

Practice

Table of Contents

<i>Time Efficiency</i>	3
<i>Time Efficiency: Fibonacci 1</i>	4
<i>Time Efficiency: Multiplication</i>	5
<i>Time Efficiency: Fibonacci 2</i>	6
<i>Time Efficiency: Fibonacci 3</i>	7
<i>Time Efficiency: Selection Sort</i>	8
<i>Modular Simplification</i>	9
<i>Proof: Multiplicative Inverse</i>	11
<i>Proof: Recurrence Correctness 1</i>	13
<i>Proof: Recurrence Correctness 2</i>	14
<i>Proof: Recurrence Correctness 3</i>	16
<i>Proof: Master Theorem Correctness</i>	18
<i>Proof: Greedy Choice</i>	21
<i>Graph Algorithm 1</i>	23
<i>Graph Algorithm 2</i>	25
<i>Proof: Directed Acyclic Graph (DAG) Algorithm</i>	26
<i>Proof: Depth First Search (DFS) Algorithm</i>	27
<i>Proof: Shortest Path Algorithm</i>	28
<i>Proof: Dijkstra's Algorithm</i>	29
<i>Proof: Bellman-Ford Algorithm</i>	30
<i>Minimum Spanning Tree 1</i>	32
<i>Minimum Spanning Tree 2</i>	33

Practice

<i>Min-Cut and Optimal Substructure</i>	<i>34</i>
<i>Optimal Substructure Algorithm</i>	<i>36</i>
<i>Time Efficiency: k-ary Search.....</i>	<i>37</i>
<i>Time Efficiency: Binary Search</i>	<i>39</i>
<i>Integer Linear Program: Vertex Cover.....</i>	<i>41</i>
<i>Integer Linear Program: Algorithm.....</i>	<i>42</i>

Practice

TIME EFFICIENCY

$f(n)$	$g(n)$	$O/\Omega/\Theta$
$n - 100$	$n - 200$	Θ
$n^{1/2}$	$n^{2/3}$	O
$100n + \log n$	$n + (\log n)^2$	$\Theta (a)$
$\log 2n$	$\log 3n$	$\Theta (b)$
$10 \log n$	$\log n^2$	$\Theta (c)$
$n^{1/2}$	$5^{\log_2 n}$	$O (d)$
2^n	2^{n+1}	$\Theta (e)$

(a): n dominates $(\log n)^c \rightarrow n + (\log n)^2 = \Theta(n)$

(b): $\log ab = \log a + \log b$

(c): $\log a^b = b \log a$

(d): $5 = 2^{2x}$ where $x > 0 \rightarrow 5^{\log_2 n} = (2^{2x})^{\log_2 n} = (2^{\log_2 n})^{2x} = \Omega(n^{1/2})$

(e): $2^{n+1} = 2 \times 2^n$

Practice

TIME EFFICIENCY: FIBONACCI 1

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{otherwise} \end{cases}$$

Prove: $F_n = \Omega(\sqrt{2^n})$.

By trial and error: It appears that $F(n) \geq 2^{n/2}$ for all $n \geq 7$

To prove: For all positive integers $n \geq 7 \rightarrow F_n \geq 2^{n/2}$

By induction on n . Base case: $n = 7$

Step, assume: Indeed, true that for all $i = 7, 8, \dots, k \rightarrow F_i \geq 2^{i/2}$

To prove: $F_{k+1} \geq 2^{(k+1)/2}$

LHS: $F_{k+1} = F_k + F_{k-1} \geq 2^{k/2} + 2^{(k-1)/2}$

Suffices to prove: $2^{k/2} + 2^{(k-1)/2} \geq 2^{(k+1)/2}$

$2^{1/2} + 1 \geq 2^{2/2}$ (by dividing the above by $2^{(k-1)/2}$)

It is indeed true that $2^{1/2} + 1 \geq 2^{2/2} = 2$

Practice

TIME EFFICIENCY: MULTIPLICATION

Figure 1.1 Multiplication à la Français.

```
function multiply(x, y)
```

Input: Two n -bit integers x and y , where $y \geq 0$

Output: Their product

```
if  $y = 0$ : return 0
```

```
 $z = \text{multiply}(x, \lfloor y/2 \rfloor)$ 
```

```
if  $y$  is even:
```

```
    return  $2z$ 
```

```
else:
```

```
    return  $x + 2z$ 
```

Suppose instead of both x and y being n -bit, x is n -bit and y is m -bit. What is the worst-case time efficiency of *multiply*?

Proposed: $O(nm)$

Time Efficiency:

- # recursive calls \times time/call
- # worst case recursive calls = $O(m)$
- Worst case time/call =
 - $2z$ is at worst $O(n + m) \rightarrow$ because very last addition is $2z = xy - x$
 - x is n bits
 - So, addition's time: $O(\max\{n, n + m\}) = O(\max\{n, m\})$

So, final answer: $O(m \times \max\{n, m\})$

Practice

TIME EFFICIENCY: FIBONACCI 2

Let F_n be the n^{th} Fibonacci number, Prove $F_n = O(2^n)$.

- Somewhere, we have shown: $F_n = \Omega(\sqrt{2}^n)$
- But here, seek to show: There exists positive real $F_n \leq c \cdot 2^n$, for all n in N
- Natural proof strategy for “there exists” – construction (i.e., propose some concrete c , and show that it works)
- Try some small values for n , and see what c would work
 - $n = 0, F_0 = 0, 2^0 = 1 \rightarrow c = 1 \text{ works}$
 - $n = 1, F_1 = 1, 2^1 = 2 \rightarrow c = 1 \text{ works}$
 - $n = 2, F_2 = 1, 2^2 = 4 \rightarrow c = 1 \text{ works}$
 - $n = 3, F_3 = 2, 2^3 = 8 \rightarrow c = 1 \text{ works}$
 - $n = 4, F_4 = 3, 2^4 = 16 \rightarrow c = 1 \text{ works}$
- Appears that $c = 1$ works. Adopt it and check if proof goes through. Now, proof by induction with $c = 1$
- Base case, $n = 1, F_1 = 1, 2^1, 1 \leq 2 \rightarrow \text{True}$
- Step: Seek to show $F_n \leq 2^n$ given that $F_k \leq 2^k$ for all $k = 1, 2, \dots, n - 1$
- $F_n = F_{n-1} + F_{n-2} \leq 2^{n-1} + 2^{n-2}$ by induction assumption
- $F_n = 2^{n-2} (2 + 1) = 3 \times 2^{n-2} \leq 2^n = 2^2 \times 2^{n-2} = 4 \times 2^{n-2} \rightarrow \text{Done}$

Practice

TIME EFFICIENCY: FIBONACCI 3

Let F_n be the n^{th} Fibonacci number, Prove $F_n \neq O(n^2)$.

