Open-Source Operating Systems

Advanced Algorithms
TX00CO27

Lecture 4 - 13.11.2017

Jarkko.Vuori@metropolia.fi

Divide And Conquer

- The divide-and-conquer strategy so successfully used by monarchs and colonizers may also be applied to the development of efficient computer algorithms
    1. *Divide*: Smaller problems are solved recursively (except, of course, base cases)
    2. *Conquer*: The solution to the original problem is then formed from the solutions to the subproblems
- Distinguish between small and large instances
- Small instances solved differently from large ones
- Those smaller instances are often independent, therefore they can be worked on by different processor of the parallel computer

Small And Large Instance

- Small instance
  - Sort a list that has $n \leq 10$ elements
  - Find the minimum of $n \leq 2$ elements
- Large instance
  - Sort a list that has $n > 10$ elements
  - Find the minimum of $n > 2$ elements

Solving A Small Instance

- A small instance is solved using some direct/simple strategy
  - Sort a list that has $n \leq 10$ elements
    - Use count, insertion, bubble, or selection sort
  - Find the minimum of $n \leq 2$ elements
    - When $n = 0$, there is no minimum element
    - When $n = 1$, the single element is the minimum
    - When $n = 2$, compare the two elements and determine which is smaller

Solving A Large Instance

- A large instance is solved as follows:
  - Divide the large instance into $k \geq 2$ smaller instances
  - Solve the smaller instances somehow
  - Combine the results of the smaller instances to obtain the result for the original large instance

Sort A Large List

- Sort a list that has n > 10 elements
  - Sort 15 elements by dividing them into 2 smaller lists
    - One list has 7 elements and the other has 8
  - Sort these two lists using the method for small lists
  - Merge the two sorted lists into a single sorted list

Find The Min Of A Large List

- Find the minimum of 20 elements
  - Divide into two groups of 10 elements each
  - Find the minimum element in each group somehow
  - Compare the minimums of each group to determine the overall minimum

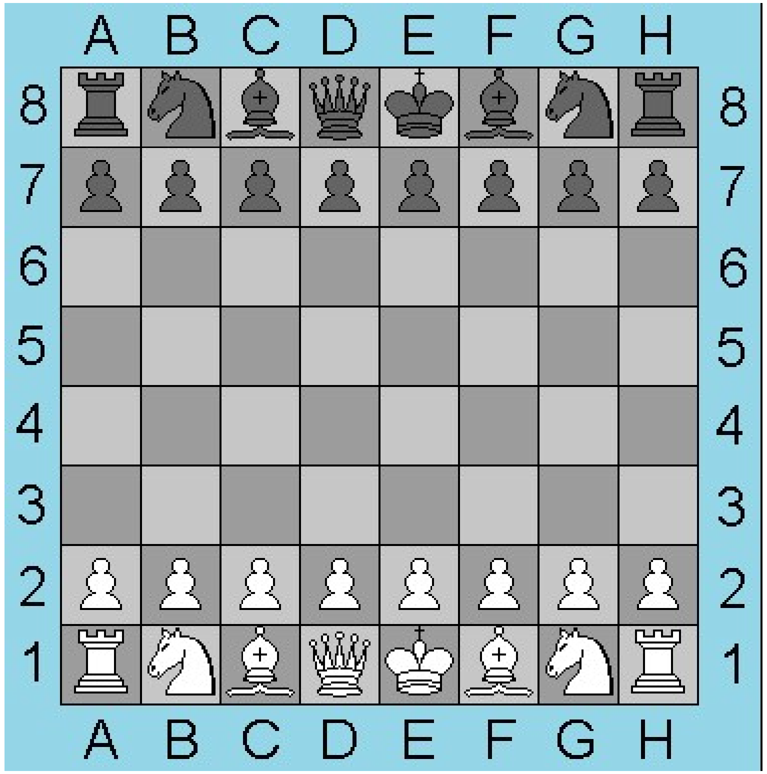
Recursion In Divide And Conquer

- Often the smaller instances that result from the divide step are instances of the original problem (true for our sort and min problems). In this case,
  - If the new instance is a small instance, it is solved using the method for small instances
  - If the new instance is a large instance, it is solved using the divide-and-conquer method recursively
- Generally, performance is best when the smaller instances that result from the divide step are of approximately the same size
- For example, nearly all the binary tree algorithms use this technique
  - To traverse a binary tree tree → small instance == NULL; large instance → every non empty binary tree; traversal→ NULL→ do nothing; non NULL → traverse left, traverse right, visit (say)

Recursive Find Min

- Find the minimum of 20 elements
  - Divide into two groups of 10 elements each
  - Find the minimum element in each group recursively. The recursion terminates when the number of elements is ≤ 2. At this time the minimum is found using the method for small instances
  - Compare the minimums of the two groups to determine the overall minimum

