CS 457, Fall 2019

Drexel University, Department of Computer Science

Lecture 5

Running Time of Divide & Conquer

- We first discussed how the worst-case running time can be thought of as a function of the input size
- When the running time of an algorithm depends on the running time for solving smaller instances of the same problem, we get a recurrence
- Recurrence equation for divide and conquer algorithms:
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$

Methods for Solving Recurrences

Three methods:

1. Recursion-tree method
   - Covert into a tree and measure cost incurred at the various levels
2. Substitution method
   - Guess a bound and use mathematical induction to prove its correctness
3. Master method
   - Directly provides bounds for recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Maximum Subarray Problem

- You are given an array A of n numbers (both positive and negative)
  - Find a contiguous subarray with the maximum sum of numbers
  - In other words: find i, j such that $1 \leq i \leq j \leq n$ and maximize $\sum_{x=i}^{j} A[x]$
  - For example, consider the following array:

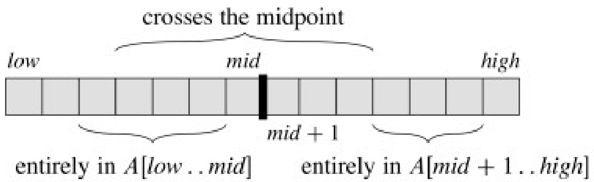
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray

  - What is the first, simple, algorithm that comes to mind?
  - What is the running time of this algorithm?
  - Can you come up with a divide & conquer algorithm?
    - How can we analyze the (worst case) running time of such algorithms?

Maximum Subarray Problem

What does the recurrence equation look like?



(a)

Since we know that the max crossing subarray will comprise a suffix of the left subarray and a prefix of the right subbaray, we can actually compute it in time $O(n_{left} + n_{right})$

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i . . mid].
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 . . j].
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```

Maximum Subarray Problem

Divide-and-conquer procedure for the maximum-subarray problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])            // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

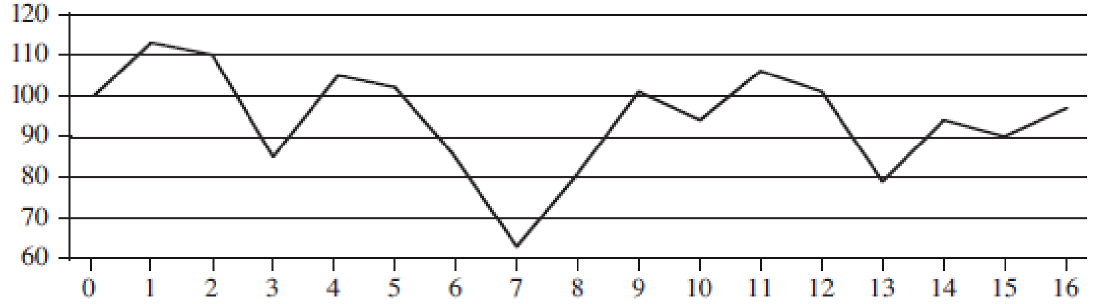
Initial call: FIND-MAXIMUM-SUBARRAY$(A, 1, n)$

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

As we showed for merge sort, this recurrence equation leads to a bound of $O(n \log n)$

Profit Maximizing Stock Trade

- Input: $\boldsymbol{n}$ price points $(t_1, t_2, \ldots, t_n)$
- Output: $(\boldsymbol{t_b}, \boldsymbol{t_s})$ s.t. $\boldsymbol{0 \le t_b < ts \le n}$, and $\boldsymbol{p(t_s) - p(t_b)}$ is maximized



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- How would you solve this problem?
  - This problem can be easily reduced to the maximum subarray problem

Merge Sort

Sorting: Given a list A of n integers, create a sorted list of these integers

- Divide
  - *Split the problem into smaller sub-problems of the same structure*
  - Split the list A into two smaller lists of size n_1 and n_2
- Conquer
  - *If sub-problem size is small enough, solve directly, o/w, solve sub-problems recursively*
  - Sort the two smaller lists recursively using merge sort, unless their size is small
- Combine
  - *Merge the solutions of sub-problems into a solution of the original problem*
  - Merge the two sorted lists into one, and return the result