- Recall from logic: not (there exists an egg-laying mammal) = for all mammals m , m is not egg-laying
- Here, $f = O(g)$: There exists positive real c , for all natural n , $f(n) \leq c \cdot g(n)$
- So here, need to prove: Given any positive real c , it is true that there exists n such that $F_n > c \cdot n^2$
- By contradiction: Suppose that there exists positive real c , such that, for all natural n , $F_n \leq c \cdot n^2$
- Then: $F_n = F_{n-1} + F_{n-2} \leq c(n-1)^2 + c(n-2)^2 = c(n^2 - 2n + 1 + n^2 - 4n + 4) = c(2n^2 - 6n + 5) \leq cn^2$
- $2n^2 - 6n + 5 \leq n^2$
- $2 - \frac{1}{n^2}(6n - 5) \leq 1$
- This is true only if $\frac{1}{n^2}(6n - 5)$ is “large” compared to $2n^2$
- What is large? We need $\frac{1}{n^2}(6n - 5) \geq 1 \rightarrow \text{true for } n = 1$
- Try $n = 2$: $\frac{1}{4}(12 - 5) = \frac{7}{4} \geq 1$
- Try $n = 3$: $\frac{1}{8}(18 - 5) = \frac{13}{8} \geq 1$
- Try $n = 4$: $\frac{1}{16}(24 - 5) = \frac{19}{16} \geq 1$
- Try $n = 5$: $\frac{1}{25}(30 - 5) = 1$
- Try $n = 6$: $\frac{1}{36}(36 - 5) < 1$
- Try $n = 7$: $\frac{1}{49}(42 - 5) < 1$
- Prove by induction: $6n - 5 < n^2$ for all natural $n > 5$
- Base case $n = 6$: See above
- Step: $6(n-1) - 5 \leq (n-1)^2 \rightarrow \text{from induction assumption}$
- $6n - 5 - 6 < n^2 - 2n + 1$
- $6n - 5 \leq n^2 - (2n - 7) \leq n^2 \text{ whenever } 2n - 7 \geq 0 \rightarrow \text{which it is for } n \geq 6$
- So far: We have shown that indeed, for $n \geq 6$, $F_n < cn^2 \rightarrow \text{Done}$

Practice

TIME EFFICIENCY: SELECTION SORT

```
SELECTIONSORT( $A[1, \dots, n]$ )
  foreach  $i$  from 1 to  $n$  do
     $m \leftarrow i - 1 + \text{INDEXOFMIN}(A[i, \dots, n])$ 
    if  $i \neq m$  then swap  $A[i], A[m]$ 

    INDEXOFMIN( $B[1, \dots, m]$ )
       $\text{min} \leftarrow B[1], \text{idx} \leftarrow 1$ 
      foreach  $j$  from 2 to  $m$  do
        if  $B[j] < \text{min}$  then
           $\text{min} \leftarrow B[j], \text{idx} \leftarrow j$ 
      return  $\text{idx}$ 
```

What is a meaningful characterization of the time efficiency of *SELECTIONSORT*?

- Suppose we invoke $\text{INDEXOFMIN}(A[5, \dots, 13])$. In $\text{INDEXOFMIN}: B[1, \dots, 9]$. Suppose now, min is at index 3 in $B[1, \dots, 9]$. This \rightarrow index of a min in $A[5, \dots, 13]$ is at index $(5 - 1) + 3 = 7$
- Suppose on input: $A[1, \dots, 5] = [13, -23, 45, -23, 1]$. Then A evolves in *SELECTIONSORT* as follows:
 - $i = 1, m = 2, [-23, 13, 45, -23, 1]$
 - $i = 2, m = 4, [-23, -23, 45, 13, 1]$
 - $i = 3, m = 4, [-23, -23, 13, 45, 1]$
- For time efficiency: Need to make meaningful assumption(s)
- Customary Assumptions: (1) n is unbounded, (ii) each $A[i]$ is bounded
- What should we count? Suppose we all agree that counting # swaps is a meaningful measure for time efficiency
- Then: *Worst case # swaps* $= n - 1 = \Theta(n)$
- Now, let's say we want to get a bit more fine-grained. Incorporate (worst case) time for each swap x # swaps
- So now, time efficiency: $(n - 1) + (n - 2) + \dots + 1 = \Theta(n^2)$

Practice

MODULAR SIMPLIFICATION

1. Is $6^6 \equiv 5^3 \pmod{31}$?

$$6 \times 6 = 36 \equiv 5 \pmod{31}$$

$$\text{So: } (6^2)^3 \equiv (5)^3 \pmod{31}$$

2. $2^{125} \equiv ? \pmod{127}$

$$2^7 = 128 = 127 + 1$$

$$\text{So: } 128 \pmod{127} = 1$$

$$\text{Now: } 125/7 = 17 + 6/7$$

$$\text{So: } 2^{125} = 2^{17 \times 7 + 6} = 2^{17 \times 7} \times 2^6$$

$$\text{So: } 2^{125} \equiv 2^{17 \times 7} \times 2^6 \equiv (2^7)^{17} \times 2^6 \equiv 1^{17} \times 2^6 \equiv 64 \pmod{127}$$

3. Is $4^{1536} - 9^{4824}$ divisible by 35?

$$4^{1536} \equiv 9^{4824} \pmod{35}$$

Trick: Keep exponentiating until numbers start to repeat.