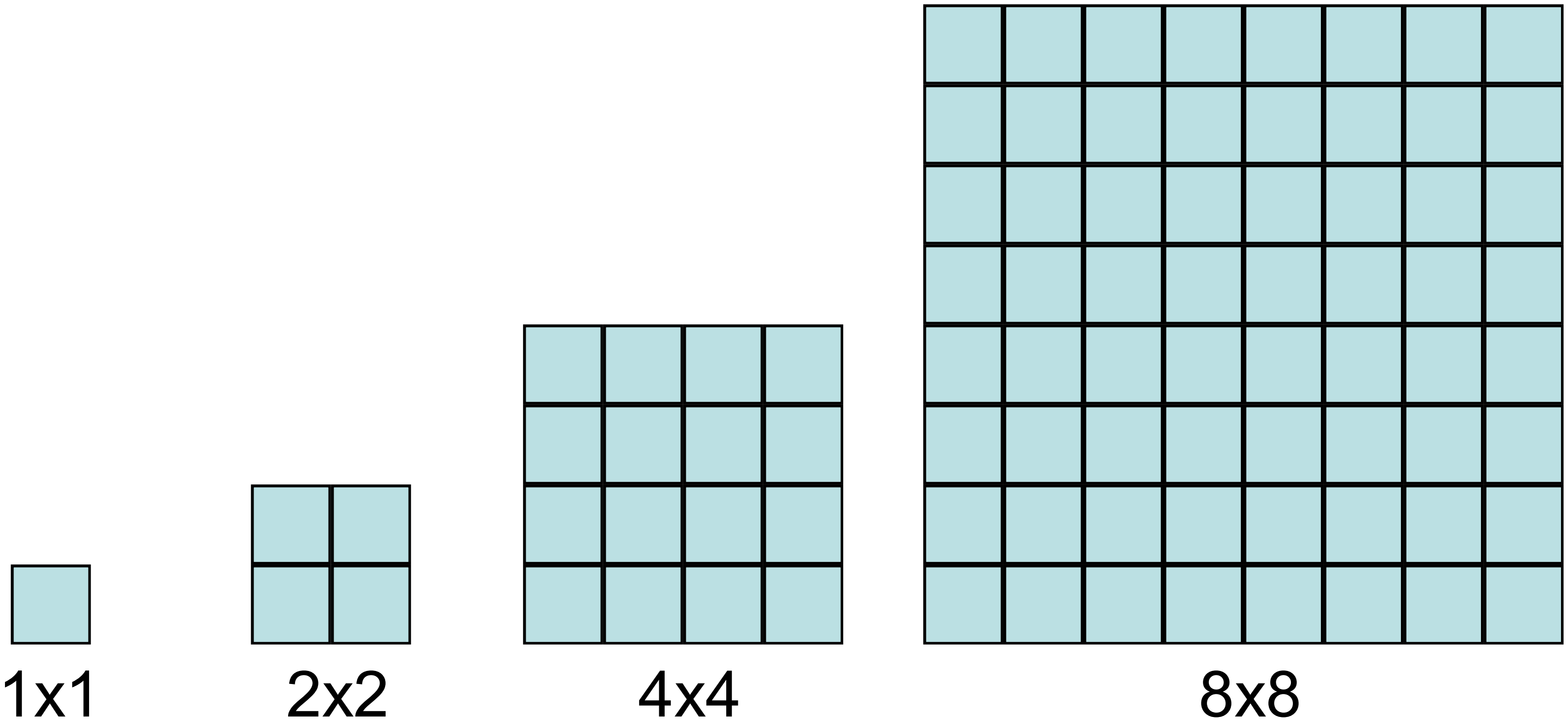
Tiling A Defective Chessboard



A real chessboard.

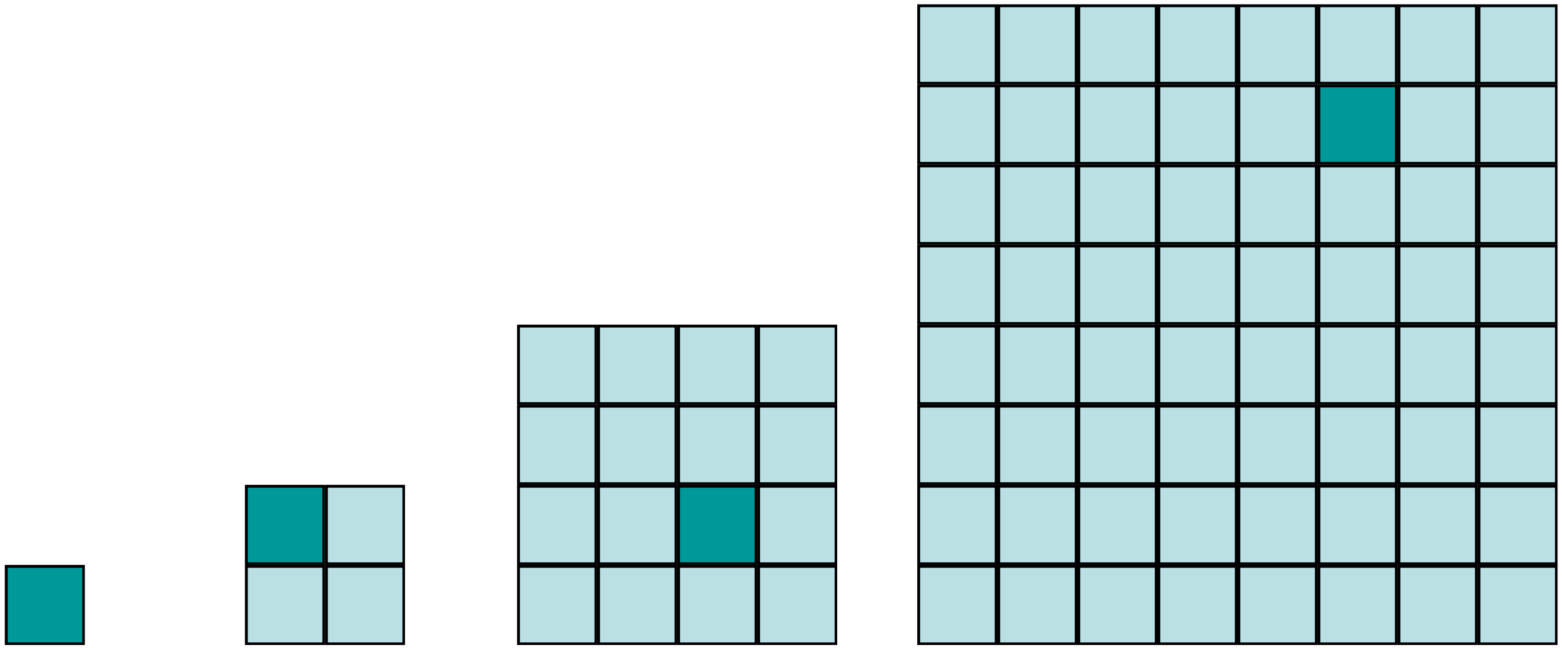
Our Definition Of A Chessboard

A chessboard is an n x n grid, where n is a power of 2.



