Divide and Conquer

An Algorithm design method

The principle

- Divide a large problem instance into one or more smaller instances
  - Solve the smaller instances by the same approach, if not trivial
  - Solve directly when instance becomes trivial
- Can you see recursion at play?
- Algorithms designed this way are typically analyzed using **recurrence equations**

General form of recurrence relations from divide and conquer – Master theorem

- $T(n) = aT\left(\frac{n}{b}\right) + cn^k$, for $n > 1$
  $T(1) = d$
- Solutions depend on the ratio b^k/a
  - $T(n) = \Theta(n^{\log_b a})$, if $a > b^k$
  - $T(n) = \Theta(n^k \log n)$, if $a = b^k$
  - $T(n) = \Theta(n^k)$, if $a < b^k$
- b – the number of sub-problems
- a – how many of the sub-problems are to be solved
- k– amount of extra work

Other Forms of Recurrence

- $T(n) = T(n-1) + n$
  $T(1) = 1$
  - Solution: Quadratic – $\Theta(n^2)$
- $T(n) = \mathrm{aT(n-1)+k}$
  $\mathrm{T(1)=k}$
  - Solution: Exponential – $\Theta(a^n)$

Example

- $T(n) = T\left(\frac{n}{2}\right) + c$, for $n > 1$
  $T(1) = c_0$
- Familiar?
  - Binary search
- Analysis
  - $a = 1, b = 2, k = 0$
    - Split in halves, discard one half; the splitting is done in constant time
  - $a = b^k$
  - $\Theta(n^k \log n) = \Theta(\log n)$ as before

Example

- $T(n) = 2T\left(\frac{n}{2}\right) + nc$, for $n > 1$
  $T(1) = c_0$
- Analysis
  - $a = b = 2, k = 1$
    - Split in halves, solve both; splitting done in linear time
  - $a = b^k$
  - $\Theta(n^k \log n) = \Theta(n \log n)$

Example

- $a = b, k = 0$
  - Partition into b partitions, solve all b partions; splitting/recombination done in constant time
- $a > b^k$
- $\Theta\left(n^{\log_b a}\right) = \Theta(n)$
- Eg: finding min/max element in an array

Example: Finding min/max (naïve)

```
public class MinMaxPair{
    public int min, max;
}

public MinMaxPair find(float a[]) {
    MinMaxPair res = new MinMaxPair();
    res.min = res.max = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[i] > a[res.max]) res.max = i;
        if (a[i] < a[res.min]) res.min = i;
    }
    return res;
}
```

Indices of smallest and largest numbers, resp.

Initially assume largest and smallest at first index

min/max (naïve): Analysis

```
public static class MinMaxPair{
    public int min, max;
}

public static MinMaxPair find(float a[]) {
    MinMaxPair res = new MinMaxPair();
    res.min = res.max = 0;
    for (int i = 1; i < a.length; i++) {
        if (a[i] > a[res.max]) res.max = i;
        if (a[i] < a[res.min]) res.min = i;
    }
    return res;
}
```

Repeats n-1 times because we start at 1 and stop at n-1

$T(n) = 2(n-1) + c$

Therefore $\Theta(n)$

Min/max(divide and conquer)

```
public static MinMaxPair find2(float a[], int l, int r) {
    MinMaxPair res = new MinMaxPair();
    if (l == r) {res.min = res.max = r; return res;}
    if (l == r - 1) {
        if (a[l] > a[r]) {
            res.min = r; res.max = l;
        } else {res.min = l; res.max = r;}
        return res;
    }
    int mid = (l+r)/2;
    MinMaxPair p1 = find2(a, l, mid);
    MinMaxPair p2 = find2(a, mid + 1, r);
    res.min = a[p1.min] < a[p2.min] ? p1.min : p2.min;
    res.max = a[p1.max] > a[p2.max] ? p1.max : p2.max;
    return res;
}
```

When sublist has only 1 item

When sublist has two items

Split list in the middle and search each sublist

Get the minimum of the 2 minima and maximum of the two maxima

Min/max(divide and conquer): Analysis

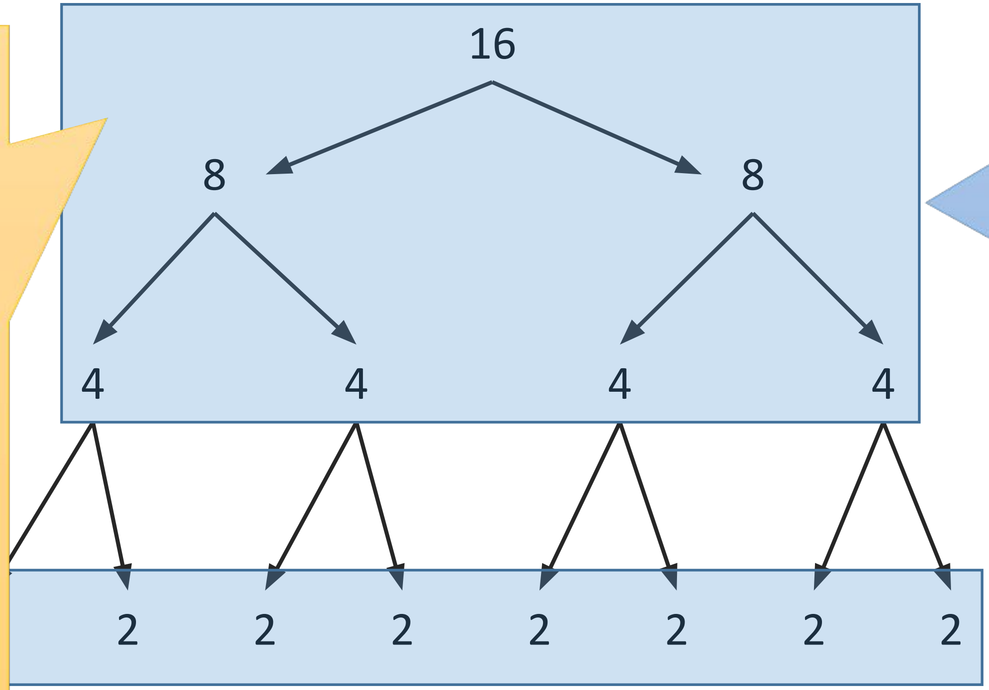
- Assume list size, n, is a power of 2
  - Eg. 16: The following tree shows subdivision to get to sublists of size 2

Total number of comparisons:

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

Therefore also $\Theta(n)$

But, actual number of comparisons is $\frac{3}{2}n$, which is a 25% improvement over the naïve version which has $2n - 2$



$7 \left(= \frac{n}{2} - 1\right)$ internal nodes requiring two comparisons each

$8\ (= n/2)$ leaves requiring one comparison each

Towers of Hanoi

- Algorithm:

```
static void TOH(int n, Pole start, Pole goal, Pole temp) {
   if (n==1)
      System.out.println("move ring from pole " +
         + start + " to pole " + goal);
   else {
      TOH(n-1, start, temp, goal);
      System.out.println("move ring from pole " +
         + start + " to pole " + goal);
      TOH(n-1, temp, goal, start);
   }
}
```

Towers of Hanoi: Analysis

- $T(n) = 2T(n-1) + 1$
  $T(1) = 1$
  - A special case of $T(n) = \mathrm{aT(n-1) + k; T(1) = k}$
    - $a = 2, k = 1$
  - Solution: $T(n) = \Theta(2^n)$
    - Can you work this out using the substitution method?
    - Then prove the result using induction.