Recurrence Relations for Complexities

Textbook Reading:

Section 3.3, pp. 97-102.

Section 3.6.4, pp. 119-121

Worst-Case Complexity Binary Search: Version with Test for Equality

Assume for convenience that $n = 2^k - 1$, so that $k = \log_2(n + 1)$.

Since *BinarySearch* makes two comparisons (= and <) and then performs a recursive call with a sublist of size $(n - 1)/2 = 2^{k-1} - 1$, we have the recurrence relation:

$$W(n) = W((n - 1)/2) + 2, \qquad \text{Initial Condition: } W(0) = 0 \text{ (empty list).}$$

Equivalently, setting $t(k) = W(n) = W(2^k - 1)$, we have the recurrence relation

$$t(k) = t(k\text{-}1) + 2, \qquad \text{Initial Condition: } t(0) = 0.$$

Applying the technique of **repeated substitution**, we obtain:

$$\begin{aligned} t(k) &= t(k-1) + 2 = (t(k-2) + 2) + 2 \\ &= t(k-2) + 4 = (t(k-3) + 2) + 4 \\ &= t(k-3) + 6 = (t(k-4) + 2) + 6 \\ &\vdots \qquad\qquad \vdots \\ &= t(0) + 2k = 2k = 2\log_2(n+1) \sim 2\log_2 n \end{aligned}$$

PSN. Obtain recurrence relation for $W(n)$ and solve for the version of Binary Search where no equality check is done until list has size 1. For convenience assume $n = 2^k$.

Mergesort vs. Quicksort

Mergesort and Quicksort are two important and widely used sorting algorithms that are described in detail in the text. They are both based on the **divide-and-conquer** design paradigm. However, in Mergesort most of the work is done in the combine stage, whereas in most of the work in Quicksort is done in the divide stage.

Mergesort

```
void Mergesort(DataType L[MAX], int low, int high)
 {   //BOOTSTRAP CONDITION -- A LIST OF SIZE AT MOST ONE
     //                       IS A PRIORI SORTED
     if (low < high)
     {
        mid = (low + high)/2;
        //RECURSIVELY SORT FIRST HALF
        Mergesort(L, low, mid);

        //RECURSIVELY SORT SECOND HALF
        Mergesort(L, mid+1, high);

        //MERGE TWO HALVES
        Merge(L, low, mid, high);
      }//END IF
  }//END Mergesort()
```

Design and Analysis of Merge

Merge() sorts the list L with the **precondition** that the first and second halves of L and the are **both sorted**.

It involves a allocating a temporary array Temp and performing a scan of both half sublists. An index pointer p is utilized for the left sublist (from low to mid) and an index pointer q for the right sublist (from mid+1 to high). Starting with i = p = low and q = mid+1, the following operation is perform until either p > mid or q > high

```
if (L[p] < L[q])
    { Temp[i] = L[p]; p = p+1; i = i + 1; }
else
    { Temp[i] = L[q]; q = q+1; i = i + 1; }
```

If p > mid the rest of the right sublist is dumped at the end of Temp. Otherwise, if q > high, the rest of the left sublist is dumped at the end of Temp. The array Temp is then copied back into L. Because of the use of this temp array Mergesort is **not** an **in-place** sorting algorithm.

It is easily verified that in the **best-case** Merge performs ***n*/2 comparisons** and in the **worst-case *n* – 1 comparisons**.

Recurrence Relation for Worst-Case Complexity of Mergesort

We choose comparison as the basic operation. For convenience we assume that the input size n is a power of 2, i.e., $n = 2^k$, $k \geq 0$. The worse case complexity $W(n)$ of Mergesort for a list of size n satisfies

$W(n) = W(n/2)$ (the number of comparisons to recursively sort first half)
$+ W(n/2)$ (the number of comparisons to recursively sort first half)
$+ n - 1$ (most comparisons to merge left and right sublists)

Thus, observing the Mergesort requires no comparisons to sort a list of size 1, and for convenience approximating n with $n - 1$, we have the following recurrence relation:

$$W(n) = 2W(n/2) + n, \text{ initial condition } W(1) = 0.$$

Solving recurrence relation for $W(n)$

$W(n) = 2W(n/2) + n$, initial condition $W(1) = 0$.