A Defective Chessboard

A defective chessboard is a chessboard that has one unavailable (defective) position



1x1 2x2 4x4 8x8

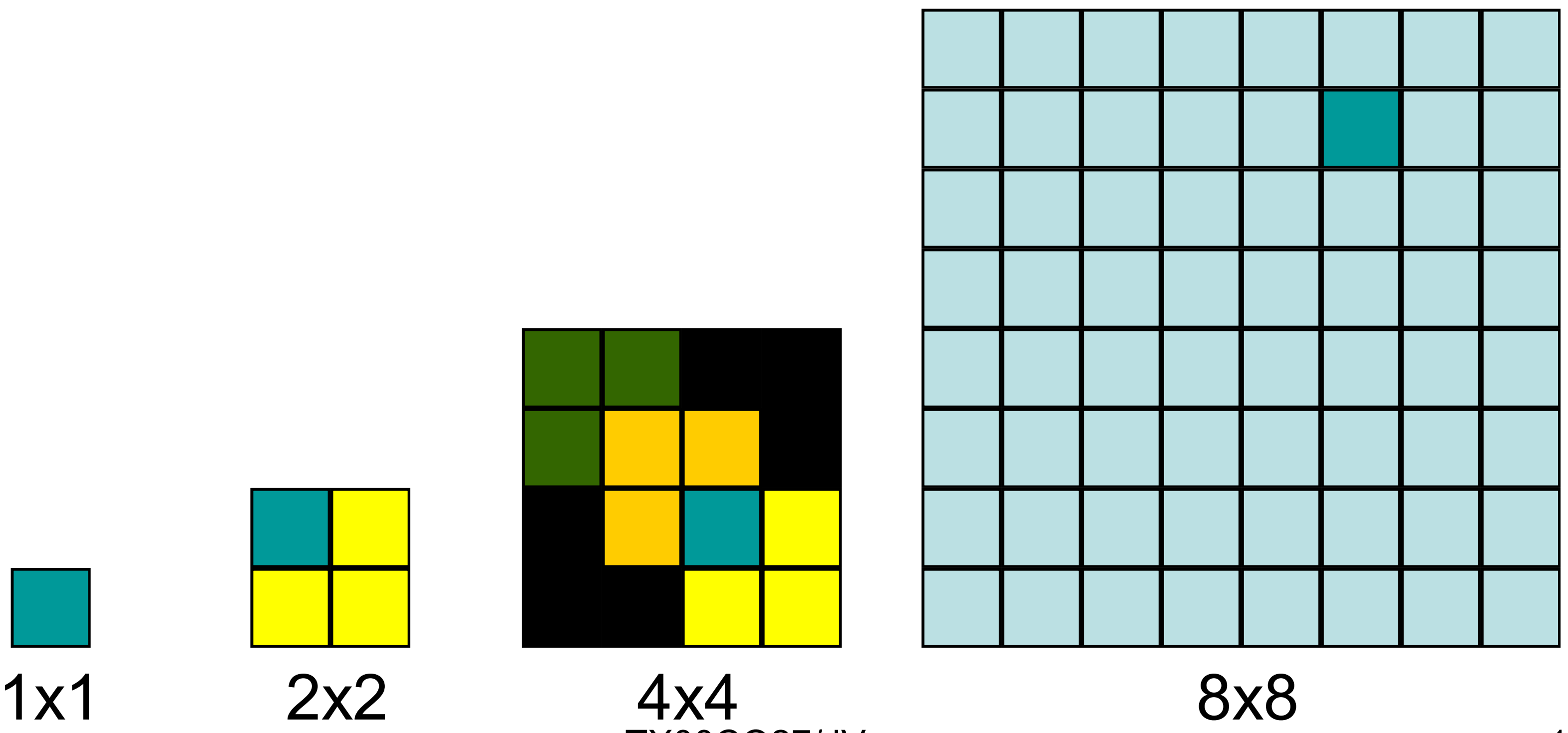
A Triomino

A triomino is an L shaped object that can cover three squares of a chessboard

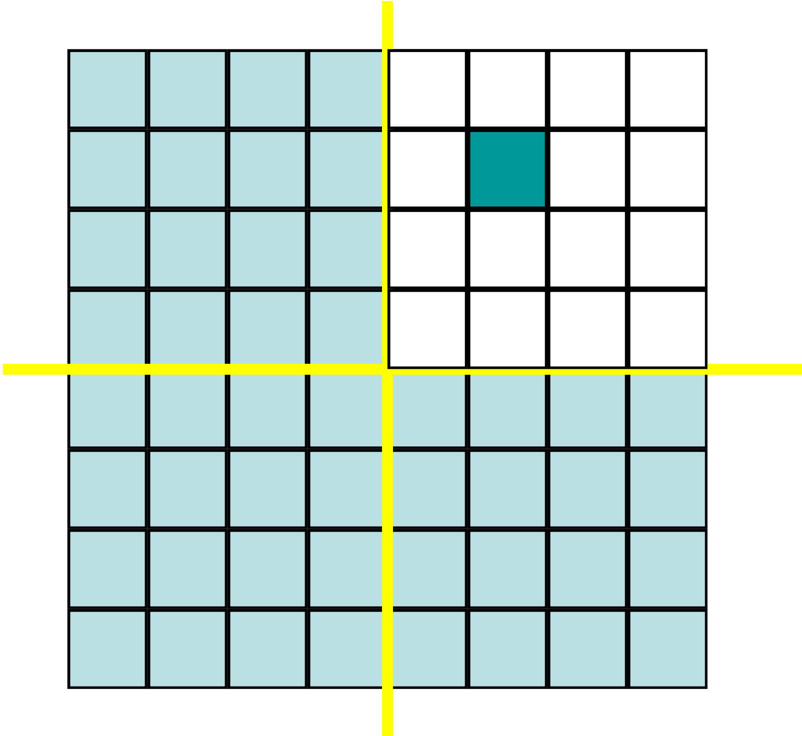
A triomino has four orientations

Tiling A Defective Chessboard

Place $(n^2 - 1)/3$ triominoes on an $n \times n$ defective chessboard so that all $n^2 - 1$ nondefective positions are covered