Suppose we repeatedly exponentiate 4:

$$4$$

$$\rightarrow 16$$

$$\rightarrow 64 \equiv 29 \pmod{35}$$

$$\rightarrow 116 = 35 \times 3 + 11 = 11 \pmod{35}$$

$$\rightarrow 9 \pmod{35}$$

$$\rightarrow 36 \equiv 1 \pmod{35}$$

$$\text{So: } 4^6 \equiv 1 \pmod{35}. \text{ And } 1536 = 6 \times 256. \text{ So } 4^{1536} \equiv 1 \pmod{35}$$

Now check whether 1536 is divisible by 4. Indeed: $1536 = 4 \times 384$

Repeat with 9. Repeated exponentiation of 9:

$$9$$

Practice

$$\rightarrow 81 \equiv 11 \pmod{35}$$

$$\rightarrow 99 \equiv 29 \pmod{35}$$

$$\rightarrow 261 = 7 \times 35 + 16 \equiv 16 \pmod{35}$$

$$\rightarrow 144 \equiv 4 \times 35 + 4 \equiv 4 \pmod{35}$$

$$\rightarrow 36 \equiv 1 \pmod{35}$$

$$\text{So: } 9^6 \equiv 1 \pmod{35}$$

$$\text{Now: } 9^{4824} = 9^{804 \times 6} \equiv 1 \pmod{35}.$$

\therefore It is divisible by 35.

4. $2^{2^{2006}} \pmod{3} = ?$

$$2^{2^{2006}} = (2^2)^{2^{2005}} = 4^{2^{2005}} \equiv 1 \pmod{3}$$

5. Is $5^{30000} - 6^{123456}$ a multiple of 31?

$$31 \text{ is prime. And } 5^{30000} = (5^{30})^{1000} \equiv 1 \pmod{31}.$$

$$\text{Compare with } 6^{123456} = 6^{123450} \times 6^6:$$

$$1 \times 6^6 \equiv 5^3 \equiv 125 \equiv 31 \times 4 + 1 \pmod{31} \equiv 1 \pmod{31}$$

\therefore It is a multiple of 31.

Practice

PROOF: MULTIPLICATIVE INVERSE

Show that if a has a multiplicative inverse modulo N , then this inverse is unique (modulo N).

Let's assume $a \in \{1, \dots, N - 1\}$.

Suppose $b, c \in \{1, \dots, N - 1\}$ are both multiplicative inverses of a modulo N . Then:

$$ab \equiv 1 \pmod{N}$$

$$ac \equiv 1 \pmod{N}$$

$$ab \equiv ac \pmod{N}$$

$$ab \cdot b \equiv ac \cdot b \pmod{N} \quad (1)$$

(1): Substitution Rule:

$$x \equiv x', y \equiv y' \pmod{N}$$

$$xy \equiv x'y' \pmod{N}$$

Then:

$$(ab) \cdot b \equiv (ab) \cdot c \pmod{N} \quad (2)$$

(2): Commutativity

$$1 \cdot b \equiv 1 \cdot c \pmod{N}$$

$$b \equiv c \pmod{N}$$

$$b = c$$

Suppose $p \equiv 3 \pmod{4}$. Show that $(p + 1)/4$ is an integer.

$$p \equiv 3 \pmod{4}$$

$$p = 4k + 3 \text{ for some } k \in \mathbb{Z}$$

So: $p + 1 = 4k + 4$, which is divisible by 4.

Practice

We say that x is a square root of y modulo a prime p if $y \equiv x^2 \pmod{p}$. Show that if (i) $p \equiv 3 \pmod{4}$ and (ii) y has a square root modulo p , then $y^{(p+1)/4}$ is such a square root.

Let x be the square root of y modulo p . Then: $y \equiv x^2 \pmod{p}$.

Write $p = 4k + 3$. Then, $\left(y^{\frac{p+1}{4}}\right)^2 = y^{2(p+1)/4} = y^{2(4k+3+1)/4} = y^{2k+2}$

Keep in mind: $(p + 1)/4 = k + 1$.

Try plugging in x in the last expression:

Is $y^{2k+2} = x^{4k+4} \equiv x^2 \pmod{p}$?

So, we're asking: Is $x^{4k+4} - x^2 \equiv 0 \pmod{p}$?

$$x^{4k+4} - x^2 = (x^{2k+2} - x)(x^{2k+2} + x)$$

So at least one of: $x^{2k+2} - x$ or $x^{2k+2} + x$ must be $\equiv 0 \pmod{p}$.

- $\frac{(p+1)}{4} = \frac{(4k+3+1)}{4} = k + 1$
- $2 \cdot \frac{(p+1)}{4} = 2k + 2$
- $p - 1 = 4k + 2$

We know: There exists $x \in \{1, \dots, p - 1\}$ such that $y \equiv x^2 \pmod{p}$.

We seek to prove: $\left(y^{\frac{(p+1)}{4}}\right)^2 \equiv y \pmod{p}$. Sufficient condition for that to be true:

$$\left(y^{\frac{(p+1)}{4}}\right)^2 \cdot y^{-1} \equiv 1 \pmod{p} \rightarrow \text{is okay, because } y \text{ is invertible modulo } p$$

$$\Rightarrow (y^{2k+2}) \cdot y^{-1} \equiv 1 \pmod{p}$$

$$\Rightarrow y^{2k+1} \equiv 1 \pmod{p}$$

$$\Rightarrow (x^2)^{2k+1} \equiv 1 \pmod{p}$$

$$\Rightarrow x^{4k+2} \equiv 1 \pmod{p}$$

$$\Rightarrow x^{p-1} \equiv 1 \pmod{p}$$

$$\Rightarrow \text{True (Fermat's little theorem)}$$

Practice

PROOF: RECURRENCE CORRECTNESS 1

Suppose $x \in \mathbb{Z}^+, y \in \mathbb{Z}_0^+$. Prove recurrence correctness.