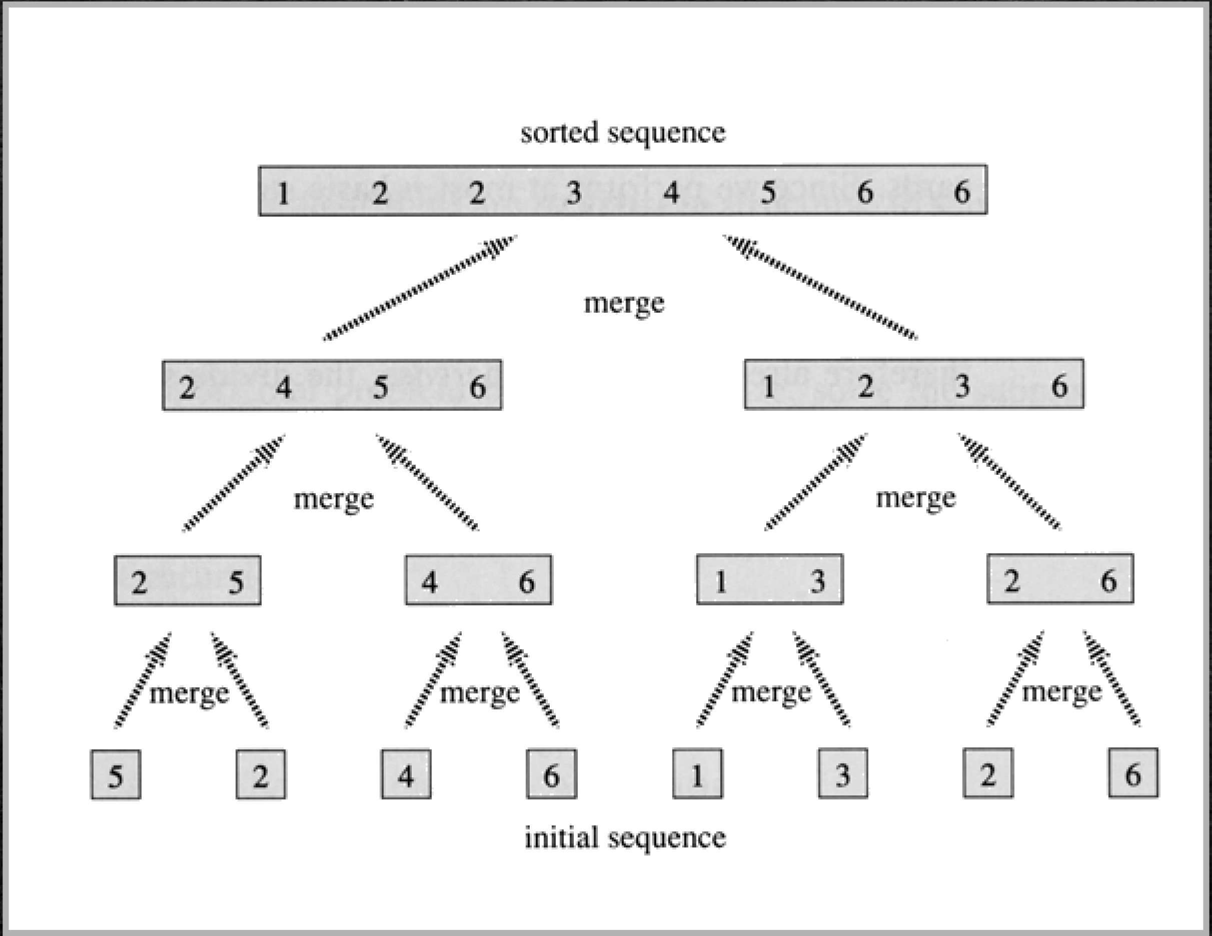
Merge Sort

MERGE-SORT (A, p, r)

```
1.    if p < r                              // Check for base case
2.        q = ⌊(p + r)/2⌋                   // Divide step
3.        MERGE-SORT (A, p, q)              // Conquer step.
4.        MERGE-SORT (A, q + 1, r)          // Conquer step.
5.        MERGE (A, p, q, r)                // Conquer step.
```

To sort A[1 .. n], make initial call to MERGE-SORT (A, 1, n)

Merge Sort



sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

Merging Two Sorted Lists (running time)

MERGE (A, p, q, r)

```
n1 = q – p + 1
n2 = r – q
Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
for i = 1 to n1
        L[i] = A[p + i – 1]
for j = 1 to n2
        R[j] = A[q + j ]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i ] ≤ R[ j]
                A[k] = L[i]
                i = i + 1
        else
                A[k] = R[j]
                j = j + 1
```