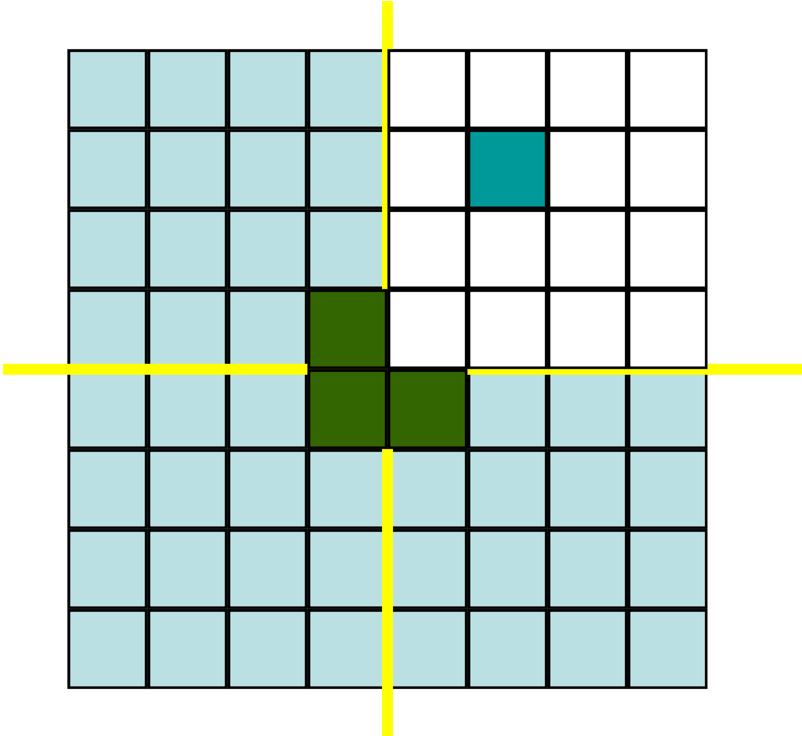
Tiling A Defective Chessboard



Divide into four smaller chessboards. 4 x 4

One of these is a defective 4 x 4 chessboard.

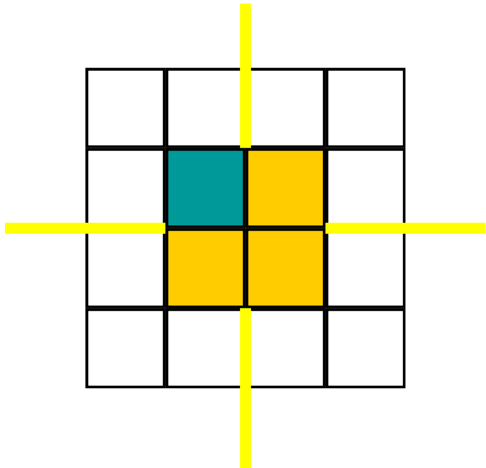
Tiling A Defective Chessboard



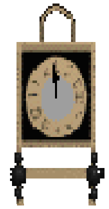
Make the other three 4 x 4 chessboards defective by placing a triomino at their common corner

Recursively tile the four defective 4 x 4 chessboards

Tiling A Defective Chessboard



Complexity



- Let $n = 2^k$
- Let $t(k)$ be the time taken to tile a 2^k x 2^k defective chessboard
- $t(0) = d$, where d is a constant
- $t(k) = 4t(k-1) + c$, when $k > 0$. Here c is a constant
- Recurrence equation for $t()$

Substitution Method

$$
\begin{aligned}
t(k) &= 4t(k-1) + c \\
&= 4[4t(k-2) + c] + c \\
&= 4^2\ t(k-2) + 4c + c \\
&= 4^2[4t(k-3) + c] + 4c + c \\
&= 4^3\ t(k-3) + 4^2c + 4c + c \\
&= \ldots \\
&= 4^k\ t(0) + 4^{k-1}c + 4^{k-2}c + \ldots + 4^2c + 4c + c \\
&= 4^k\ d + 4^{k-1}c + 4^{k-2}c + \ldots + 4^2c + 4c + c \\
&= \Theta(4^k) \\
&= \Theta(\text{number of triominoes placed})
\end{aligned}
$$

Min And Max

Find the lightest and heaviest of n elements using a balance that allows you to compare the weight of 2 elements

Minimize the number of comparisons

Max Element

- Find element with max weight from w[0:n-1]

```
maxElement = 0;
for (int i = 1; i < n; i++)
   if (w[maxElement] < w[i]) maxElement = i;
```

- Number of comparisons of w values is n-1

Min And Max

- Find the max of n elements making $n-1$ comparisons
- Find the min of the remaining $n-1$ elements making $n-2$ comparisons
- Total number of comparisons is $2n-3$

Divide And Conquer

- Small instance
  - $n \leq 2$
  - Find the min and max element making at most one comparison