$$x^y = \begin{cases} 1, & \text{if } y = 0 \\ (x^2)^{\lfloor y/2 \rfloor}, & \text{if } y \text{ is even} \\ x \cdot (x^2)^{\lfloor y/2 \rfloor}, & \text{otherwise} \end{cases}$$

Case Analysis:

1. If $y = 0$, then $x^y = x^0$. So, the recurrence is correct for the case where $y = 0$
2. If $y \neq 0, y \text{ is even}$: then $\lfloor y/2 \rfloor = y/2$. So $x^y = x^{2 \times y/2} = (x^2)^{y/2} = (x^2)^{\lfloor y/2 \rfloor}$
3. If $y \neq 0, y \text{ is odd}$: then $\lfloor y/2 \rfloor = (y - 1)/2$. So now:

$$x^y = x^{(2 \times (y-1)/2) + 1} = x^{(2 \times \lfloor y/2 \rfloor) + 1} = x \cdot x^{2 \times \lfloor y/2 \rfloor}$$

Practice

PROOF: RECURRENCE CORRECTNESS 2

Let $\langle q, r \rangle$ be the quotient and remainder of x/y and $\langle q', r' \rangle$ be the quotient and remainder of $(\lfloor x/2 \rfloor)/y$. Prove recurrence correctness.

$$\langle q, r \rangle = \begin{cases} \langle 0, 0 \rangle, & \text{if } x = 0 \\ \langle 2q', 2r' \rangle, & \text{if } x \text{ even and } 2r' < y \\ \langle 2q', 2r' + 1 \rangle, & \text{if } x \text{ odd and } 2r' + 1 < y \\ \langle 2q' + 1, 2r' - y \rangle, & \text{if } x \text{ even and } 2r' \geq y \\ \langle 2q' + 1, 2r' + 1 - y \rangle, & \text{otherwise} \end{cases}$$

To be absolutely clear, what are the quotient and remainder of x/y ?

We call q the quotient, and r the remainder if and only if q and r are non-negative integers that satisfy:

$$x = q \cdot y + r, \text{ where } r \in \{0, 1, \dots, y - 1\}$$

Proof by case analysis:

1. If $x = 0$, then $x = 0 = 0 \cdot y + 0$. So, recurrence is correct for this case.
2. If x is even and $2r' < y$: then $\lfloor x/2 \rfloor = x/2$. So:

$$\lfloor x/2 \rfloor = x/2 = q' \cdot y + r'$$

$$x = (2q') \cdot y + 2r'$$

$$q = 2q', r = 2r'$$

Where we infer the last line from the facts that: (i) equation is of the form from definition for quotient and remainder, (ii) $r' \geq 0 \rightarrow 2r' \geq 0$, and (iii) we are given $2r' \leq y - 1$.

3. If x is odd and $2r' + 1 < y$: $\lfloor x/2 \rfloor = (x - 1)/2$

$$\lfloor x/2 \rfloor = (x - 1)/2 = q' \cdot y + r'$$

$$x - 1 = (2q') \cdot y + 2r'$$

$$x = (2q') \cdot y + (2r' + 1)$$

4. x is even, $2r' \geq y$: $\lfloor x/2 \rfloor = x/2$. So:

Practice

$$\lfloor x/2 \rfloor = x/2 = q' \cdot y + r'$$

$$x = (2q') \cdot y + 2r'$$

This is of the form of the definition of quotient and remainder, except that we need to confirm that $2r'$ indeed lies between 0 and $y - 1$. Which it does not necessarily. Actually, we are given that $2r' \geq y$ and therefore not between 0 and $y - 1$. Now we observe:

$$x = (2q') \cdot y + 2r'$$

$$x = (2q' + 1) \cdot y + (2r' - y)$$

Now only question that remains: is it the case that $2r' - y \in \{0, 1, \dots, y - 1\}$?

- Is $2r' - y \geq 0$? Yes, because $2r' \geq y$
- Is $2r' - y \leq y - 1$? Yes, because:

$$r' \leq y - 1$$

$$2r' \leq 2y - 2$$

$$2r' - y \leq y - 2 \leq y - 1$$

5. x odd, $2r' + 1 \geq y$:

$$\lfloor x/2 \rfloor = (x - 1)/2 = q' \cdot y + r'$$

$$x = (2q') \cdot y + (2r' + 1)$$

$$x = (2q' + 1) \cdot y + (2r' + 1 - y)$$

Now:

- $2r' + 1 - y \geq 0$ because $2r' + 1 \geq y$.
- $2r' + 1 - y \leq y - 1$ because:

$$r' \leq y - 1$$

$$2r' + 1 \leq 2y - 1$$

$$2r' + 1 - y \leq y - 1$$

Practice

PROOF: RECURRENCE CORRECTNESS 3

Prove that *BinSearch* is correct.

BinSearch($A[1, \dots, n], lo, hi, i$)

1. **if** $lo \leq hi$ **then**
2. $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
3. **if** $A[mid] = i$ **then return true**
4. **if** $A[mid] < i$ **then return** *BinSearch*($A, mid + 1, hi, i$)
5. **else return** *BinSearch*($A, lo, mid - 1, i$)
6. **else return false**

Above is recursive version of binary search. Iterative version:

BinSearch($A[1, \dots, n], lo, hi, i$)

1. **while** $lo \leq hi$ **do**
2. $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$
3. **if** $A[mid] = i$ **then return true**
4. **if** $A[mid] < i$ **then** $lo \leftarrow mid + 1$
5. **else** $hi \leftarrow mid - 1$
6. **else return false**

Typically, for iterative algorithms, towards correctness, we articulate a *loop invariant*:

Let $lo^{(in)}$ and $hi^{(in)}$ be the values of lo and hi respectively on input. Just before we successfully enter an iteration of the **while** loop of Line (1), it is true that:

$$i \in A[lo^{(in)}, \dots, hi^{(in)}] \rightarrow i \in A[lo, \dots, hi]$$

Going back to the recursive version, what is a correctness property?

