Module 3: Sorting and Randomized Algorithms

CS 240 - Data Structures and Data Management

Reza Dorrigiv, Daniel Roche

School of Computer Science, University of Waterloo

Winter 2010

Selection vs. Sorting

We have already seen some algorithms for the *selection problem*:
Given an array A of n numbers, find the kth largest number.
(note: we always count from zero, so $0 \leq k < n$)

Best heap-based algorithm had time cost $\Theta(n + k \log n)$.
For median selection, $k = \lfloor \frac{n}{2} \rfloor$, giving cost $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?

Selection vs. Sorting

We have already seen some algorithms for the *selection problem*:
Given an array A of n numbers, find the kth largest number.
(note: we always count from zero, so $0 \leq k < n$)

Best heap-based algorithm had time cost $\Theta(n + k \log n)$.
For median selection, $k = \lfloor \frac{n}{2} \rfloor$, giving cost $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?
The *quick-select* algorithm answers this question in the affirmative.

Selection vs. Sorting

We have already seen some algorithms for the *selection problem*:
Given an array A of n numbers, find the kth largest number.
(note: we always count from zero, so $0 \leq k < n$)

Best heap-based algorithm had time cost $\Theta(n + k \log n)$.
For median selection, $k = \lfloor \frac{n}{2} \rfloor$, giving cost $\Theta(n \log n)$.

This is the same cost as our best sorting algorithms.

Question: Can we do selection in linear time?
The *quick-select* algorithm answers this question in the affirmative.

Observation: Finding the element at a given position is tough, but finding the position of a given element is simple.

Crucial Subroutines

quick-select and the related algorithm *quick-sort* rely on two subroutines:

- *choose-pivot*(A): Choose an index i such that $A[i]$ will make a good pivot (hopefully near the middle of the order).
- *partition*(A, p): Using pivot $A[p]$, rearrange A so that
  all items $\leq$ the pivot come first,
  followed by the pivot,
  followed by all items greater than the pivot.

Selecting a pivot

Ideally, we would select a median as the pivot.
But this is the problem we're trying to solve!

First idea: Always select first element in array

```
choose-pivot1(A)
1.    return 0
```

We will consider more sophisticated ideas later on.

Partition Algorithm

```
partition(A, p)
A: array of size n,   p: integer s.t. 0 ≤ p < n
 1.    swap(A[0], A[p])
 2.    i ← 1,   j ← n − 1
 3.    while i < j do
 4.        while A[i] ≤ A[0] and i < n do
 5.            i ← i + 1
 6.        while A[j] > A[0] and j > 0 do
 7.            j ← j − 1
 8.        if i < j then
 9.            swap(A[i], A[j])
10.    swap(A[0], A[j])
11.    return j
```

Idea: Keep swapping the outer-most wrongly-positioned pairs.

QuickSelect Algorithm

quick-select1(A, k)
A: array of size n, k: integer s.t. $0 \leq k < n$

```
1.    p ← choose-pivot1(A)
2.    i ← partition(A, p)
3.    if i = k then
4.        return A[i]
5.    else if i > k then
6.        return quick-select1(A[0, 1, ..., i − 1], k)
7.    else if i < k then
8.        return quick-select1(A[i + 1, i + 2, ..., n − 1], k − i − 1)
```

Analysis of quick-select1

Worst-case analysis: Recursive call could always have size $n-1$.

Recurrence given by $T(n) = \begin{cases} T(n-1)+cn, & n \geq 2 \\ d, & n = 1 \end{cases}$

Solution: $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + d \in \Theta(n^2)$

Analysis of quick-select1

Worst-case analysis: Recursive call could always have size $n-1$.

Recurrence given by $T(n) = \begin{cases} T(n-1)+cn, & n \geq 2 \\ d, & n = 1 \end{cases}$

Solution: $T(n) = cn + c(n-1) + c(n-2) + \cdots + c \cdot 2 + d \in \Theta(n^2)$

Best-case analysis: First chosen pivot could be the kth element

No recursive calls; total cost is $\Theta(n)$.