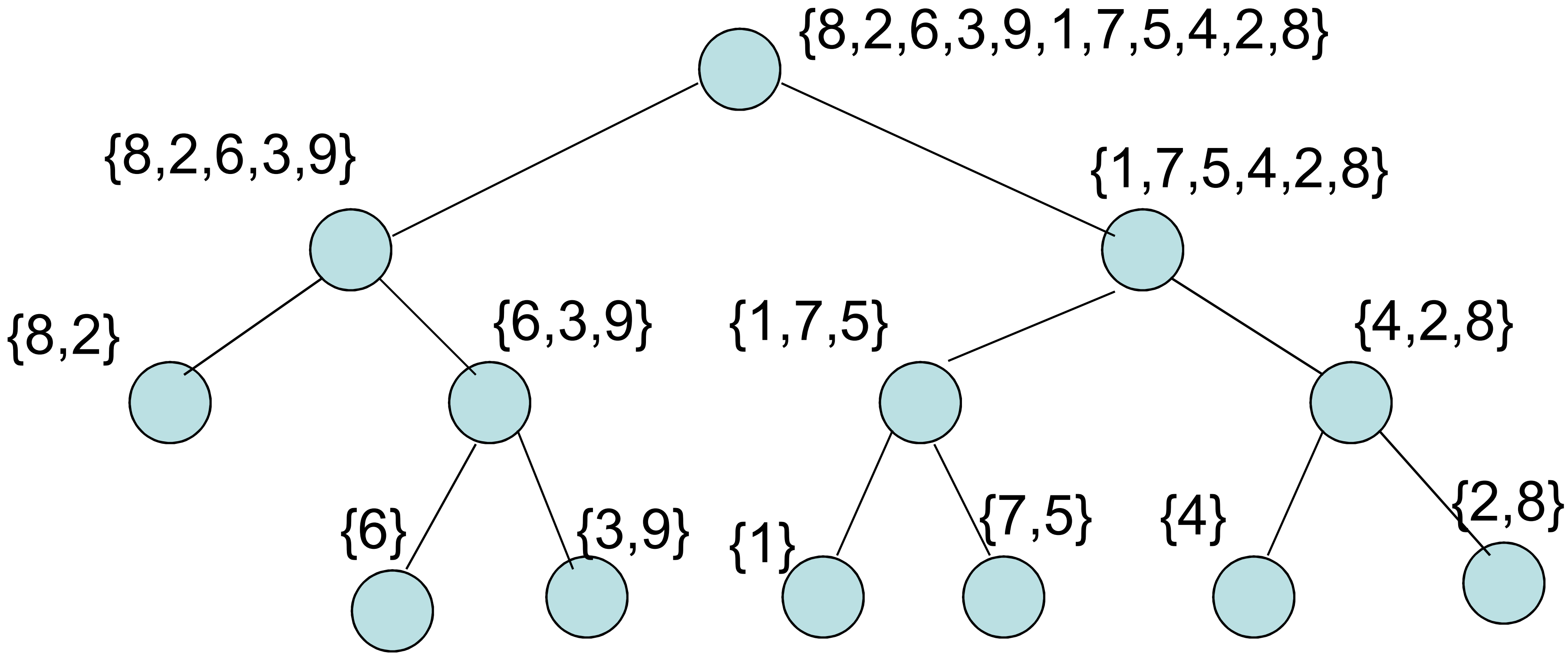
Large Instance Min And Max

- $n > 2$
- Divide the n elements into 2 groups A and B with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements, respectively
- Find the min and max of each group recursively
- Overall min is $\min\{\min(A), \min(B)\}$
- Overall max is $\max\{\max(A), \max(B)\}$

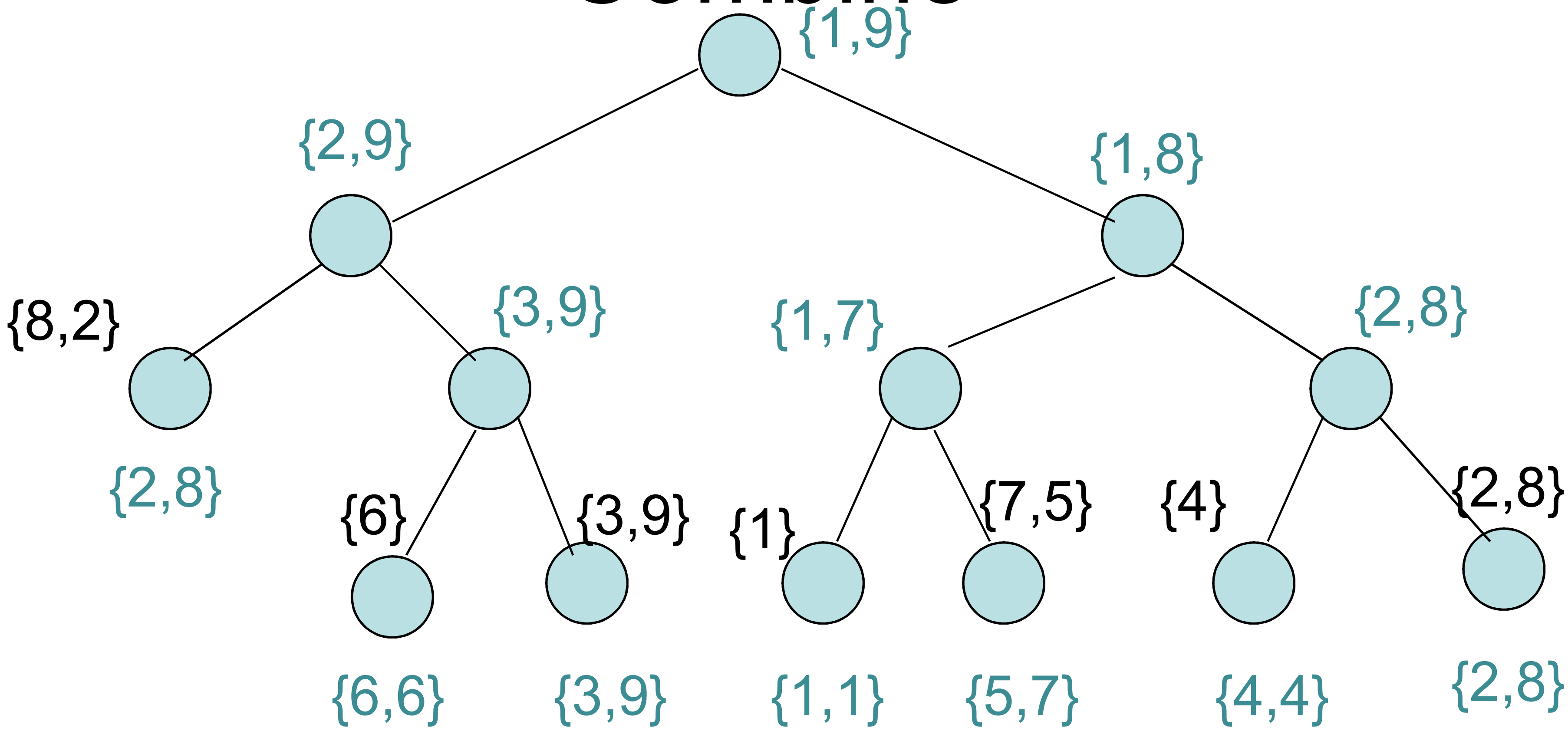
Min And Max Example

- Find the min and max of {3,5,6,2,4,9,3,1}
- Large instance
- A = {3,5,6,2} and B = {4,9,3,1}
- min(A) = 2, min(B) = 1
- max(A) = 6, max(B) = 9
- min{min(A),min(B)} = 1
- max{max(A), max(B)} = 9