Given $A[1, \dots, n]$ an array that is sorted, non-decreasing, lo, hi are each $\in \{1, \dots, n\}$ on input,

BinSearch(A, lo, hi, i) returns:

Practice

- $True \rightarrow (lo \leq hi) \text{ and } (i \in A[lo, \dots, hi])$
- $False \rightarrow \text{either } (lo > hi) \text{ or } (i \text{ is not } \in A[lo, \dots, hi])$

Proof by case analysis:

Case 1: $lo > hi$ on input: then **if** condition of Line (1) evaluates to **false**, and we correctly return **false** in Line (6). Then, this is either from (a) Line (6) without making any recursive calls, or (b) as the return value from a recursive call from one of Lines (4) or (5).

For (b), we first observe that $lo \leq hi$ because the only recursive calls are within the **if** block of Line (1). So, all that remains to be proven is that indeed: $i \notin A[lo, \dots, hi]$.

We prove that by induction on $hi - lo + 1$. Base case: $hi - lo + 1 = 1$. We claim we return **false** within the first recursive invocation. That is, we claim: (i) $mid + 1 > hi$ and $lo > mid - 1$, (ii) $mid = lo = hi$, and (iii) $i \neq A[mid]$.

(ii) easy to prove:

$$hi - lo + 1 = 1$$

$$\Rightarrow lo = hi$$

$$\Rightarrow mid = \left\lfloor \frac{(lo + hi)}{2} \right\rfloor = \left\lfloor \frac{(lo + lo)}{2} \right\rfloor = \left\lfloor \frac{(2 \cdot lo)}{2} \right\rfloor = \frac{2 \cdot lo}{2} = lo = hi$$

(iii) is **true**, because then we would have returned **true** in Line (3).

To prove (i): we simply exploit: $mid = hi = lo$

$$mid = hi \Rightarrow mid + 1 > hi$$

$$mid = lo \Rightarrow mid - 1 < lo$$

So, the algorithm is correct if it returns **false**, and $hi - lo + 1 = 1$.

For the step, we know that on input $lo < hi$. So, we returned **false** in some recursive call. So, all we have to prove to appeal to induction assumption: $hi - (mid + 1) < hi - lo$ and $(mid - 1) - lo < hi - lo$.

Practice

PROOF: MASTER THEOREM CORRECTNESS

Give a closed form solution for the following recurrence. Assume: $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is non-decreasing, $a > 0, b > 1, d \geq 0$.

$$f(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1 \\ a \cdot f\left(\frac{n}{b}\right) + \Theta(n^d), & \text{otherwise} \end{cases}$$

Proposed approach: Inductive “rewriting” of the function f . But first: adopt concrete functions wherever we have $\Theta(\cdot)$, $O(\cdot)$ or $\Omega(\cdot)$. In this case: adopt 1 for $\Theta(1)$, and n^d for $\Theta(n^d)$. Now onto the rewriting:

$$\begin{aligned} f(n) &= a \cdot f\left(\frac{n}{b}\right) + n^d \\ &= a \cdot \left(a \cdot f\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^d\right) + n^d \\ &= a^2 \cdot f\left(\frac{n}{b^2}\right) + a \cdot \left(\frac{n}{b}\right)^d + n^d \\ &= a^2 \left(a \cdot f\left(\frac{n}{b^3}\right) + \left(\frac{n}{b^2}\right)^d\right) + a \cdot \left(\frac{n}{b}\right)^d + n^d \\ &= a^3 \cdot f\left(\frac{n}{b^3}\right) + a^2 \cdot \left(\frac{n}{b^2}\right)^d + a \cdot \left(\frac{n}{b}\right)^d + n^d \\ &= a^3 \cdot f\left(\frac{n}{b^3}\right) + n^d \left(\left(\frac{a}{b^d}\right)^2 + \left(\frac{a}{b^d}\right)^1 + \left(\frac{a}{b^d}\right)^0\right) \\ &= a^4 \cdot f\left(\frac{n}{b^4}\right) + n^d \left(\left(\frac{a}{b^d}\right)^3 + \left(\frac{a}{b^d}\right)^2 + \left(\frac{a}{b^d}\right)^1 + \left(\frac{a}{b^d}\right)^0\right) \\ &\dots \\ &= a^{\log_b n} \cdot f(1) + n^d \cdot \left(\left(\frac{a}{b^d}\right)^{(\log_b n)-1} + \left(\frac{a}{b^d}\right)^{(\log_b n)-2} + \dots + \left(\frac{a}{b^d}\right)^0\right) \\ &= a^{\log_b n} + n^d \cdot \left(\left(\frac{a}{b^d}\right)^{(\log_b n)-1} + \left(\frac{a}{b^d}\right)^{(\log_b n)-2} + \dots + \left(\frac{a}{b^d}\right)^0\right) \end{aligned}$$

To figure out the power of a in that last term:

Power of a is the same as the power of b inside the $f\left(\frac{n}{b^x}\right)$. In other words: what is the power of b , i.e., x for which $\frac{n}{b^x} = 1$? Answer: $\frac{n}{b^x} = 1 \Leftrightarrow n = b^x \Leftrightarrow x = \log_b n$.

Practice

Our next step: Simplify/figure out:

$$S = \left(\frac{a}{b^d}\right)^{(\log_b n)-1} + \left(\frac{a}{b^d}\right)^{(\log_b n)-2} + \dots + \left(\frac{a}{b^d}\right)^0$$

Suppose:

$$\begin{aligned} T &= r^{q-1} + r^{q-2} + \dots + r^0 \\ \Rightarrow r \cdot T &= r^q + r^{q-1} + \dots + r \end{aligned}$$

Now subtract one from the other:

$$\begin{aligned} \Rightarrow T - r \cdot T &= r^0 - r^q \\ \Rightarrow (1 - r) \cdot T &= 1 - r^q \\ \Rightarrow T &= \frac{1 - r^q}{1 - r}, \text{ provided } r \neq 1 \end{aligned}$$

When $r = 1$, how do we figure out what T is? Answer: then, T is:

$$\begin{aligned} T &= 1^{q-1} + 1^{q-2} + \dots + 1^0 \\ &= 1 + 1 + \dots + 1 \rightarrow q \text{ instances of } 1 \\ &= q \end{aligned}$$

So, going back to our S :

$$\begin{aligned} S &= \left(\frac{a}{b^d}\right)^{(\log_b n)-1} + \left(\frac{a}{b^d}\right)^{(\log_b n)-2} + \dots + \left(\frac{a}{b^d}\right)^0 \\ \Rightarrow S &= \frac{1 - \left(\frac{a}{b^d}\right)^{\log_b n}}{1 - \left(\frac{a}{b^d}\right)}, \text{ provided } \frac{a}{b^d} \neq 1 \end{aligned}$$

And:

$$S = \log_b n, \text{ when } \frac{a}{b^d} = 1$$

When is $\frac{a}{b^d} = 1$? Answer: $d = \log_b a$.

