this algorithm satisfies a recurrence that looks something like

$$T(n) = 2T\left(\frac{n}{2}\right) + O(???)$$

  - What goes into the $O(???)$?
  - How long does it take to "combine". In other words, how long does it take to check if there is not a bigger overlap between sublists?

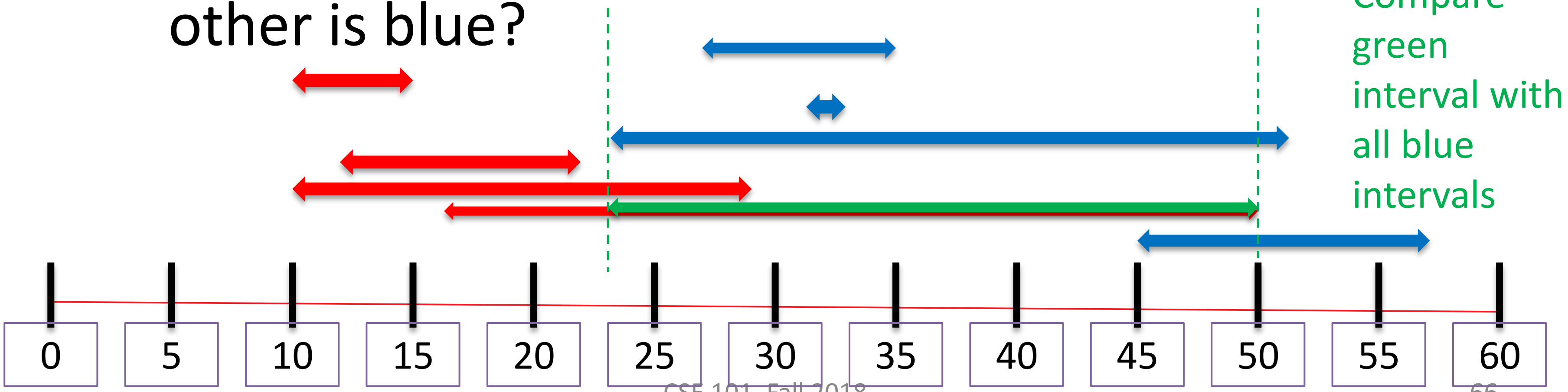
Greatest overlap

- Combine the results
  - What is an efficient way to determine the greatest overlap of intervals where one is red and the other is blue?



Greatest overlap

- Combine the results
  - What is an efficient way to determine the greatest overlap of intervals where one is red and the other is blue?



Greatest overlap between sets

- Let's formalize our algorithm that finds the greatest overlap of two intervals such that they come from different sets sorted by starting point

procedure **overlapbetween** ($[[a_1, b_1], \ldots [a_\ell, b_\ell]]$, $[[c_1, d_1], \ldots [c_k, d_k]]$)

$$(a_1 \leq a_2 \leq \cdots \leq a_\ell \leq c_1 \leq c_2 \leq \cdots \leq c_k)$$

Greatest overlap between sets

- Let's formalize our algorithm that finds the greatest overlap of two intervals such that they come from different sets sorted by starting point

procedure **overlapbetween** $([[a_1, b_1], \ldots [a_\ell, b_\ell]], [[c_1, d_1], \ldots [c_k, d_k]])$

$$(a_1 \leq a_2 \leq \cdots \leq a_\ell \leq c_1 \leq c_2 \leq \cdots \leq c_k)$$

```
if k==0 or ℓ == 0 then return 0
minc = c_1
maxb = 0
olap = 0
for i from 1 to ℓ:
    if maxb < b_i:
        maxb = b_i
for j from 1 to k:
    if olap < overlap([minc,maxb],[c_k, d_k]):
        olap = overlap([minc,maxb],[c_k, d_k])
return olap
```

Next lecture

- Divide and conquer algorithms
  - Reading: Chapter 2