Lecture 6: Divide-and-Conquer

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Types of Algorithms

In COSC242, we will be looking at 3 general types of algorithms:

1. divide-and-conquer algorithms
2. greedy algorithms
3. dynamic programming algorithms

Each type of algorithm can be used to naturally/easily solve a particular type of problem, so it is useful to keep a list of the typical problems that a given type of algorithm is good for.

Today: divide-and-conquer

Divide and conquer

Divide-and-conquer algorithms usually work on sequential data structures of known size. That is arrays.

A *divide-and-conquer* algorithm processes the data structure X in the following recursive way:

1: **if** X is an atom **then**
2: process X directly.
3: **else**
4: divide X into two or more smaller pieces
5: apply the algorithm to each piece "recursively"
6: combine the processed pieces (if necessary).

Binary Search

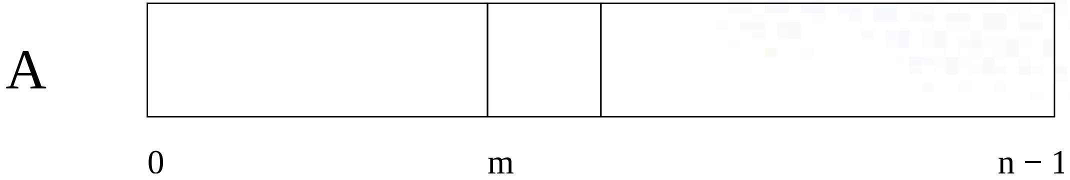
Consider an array A$[0 \,..\, n-1]$ of sorted keys, and suppose we want to locate a target value x. The simplest way to search is sequentially, but sequential search is linear: $O(n)$.

Faster than sequential search is *binary search*.

Find the index $m = \lfloor (n-1)/2 \rfloor$ of the middle element, then compare x with A$[m]$.

There are 4 possibilities.
If x $=$ A$[m]$, we have found x.
If x $<$ A$[m]$, we search the smaller array A$[0 \,..\, m-1]$.
If x $>$ A$[m]$, we search the smaller array A$[m+1 \,..\, n-1]$.
If the breaking down process ever gives an empty array, we've gone too far and can stop.

Binary Search



Binary Search Pseudo-code

```
Binary_search(A, x, low, high):
if low > high then
    report failure and stop
else
    mid ← (low + high) / 2
    if x = A[mid] then
        report success and return mid
    else if x < A[mid] then
        return Binary_search(A, x, low, mid - 1)
    else if x > A[mid] then
        return Binary_search(A, x, mid + 1, high)
```

Binary Search Analysis

To look for x in an array A of length n, you would call
Binary_search(A, x, 0, n - 1)

Binary Search Analysis

To look for x in an array A of length *n*, you would call
Binary_search(A, x, 0, *n* - 1)

Analysis: How many times (in the worst case) can we divide the length n in half? Try it for 8, 16, 32, 64.

Binary Search Analysis

To look for x in an array A of length *n*, you would call
Binary_search(A, x, 0, *n* - 1)

Analysis: How many times (in the worst case) can we divide the length n in half? Try it for 8, 16, 32, 64.

$2^n = 2.2.2.2\ldots2$. How many times can I divide that by 2?

Binary Search Analysis

To look for x in an array A of length *n*, you would call
Binary_search(A, x, 0, *n* - 1)

Analysis: How many times (in the worst case) can we divide the length n in half? Try it for 8, 16, 32, 64.

$2^n = 2.2.2.2 \ldots 2$. How many times can I divide that by 2?

So the time complexity function of the algorithm is $f =$ ____________.

Merge

Suppose we have two arrays X and Y each in sorted order and want to build array Z containing all the keys of X and Y in sorted order. Suppose length(X) $= l$, length(Y) $= m$:

initialise i, j, k to 0 (i, j, k index the arrays X, Y, Z)
while $i < l$ and $j < m$ **do**
    if X[i] $<$ Y[j] **then**
        Z[k] $\leftarrow$ X[i]
        $i \leftarrow i + 1$
    else if X[i] $\geq$ Y[j] **then**
        Z[k] $\leftarrow$ Y[j]
        $j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i \geq l$ then copy the end of Y to the end of Z
else copy the end of X to the end of Z

Merge Analysis

How many times through the merge while loop?

One thing we notice is that i, j, k all start at 0. Each time through the loop k is incremented and either i or j is incremented. Neither i, j or k ever gets smaller.

At the end of the loop:

- either $(i = l$ and $j < m)$ or $(j = m$ and $i < l)$,
- k is just the sum of i and j, so $k < l + m$.

Since k is incremented every time through the loop, the number of times through the loop is less than $l + m$. Then, whichever array has not been fully scanned is copied onto the end of Z.

So the number of operations is some constant times $l + m$. Let $n = l + m$, so Merge is $O(n)$ (actually $\Theta(n)$).

Mergesort

To mergesort an array A$[0 \mathrel{..} n-1]$ of keys, we repeatedly split A, and after getting to the bottom we rebuild by merging the pieces. To identify the pieces that must be split or patched together, we use indices *left* and *right*.

Mergesort(A, *left*, *right*) sorts the keys in A[*left* .. *right*].

```
if left ≥ right then
    stop since A[left .. right] is sorted
else
    mid ← (left + right) / 2
    Mergesort(A, left, mid)
    Mergesort(A, mid + 1, right)
    Merge subarrays A[left .. mid] and A[mid + 1 .. right]
```

Mergesort Analysis

Let's call the time complexity function T.

If $n = 1$, then mergesort takes constant time, so $T(1) = 1$.

Otherwise the number of operations needed to mergesort n keys is equal to the number of operations needed to do two mergesorts of size $n/2$ (the recursive calls) plus the merge needed to patch the two sorted arrays of length $n/2$ together (which is linear).

So $T(n) = 2T(n/2) + n$.

We'll see how to analyse these sorts of complexity functions in the next lecture.