COT 6405 Introduction to Theory of Algorithms

Midterm II review

Overview

- Exam time: Nov 5th 3:30pm to 4:45pm
- Exam location: ENB 118 (regular session) and ENB 313 (online session)
- Coverage:
  - Lectures 8, 9, 10, 11, and midterm II review

Quicksort

- Sorts “in place”
  - Only a constant number of elements stored outside the sorted array
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
- So why people use it instead of merge sort?
  - Merge sort does not sort “in place”

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}  // what is the initial call?
```

Partition

- Clearly, all the actions take place in the **`partition()`** function
  - Rearranges the subarray “in place”
  - End result:
    - Two subarrays
    - All values in 1st subarray < all values in 2nd
  - Returns the index of the “pivot” element separating the two subarrays
- How do we implement this function?

Partition array $A[p..r]$

```
PARTITION(A, p, r)
    x ← A[r]          // select the pivot
    i ← p – 1
    for j ← p to r – 1
        if A[ j ] ≤ x
            i ← i + 1
            exchange A[i ] ↔ A[ j ]
    // move the pivot between the two subarraies
    exchange A[i + 1] ↔ A[r]
    // return the pivot
    return i + 1
```

What is the running time of `partition()`?

Performance of quicksort

- The running time of quicksort depends on the partitioning of the subarrays:
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.
  - If the subarrays are balanced, then quicksort can run as fast as mergesort. The following inequality is used for the average case analysis of quicksort

$$\sum_{k=1}^{k=n-1} k\lg k \le \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$$

Improving quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on an already-sorted input
- How to avoid this?
- Two solutions
  - Randomize the input array
  - Pick a random pivot element
- How will these solve the problem?
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Randomized version of quicksort

- We add randomization to quicksort.
  - We could randomly permute the input array: very costly
  - Instead, we use **random sampling** to pick one element at random as the pivot
    - Don't always use $A[r]$ as the pivot.

Analysis of quicksort

- We analyzed
  - the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT
  - the expected (average-case) running time of QUICKSORT and RANDOMIZED-QUICKSORT

Worst-case analysis

- We saw a worst-case split (0:n-1) at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which,
  - Intuitively, is the worst-case running time
- We have prove this assertion

Average case analysis

- The dominant cost of the algorithm is partitioning.
- What is the maximum number of calls to the function PARTITION?
  - PARTITION is called at most n times.

Average case analysis (cont'd)

Lemma 7.1: Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

The amount of work of each call to PARTITION is a constant plus the number of comparisons performed in its for loop
The expectation of X is the average case running time, and
E(x) = O(nlgn)

Decision trees

- We can view comparison sorts abstractly in terms of decision trees
  - A decision tree is a full binary tree that represents the comparisons between elements
  - Each node on the tree is a comparison of *i*:*j*, i.e., a_i v.s. a_j

Theorem 8.1

- Any comparison sort algorithm requires $\Omega(nlgn)$ comparisons in the worst case
- How to prove?
  - By proving that the height of the decision tree is $\Omega(nlgn)$
  - What's the # of leaves of a decision tree? l = ?
  - What's the maximum # of leaves of a general binary tree? l_{max} = ?

Proof

- l_{min} = n! and $l_{max} = 2^h$
- Clearly, the minimum # of leaves l_{min} is less than or equal to the maximum # of leaves, l_{max}
- So we have: $n! \leq 2^h$
- Taking logarithms: $\lg(n!) \leq h$

Proof (cont'd)

- Stirling's approximation tells us:

$$n! > \left(\frac{n}{e}\right)^n$$

- Thus, $h \geq \lg(n!)$

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Sorting in linear time

- Counting sort
  - No direct comparisons between elements!
  - Depends on assumption about the numbers being sorted
    - We assume numbers are in the range [0.. k]
  - The algorithm is NOT "in place"
    - Input: A[1..n], where A[j] $\in$ {0, 2, 3, ..., k}
    - Output: B[1..n], sorted
    - Auxiliary counter storage: Array C[0..k]
    - notice: A[], B[], and C[] → <u>not sorting in place</u>

Counting sort