To solve this recurrence relation we use **repeated substitution**:

$$W(n) = 2W(n/2) + n = 2(2W(n/2^2) + n/2) + n$$

$$= 2^2W(n/2^2) + 2n = 2^2(2W(n/2^3) + n/2^2) + 2n$$

$$= 2^3W(n/2^3) + 3n$$

$$\vdots \qquad\qquad \vdots$$

$$= 2^kW(n/2^k) + kn$$

Since $n = 2^k$ we have that

$$W(n) = 2^kW(1) + kn \ .$$

But by the initial condition, W(1) = 0 and since $n = 2^k$ we have that $k = \log_2 n$. Therefore,

$$\boldsymbol{W(n) = n\log_2 n} \ .$$

PSN. Obtain recurrence relation for the Best-Case Complexity $B(n)$ of Mergesort and solve using repeated substitution.

Average Complexity of Mergesort

For any probability distribution on the sample space of input lists of size n, we have

$$B(n) \leq A(n) \leq W(n).$$

Thus,

$$\tfrac{1}{2}\, n\log_2 n \leq A(n) \leq n\log_2 n\ ,$$

so that

$$A(n) \in \Theta(n \log n)$$

Quicksort

- Choose first element of the list to be the pivot element, i.e., *L*[0] is pivot element.
- Call a subroutine Partition that rearranges the list so that every element to the left of (having a lower index than) the pivot element is smaller than or equal to the pivot element and every element to the right of (having a higher index than) the pivot element is greater than or equal to the pivot element.
- This rearrangement would automatically mean that the pivot element has been placed in its proper (sorted) position in the list.
- We then sort the list by recursive sorting left and right sublists.

C++ Code for Quicksort

```
void Quicksort(DataType L[MAX], int low, int high) {
    // BOOTSTRAP CONDITION -- A LIST OF SIZE AT MOST ONE IS A PRIORI
    // SORTED
     if (low < high) {
        // REARRANGE LIST SO ELEMENTS TO THE LEFT OF PIVOT ELEMENT
        // ARE SMALLER THAN PIVOT ELEMENT AND ELEMENTS TO THE RIGHT
        // ARE GREATER
        // RETURN INDEX WHERE PIVOT ELEMENT LIES AFTER REARRANGEMENT
        Partition(L, low, high, PivotIndex);
        //RECURSIVELY SORT FIRST HALF
        Quicksort(L, low, PivotIndex - 1);
        //RECURSIVELY SORT SECOND HALF
        Quicksort(List, PivotIndex + 1, high);
      }//END IF
  }//END Quicksort()
```

Design of Partition

Partition rearranges the list so elements to the left of the pivot element are smaller than (or equal to) the pivot element and elements to the right are greater than (or equal to) the pivot element. This forces the pivot element to be in its sorted position in the list. Partition returns the index of this position.

There are numerous strategies for *Partition* in the literature. Here is one strategy discussed in textbook based on utilizing two moving variables ***moveright*** and ***moveleft***, which contain the indices of elements in *L* and are initialized to *low* + 1 and *high*, respectively.

while *moveright* < *moveleft* **do**
    moveright moves to the right (one index at a time) until it assumes the index of a list element not smaller than *x*, then it stops.
    moveleft moves to the left (one index at a time) until it assumes the index of a list element not larger than *x*, then it stops.
    if *moveright* < *moveleft* **then**
        interchange *L*[*moveright*] and *L*[*moveleft*]
    endif
endwhile

To guarantee that *moveright* actually finds an element not smaller than *x*, we assume that *L*[*high* + 1] is defined and is not smaller than *L*[*low*]. To guarantee that this holds initially, we introducing a sentinel value *L*[*n*] = +∞.

Pseudocode for Partition

```
procedure Partition(L[0:n – 1], low, high,position)
Input: L[0:n – 1] (an array of n list elements)
        low, high (indices of L[0:n – 1])
       //L[high+1] is assumed defined and ≥ L[low]
Output: a rearranged sublist L[low:high] such that L[i] ≤ L[position]
                low ≤ i ≤ position, L[i] ≥ L[position], position ≤ i ≤ high
                where, originally, L[low] = L[position]
         position (the position of a proper placement of the original element
                          L[low] in the list L[low:high])
        moveright ← low
        moveleft ← high + 1
        x ← L[low]
        while moveright < moveleft do
                repeat
                      moveright ← moveright + 1
                until L[moveright] ≥ x
                repeat
                       moveleft ← moveleft – 1
                until L[moveleft] ≤ x
                if moveright < moveleft then
                        interchange(L[moveright],L[moveleft])
                endif
         endwhile
        position ← moveleft
        L[low] ← L[position]
        L[position] ← x
end Partition
```

