“Notes” slides from before lecture

CSE 21, Winter 2017, Section A00

Lecture 2 Notes

Class URL: http://vlsicad.ucsd.edu/courses/cse21-w17/
Notes January 11 (1)

- **Schedule** (*with readings*) is up on class website
  - If any issues with OHs, please contact instructors ASAP
- **Notations and conventions** (e.g., 1-based indexing) – Piazza @22 (pinned)
- HW1 was posted on Monday afternoon (due Tuesday, January 17)
- Annotated PDF of Day 1 lecture was replaced yesterday (+ announcement on Piazza) – some added comments, screenshots of python executions, etc.
- In case of interest: “How To Solve It” (Polya) and “The Art and Craft of Problem Solving” (Zeitz) – see http://vlsicad.ucsd.edu/courses/cse101-w16/
- **Today:** Remaining Day 2 + Day 3 posted slides == Sorting and Searching Algorithms
  - **Strong advice:** Please do not fall behind in this class!
- My email: abk@cs.ucsd.edu
  - If you email me about this course, please put [cse21w17] into the subject line!
- My personal home page: http://vlsicad.ucsd.edu/~abk
From "How" to "Why"

What makes this algorithm work?

How do you know that the resulting list will be sorted?

For loop-based algorithms:

What’s the effect of each loop iteration on the list?

Have we made progress?

(We’ll start here on Wednesday. Please review slides through end of Day 3!)

(i.e., iterative)
A **loop invariant** is a property that remains true after each time the body of a loop is executed.

For an iterative algorithm:

1. **Look for a loop invariant**
   - State the property precisely

2. **Prove that it is invariant**
   - It must be true after any number of loop iterations

3. **Use the invariant to prove correctness**
   - Show that when the loop is finished, the invariant guarantees that we've reached a solution
Selection Sort (MinSort)

"Find the first name alphabetically, move it to the front. Then look for the next one, move it, etc."
Selection Sort (MinSort) Illustration