```
CountingSort(A, B, k)
    for i= 0 to k   // counter initialization
        C[i]= 0;
    for j= 1 to A.length
        C[A[j]] += 1;
    for i= 1 to k   // aggregate counters
        C[i] = C[i] + C[i-1];
    for j= A.length downto 1 //move results
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Counting sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$ → $k < c\,n$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$! Contradiction?
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is stable
    - The elements with the same value is in the same order as the original
    - index $i < j$, $a_i = a_j$ → new index $i' < j'$

Counting Sort

- Why don't we always use counting sort?
- Because it depends on range *k* of elements
- Could we use counting sort to sort 32 bit integers? Why or why not?
- Answer: no, *k* too large (2^{32} = 4,294,967,296)
  - We need huge arrays, e.g., C[4,294,967,296]?
  - k >> n → O(n+k) = O(k)

Least significant digit (LSD) Radix Sort

- Key idea: sort the least significant digit first
- Assume we have d-digit numbers in A

```
RadixSort(A, d)
        for i= 1 to d
            StableSort(A) on digit i
```

Radix Sort

- What sort will we use to sort on digits?
- <u>Counting sort</u> is obvious choice:
  - Sort n numbers on digits that range from 1..k
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so the total time $O(dn+dk)$
  - When d is constant and $k= O(n)$, takes $O(n)$ time

How to break words into digits?

- We have n word
- Each word is of b bits
- We break each word into r-bit digits, $d = \lceil b/r \rceil$
- Using counting sort, $k = 2^r - 1$
- E.g., 32-bit word, we break into 8-bit digits
  - $d = \lceil 32/8 \rceil = 4$, $k = 2^8 - 1 = 255$
- $T(n) = \Theta(\ d*(n+k)\ ) = \Theta(b/r * (n + 2^r))$

How to choose r?

How to choose r? Balance b/r and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta(bn/\lg n)$.

Still in O(n)

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

Bucket Sort

- Assumes the input is generated by a random process that distributes elements uniformly over $[0, 1)$.
- ***Idea:***
  - Divide $[0, 1)$ into n equal-sized *buckets*.
  - Distribute the n input values into the buckets.
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one.

Bucket Sort (cont’d)

- Input:
  - $A[1\ .\ .\ n]$, where $0 \le A[i\ ] < 1$ for all i .
- Auxiliary array:
  - $B[0\ .\ .\ n - 1]$ of linked lists, each list initially empty.

Bucket sort Implementation

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$

  do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order

return the concatenated lists

Easily compute the bucket index $\lfloor n \cdot A[\mathrm{i}] \rfloor$

Formal Analysis

- Define a random variable:

  n_i = the number of elements placed in bucket $B[i]$
- Because insertion sort runs in quadratic time, bucket sort time is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Formal Analysis (Cont'd)

Take expectations of both sides:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \qquad \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) \qquad (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}
$$

n_i = the number of elements placed in bucket $B[i]$

n_i = the number of elements placed in bucket $B[i\,]$

Claim

$\mathrm{E}\left[n_i^2\right] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

$$X_{i,j} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}.$$

$$= \begin{cases} 1 \text{ if } A[j] \text{ falls in bucket } i \\ 0 \text{ if } A[j] \text{ doesn't fall in bucket } i \end{cases}$$

The Claim

Then

$$
\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}\left[X_{ij}X_{ik}\right] \quad \text{(linearity of expectation)}
\end{aligned}
$$

$(x_1+x_2+x_3)(x_1+x_2+x_3)$

$= x_1^2 + x_1x_2 + x_1x_3$

$+ x_2^2 + x_1x_2 + x_2x_3$

$+ x_3^2 + x_1x_3 + x_2x_3$

$$
\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}
$$

Analysis

$\mathrm{E}[X_{ij}X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$
\begin{aligned}
\Rightarrow \mathrm{E}[X_{ij}X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2}
\end{aligned}
$$

Therefore:

$$
\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{n^2}
$$

Analysis (Cont'd)

$$
\begin{aligned}
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n} \qquad \blacksquare \text{ (claim)}
\end{aligned}
$$

Therefore:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

Order statistic

- The *i*-th order statistic in a set of *n* elements is the *i*-th smallest element
  - The *minimum* is thus the 1st order statistic
  - The *maximum* is the *n*-th order statistic
  - The *median* is the *n*/2 order statistic
    - If *n* is even, we have 2 medians: lower median $n/2$ and upper median n/2+1
    - By our convention, "median" normally refers to the lower median

Can we reduce the cost?