This needs time $\Theta(n_1 + n_2) = \Theta(r - p + 1)$

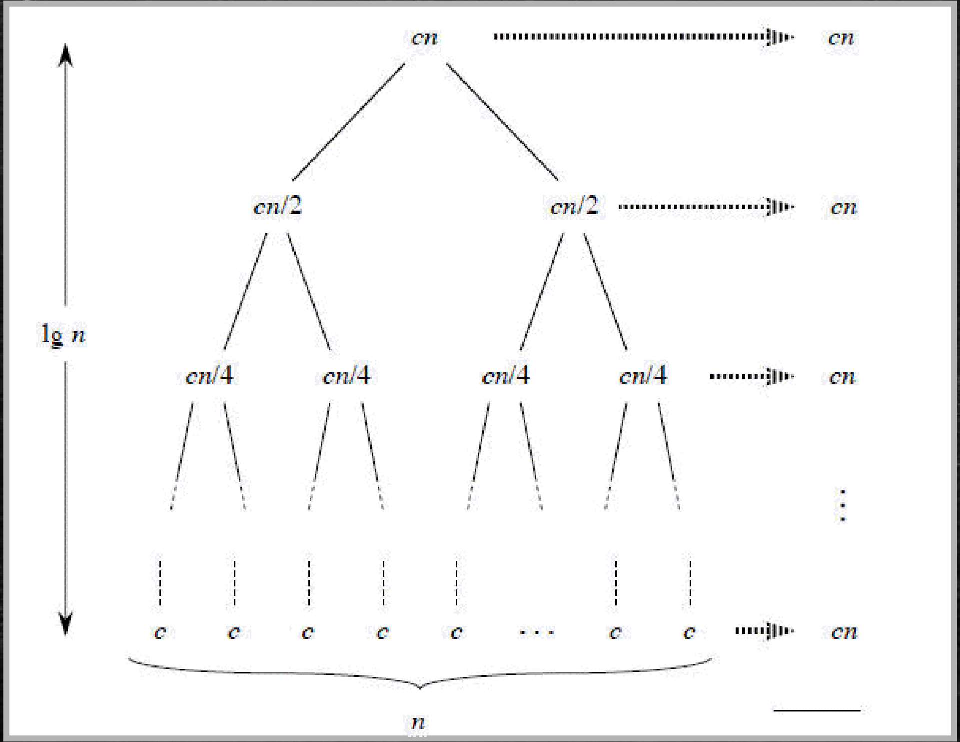
How about MERGE-SORT *(A, 1, n)*?

Running Time

- Recurrence equation for divide and conquer algorithms:
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$
- Recurrence equation for Merge Sort
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$

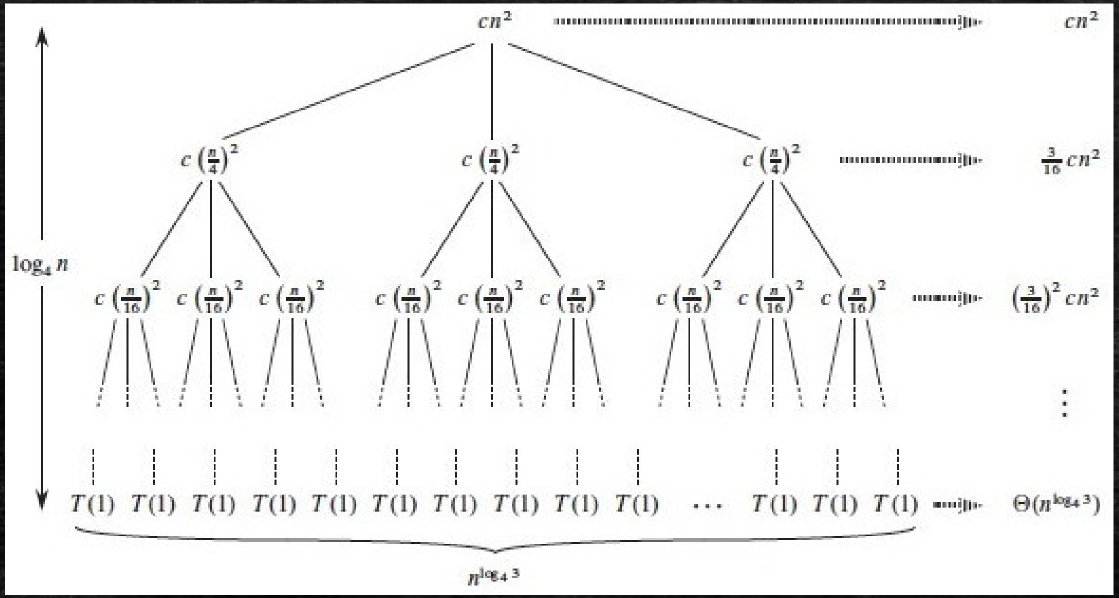
Recursion-Tree Method

- Recurrence equation for Merge Sort
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise} \end{cases}$



Recursion-Tree Method

- Recurrence equation
  - $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(\mathrm{n}^2) & \text{otherwise} \end{cases}$



Substitution Method

1. Guess the form of the solution
2. Use mathematical induction to show that it works (for appropriate constants)

E.g., $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n) = 2T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$

Why wouldn't this work for $\boldsymbol{T(n) \leq cn}$ as well? (verify it!)