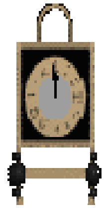
Dividing Into Smaller Instances



Solve Small Instances And Combine



Time Complexity



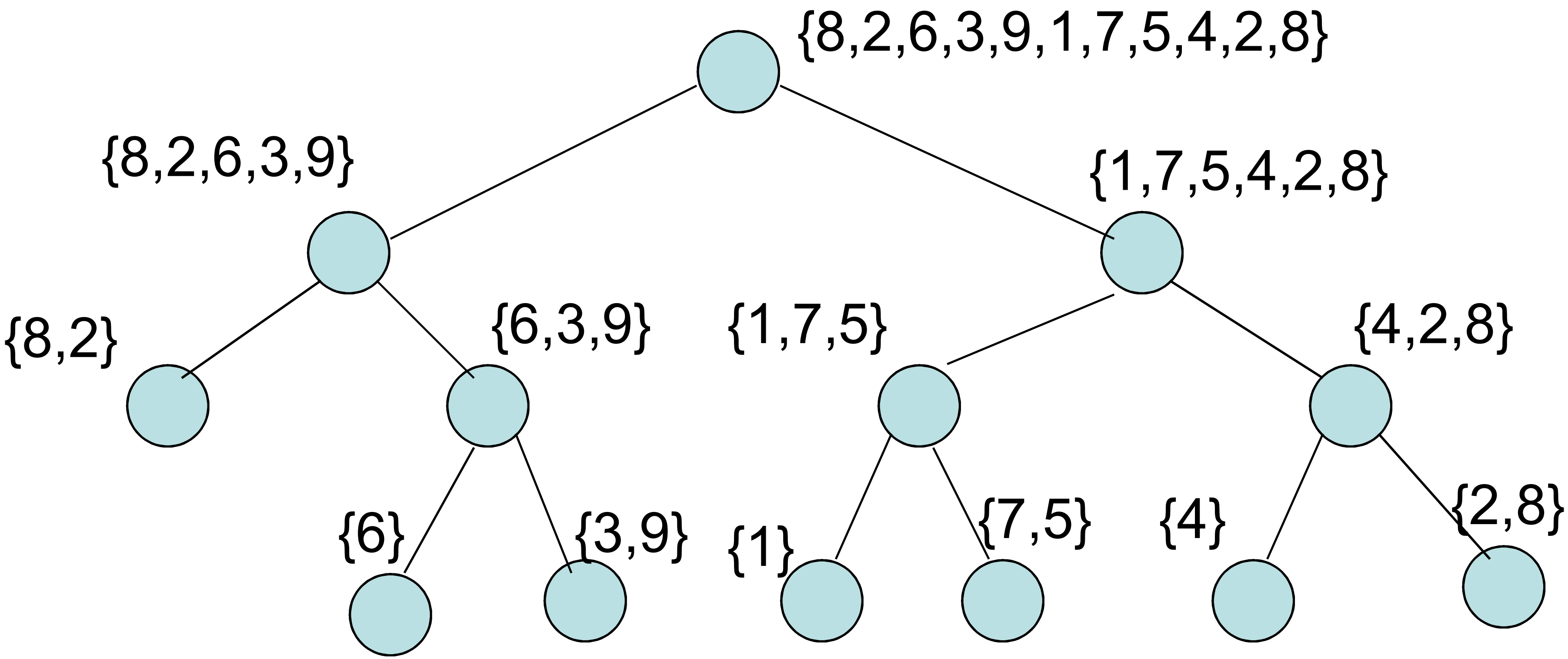
- Let $c(n)$ be the number of comparisons made when finding the min and max of n elements
- $c(0) = c(1) = 0$
- $c(2) = 1$
- When $n > 2$,

  $c(n) = c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 2$
- To solve the recurrence, assume n is a power of 2 and use repeated substitution
- $c(n) = \lceil 3n/2 \rceil - 2$ (compare this to $2n-3$ without divide and conquer method)