Comment: From python execution in class (should be on podcast).

```python
[54, 26, 93, 17, 77, 31, 44, 55, 20]  <-- input list
[17, 26, 93, 54, 77, 31, 44, 55, 20]
[17, 20, 93, 54, 77, 31, 44, 55, 26]
[17, 20, 26, 54, 77, 31, 44, 55, 93]
[17, 20, 26, 31, 77, 54, 44, 55, 93]
[17, 20, 26, 31, 44, 54, 77, 55, 93]
[17, 20, 26, 31, 44, 54, 77, 55, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

- `i`  
- `min`  
- `swap`
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

procedure selection sort(a1, a2, ..., an: real numbers with n >=2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (a_j < a_m) then m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order }

What is the loop invariant for this iterative algorithm?
Selection Sort (MinSort) Correctness: WHY

1. **Loop invariant**: After the $k^{th}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.
Selection Sort (MinSort) Correctness: WHY

1. **Loop invariant:** [After the $k^{th}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.]

2. How can we show that this loop invariant is true?

3. Once we do, why can we conclude that the program is correct?
Selection Sort (MinSort) Correctness: WHY

1. **Loop invariant:** After the $k^{\text{th}}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.

2. How can we show that this loop invariant is true?

3. Once we do, why can we conclude that the program is correct?

*Start with this (quicksort)*
Selection Sort (MinSort) Correctness: WHY

Loop invariant: After the $k^{th}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.

Therefore,

How would you use the loop invariant to prove the correctness of MinSort?

- We execute loop exactly $n$ times
- Loop invariant with $k = n-1$ $\rightarrow$ all $n$ elements of list will be in correct order (at end of execution, as we require)
Loop invariant: After the $k$th time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.

How can we show that this loop invariant is true?

Once we do, why can we conclude the program is correct?
Loop invariant: After the $k^{th}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.

How can we show that this loop invariant is true?

2 Induction...... What variable do we induct on?
Induction variable (k): *the number of times through the loop.*

Base Case: It’s true when k=0 (before the loop.)

Induction Step: If it’s ever true, then going through the loop one more time keeps it true.

2

- After Ø times through loop
- First Ø elements (Ø) are the Ø smallest list elements in order

(Weak induction)

(aka “ordinary”)
Induction variable (k): *the number of times through the loop*.

**Base Case:** It’s true when \( k = 0 \) (before the loop.)

**Induction Step:** If it’s ever true, then going through the loop one more time keeps it true.

Therefore: It’s true for all values of \( k \geq 0 \).
Structure of the Induction Proof

Induction variable (k): the number of times through the loop.

**Base Case:** It's true when k=0 (before the loop.)

**Induction Step:** If it's ever true, then going through the loop one more time keeps it true.

Therefore: It's true for all values of \( k \geq 0 \).
Induction variable (k): the number of times through the loop.

Base Case: It's true when k=0 (before the loop.)

Induction Step: If it's ever true, then going through the loop one more time keeps it true.

Therefore: It's true for all values of k\geq0.

0 \rightarrow 1

k-1 \rightarrow k \quad \text{OR} \quad k \rightarrow k+1
**Structure of the Induction Proof**

Induction variable (k): *the number of times through the loop.*

**Base Case:** It's true when k=0 (before the loop.)

**Induction Step:** If it's ever true, then going through the loop one more time keeps it true.

**Therefore:** It's true for all values of k≥0.
Structure of the Induction Proof

Induction variable (k): the number of times through the loop.

Base Case: It's true when k=0 (before the loop.)

Induction Step: If it's ever true, then going through the loop one more time keeps it true.

Therefore: It's true for all values of k\geq0.
Structure of the Induction Proof

**Induction variable (k):** the *number of times through the loop.*

**Base Case:** It's true when \( k=0 \) (before the loop.)

**Induction Step:** If it's ever true, then going through the loop one more time keeps it true.

Therefore: It's true for all values of \( k \geq 0 \).
Statement: After the $k^{th}$ time through the outer loop, the first $k$ elements of the list are the $k$ smallest list elements in order.

Induction variable (k): the number of times through the loop.

Base case: Need to show the statement holds for $k=0$, before the loop.

Inductive step: Let $k$ be a positive integer.

Induction hypothesis: Suppose the statement holds after $k-1$ times through the loop.

Need to show that the statement holds after $k$ times through the loop.
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

Q: What happens in the $k^{th}$ iteration?

procedure selection sort($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $i := 1$ to $n-1$
    $m := i$
    for $j := i+1$ to $n$
        if ($a_j < a_m$) then $m := j$
    interchange $a_i$ and $a_m$

{ $a_1, \ldots, a_n$ is in increasing order }
Proving Loop Invariants

Induction variable (k): the number of times through the loop.

Base case: Need to show the statement holds for k = 0 before the loop.

Inductive step: Let k be a positive integer.

Induction hypothesis: Suppose the statement holds after k - 1 times through the loop.

Need to show that the statement holds after k times through the loop.
Why sort?

A TA facing a stack of exams needs to input all 400 scores into a spreadsheet where the students are listed in alphabetical order.

OR

You want to find all the duplicate values in a long list.

It's easier to access data when it is sorted because you know exactly where to find it.

Really???
Sorting helps with searching

Two searching algorithms:

One that works for any data, sorted or not

One that is much faster, but relies on the data being sorted

More on this next time!
End of Day 2 (MWF schedule)
Start of Day 3 (not repeated from Day 2)  
(MWF schedule)
Searching Algorithms

CSE21 Winter 2017, Day 3 (B00), Day 2 (A00)

January 13, 2017
Why sort?

A TA facing a stack of exams needs to input all 400 scores into a spreadsheet where the students are listed in alphabetical order.

OR

You want to find all the duplicate values in a long list.

It's easier to access data when it is sorted because you know exactly where to find it.

Really???
Two searching algorithms:

- One that works for any data, sorted or not
- One that is much faster, but relies on the data being sorted
Given a list $a_1, a_2, \ldots, a_n$ and a target value $x$ find an index $j$ where $x = a_j$ or determine that there is no such index.

(e.g., return $\emptyset$)

Values can be any type. For simplicity, use integers.

Example: $4, 2, 7, 3, 6$

$x = 7$

Value returned: $3$
**Pseudocode:**