Guess that $T(n) = O(n \log n)$

Then, assume $T(n') \leq cn' \log n'$ for all $n' < n$, and show $T(n) \leq cn \log n$

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2[c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)] + n \\ &\leq cn \log(n/2) + n \\ &\leq cn \log n - cn \log 2 + n \\ &\leq cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant ε, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant ε, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n ,then $T(n) = \Theta(f(n))$

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n
  1. $T(n) = 4T\left(\frac{n}{3}\right) + n\log n$
  2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
  3. $T(n) = 3T\left(\frac{n}{3}\right) + n/\log n$

We first seek to apply the master theorem. If we let $f(n) = n/\log n, a = 3$, and $b = 3$, we see that $n^{\log_b a} = n^{\log_3 3} = n$. Since $n/\log n \in o(n)$, it is clear that the second and third rule of the master theorem cannot apply. But, as it happens, the first rule does not apply either, since $n/\log n \in \omega(n^{1-\epsilon})$ for any constant $\epsilon > 0$.

Examples

- Give asymptotic upper and lower bounds for $T(n)$. Assume that $T(n)$ is constant for sufficiently small n

  1. $T(n) = 4T\left(\frac{n}{3}\right) + n\log n$
  2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
  3. $T(n) = 3T\left(\frac{n}{3}\right) + n/\log n$

Using a recursion tree, we observe that its depth for this equation is $\Theta(\log n)$ and the total cost at depth d is $\frac{n}{\log(\frac{n}{3^d})}$. This leads to:

$$T(n) \approx \sum_{d=1}^{\Theta(\log n)} \frac{n}{\log\left(\frac{n}{3^d}\right)} \approx n \sum_{d=1}^{\Theta(\log n)} \frac{1}{\log n - d} \approx n \sum_{d=1}^{\Theta(\log n)} \frac{1}{d} \approx \Theta(n \log\log n).$$

We therefore guess that $T(n) \in \Theta(n\log\log n)$, and prove this using the substitution method.

Today's Lecture and Next Homework

- Analysis of recurrence equations
- More divide and conquer algorithms

- Second homework will be available tomorrow
- It is due Wednesday 10/16
- No collaboration is allowed, so please ask questions early

Change of Variables in Recurrences

- In the recitation, we solved the recurrence: $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

In order to solve this recurrence equation we will be performing a change of variables (see Page 86 of your textbook). In particular, just like in the textbook, we will be renaming $m = \log n$, which yields $T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m$. If we divide both sides with 2^m, we get

$$\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + 1.$$

We can now rename $S(m) = \frac{T(2^m)}{2^m}$, which reduces our initial recurrence equation to $S(m) = S(m/2) + 1$, which is much easier to solve. Using the master theorem to solve $S(m)$, we have $a = 1$, $b = 2$ and $f(n) = 1$, so $n^{\log_b a} = n^0 = 1$. Therefore, $f(1) = 1 \in \Theta(1) = \Theta(n^{\log_b a})$ and, using the second case of the master theorem, we get $S(m) \in \Theta(\log m)$. Since $S(m) = \frac{T(2^m)}{2^m}$, th implies that $T(2^m) = 2^m S(m) \in \Theta(2^m \log m)$. Finally, since $m = \log n$, we conclude that

$$T(n) = \Theta(n \log \log n).$$

What is wrong with this argument?

- Doesn't that seem to suggest a sorting algorithm faster than $O(n \log n)$?
  - First, divide the array of n elements into $\sqrt{n}$ parts of size $\sqrt{n}$ each
  - Then, sort these parts recursively, and merge them to get the final sorted list
  - The solution suggests that the running time would be $O(n \log \log n)$!!!