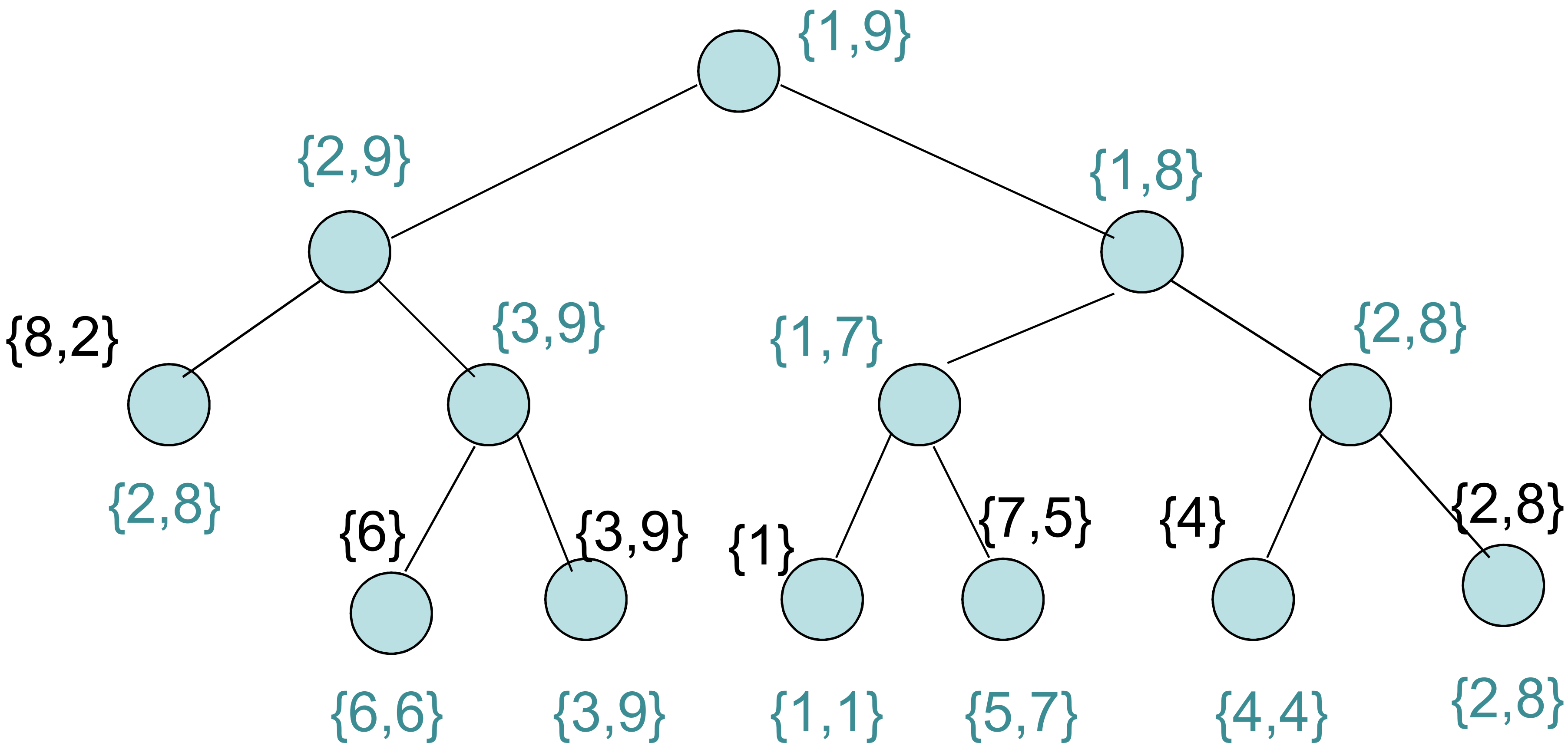
Interpretation Of Recursive Version

- The working of a recursive divide-and-conquer algorithm can be described by a tree—recursion tree
- The algorithm moves down the recursion tree dividing large instances into smaller ones
- Leaves represent small instances
- The recursive algorithm moves back up the tree combining the results from the subtrees
- The combining finds the min of the mins computed at leaves and the max of the leaf maxs

Downward Pass Divides Into Smaller Instances



Upward Pass Combines Results From Subtrees



Iterative Version

- Start with n/2 groups with 2 elements each and possibly 1 group that has just 1 element
- Find the min and max in each group
- Find the min of the mins
- Find the max of the maxs

Iterative Version Example

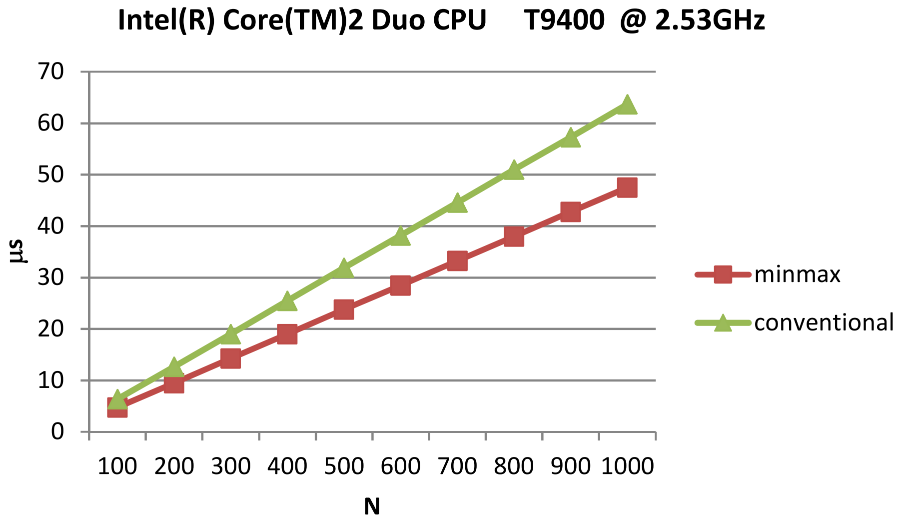
- {2,8,3,6,9,1,7,5,4,2,8 }
- {2,8}, {3,6}, {9,1}, {7,5}, {4,2}, {8}
- mins = {2,3,1,5,2,8}
- maxs = {8,6,9,7,4,8}
- minOfMins = 1
- maxOfMaxs = 9

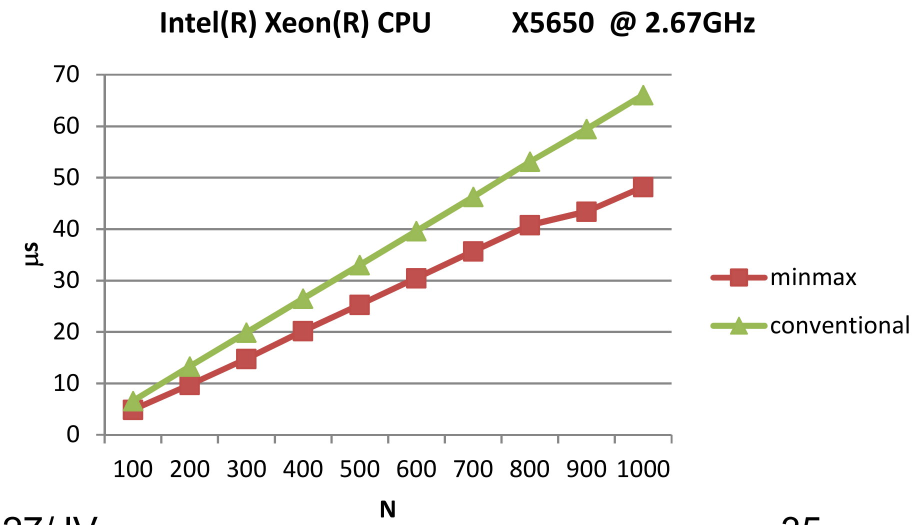
Comparison Count

- Start with $n/2$ groups with 2 elements each and possibly 1 group that has just 1 element
  - No compares
- Find the min and max in each group
  - $\lfloor n/2 \rfloor$ compares
- Find the min of the mins
  - $\lceil n/2 \rceil - 1$ compares
- Find the max of the maxs
  - $\lceil n/2 \rceil - 1$ compares
- Total is $\lceil 3n/2 \rceil - 2$ compares