Average-case analysis of quick-select1

Assume all $n!$ permutations are equally likely.
Average cost is sum of costs for all permutations, divided by $n!$.

Define $T(n, k)$ as average cost for selecting kth item from size-n array:

$$T(n,k) = cn + \frac{1}{n}\left(\sum_{i=0}^{k-1} T(n-i-1, k-i) + \sum_{i=k+1}^{n-1} T(i,k)\right)$$

We could analyze this recurrence directly,
or be a little lazier and still get the same asymptotic result.

For simplicity, define $T(n) = \max_{0 \le k < n} T(n,k)$.

Average-case analysis of quick-select1

The cost is determined by i, the position of the pivot $A[0]$.
For more than half of the $n!$ permutations, $\frac{n}{4} \leq i < \frac{3n}{4}$.

In this case, the recursive call will have length at most $\left\lfloor \frac{3n}{4} \right\rfloor$, for any k.
The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}\Big(T(n) + T\big(\lfloor 3n/4 \rfloor\big)\Big), & n \geq 2 \\ d, & n = 1 \end{cases}$$

Average-case analysis of quick-select1

The cost is determined by i, the position of the pivot $A[0]$.
For more than half of the $n!$ permutations, $\frac{n}{4} \leq i < \frac{3n}{4}$.

In this case, the recursive call will have length at most $\left\lfloor \frac{3n}{4} \right\rfloor$, for any k.
The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}\Big(T(n) + T\big(\lfloor 3n/4 \rfloor\big)\Big), & n \geq 2 \\ d, & n = 1 \end{cases}$$

Rearranging gives:

$$\begin{aligned} T(n) &\leq 2cn + T\big(\lfloor 3n/4 \rfloor\big) \leq 2cn + 2c(3n/4) + 2c(9n/16) + \cdots + d \\ &\leq d + 2cn \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \in O(n) \end{aligned}$$

Since $T(n)$ *must* be $\Omega(n)$ (why?), $T(n) \in \Theta(n)$.

Randomized algorithms

A *randomized algorithm* is one which relies on some random numbers in addition to the input.
The cost will depend on the input and the random numbers used.

Randomized algorithms

A *randomized algorithm* is one which relies on some random numbers in addition to the input.
The cost will depend on the input and the random numbers used.

Generating random numbers: Computers can't generate randomness.
Instead, some external source is used (e.g. clock, mouse, gamma rays,. . .)

This is expensive, so we use a *pseudo-random number generator (PRNG)*, a deterministic program that uses a true-random initial value or *seed*.
This is much faster and often indistinguishable from truly random.

Randomized algorithms

A *randomized algorithm* is one which relies on some random numbers in addition to the input.
The cost will depend on the input and the random numbers used.

Generating random numbers: Computers can't generate randomness.
Instead, some external source is used (e.g. clock, mouse, gamma rays,. . .)

This is expensive, so we use a *pseudo-random number generator (PRNG)*, a deterministic program that uses a true-random initial value or *seed*.
This is much faster and often indistinguishable from truly random.

Goal: To shift the probability distribution from what we can't control (the input), to what we can control (the random numbers).
There should be no more bad instances, just unlucky numbers.

Expected running time

Define $T(I, R)$ as the running time of the randomized algorithm for a particular input I and the sequence of random numbers R.

The *expected running time* $T_A^{(exp)}(I)$ of a randomized algorithm A for a particular input I is the "expected" value for $T(I, R)$:

$$T_A^{(exp)}(I) = \mathbf{E}[T(I, R)] = \sum_R T(I, R) \cdot Pr[R]$$

Expected running time

Define $T(I, R)$ as the running time of the randomized algorithm for a particular input I and the sequence of random numbers R.

The *expected running time* $T_A^{(exp)}(I)$ of a randomized algorithm A for a particular input I is the "expected" value for $T(I, R)$:

$$T_A^{(exp)}(I) = \mathbf{E}[T(I, R)] = \sum_R T(I, R) \cdot Pr[R]$$

The *worst-case expected running time* is then

$$T_A^{(\exp)}(n) = \max_{size(I)=n} T_A^{(\exp)}(I).$$

For many randomized algorithms, worst-, best-, and average-case expected times are the same (why?).