```
procedure mystery (x: integer, a_1, a_2, ..., a_n: distinct integers )
    i := 1
    while (i <= n and x ≠ a_i)
        i := i+1
    if i <= n then location := i
    else location := 0
    return location
```

\{ location is the subscript of the term that equals x, or is 0 if x is not found \}

**Scan (linear search)**

- If \( x = a_i \) return \( i \)
- \( i > n \) return \( \emptyset \)
- Stops when either

In groups, describe in English (high-level), what the algorithm is doing.
Correctness of Linear Search: WHY

What's the loop invariant?

After 1 iteration (through loop), ____________
After 2 iterations, ____________
After k iterations, ____________

Try to write it down yourself, first.

X is not equal to any of the first k elements of the list
Step 1: Loop invariant is:
"After the $t^{th}$ iteration of the while loop, guarantee that $x$ is not equal to any of the first $t$ entries of the list."

Step 2: Prove that this loop invariant holds.

Step 3: Use the invariant to prove that linear search is correct.
Using order in search

**Linear Search:**
Starting at the beginning of the list, compare items one by one with $x$ until find it or reach the end.

✗ We didn't assume anything about the list!!!
Using order in search

Linear Search:
Starting at the beginning of the list, compare items one by one with \( x \) until find it or reach the end.

We didn't assume **anything** about the list!!!

Can we do better if the list is sorted?

A. Yes, we can modify Linear Search to take advantage of sorted order.
B. Yes, we can devise a totally new algorithm which uses the sorted order.
C. Yes, even though we might always need to look at all list elements in the worst case, we'll be able to do better on average.
D. No.
Suppose our list is sorted. If we probe the list at position \( m \), what do we learn if ... 

\[ x = a_m \]  

\( x = a_m \) ? We're done!

\( \text{return } m \)

i.e., look at
Using order in search

Suppose our list is sorted. If we probe the list at position \( m \), what do we learn if …

\[ x = a_m \] ? We're done!

\[ x < a_m \] ?

A. if \( x \) is in the list, it occurs BEFORE position \( m \)
B. if \( x \) is in the list, it occurs AFTER position \( m \)
C. \( x \) occurs at position \( m \)
D. \( x \) is not in the list
Using order in search

Suppose our list is sorted. If we probe the list at position $m$, what do we learn if …

- $x = a_m$ ? We’re done!
- $x < a_m$ ? Need to check positions $1 \ldots m-1$
- $x > a_m$ ? Need to check positions $m+1 \ldots n$
Using order in search

Suppose our list is sorted. If we probe the list at position $m$, what do we learn if ...

- $x = a_m$? We're done!
- $x < a_m$? Need to check positions 1 … $m-1$
- $x > a_m$? Need to check positions $m+1$ … $n$

Have we made any progress? (yes)
Binary Search: HOW

Starting at the middle of the list and based on what you find, determine which half to search next. Continue until target is found or sure that it is missing.

\[
\text{procedure } \text{binary search} \ (x: \text{integer}, \ a_1, \ a_2, \ \ldots, \ a_n: \text{increasing integers})
\]

location is the subscript of the term that equals x, or is 0 if x is not found.
Starting at the middle of the list and based on what you find, determine which half to search next. Continue until target is found or sure that it is missing.

procedure binary search (x: integer, a₁, a₂, ..., aₙ: increasing integers )
i := 1
j := n
while ????
m := floor((i+j)/2)
if x > aₘ then i := m+1
else j := m
if x=aᵢ then location := i
else location := 0
return location

{ location is the subscript of the term that equals x, or is 0 if x is not found }
Binary Search: HOW

Starting at the middle of the list and based on what you find, determine which half to search next. Continue until target is found or sure that it is missing.

procedure binary search (x: integer, a₁, a₂, ..., aₙ: increasing integers )
i := 1
j := n
while ????
m := floor( (i+j)/2 )
if x > aₘ then i := m+1
else j := m
if x=aᵢ then location := i
else location := 0
return location

{ location is the subscript of the term that equals x, or is 0 if x is not found

What is “???”

i<n
j<n
i<j
j<i

None of the above

if i<j, 2 or more elements under consideration.

x=5
A = 1, 3, 4, 4, 5, 7, 8
m = ⌊(1+7)/2⌋ = 4
x > a₄ (5>4)
procedure binary search (x: integer, a1, a2, ..., an: increasing integers)
i := 1
j := n
while i<j
  m := floor((i+j)/2)
  if x > am then i := m+1
  else j := m
if x=a_i then location := i
else location := 0
return location

{ location is the subscript of the term that equals x, or is 0 if x is not found }

How do we prove this? Informally, the list to consider gets smaller and ai <= x <= aj
General questions to ask about algorithms

1. **What** problem are we solving? **SPECIFICATION**
2. **How** do we solve the problem? **ALGORITHM DESCRIPTION**
3. **Why** do these steps solve the problem? **CORRECTNESS**
4. **When** do we get an answer? **RUNNING TIME PERFORMANCE**

\[
\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc
\]

Gaussian elimination: \( O(n^3) \) upper triangular: \( \det A = \prod_{i=1}^{n} A_{i,i} \)

\( \det (A_{i,j}) \) ... (recursive)

\( n! \)
Counting comparisons: WHEN

Measure ...

Time

Number of operations

Comparisons of list elements!

For selection sort (MinSort), how many times do we have to compare the values of some pair of list elements?
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_m ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}
(Nested For Loops Examples)

