Code: AGRJB

CS 590: Algorithms

Sorting and Order Statistics I: Heapsort / Quicksort

Outline

4.1. Heapsort
- 4.1.1. Tree & Binary Tree
- 4.1.2. Heap
- 4.1.3. Heapsort
- 4.1.4. Priority Queues

4.2. Quick Sort
- 4.2.1. Description of Quicksort
- 4.2.2. Performance of Quicksort
- 4.2.3. Randomized Quicksort
- 4.2.4. Analysis of Quicksort

4.2.1. Description of Quicksort

Quicksort has a similar approach as the merge sort.

- It is another sorting algorithm based on the divide-and-conquer process.
- Divide the partition $A[p \ldots r]$ into two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$.
- Note: Elements will be arranged before the split!
  - each element in $A[p \ldots q-1] \leq A[q]$ and
  - $A[q] \leq A[q+1 \ldots r]$.
- Conquer: We sort the two subarrays by recursive calls to QUICKSORT.
- Combine: No need to combine the subarrays, because they are sorted in place.

4.2.1. Description of Quicksort

```
QUICKSORT(A, p, r) //initial call (A,1,n)
if (p < r) then
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)
```

4.2.1. Description of Quicksort

```
PARTITION(A, p, r)
x = A[r] //the last element (call it a pivot)
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1 //returns q the index to split
```

4.2.1. Description of Quicksort

- PARTITION( ) rearranges the subarray in place before the split.
- PARTITION( ) always selects the last element $A[r]$ in the subarray $A[p \dots r]$ as the pivot (the element around which to partition).
- As the procedure executes, the array is partitioned into four regions, which may be empty.

4.2.1. Description of Quicksort

A = [8, 1, 6, 4, 0, 3, 9, 5] becomes [1, 4, 0, 3, 5, 8, 9, 6] when PARTITION(A, 1, n) is called.

```
PARTITION(A, p, r)

x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

$x = 5$ [A, 1, 8]

$i = 0$

1st $p = 1, j = 1$, $r-1 = 7$

$A[1] = 8$, $8 > 5$

$i = 0$

2nd $j = 2$

$A[2] = 1$, $1 < 5$ ✓

$i = 1$

[1, 8, ... 5]

4.2.1. Description of Quicksort

A = [8, 1, 6, 4, 0, 3, 9, 5] becomes [1, 4, 0, 3, 5, 8, 9, 6] when PARTITION(A, 1, n) is called.

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

3rd $j = 3$

$A[3] = 6 \quad 5 \not> 6$

4th $j = 4$

$A[4] = 4 \quad 5 > 4$

$i = 2$, [1, 4, 8, 6, 0, 3, 9, 5]

5th $j = 5$

$A[5] = 0$, $\quad 5 > 0$

$i = 3$

[1, 4, 0, 6, 8, 3, 9, 5]

4.2.1. Description of Quicksort

A = [8, 1, 6, 4, 0, 3, 9, 5] becomes [1, 4, 0, 3, 5, 8, 9, 6] when PARTITION(A, 1, n) is called.

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

6th $j = 6$

$A[6] = 3$ $\quad$ $5 > 5$

$i = 4$ $\quad$ $A[4] = 6$

[1, 4, 0, 3, 8, 6, 9, 5]

$A[i+1] = A[5] = 8$

swap w/ $A[8] = 5$

[1, 4, 0, 3 | 5 | 6, 9, 8]

$q = 5$,

4.2.1. Description of Quicksort

A = [8, 1, 6, 4, 0, 3, 9, 5] becomes [1, 4, 0, 3, 5, 8, 9, 6] when PARTITION(A, 1, n) is called.

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for (p <= j <= r-1)
    if (A[j]<= x):
        i = i + 1
        swap A[i] and A[j]
swap A[i+1] and A[r]
return i+1
```

QS(A, 1, 4)

[1, 0, 3, 4]

QS(A, 1, 3)

[0, 1, 3]

QS(A, 6, 8]

[6, 8, 9]

QS(A, 7, 8]

[8, 9]

2nd r.

[1, 0, 3, 4, 5, 6, 8, 9]

3rd r.

[0, 1, 3, 4, 5, 6, 8, 9]

4.2.2. Performance of Quicksort

Loop invariant:

1. Array elements are arranged before a call.
   - All entries in A[p…i] are < x (A[r]).
   - All entries in A[i+1,…r-1] are > x.
2. Elements move to the positions to maintain the arrangement rules. (not sorted)
   - The pivot stays at the end of the array.
3. All elements are positioned, and arrangement rules are satisfied.

Note: The additional region $A[j \ldots r - 1]$ consists of elements that have not yet been processed. We do not yet know how they compare to the pivot element.

4.2.2. Performance of Quicksort

The running time of **QUICKSORT** depends on the partitioning of the subarrays.

- **QUICKSORT** is as fast as **MERGE-SORT** if the partitioned subarrays are balanced (even-sized).
- **QUICKSORT** is as slow as **INSERTIONSORT** if the partitioned subarrays are unbalanced (uneven-sized).

4.2.2. Performance of Quicksort

Worst case: Subarrays completely unbalanced.

- Have 0 elements in one subarray and $n-1$ elements in the other subarray.
- The recurrence running time:

$$T(n) = T(n-1) + T(0) + \Theta(n) = O(n^2)$$

- The running time is like INSERTION-SORT.