Time Complexity

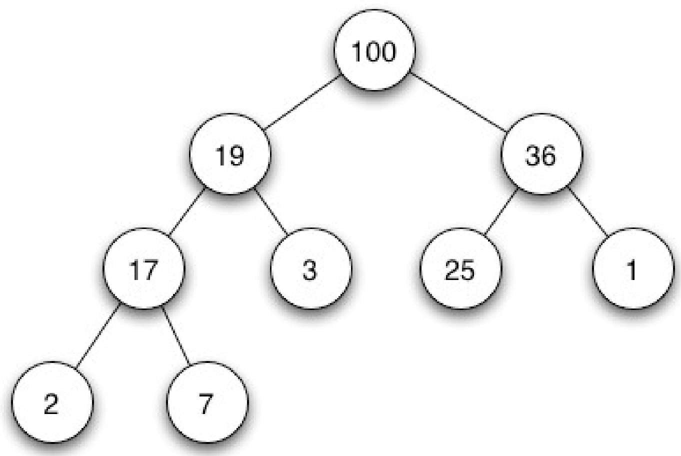
```
/* find min and max at the same time */
#define VAL(n) ((Titem *)((char *)base+(n)*s))
void minmax(void *base, int s, int n, int (*cmp)(const void *, const void *), int *min, int
*max) {
    int i, start;

    // check simple cases first
    if (n < 1) return;
    else if (n == 1) {
        *min = 0; *max = 0;
        return;
    }

    start = 1;
    if (n % 2 == 1) { // odd lenght
        *min = 0; *max = 0;
    } else {          // even lenght
        if (cmp(VAL(0), VAL(1)) > 0) {
            *min = 1; *max = 0;
        } else {
            *min = 0; *max = 1;
        }
        start = 2;
    }

    //  compare remaining pairs
    for (i = start; i < n; i += 2) {
        // find larger of base[i] and base[i+1], then compare larger one
        // with base[max] and smaller one with base[min]
        if (cmp(VAL(i), VAL(i+1)) > 0) {
            if (cmp(VAL(i),   VAL(*max)) > 0) *max = i;
            if (cmp(VAL(i+1), VAL(*min)) < 0) *min = i+1;
        } else {
            if (cmp(VAL(i+1), VAL(*max)) > 0) *max = i+1;
            if (cmp(VAL(i),   VAL(*min)) < 0) *min = i;
        }
    }
}


/* conventional implementation for the minmax */
void minmax2(void *base, int s, int n,
           int (*cmp)(const void *, const void *), int *min, int *max) {
    int i;

    *min = 0; *max = 0;
    for (i = 0; i < n; i++) {
        if (cmp(VAL(i), VAL(*min)) < 0) *min = i;
        else if (cmp(VAL(i), VAL(*max)) > 0) *max = i;
    }
}
```





Initialize A Heap

- A max heap (min heap) is a binary max tree (min tree) in which the value in each node is greater (less) than or equal to those in its children (if any)
- $n > 1$:
  - Initialize left subtree and right subtree recursively
  - Then do a trickle down operation at the root
- $t(n) = c,\ n \leq 1$
- $t(n) = 2t(n/2) + d * \text{height},\ n > 1$
- c and d are constants
- Solve to get $t(n) = O(n)$



Running time of Divide and Conquer Algorithms

- At the lecture 1 we found the running time equation T(*N*)=2T(*N*/2)+O(*N*) has solution O(*N*log*N*)
- The general solution to the equation T(N)=aT(N/b)+Θ(N^k), where a≥1 and b≥1, is

$$T(N) = \begin{cases} O\left(N^{\log_b a}\right) & if\ a > b^k \\ O\left(N^k \log N\right) & if\ a = b^k \\ O\left(N^k\right) & if\ a < b^k \end{cases}$$

Multiplying integers

- We want to multiply two N-digit numbers X and Y
  - We assume that the sign is handled separately and therefore $X,Y \geq 0$
- The algorithm we usually use (when calculating the multiplication by hand) requires $\Theta(N^2)$ operations (each digit in X is multiplied by each digit in Y)
- If X= 61 438 521 and Y= 94 736 407 then XY= 5 820 464 730 934 047
- If we break X and Y into two halves, consisting most significant and least significant digits, respectively
  - Then X_L=6 143, X_R=8 521, Y_L=9 473, and Y_R=6 407
  - We also have $X=X_L10^4 + X_R$ and $Y=Y_L10^4 + Y_R$. It follows that $XY=X_LY_L10^8 + (X_LY_R+X_RY_L)10^4 + X_RY_R$
  - This equation contains four multiplications which are half the size of the original problem (N/2 digits), multiplication by 10^4 and 10^8 amount to the placing of zeros. This and the subsequent additions add only O(N) additional work
  - If these four multiplications are performed recursively, stopping at the appropriate base case, we obtain the recurrence $T(N)=4T(N/2)+O(N)$ which can be seen $T(N)=O(N^2)$
    - So, unfortunately, we have not improved the algorithm
    - To achieve subquadratic algorithm, we must use less than four recursive calls

Multiplying integers

- The mathematican Gauss once noticed that although the product of two complex numbers (a+bi)(c+di)=ac-bd+(bc+ad)i seems to involve four real-number multiplications, it can in fact be done with just three, since bc+ad=(a+b)(c+d)-ac-bd
- This is the key observation in reducing the number of multiplications that $X_L Y_R + X_R Y_L = (X_L - X_R)(Y_R - Y_L) + X_L Y_L + X_R Y_R$
  - Thus, instead of using two multiplications to compute the coefficient of 10^4, we can use one multiplication, plus the result of two multiplications that have already been performed
  - Now we need only three multiplications (= recursive subproblems) to be solved
  - The recurrence equation is now T(N)=3T(N/2) + O(N) and so we obtain $T(N) = O\left(N^{\log_2 3}\right) = O\left(N^{1,59}\right)$
    - To complete the algorithm, we must have a base case, which can be solved without recursion
      - When both numbers are one-digit, we can do the multiplication by table lookup
      - If one number has zero digits, then we return zero

Multiplying integers

- When both numbers are one digit, we can do multiplication by table lookup
- If one number has zero digits, then we return zero
- In practice, the base case should be set to that which is most convenient for the machine
- If numbers are not equal in size, a smaller one must be zero padded, e.g. 3×15 must be adjusted to 03×15

```
function multiply(x,y)
Input: two n-digit numbers x and y
Output: their product

if n=1: return xy

xl,xr = leftmost, rightmost ⌈n/2⌉ digits of x
yl,yr = leftmost, rightmost ⌈n/2⌉ digits of y

p1 = multiply(xl,yl)
p2 = multiply(xr,yr)
p3 = multiply(xl+xr,yl+yr)

return P1×10^n+(P3-P1-P2)×10^(n/2)+P2
```