Change of Variables in Recurrences

- In the recitation, we solved the recurrence: $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

In order to solve this recurrence equation we will be performing a change of variables (see Page 86 of your textbook). In particular, just like in the textbook, we will be renaming $m = \log n$, which yields $T(2^m) = 2^{m/2} T(2^{m/2}) + 2^m$. If we divide both sides with 2^m, we get

$$\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + 1.$$

We can now rename $S(m) = \frac{T(2^m)}{2^m}$, which reduces our initial recurrence equation to $S(m) = S(m/2) + 1$, which is much easier to solve. Using the master theorem to solve $S(m)$, we have $a = 1$, $b = 2$ and $f(n) = 1$, so $n^{\log_b a} = n^0 = 1$. Therefore, $f(1) = 1 \in \Theta(1) = \Theta(n^{\log_b a})$ and, using the second case of the master theorem, we get $S(m) \in \Theta(\log m)$. Since $S(m) = \frac{T(2^m)}{2^m}$, this implies that $T(2^m) = 2^m S(m) \in \Theta(2^m \log m)$. Finally, since $m = \log n$, we conclude that

$$T(n) = \Theta(n \log \log n).$$

Merging is not linear anymore!

- Doesn't that seem to suggest a sorting algorithm faster than $O(n \log n)$?
  - First, divide the array of n elements into $\sqrt{n}$ parts of size $\sqrt{n}$ each
  - Then, sort these parts recursively, and merge them to get the final sorted list
  - The solution suggests that the running time would be $O(n \log \log n)$!!!

Quicksort

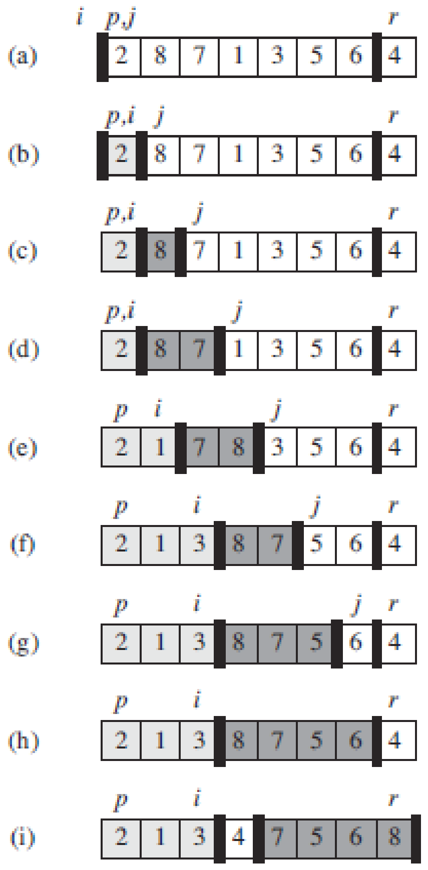
QUICKSORT (*A*, *p*, *r*)

```
if p < r                                   // Check for base case
    q = PARTITION(A, p, r)                 // Divide step
    QUICKSORT (A, p, q - 1)                // Conquer step.
    QUICKSORT (A, q + 1, r)                // Conquer step.
```