So, going back to our $f(n)$: first, the case that $d = \log_b a$.

But even before that: rewrite $a^{\log_b n} = n^{\log_b a}$. Because:

$$x = a^{\log_b n} \Leftrightarrow \log_b x = \log_b a \cdot \log_b n \Leftrightarrow x = n^{\log_b a}$$

Practice

$$f(n) = n^{\log_b a} + n^d \cdot S$$

So, when $d = \log_b a$, $S = \log_b n$. So, in this case:

$$\begin{aligned} f(n) &= n^d + n^d \cdot \log_b n \\ &= \Theta(n^d \cdot \log n) \end{aligned}$$

Onto the other two cases: $d \neq \log_b a$.

$$f(n) = n^{\log_b a} + \dots + n^d \cdot S$$

Before we continue: a closer look at $\left(\frac{a}{b^d}\right)^{\log_b n}$:

$$\begin{aligned} \left(\frac{a}{b^d}\right)^{\log_b n} &= \frac{a^{\log_b n}}{(b^d)^{\log_b n}} \\ &= \frac{n^{\log_b a}}{(b^{\log_b n})^d} \\ &= \frac{n^{\log_b a}}{n^d} \end{aligned}$$

So: when $d \neq \log_b a$

$$S = \frac{1 - \frac{n^{\log_b a}}{n^d}}{1 - \left(\frac{a}{b^d}\right)}$$

So, going back to $f(n)$:

$$\begin{aligned} f(n) &= n^{\log_b a} + n^d \cdot S \\ &= n^{\log_b a} + \frac{1}{1 - \left(\frac{a}{b^d}\right)} \cdot (n^d - n^{\log_b a}) \\ &= c \cdot n^{\log_b a} + c' \cdot n^d, \text{ for positive constants } c, c' \end{aligned}$$

So, if $d > \log_b a$: $f(n) = \Theta(n^d)$

And if $d < \log_b a$: $f(n) = \Theta(n^{\log_b a})$

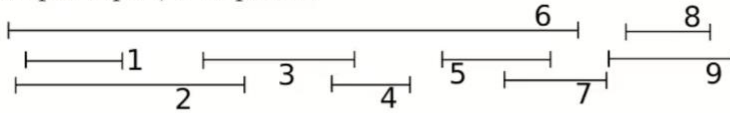
Practice

PROOF: GREEDY CHOICE

Given as input n meeting requests, $\langle s_1, f_1 \rangle, \langle s_2, f_2 \rangle, \dots, \langle s_n, f_n \rangle$, where each $s_i, f_i \in \mathbb{Z}^+$ is a start- and finish-time and $s_i < f_i$. We want a subset of those requests that is of maximum size that are pairwise conflict-free. \hookrightarrow

Two requests $\langle s_i, f_i \rangle, \langle s_j, f_j \rangle$ are in conflict if $s_i \leq f_j$, and $s_j \leq f_i$, or vice versa.

Example input, 9 requests:



Request 5 is in conflict with each Request 6 and 7. But is conflict-free with Request 2.

An optimal (maximum-sized) conflict-free set: $\{1, 3, 5, 9\}$. Another: $\{1, 4, 7, 8\}$.

Prove: this problem possesses a greedy choice.

Candidate greedy choice: request with earliest finish time.

Proof strategy: “cut and paste.”

For this problem, we prove two claims in order:

Claim 1: Suppose for some input of n requests, $O = \{o_1, \dots, o_k\}$ is an optimal (maximum-sized) set of requests which are pairwise conflict-free ordered in increasing finish time. Suppose our greedy algorithm outputs $G = \{g_1, \dots, g_l\}$, ordered in increasing finish time. Then, it is true that: for every $i = 1, 2, \dots, l, f(g_i) \leq f(o_i)$.

Proof. Note: it must be the case that $l \leq k$. And therefore, $k = l$, i.e., greedy is optimal.

Proof by induction on i . Base case: $i = 1$. In our greedy algorithm, we first pick exactly a meeting that finishes earliest amongst all requests. Therefore, immaterial of what o_1 is, $f(g_1) \leq f(o_1)$.

Induction assumption: for $i = j - 1$, it is true that $f(g_i) \leq f(o_i)$.

Practice

Step: to prove that $f(g_j) \leq f(o_j)$. We observe:

- $f(o_{j-1}) \leq s(o_j)$ – because the set O is conflict-free requests, ordered in increasing finish, and therefore, start times.
- $f(g_{j-1}) \leq f(o_{j-1})$ – induction assumption.
- Therefore, $f(g_{j-1}) \leq s(o_j)$. Therefore $f(g_j) \leq f(o_j)$ – because after we greedily choose g_{j-1} and eliminate all requests that are in conflict, o_j still remains. And our greedy choice is exactly to pick a request that remains that finishes earliest, and we happened to pick g_j .

Claim 2: Given sets O, G as in Claim 1, o_{l+1} cannot exist in O .

Proof. By Claim 1, $f(g_l) \leq f(o_l)$. And because the O set is all conflict-free, $f(o_l) \leq s(o_{l+1})$. Therefore, $f(g_l) \leq s(o_{l+1})$. So, o_{l+1} not in conflict with g_l , and so was available to be chosen after g_l was chosen and all conflicts were eliminated.

Contradiction to the assumption that greedy algorithm terminates only when no more requests available to choose from.

