Divide and Conquer

- What's divide-and-conquer
- How to analyze a divide-and-conquer algorithm
- An example: binary search

What is divide and conquer

- A technique for designing algorithms that decompose instance into smaller sub-instance of the same problem
  - Solving the sub-instances independently
  - Combining the sub-solutions to obtain the solution of the original instance

A general template

```
DC(x)
{
    if (x is sufficiently small or simple)
            adhoc(x);   // use a basic sub-algorithm

    decompose x into small instances x[0],..,x[l-1];  // divide

    for (i=0;  i<l; i++)                          //conquer
            s[i] = DC(x[i]);

    combine s[0],.., s[l-1] to obtain solution s for x;  // combine
    return s;
}
```

- Three conditions to be considered
  - When to use the basic sub-algorithm
  - Efficient decomposition and recombination
  - The sub-instances must be roughly the same size

Running-time analysis

- Assume that the *l* sub-instances have roughly the same size n/*b* for some constant *b*
- Let g(n) be the time required by DC for dividing and combining on instances of size n,
  - g(n) is the total time excluding the times need for the recursive calls.
  - We have $t(n) = l \cdot t(n/b) + g(n)$
- If $g(n) \in \Theta(n^k)$ for an integer k, we have

$$t(n) \in \begin{cases} \Theta(n^k) & if \quad l < b^k \\ \Theta(n^k \log n) & if \quad l = b^k \\ \Theta(n^{\log_b l}) & if \quad l > b^k \end{cases}$$

Sequential Search from a sorted sequence

- T[] is a sequence in nondecreasing order
- Return the insertion position of a new value *x*

```
sequentialSearch(T[], x)
{
    for (i=0; i<n; i++) {
        if (T[i] >= x) // T[i-1]<x<=T[i]
            return i;
    }

}
```

Cost: best, worst, average?

Binary Search

```
binarySearch(T[], x)
{
    if (n==0 || x>T[n])
        return n;
    else
        return binaryRecursive(T, 1, n, x);
}
```

```
binaryRecursive(T[], i, j, x)
{
    // we know T[i-1] < x <= T[j]
    // assume T[-1] is sufficiently small
    if (i==j)
        return i;
    k = (i+j)/2;
    if (x <= T[k])
        return binaryRecursive(T, i, k, x);
    else
        return binaryRecursive(T, k+1, j, x);
}
```

Cost?

Cost of binary search

- $T(n) = T(n/2) + \Theta(1)$
- $T(n) \in \Theta(\lg n)$

Binary Search (iterative)

```
int binarySearch(int A[], int n, int x)
{
  int i, j, k;

  i=1; j=n;
  while (i<j) {
    k = (i+j)/2;
    if (x<=A[k]) j=k;
    else i = k+1;
  }
  return i;
}
```