$T(n) = dn^2 \quad + \underbrace{cn}$

$T(n) \leq dn^2$

guess

new guess

$T(n) \leq dn^2 - d'n$

QS worst Case.

$$T(n) = T(n-1) + \Theta(n) \qquad \text{Guess } T(n) = O(n^2)$$

let q be $0 \le q \le n-1$

$$= T(q) + T(n-q-1) + \Theta(n) \qquad (1)$$

let $T(q) \le cq^2$

$$= (cq^2 + c[n-q-1]^2) + \Theta(n)$$

$$\rightarrow \quad q^2 + (n-q-1)^2 \le \underbrace{(n-1)^2}_{\text{worst}} = n^2 - 2n + 1$$

rewrite (1) w/ $(n-1)^2$

$$T(n) \le cn^2 - c(2n-1) + \Theta(n)$$

let $\Theta(n) = dn$ $d>0, c>0$

$$\le cn^2 - c(2n-1) + dn$$

$$= O(n^2)$$

4.2.2. Performance of Quicksort

Best case: Subarrays are always completely balanced.

- Each subarray has $\leq n/2$ elements.
- The recurrence running time:

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\
&= \Theta(n \lg n)
\end{aligned}
$$

4.2.2. Performance of Quicksort

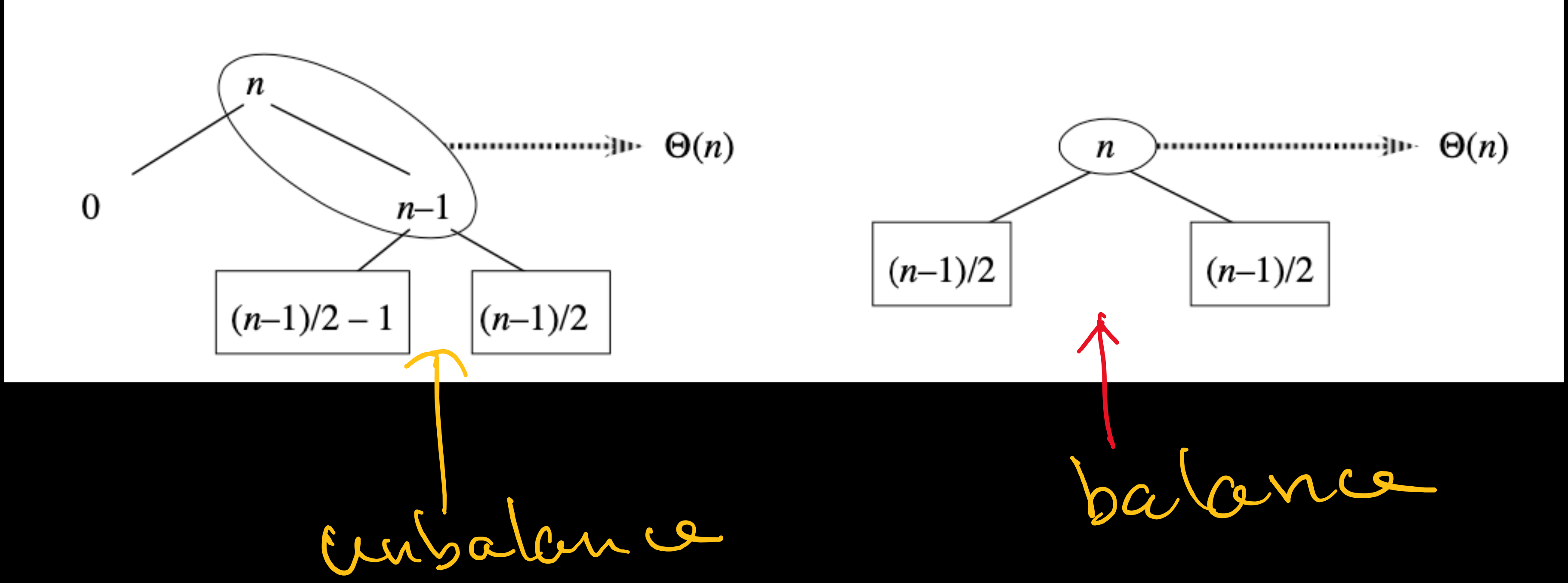
Balanced partitioning:

- Let's assume that PATITION always produces a 9 to 1 split.
- Then, the recurrence is

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) = O(n \lg n)$$

- As long as it's a constant, the base of the log does not matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth $\Theta(\lg n)$.

4.2.2. Performance of Quicksort



4.2.2. Performance of Quicksort

Average case:

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of "good" and "bad" splits throughout the recursion tree.
- Assuming that levels alternate between best- and worst-case splits does not affect the asymptotic running time.
- The bad split only adds to the constant hidden in Θ notation.
- The same number of subarrays to sort, but twice as much work is needed.
- Both splits result in $\Theta(n \lg n)$ time, though the constant on the bad split is higher.

4.2.3 Randomized Quicksort

- We assumed so far that all input permutations are equally likely, which is not always the case.
- We introduce randomization to improve the quicksort algorithm.
- One option would be to use a random permutation of the input array.
- We use random sampling instead, which is picking one element at random.
- Instead of using $A[r]$ as the pivot element we randomly pick an element from the subarray.

4.2.3 Randomized Quicksort

```
RANDOMIZED-PARTITION(A, p, r)
i = random(p, r)
exchange A[r] and A[i]
return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r:
    q = RANDOMMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q-1)
    RANDOMIZED-QUICKSORT(A, q+1, r)
```

4.2.4 Analysis of Quicksort

Work on board