Randomized QuickSelect

random(n) returns an integer uniformly from $\{0, 1, 2, \ldots, n-1\}$.

First idea: Randomly permute the input first using *shuffle*:

```
shuffle(A)
A: array of size n
1.      for i ← 0 to n − 2 do
2.              swap(A[i], A[i + random(n − i)])
```

Expected cost becomes the same as the average cost, which is $\Theta(n)$.

Randomized QuickSelect

Second idea: Change the pivot selection.

```
choose-pivot2(A)
1.    return random(n)
```

```
quick-select2(A, k)
1.    p ← choose-pivot2(A)
2.    ...
```

With probability at least $\frac{1}{2}$, the random pivot has position $\frac{n}{4} \leq i < \frac{3n}{4}$, so the analysis is just like that for the average-case.
The expected cost is again $\Theta(n)$.

Worst-case linear time

Blum, Floyd, Pratt, Rivest, and Tarjan invented the "medians-of-five" algorithm in 1973 for pivot selection:

```
choose-pivot3(A)
A: array of size n
1.    m ← ⌊n/5⌋ − 1
2.    for i ← 0 to m do
3.        j ← index of median of A[5i, ..., 5i + 4]
4.        swap(A[i], A[j])
5.    return quick-select3(A[0, ..., m], ⌊m/2⌋)
```

```
quick-select3(A, k)
1.    p ← choose-pivot3(A)
2.    ...
```

This *mutually recursive* algorithm can be shown to be $\Theta(n)$ in the worst case, but it's a little beyond the scope of this course.

QuickSort

QuickSelect is based on a sorting method developed by Hoare in 1960:

quick-sort1(A)
A: array of size n
1. **if** $n \leq 1$ **then return**
2. $p \leftarrow$ *choose-pivot1*(A)
3. $i \leftarrow$ *partition*(A, p)
4. *quick-sort1*$(A[0, 1, \ldots, i-1])$
5. *quick-sort1*$(A[i+1, \ldots, size(A)-1])$

QuickSort

QuickSelect is based on a sorting method developed by Hoare in 1960:

```
quick-sort1(A)
A: array of size n
1.    if n ≤ 1 then return
2.    p ← choose-pivot1(A)
3.    i ← partition(A, p)
4.    quick-sort1(A[0, 1, ..., i − 1])
5.    quick-sort1(A[i + 1, ..., size(A) − 1])
```

Worst case: $T^{(\text{worst})}(n) = T^{(\text{worst})}(n-1) + \Theta(n)$
Same as *quick-select1*; $T^{(\text{worst})}(n) \in \Theta(n^2)$

QuickSort

QuickSelect is based on a sorting method developed by Hoare in 1960:

```
quick-sort1(A)
A: array of size n
1.    if n ≤ 1 then return
2.    p ← choose-pivot1(A)
3.    i ← partition(A, p)
4.    quick-sort1(A[0, 1, ..., i − 1])
5.    quick-sort1(A[i + 1, ..., size(A) − 1])
```

Worst case: $T^{(\text{worst})}(n) = T^{(\text{worst})}(n-1) + \Theta(n)$
Same as *quick-select1*; $T^{(\text{worst})}(n) \in \Theta(n^2)$

Best case: $T^{(\text{best})}(n) = T^{(\text{best})}(\lfloor \frac{n-1}{2} \rfloor) + T^{(\text{best})}(\lceil \frac{n-1}{2} \rceil) + \Theta(n)$
Similar to *merge-sort*; $T^{(\text{best})}(n) \in \Theta(n \log n)$

Average-case analysis of quick-sort1

Of all $n!$ permutations, $(n-1)!$ have pivot $A[0]$ at a given position i.

Average cost over all permutations is given by:

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-i-1)\big) + \Theta(n), \qquad n \geq 2$$

It is possible to solve this recursion directly.