Quicksort

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

```
x = A[r]
i = p – 1
for  j = p  to  r – 1
        if  A[ j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```



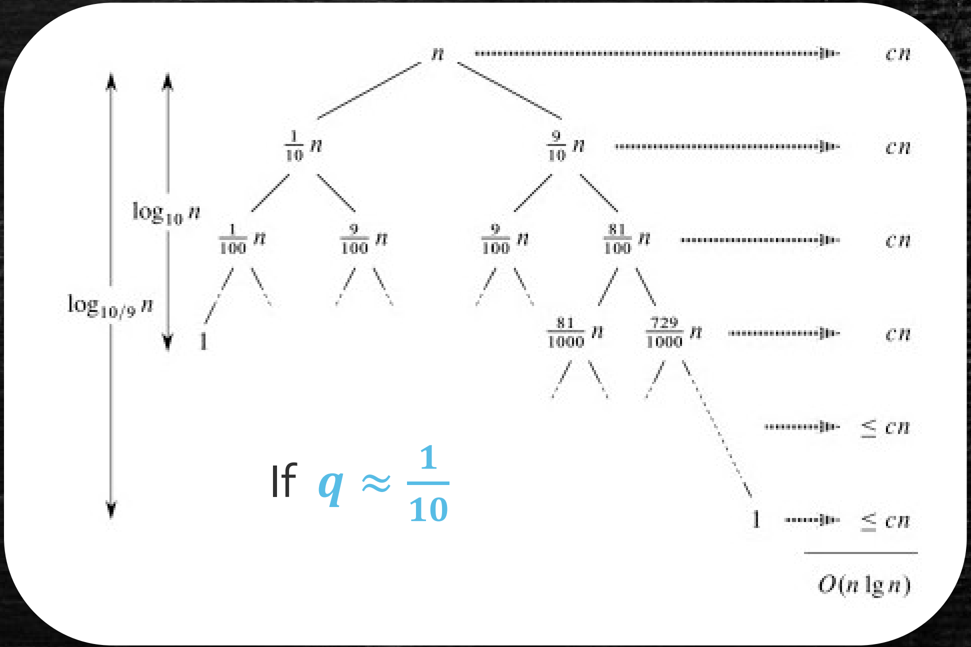
Quicksort (Running Time)

QUICKSORT (A, p, r)

1. **if** $p < r$ // Check for base case
2. q = PARTITION(A, p, r) // Divide step
3. QUICKSORT (A, p, q - 1) // Conquer step
4. QUICKSORT (A, q + 1, r) // Conquer step

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(q) + T(n-q-1) + \Theta(n) & \text{otherwise} \end{cases}$$

- This leads to $O(n^2)$ in the worst case
- But, if $q = cn$ for some constant c, it is $O(n \log n)$



Order Statistics and the Selection Problem

The i^{th} **order statistic** of a set of n numbers: the i^{th} smallest number in sorted sequence:

A	4	1	3	2	16	9	10	14	8	7

- Minimum or first order statistic: $\mathbf{1}$
- Maximum or n^{th} order statistic: $\mathbf{16}$
- Median or $(n/2)^{\text{th}}$ order statistic: $\mathbf{7}$ or $\mathbf{8}$ (both are medians, when n is even)

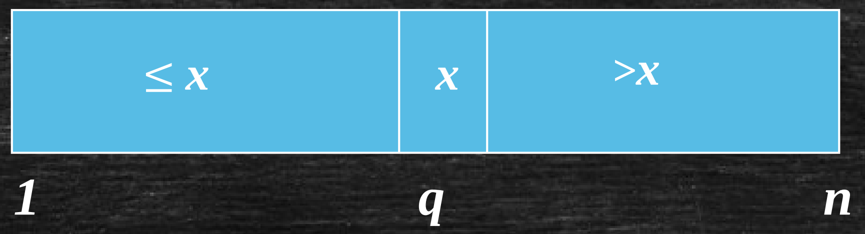
- **Selection Problem**
  - **Input:** An array $\mathbf{A}$ of distinct numbers of size n, and a number $\boldsymbol{i}$
  - **Output:** The element $\boldsymbol{x}$ in $\mathbf{A}$ that is larger than exactly $\boldsymbol{i}-\mathbf{1}$ oth

  - Finding ***maximum*** and ***minimum?***
  - Can be easily solved in linear time (i.e., $O(n)$. It's actually $\Theta(n)$)
  - What about finding the i^{th} order statistic for any given $i \in [1, n]$?

We could always sort the numbers, but that would need $\Theta(n \log n)$ time

Selection Algorithms using Pivot Element

- Choose a pivot element x and partition the subarray $A[1, \ldots, n]$ around it



- If $q == i$, then x is the i^{th} order statistic
- If $q > i$, then we want the i^{th} order statistic of subarray $[1, \ldots, q-1]$
- If $q < i$, then we want the $(i-q)^{\text{th}}$ order statistic of subarray $[q+1, \ldots, n]$

- But, how do we choose this pivot element?

Simple Selection Algorithm

Select(A, p, r, i)

