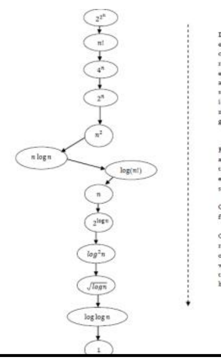
Analysis of sorting and searching algorithms

Relationship between different rates of growth



Binary search

- If we consider searching of a word in a dictionary, in general we directly go some approximate page [generally middle page] start searching from that point.
- If that we are searching is same then we are done with the search.
- If the page is before the selected pages then apply the same process for the first half otherwise apply the same process to the second half.

Analysis of BINARY-SEARCH

Alg.: BINARY-SEARCH (A, lo, hi, x)

```
if (lo > hi)                                  ← constant time: c1
    return FALSE
mid ← ⌊(lo+hi)/2⌋                             ← constant time: c2
if x = A[mid]                                 ← constant time: c3
    return TRUE
if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)           ← same problem of size n/2
if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)           ← same problem of size n/2
```

- $T(n) = c + T(n/2)$
  - $T(n)$ – running time for an array of size n

4

The Iteration Method

$$T(n) = c + T(n/2)$$

$T(n) = c + T(n/2)$ $\qquad$ $T(n/2) = c + T(n/4)$

$\quad = c + c + T(n/4)$ $\qquad$ $T(n/4) = c + T(n/8)$

$\quad = c + c + c + T(n/8)$

Assume $n = 2^k \rightarrow k = \lg n$

$T(n) = \underbrace{c + c + \ldots + c}_{k \text{ times}} + T(1)$

$\quad = ck + T(1)$

$\quad = c\lg n + T(1)$

$\quad = \Theta(\lg n)$

5

Example: Binary Search

$$T(n) = c + T(n/2)$$

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le d \lg(n/2)$
- Proof of induction goal:

$T(n) = T(n/2) + c \le d \lg(n/2) + c$

$\qquad \le d \lg n - d + c$

$T(n) \le d \lg n$

$(d \lg n - d + c \le d \lg n)$ $\quad$ if: $-d + c \le 0,\ d \ge c$

Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - $T(n)$ – running time on a problem of size n
  - **Divide** the problem into a subproblems, each of size n/b: takes $D(n)$
  - **Conquer** (solve) the subproblems $aT(n/b)$
  - **Combine** the solutions $C(n)$

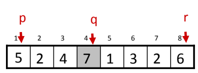
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge Sort Approach

- To sort an array $A[p\,.\,.\,r]$
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort

Alg.: MERGE-SORT(A, p, r)

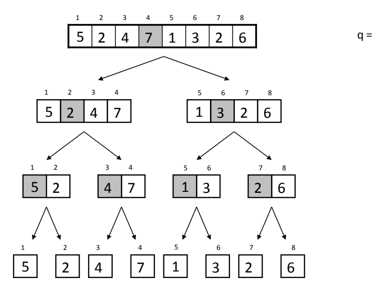


if p < r	▷ Check for base case
then q ← ⌊(p + r)/2⌋	▷ Divide
MERGE-SORT(A, p, q)	▷ Conquer
MERGE-SORT(A, q + 1, r)	▷ Conquer
MERGE(A, p, q, r)	▷ Combine

- Initial call: MERGE-SORT(A, 1, n)

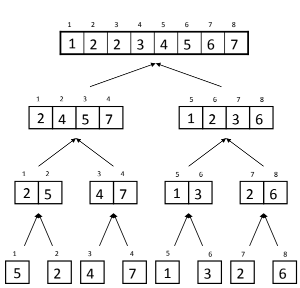
Example – n Power of 2

Divide



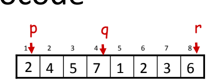
Example – n Power of 2

Conquer and Merge

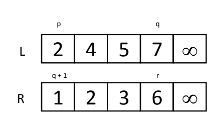


Merge - Pseudocode

Alg.: MERGE(A, p, q, r)



1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\,.\,.\,n_1 + 1]$ and the next n_2 elements into $R[1\,.\,.\,n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** k ← p **to** r
6. **do if** L[i] ≤ R[j]
7. **then** A[k] ← L[i]
8. i ← i + 1
9. **else** A[k] ← R[j]
10. j ← j + 1



MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n\lg n)$