Action of Partition for a Sample List

L[0:6] = 23 9 23 52 15 19 47

Initially:	index	0	1	2	3	4	5	6	7
	list element	23	9	23	52	15	19	47	+∞
		mr							*ml*
		Rearrange Step							
				19			23		
1st iteration:		23	9	23	52	15	19	47	+∞
				mr			*ml*		
					15	52			
2nd iteration:		23	9	19	52	15	23	47	+∞
					mr	*ml*			
3rd iteration:		23	9	19	15	52	23	47	+∞
					ml	*mr*			
		Place Step							
		15			23				
After completion of *Partition:*		23	9	19	15	52	23	47	+∞
					↑				
					PivotIndex = 3				

Worst-Case Complexity of Quicksort

Unfortunately, the worst-case of Quicksort is realized when the list is already **sorted**! *Partition* requires $n + 1$ comparisons to rearrange a list of size n, we have:

$$W(n) = W(n-1) + n + 1, \quad \text{Initial Condition: } W(1) = 0.$$

Note that the bootstrap condition may vary slightly depending on how Quicksort is written. In the textbook the initial condition is W(2) = 3 and the implementation of partition requires n + 1 comparisons, but this makes only a difference of a small difference in the result. Solving the above recurrence relation using **repeated substitution**, we obtain:

$W(n) = W(n-1) + n+1$

$= (W(n-2) + n) + n+1 =$

$= (W(n-3) + n - 1) + n + n+1$

⋮ ⋮

$= (W(1) + 3) + 4 + 5 + \ldots + n + n+1$

$= (1 + 2 + \ldots + n+1) - 3 = (n+1)(n+2)/2 - 3 \sim \boldsymbol{\frac{n^2}{2}}$.

Average Complexity of Quicksort

τ_1 = number of comparisons performed by *Partition.*

τ_2 = number of comparisons performed by the two recursive calls.

$$\tau = \tau_1 + \tau_2.$$

$$\mathrm{A}(n) = E[\tau] = E[\tau_1] + E[\tau_2] = n + 1 + E[\tau_2].$$

Obtaining recurrence relation for $\mathrm{E}[\tau_2]$

Let *Y* be the random variable that maps an input list *L*[0:*n*-1] onto the index the pivot element ends up at, i.e., index where the list is split.

$$E[\tau_2 \mid Y = i] = \mathrm{A}(i) + \mathrm{A}(n-i-1), \quad i = 0, \ldots, n-1.$$

Obtaining recurrence relation for $\mathrm{E}[\tau_2]$ cont'd

$$\mathrm{A}(n) = (n+1) + \sum_{i=0}^{n-1} E[\tau_2 \mid Y = i]P(Y = i)$$

$$= (n+1) + \sum_{i=0}^{n-1} (\mathrm{A}(i) + \mathrm{A}(n-i-1))\left(\frac{1}{n}\right)$$

$$= (n+1) + \frac{2}{n}(\mathrm{A}(0) + \mathrm{A}(1) + \cdots + \mathrm{A}(n-1)),$$

init. cond. $\mathrm{A}(0) = \mathrm{A}(1) = 0.$

Simplifying **full-history** recurrence relation from previous slide

$$nA(n) = n(n+1) + 2(A(0) + A(1) + \cdots + A(n-2) + A(n-1)).$$

now substituting $n - 1$ for n we obtain

$$(n-1)A(n-1) = n(n-1) + 2(A(0) + A(1) + \cdots + A(n-2)).$$

Subtracting two equations yields:

$$nA(n) - (n-1)A(n-1) = 2n + 2A(n-1).$$

Or equivalently

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2}{n+1}.$$

Letting $t(n) = A(n)/(n + 1)$, we obtain recurrence relation:

$$t(n) = t(n-1) + \frac{2}{n+1}.$$

with initial condition $t(1) = 0$.

$$t(n) = 2\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{(n+1)}\right)$$

$$= 2H(n+1) - 3,$$

where $H(n)$ is the harmonic series. Thus,

$$t(n) \sim 2 \ln n,$$

so that the average complexity $A(n)$ of *QuickSort* satisfies:

$$A(n) \sim 2 n \ln n.$$

Mergesort vs. Quicksort Pros and Cons

- A drawback with Quicksort is its quadratic worst-case complexity when the input list is already sorted. However, it achieves a good average complexity of $2n \ln n$ for a uniform distribution.
- Mergesort is a very fast sorting algorithm achieving worst-case complexity $n\log_2 n$.
- However, Quicksort has the advantage that it is **in-place**, i.e., only a constant amount of extra space is needed, whereas Mergesort has the drawback that it is not in-place, requiring allocating a temporary array the size of the input list.

Knock-knock joke with recursion

Knock knock.

Who's there?

Déja.

Déja who?

Knock knock.