```
if p == r
    return A[p]
q = Partition(A, p, r)
k = q − p + 1
if i == k
    return A[q]
else if  i ≤ k
    Select(A, p, q − 1, i)
else
    Select(A, q + 1, r, i − k)
```

Partition (A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Partitioning

Partition (A, p, r)

$x = A[r]$

$i = p - 1$

for $j = p$ to $r - 1$

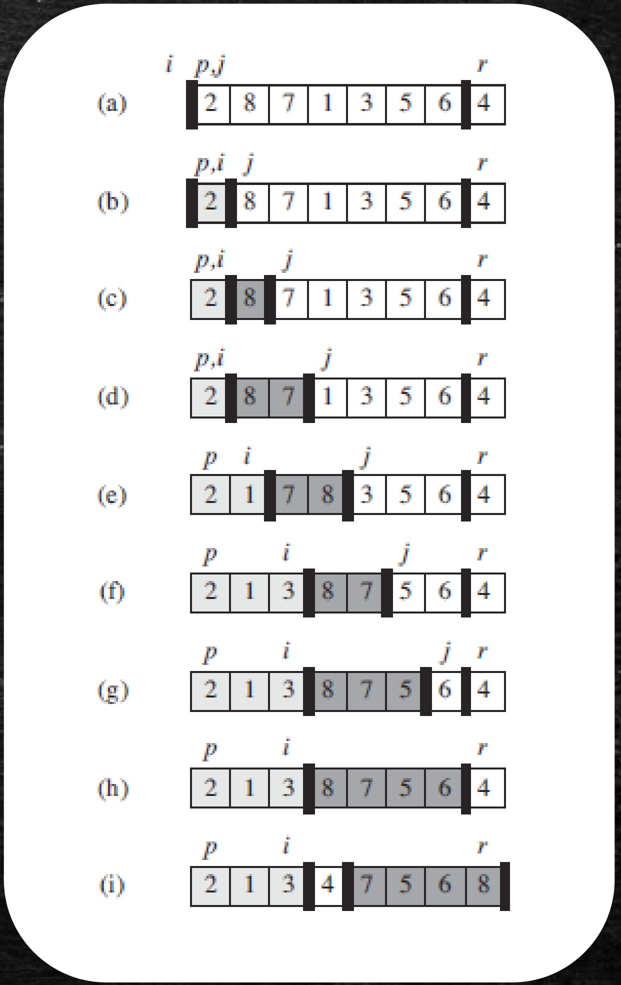
  if $A[j] \leq x$

    $i = i + 1$

    exchange $A[i]$ with $A[j]$

exchange $A[i + 1]$ with $A[r]$

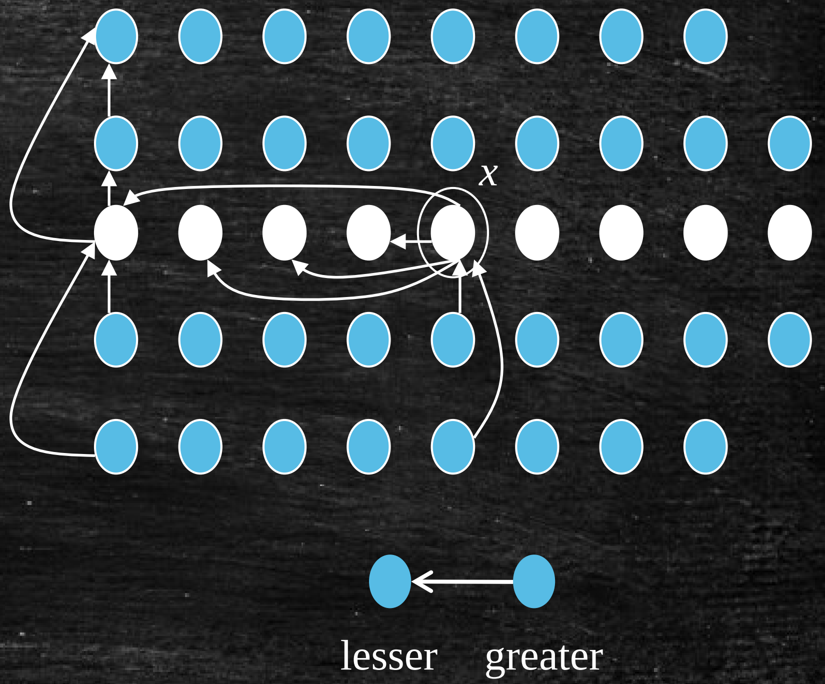
return $i + 1$



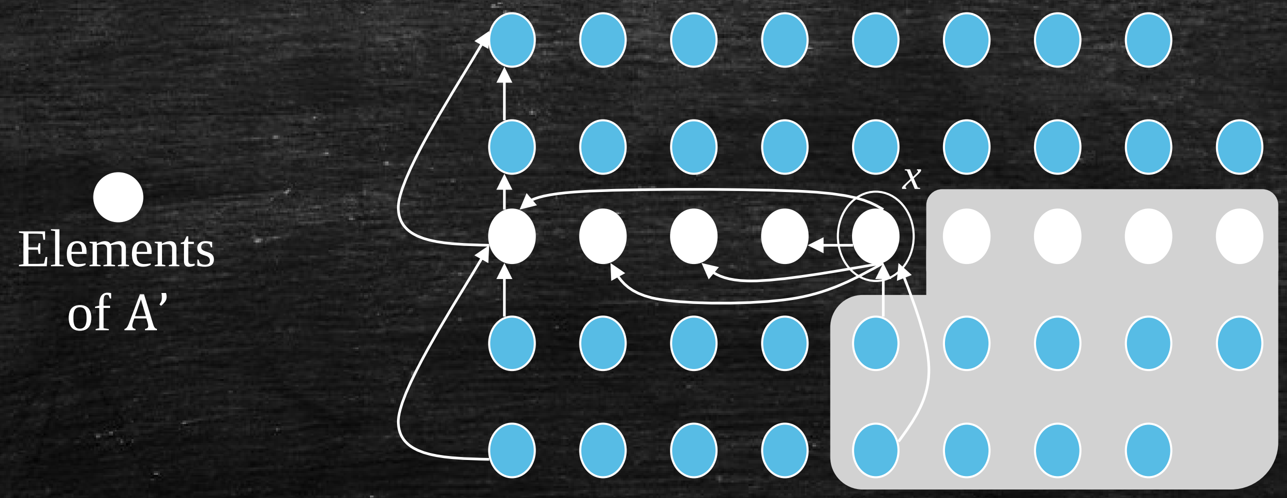
Worst case linear time selection

Select(A,p,r,i)

1. Divide **A** into $n/5$ groups of size 5.
2. Find the median of each group of 5 by brute force, and store them in a set $\mathbf{A'}$ of size $n/5$.
3. Recursively use **Select**$(\mathbf{A'},\ \mathbf{1},\ n/5, n/10)$ to find the median $\mathbf{x}$ of $n/5$ medians.
