Deterministic AA

- Introduction, loop counting
- Euclidian algorithm for GCD
- Divide and conquer
- Traversing binary trees
- Sorting algorithms; lower bound for the comparison based algorithms
- Pattern matching

Analysis of Algorithms

- **Analysis of algorithms** is the science of determining the amounts of time and space required by computers to carry out various procedures.
- Often we need to choose between the space and time requirements.
- In general we try to choose the most efficient algorithm to solve a problem, but the efficient algorithms are in general more complex.

Example A1

```
template <class otype>
int location_of_max (const otype
  a[], int first, int last)
{
  int max_loc = first;
  for (++first; first <= last;
                ++first)
    if (a[first] > a[max_loc])
         max_loc = first;
  return max_loc;
}
```


Example A2

```
template <class otype>
int location_by_linear_search
  (const otype a[], const otype
  &target, int first, int last)
{
   while (first <= last &&
          a[first] != target )
      ++first;
   return first;
}
```

```
template <class otype>
void binary_search (const otype a[], const otype
   & target, int first, int last, bool & found,
   int & subscript)
{
    int mid;
    found = false;
    while (first <= last && !found) {
       mid = (first + last)/2;
       if (target < a[mid])
          last = mid - 1;
       else if (a[mid] < target)
          first = mid + 1;
       else
          found = true;
    }
    if (found)
       subscript = mid;
    else
       subscript = first;
}
```


Complexity of the Binary Search

- Let M(n) be the minimum number of times that the body of the loop will be executed during an unsuccessful binary search of a subarray of length n.
- M(0)=0, M(1)=1
- $M(n)=1+M(\lfloor (n-1)/2 \rfloor) \leq 1+M(\lfloor n/2 \rfloor)$
- We can show that
- $M(n) \leq \lfloor \lg n \rfloor = O(\log n)$

Theorem A4

- Let p , q, and M be positive integers. Then pq > M if and only if p > $\lfloor M/q \rfloor$.

Euclid's Algorithm for GCD

```
int g_c_d (int m, int n)
{
   int dividend  = larger (m, n);
   int divisor   = smaller (m, n);
   int remainder = dividend % divisor;
   while (remainder != 0)
   {
      dividend  = divisor;
      divisor   = remainder;
      remainder = dividend % divisor;
   }
   return divisor;
}
```

Euclid

- **Theorem:** Let m and n be positive integers, with $m \leq n$. Let r denote the remainder when n is divided by m . Then the g.c.d. of r and m is equal to the g.c.d. of m and n .

- Complexity in the worst case :
- $\Theta(\log(\min(m, n)))$

Divide and Conquer

- Algorithms that solve a problem by dividing it in similar problems of smaller size.
- Examples: binary search, merge sort, quick sort.

```
template <class otype>
bool merge_arrays (const otype a[], int afirst,
  int alast, const otype b[], int bfirst, int
  blast, otype c[], int cfirst, int clast)
{
  if (clast - cfirst + 1 < (alast - afirst + 1)
  + (blast - bfirst + 1))
    return false;
  else {
    while (afirst <= alast && bfirst <= blast)
      if (a[afirst] <= b[bfirst])
         c[cfirst++] = a[afirst++];
      else
         c[cfirst++] = b[bfirst++];
    while (afirst <= alast)
      c[cfirst++] = a[afirst++];
    while (bfirst <= blast)
      c[cfirst++] = b[bfirst++];
    return true;
}
```



```
template <class otype>
void merge_sort (otype a[], int first, int last,
  otype *aux = NULL)
{
   if (last <= first)  return;
   bool initial_call = !(aux);
   if (initial_call)
      aux = new otype[last - first + 1];
   int mid = (first + last) / 2;
   merge_sort (a, first, mid,  aux);
   merge_sort (a, mid+1, last, aux);
   merge_arrays (a, first, mid, a, mid+1, last,
                 aux, 0, last);
   for (int i=first, j=0; i<=last; ++i, ++j)
      a[i] = aux[j];
   if (initial_call)
      delete [] aux;
}
```

Complexity of the Merge Sort

- Let T(n) be the complexity of the merge sort for a problem of size n.
- All the operations preceding the two recursive calls: $\Theta(1)$
- Calling the merge sort recursively: $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$.
- Merging the two arrays: $\Theta(n)$
- Copying the array from aux into a: $\Theta(n)$.
- $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$
- $T(n) = \Theta(n \log n)$


```
template <class otype>
void quicksort (otype a[], int first, int last)
{
   if (last <= first)  return;
   int i = first + 1,  j = last;
   while (i <= last && a[i] < a[first]) i++;
   while (a[j] > a[first]) --j;
   while (i < j)  {
      swap (a[i], a[j]);
      do
         ++i;
      while (a[i] < a[first]);
      do
         --j;
      while (a[j] > a[first]);
   }
   swap (a[first], a[j]);
   quicksort (a, first, j-1);
   quicksort (a, j+1, last);
}
```

Complexity of the Quicksort

- In general splitting the array in two will be of complexity $\Theta(n)$.
- If j is the position of the pivot and k=j-first, we can write
- $T(n) = T(k) + T(n-k-1) + \Theta(n)$
- **Worst case**: If the array is sorted then j will be equal to the first , so that k=0. Then we have
- $T(n)=T(n-1)+ \Theta(n)$
- $T(n) = \Theta(n^2)$

Complexity of the Quicksort

- **Best case**: $k = \lfloor (n-1)/2 \rfloor$
- $T(n)=T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) + \Theta(n)$
- $T(n)=\Theta(n \log n)$
- Worst case: when the array is already sorted.
- Median of three: the worst case become the best case.