Average-case analysis of quick-sort1

Of all $n!$ permutations, $(n-1)!$ have pivot $A[0]$ at a given position i.

Average cost over all permutations is given by:

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}\big(T(i) + T(n-i-1)\big) + \Theta(n), \qquad n \geq 2$$

It is possible to solve this recursion directly.

Instead, notice that the cost at each level of the recursion tree is $O(n)$.
So let's consider the height (or "depth") of the recursion, on average.

Average depth of recursion for quick-sort1

Define $H(n)$ as the average recursion depth for size-n inputs. So

$$H(n) = \begin{cases} 1 + \frac{1}{n}\sum_{i=0}^{n-1} \max\left(H(i), H(n-i-1)\right), & n \geq 2 \\ 0, & n \leq 1 \end{cases}$$

- Let i be the position of the pivot $A[0]$.
  Again, $\frac{n}{4} \leq i < \frac{3n}{4}$ for more than half of all permutations.
- Then larger recursive call has length at most $\left\lfloor \frac{3n}{4} \right\rfloor$.
  This will determine the recursion depth at least half the time.

Average depth of recursion for quick-sort1

Define $H(n)$ as the average recursion depth for size-n inputs. So

$$H(n) = \begin{cases} 1 + \frac{1}{n}\sum_{i=0}^{n-1} \max\big(H(i), H(n-i-1)\big), & n \geq 2 \\ 0, & n \leq 1 \end{cases}$$

- Let i be the position of the pivot $A[0]$.
  Again, $\frac{n}{4} \leq i < \frac{3n}{4}$ for more than half of all permutations.
- Then larger recursive call has length at most $\lfloor \frac{3n}{4} \rfloor$.
  This will determine the recursion depth at least half the time.
- Therefore $H(n) \leq 1 + \frac{1}{4}\big(3H(n) + H(\lfloor \frac{3n}{4} \rfloor)\big)$ for $n \geq 2$,
  which simplifies to $H(n) \leq 4 + H(\lfloor \frac{3n}{4} \rfloor)$.
- So $H(n) \in O(\log n)$.
  Average cost is $O(nH(n)) \in O(n \log n)$.
  Since best-case is $\Theta(n \log n)$, average must be $\Theta(n \log n)$.

More notes on QuickSort

- We can randomize by using *choose-pivot2*, giving $\Theta(n \log n)$ expected time for *quick-sort2*.
- We can use *choose-pivot3* (along with *quick-select3*) to get *quick-sort3* with $\Theta(n \log n)$ worst-case time.
- We can use tail recursion to save space on one of the recursive calls. By making sure the other one is always smaller, the auxiliary space is $\Theta(\log n)$ in the worst case, even for *quick-sort1*.
- QuickSort is often the most efficient algorithm in practice.

Lower bounds for sorting

We have seen many sorting algorithms:

Sort	Running time	Analysis
Selection Sort	$\Theta(n^2)$	worst-case
Insertion Sort	$\Theta(n^2)$	worst-case
Merge Sort	$\Theta(n \log n)$	worst-case
Heap Sort	$\Theta(n \log n)$	worst-case
quick-sort1	$\Theta(n \log n)$	average-case
quick-sort2	$\Theta(n \log n)$	expected
quick-sort3	$\Theta(n \log n)$	worst-case

Question: Can one do better than $\Theta(n \log n)$?

Answer: Yes and no! *It depends on what we allow*.

- No: Comparison-based sorting lower bound is $\Omega(n \log n)$.
- Yes: Non-comparison-based sorting can achieve $O(n)$.

The Comparison Model

In the *comparison model* data can only be accessed in two ways:

- comparing two elements
- moving elements around (e.g. copying, swapping)

This makes very few assumptions on the kind of things we are sorting.
We count the number of above operations.

All sorting algorithms seen so far are in the comparison model.

Lower bound for sorting in the comparison model

Theorem. Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations.

Proof.

- A correct algorithm takes different *actions* (moves, swaps, etc.) for each of the $n!$ possible permutations.
- The choice of actions is determined only by comparisons.