4. Partition elements of **A** around $\mathbf{x}$.
   Let k be the order of $\mathbf{x}$ found in the partitioning.
5. if $i = k$
6. return $\mathbf{x}$
7. else if $i < k$
8. **Select**$(\mathbf{A},\ \mathbf{p},\ \mathbf{q} - \mathbf{1},\ i\,)$
9. else
10. **Select**$(\mathbf{A}, \mathbf{q} + \mathbf{1}, \mathbf{r}, i - k)$

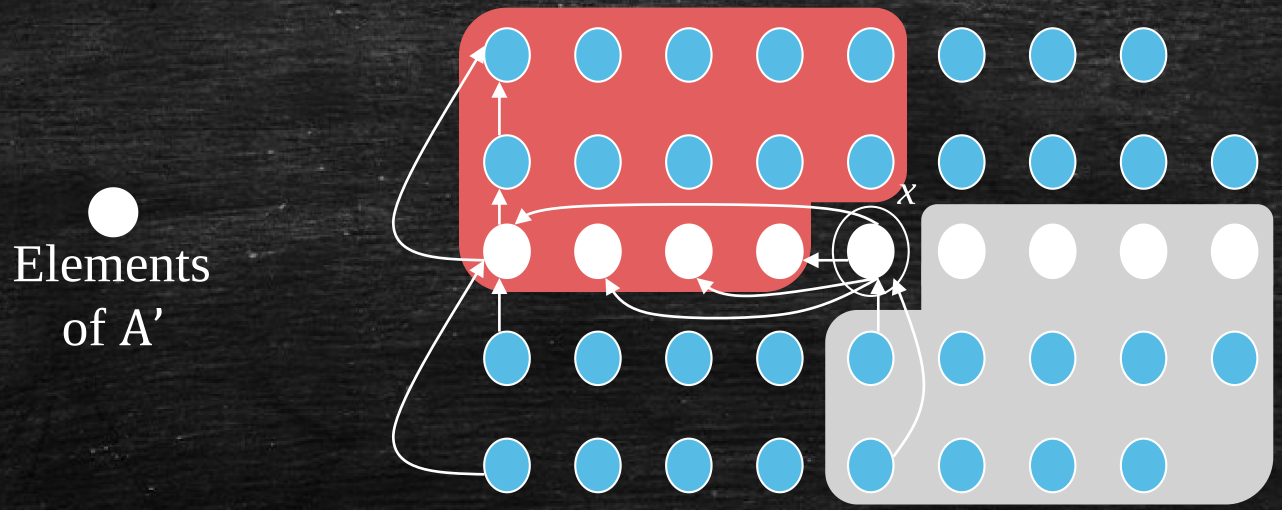


Analysis



- At least **half** of the $\lceil n/5 \rceil$ elements in $\mathbf{A}'$ are $> \boldsymbol{x}$
- Groups whose median $> \boldsymbol{x}$ have at least **3** elements $> \boldsymbol{x}$.
- Therefore, at least $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements of ***A*** are $> \boldsymbol{x}$.

Analysis



- At least **half** of the $\lceil n/5 \rceil$ elements in **A'** are $< x$
- Groups whose median $< x$ have at least **3** elements $< x$.
- Therefore, at least $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements of ***A*** are $< x$.