- Can we find the minimum and maximum with less than twice the cost, 2(n-1) ?
- Yes: Walk through elements by pairs
  - Compare each element in pair to the other
  - Compare the larger one to maximum, the smaller one to minimum
- Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding order statistics: The Selection Problem

- A more interesting problem is the selection problem
  - finding the *i*-th smallest element of a set
- A naïve way is to sort the set
  - Running time takes O(nlgn)
- We will study a practical randomized algorithm with O(n) expected running time
- We will then study an algorithm with O(n) worst-case running time

Randomized Selection

- Key idea: use partition() from Quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a randomized partition

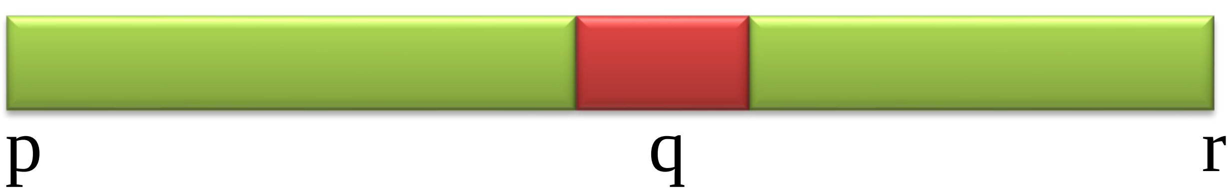
q = RANDOMIZED-PARTITION*(A, p, r)*

RANDOMIZED-PARTITION*(A, p, r)*

$i \leftarrow$ RANDOM*(p, r)*

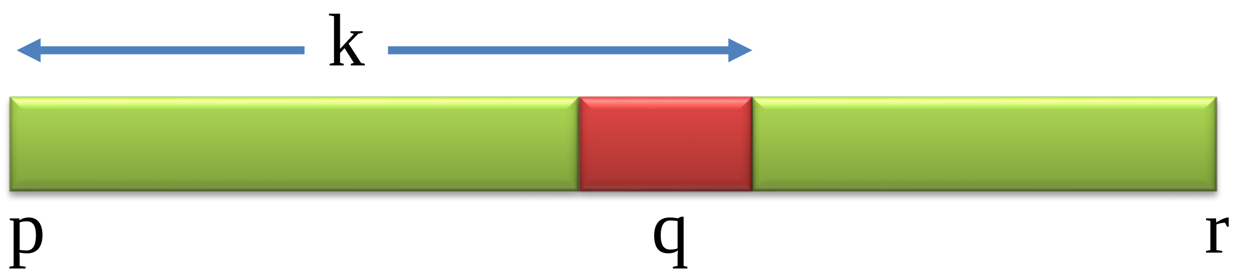
exchange $A[r] \leftrightarrow A[i]$

return PARTITION*(A, p, r)*



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A,q+1,r, i-k);
```



Analyzing Randomized-Select()

- Worst case: partition always 0:n-1
  - $T(n) = T(n-1) + O(n) = O(n^2)$
  - No better than sorting!
- "Best" case: suppose a 9:1 partition
  - $T(n) = T(9n/10) + O(n) = O(n)$ (why?)
  - Master Theorem, case 3
  - Better than sorting!

Worst-Case Linear-Time Selection

- Randomized selection algorithm works well in practice
- We now examine a selection algorithm whose running time is $O(n)$ in the worst case.

Worst-Case Linear-Time Selection

- The worst-case happens when a 0:n-1 split is generated. Thus, to achieve O(n) running time, we *guarantee* a good split upon partitioning the array.
- Basic idea:
  – Generate a good partitioning element

Selection algorithm

1. Divide *n* elements into groups of 5
2. Find median of each group (How? How long?)
3. Use Select() recursively to find median *x* of the $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

   if (i < k) **then**

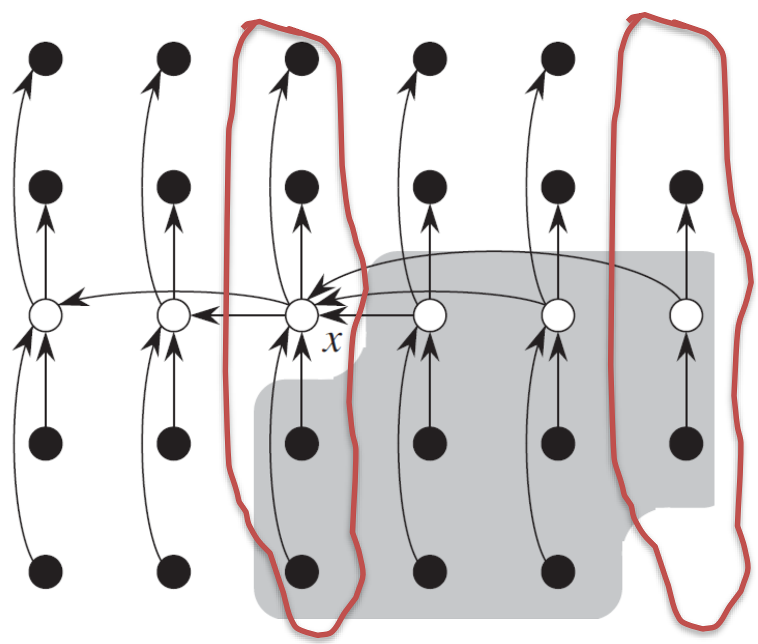