Lower bound for sorting in the comparison model

Theorem. Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations.

Proof.

- A correct algorithm takes different *actions* (moves, swaps, etc.) for each of the $n!$ possible permutations.
- The choice of actions is determined only by comparisons.
- The algorithm can be viewed as a *decision tree*.
  Each internal node is a comparison, each leaf is a set of actions.
- Each permutation must correspond to a leaf.

Lower bound for sorting in the comparison model

Theorem. Any correct comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparison operations.

Proof.

- A correct algorithm takes different *actions* (moves, swaps, etc.) for each of the $n!$ possible permutations.
- The choice of actions is determined only by comparisons.
- The algorithm can be viewed as a *decision tree*.
  Each internal node is a comparison, each leaf is a set of actions.
- Each permutation must correspond to a leaf.
- The worst-case number of comparisons is the longest path to a leaf.
- Since the tree has at least $n!$ leaves, the height is at least $\lg n!$.
- Therefore worst-case number of comparisons is $\Omega(n \log n)$. □

Counting Sort

Requirement: Each $A[i]$ satisfies $0 \leq A[i] < k$; k is given.

```
counting-sort(A, k)
A: array of size n,   k: positive integer
1.    C ← array of size k, filled with zeros
2.    for i ← 0 to n − 1 do
3.        increment C[A[i]]
4.    for i ← 1 to k − 1 do
5.        C[i] ← C[i] + C[i − 1]
6.    B ← copy(A)
7.    for i ← n − 1 down to 0 do
8.        decrement C[B[i]]
9.        A[C[B[i]]] ← B[i]
```

Counting Sort

Requirement: Each $A[i]$ satisfies $0 \leq A[i] < k$; k is given.

```
counting-sort(A, k)
A: array of size n,   k: positive integer
1.    C ← array of size k, filled with zeros
2.    for i ← 0 to n − 1 do
3.        increment C[A[i]]
4.    for i ← 1 to k − 1 do
5.        C[i] ← C[i] + C[i − 1]
6.    B ← copy(A)
7.    for i ← n − 1 down to 0 do
8.        decrement C[B[i]]
9.        A[C[B[i]]] ← B[i]
```

- **Time cost**: $\Theta(n + k)$, which is $\Theta(n)$ if $k \in O(n)$.
- **Auxiliary space**: $\Theta(n + k)$, can be made $\Theta(k)$
- *counting-sort* is *stable*: equal items stay in original order.

Radix Sort

Requirement: Each $A[i]$ is a string of d digits $x_{d-1}x_{d-2}\cdots x_0$, and each x_i satisfies $0 \leq x_i < k$.
Example: integers between 0 and $k^d - 1$

radix-sort(A, d, k)
A: array of size n, d: positive integer, k: positive integer
1. **for** $i \leftarrow 0$ to $d-1$ **do**
2. Call *counting-sort*(A, k) with each x_i as the key

Radix Sort

Requirement: Each $A[i]$ is a string of d digits $x_{d-1}x_{d-2}\cdots x_0$, and each x_i satisfies $0 \leq x_i < k$.
Example: integers between 0 and $k^d - 1$

radix-sort(A, d, k)
A: array of size n, d: positive integer, k: positive integer
1. **for** $i \leftarrow 0$ to $d-1$ **do**
2. Call *counting-sort*(A, k) with each x_i as the key

- **Time cost**: $\Theta(d(n+k))$
- **Auxiliary space**: $\Theta(n+k)$

Summary of sorting

- Randomized algorithms can eliminate "bad cases"
- Best-case, worst-case, average-case, expected-case can all differ
- Sorting is an important and *very* well-studied algorithm
- Can be done in $\Theta(n \log n)$ time; faster is not possible for general input
- HeapSort is the only fast algorithm we have seen with $O(1)$ auxiliary space.
- QuickSort is often the fastest in practice
- MergeSort is also $\Theta(n \log n)$, selection & insersion sorts are $\Theta(n^2)$.
- CountingSort, RadixSort can achieve $o(n \log n)$ if the input is special