Practice

GRAPH ALGORITHM 1

Given an undirected graph $G = \langle V, E \rangle$ encoded as an adjacency list, define an array $\text{snd}[\cdot]$ as: for each $u \in V$, $\text{snd}[u]$ is the sum of the degrees of the neighbours of u .

Devise an algorithm that given input G , computes and outputs an array snd .

```
SNDStraightForward( $G = \langle V, E \rangle$ )
1.  $\text{snd} \leftarrow$  new array of  $|V|$  entries
2. foreach  $u \in V$  do  $\text{snd}[u] \leftarrow 0$ 
3. foreach  $u \in V$  do
4.   foreach  $v \in \text{Adj}[u]$  do
5.      $\text{degree}_v \leftarrow 0$ 
6.     foreach  $w \in \text{Adj}[v]$  do  $\text{degree}_v \leftarrow \text{degree}_v + 1$ 
7.      $\text{snd}[u] \leftarrow \text{snd}[u] + \text{degree}_v$ 
8. return  $\text{snd}$ 
```

Time efficiency of *SNDStraightForward*: $O(|V| \cdot (|E|)^2)$

Perhaps a better (more efficiency) approach:

- Visit each vertex as though it is someone's neighbor.
- Measure its degree.
- Walk its adj list again and inform each neighbor of the degree so they can update their snd .

```
SNDLinearTime( $G = \langle V, E \rangle$ )
1.  $\text{snd} \leftarrow$  new array of  $|V|$  entries
2. foreach  $u \in V$  do  $\text{snd}[u] \leftarrow 0$ 
3.  $\text{deg} \leftarrow$  new array of  $|V|$  entries
4. foreach  $u \in V$  do  $\text{deg}[u] \leftarrow 0$ 
```

Practice

```
5. foreach  $u \in V$  do  
5.   $\text{deg}[u] \leftarrow 0$   
6.  foreach  $v \in \text{Adj}[u]$  do  $\text{deg}[u] \leftarrow \text{deg}[u] + 1$   
7.  foreach  $v \in \text{Adj}[u]$  do  $\text{snd}[v] \leftarrow \text{snd}[v] + \text{deg}[u]$   
8. return  $\text{snd}$ 
```

Time efficiency:

- We visit each vertex once – Line (4) *foreach* loop.
- We visit each edge four times – Line (6) and Line (7), we walk each adj list twice.
- So total time: $O(|V| + |E|)$.

Practice

GRAPH ALGORITHM 2

Given an undirected graph G as an adjacency list and an edge e in it, devise a linear-time algorithm to determine whether there is a cycle in G that contains e .

“Go-to” linear time algorithms for graphs: DFS and BFS.

- DFS, check if back edge results in DFS tree.
- In fact, edit the explore routine as follows:
 - Keep track of parent in DFS tree.
 - Every time we hit a vertex, check if edge to root of DFS tree, and root is not parent in DFS tree.
 - If yes, immediately output **true**.

$HasCycle(G = \langle V, E \rangle, e = \langle u, v \rangle)$

1. **foreach** $u \in V$ **do**

2. $visited(u) \leftarrow false$

3. $\pi(u) \leftarrow NIL$

4. **return** $ExploreModified(\langle V, E \rangle, u, u)$

$ExploreModified(\langle V, E \rangle, \langle u, v \rangle, x)$

1. $visited(x) \leftarrow true$

2. **foreach** $y \in Adj[x]$ **do**

3. **if** $visited(y) = false$ **then**

4. **if** $(x \neq u)$ or $(x = u \text{ and } y = v)$ **then**

5. $\pi(y) \leftarrow x$

6. $ret \leftarrow ExploreModified(\langle V, E \rangle, \langle u, v \rangle, y)$

7. **if** $ret = true$ **then**

8. **return** $true$

9. **else**

10. **if** $y = r$ and $\pi(x) \neq u$ **then**

11. **return** $true$

12. **return** $false$

Practice

PROOF: DIRECTED ACYCLIC GRAPH (DAG) ALGORITHM

Show that the following algorithm to linearize a DAG can be realized in linear time.

Find a source, output it, and delete it from the graph.

Repeat until the graph is empty.

We assume adjacency list representation of the input DAG.

Suppose we first create a new array, call it ni of size $|V|$, where $ni[u]$ is the number of edges incident in $u \in V$ at the start. Can do this in one pass of entire adj list of the graph.

From ni , we can identify all sources. Suppose we create a list of source vertices, call it $srclist$. Then, we remove a vertex from $srclist$ and proceed...

1. $ni \leftarrow$ new array of size $|V|$
2. **foreach** $u \in V$ **do** $ni[u] \leftarrow 0$
3. **foreach** $u \in V$ **do**
4. **foreach** $v \in Adj[u]$ **do**
5. $ni[v] \leftarrow ni[v] + 1$
6. $srclist \leftarrow$ new empty linked list
7. **foreach** $u \in V$ **do**
8. **if** $ni[u] = 0$ **then** Insert u at head of $srclist$
9. **while** $srclist$ is not empty **do**
10. $u \leftarrow$ remove vertex from head of $srclist$
11. **foreach** $v \in Adj[u]$ **do**
12. $ni[v] \leftarrow ni[v] - 1$
13. **if** $ni[v] = 0$ **then**
14. Add v to head of $srclist$
15. Output u

Practice

PROOF: DEPTH FIRST SEARCH (DFS) ALGORITHM

Prove that DFS on an undirected graph can result in no cross edges.

An edge $\langle u, v \rangle$ is a cross edge if and only if: $pre[v] < post[v] < pre[u] < post[u]$.

Suppose a cross edge, $\langle u, v \rangle$ exists after a run of DFS on an undirected graph G .

At the time $post[v]$ and at all times prior since initialization, $visited[u] = false$.

But that means that in the for loop that immediately precedes $postvisit(v)$, we would have invoked $explore(u)$, thereby setting $visited[u]$ to **true** before the time $post[v]$.

Therefore, we have a contradiction.