   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis

- At least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x,
  - except for the one group that has fewer than 5 elements, and the one group containing x itself



Running time analysis (Cont'd)

- The number of elements greater than x is at least

$$3(\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\text{-}2) \geq \frac{3n}{10} - 6$$

- Similarly, at least $\frac{3n}{10}$ - 6 elements are less than x. Thus, in the worst case, step 5 calls SELECT recursively on at most $\frac{7n}{10}$ + 6 elements.

Running time analysis (cont'd)

- Step 1 takes O(n) time
- Step 2 consists of O(n) calls of insertion sort on sets of size O(1)
- Step 3 takes time T($\lceil n/5 \rceil$)
- Step 4 takes O(n) time
- Step 5 takes time at most T(7n/10 + 6)

1. Divide *n* elements into groups of 5
2. Find median of each group (How? How long?)
3. Use Select() recursively to find median *x* of the $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

   if (i < k) **then**

   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis (cont'd)

- We can therefore obtain the recurrence
- T(n) $\leq$ T($\lceil n/5 \rceil$) + T(7n/10 + 6) + O(n)
- Assume T(k) $\leq$ ck for k < n, use the substitution method
- T(n) $\leq$ c$\lceil n/5 \rceil$ + c(7n/10 + 6) + an

  $\leq$ cn/5 + c + 7cn/10 + 6c + an

  = 9cn/10 + 7c +an

  = cn + (-cn/10 + 7c + an)

Running time analysis (cont'd)

- T(n) $\leq$ cn + (-cn/10 + 7c + an)
- Which is at most cn if
  - -cn/10 + 7c + an ≤ 0
  - c $\geq 10a(n/(n-70))$ when n > 70

Linear-Time Median Selection

- Given a "black box" $O(n)$ median algorithm, what can we do?
  - *i*-th order statistic:
    - Find median x
    - Partition input around x
    - if ($i \leq (n+1)/2$) recursively find i-th element of first half
    - else find ($i - (n+1)/2$)-th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$ (why?)

Worst-case quicksort

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - T(n) = 2T(n/2) + O(n) = O(n lg n)
  - Input assumption?