```plaintext
for i := 1 to 5
    for j := 1 to 4
        ( . . . )

How many times is (...) executed?

20 = 5 * 4
```

```plaintext
for i := 1 to 5
    for j := 1 to i
        ( . . . )

How many times is (...) executed?

15 = 1 + 2 + 3 + 4 + 5
```
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

**Pseudocode**

```plaintext
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >= 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (a_j < a_m) then m := j
    interchange a_i and a_m
{ a_1, ..., a_n is in increasing order }
```

In the body of the outer loop, when i=1, how many times do we compare pairs of elements?

A. 1
B. n
C. n-1
D. n+1
E. None of the above.
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_m ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n is in increasing order}
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
    for i := 1 to n-1
        m := i
        for j := i+1 to n
            if (a_j < a_m) then m := j
        interchange a_i and a_m
    \{ a_1, ..., a_n is in increasing order\}

For each value of i, compare (n-i) pairs of elements.

(n-1) + (n-2) + ... + (1)
Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

```
procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n ≥ 2)
  for i := 1 to n-1
    m := i
    for j := i+1 to n
      if (a_j < a_m) then m := j
    interchange a_i and a_m

{ a₁, ..., aₙ is in increasing order }
```

For each value of i, compare
(n-i) pairs of elements.

Sum of positive integers up to (n-1)
procedure selection sort(a₁, a₂, ..., aₙ: real numbers with n ≥ 2)
for i := 1 to n-1
    m := i
    for j := i+1 to n
        if (a_j < a_m) then m := j
    interchange aᵢ and aᵣ

{ a₁, ..., aₙ is in increasing order}

For each value of i, compare (n-i) pairs of elements.

\[ \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} \]
by induction

Prove

Sum of positive integers up to (n-1)

\[ (n-1) + (n-2) + \ldots + 1 \]
\[ = n(n-1)/2 \]
Announcements

Things to do:
- Sign up on Gradescope
- Enroll in and attend your discussion section (A00 Thursday unless you have course conflict)
- Sign up on Piazza

HW1 due
Tuesday 1/17 at 11:59PM.
One submission/group

FRI 7pm
CSB002 session on invariants, induction

Office Hours
See the calendar on the class website.

On course website:
- Guide to proofs and logic
- Guide to correctness proofs
- Extra practice problems on invariants and induction
End of Day 3 (MWF schedule)
Runtime performance

Goal:

Ignore what we can't control

Focus on how time scales for large inputs

Estimate time as a function of the size of the input, n

Will see these slides next time… !
Please do your reading !
Ignore what we can't control

Rate of growth

Focus on how time scales for large inputs

Which of these functions do you think has the "same" rate of growth?

A. All of them
B. $2^n$ and $n^2$
C. $n^2$ and $3n^2$
D. They're all different
Definition of Big O

Ignore what we can't control

Focus on how time scales for large inputs

For functions $f(n) : \mathbb{N} \to \mathbb{R}, g(n) : \mathbb{N} \to \mathbb{R}$ we say

$$f(n) \in O(g(n))$$

to mean there are constants, $C$ and $k$ such that

$$|f(n)| \leq C|g(n)|$$

for all $n > k$. 

Rosen p. 205
For functions $f(n) : \mathbb{N} \to \mathbb{R}, g(n) : \mathbb{N} \to \mathbb{R}$, we say $f(n) \in O(g(n))$ to mean there are constants, $C$ and $k$ such that $|f(n)| \leq C|g(n)|$ for all $n > k$. 

Rosen p. 205