Practice

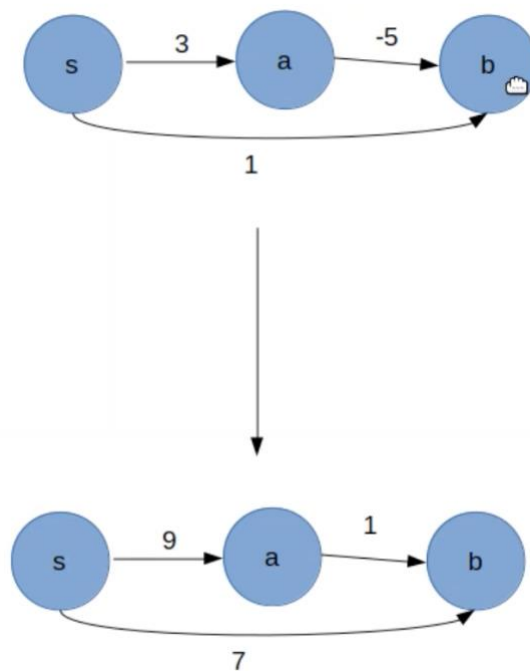
PROOF: SHORTEST PATH ALGORITHM

Professor F. Lake suggests the following algorithm for finding the shortest path from node s to a node t in a directed graph with some negative weight edges: add a large constant to each edge weight so all the weights become positive, then run Dijkstra's algorithm starting at node s , and return the shortest path found to node t .

Is this a valid method? Either prove that it works correctly or give a counterexample.

Directed graph with weights on edges is: $G = \langle V, E, l \rangle$, where $E \subseteq V \times V$, and $l: E \rightarrow \mathbb{R}$.

Counterexample, add a constant of 6 to the graph below:



In the unmodified graph, the shortest path is $s \rightarrow a \rightarrow b$ (-2), but in the modified graph, the shortest path becomes $s \rightarrow b$ (7). Since the shortest path changes, this is not a valid method.

Practice

PROOF: DIJKSTRA'S ALGORITHM

Prove: if we initialize $dist(u)$ to ∞ , and at the end of a run of Dijkstra's algorithm on $G = \langle V, E, l \rangle$ with source $s \in V$ it is the case that $dist(u) \neq \infty$, then there exists a path $s \rightsquigarrow u$ in G .

Contrapositive: if there exists no path $s \rightsquigarrow u$ in G , then at the end of any run of Dijkstra, $dist(u) = \infty$.

We first observe: the only way $dist(u)$ can change after initialization is via a call $update(e)$ where $e \in E$ is incident on u , i.e., some $\langle v, u \rangle \in E$.

So proof strategy: induction on number of invocations to $update(\cdot)$ that the run of Dijkstra does. Call this number k .

If $k = 0$, then this can only be because $E = \emptyset$. Then, there is no path $s \rightsquigarrow u$. And as we have not changed $dist(u)$ from its initial value, at the end of the run of Dijkstra, $dist(u) = \infty$ as desired.

For the step, we consider two cases.

- (i) No edge is incident on u . Then, we know that no $update(\cdot)$ affects $dist(u)$, and therefore $dist(u) = \infty$ as desired.
- (ii) There exists some $\langle v, u \rangle \in E$. If the last $update(\cdot)$ we performed is not on any edge incident on u , then $dist(u)$ is the same as it was after $k - 1$ invocations to $update(\cdot)$, and by the induction assumption $dist(u)$ in that case $= \infty$.

The final (sub-)case: the k^{th} update was on some $\langle v, u \rangle$, i.e., edge incident on u . Then there is no path $s \rightsquigarrow v$. Why not? Because if there was, there would be a path to u : $s \rightsquigarrow v \rightarrow u$. And therefore, $dist(v)$ is whatever value it is after $k - 1$ invocations to $update(\cdot)$. And by the induction assumption $dist(v) = \infty$ before $update(v, u)$. Also, again by the induction assumption, $dist(u) = \infty$ before the k^{th} invocation to $update(\cdot)$. Therefore, after the k^{th} invocation, which is $update(v, u)$, $dist(u) = \infty$.

Practice

PROOF: BELLMAN-FORD ALGORITHM

Prove: suppose we run Bellman-Ford on $\langle G = \langle V, E, l \rangle, s \in V \rangle$ where we do not know whether G has a negative weight cycle. Also suppose that at the end of that run of Bellman-Ford, we carry out one more $update(e)$ on every $e \in E$. Then: some $dist(u)$ changes in this additional round of updates for some u that is reachable from s if and only if there is a negative weight cycle in G that is reachable from s .

“Only if”: we seek to prove: if $dist(u)$ changes, this implies that there is a negative weight cycle.

By Claim (2) of Lecture 5(b): if there exists a shortest path from s to u that is simple, then $|V| - 1$ invocations to $update(\cdot)$ on all edges, as Bellman-Ford does, is sufficient for $dist(u)$ to converge to $\delta(s, u)$. Given that $|V| - 1$ invocations to $update(\cdot)$ on all edges is not sufficient, this can only be because there is a shortest path $s \rightsquigarrow u$ that is not simple. And this in turn is true only if there is a negative cycle reachable from s .

“If”: we seek to prove: if there is a negative weight cycle reachable from s , then there exists some u that is reachable from s for which the additional round of $update(\cdot)$ changes $dist(u)$.

An observation: a change to $dist(u)$ has to be a decrease. Because (repeated) invocation(s) to $update(\cdot)$ can only decrease $dist(\cdot)$ value(s).

Suppose $\langle u_0, u_1, \dots, u_{k-1}, u_0 \rangle$ is a negative weight cycle that is reachable from s , where $u_0 = u_k$.

Proof idea: we know that $\sum_{i=1}^k l(u_{i-1}, u_i) < 0$.

Practice

Assume, for the purpose of contradiction, that no $dist(u)$ changed in the additional round of $update(\cdot)$'s, for any $u \in V$. Now consider the vertices u_0, \dots, u_{k-1}, u_k in the negative weight cycle above.

We first observe for all $u_i \in \{u_1, \dots, u_k\}$, it is true that: $dist(u_i) \leq dist(u_{i-1}) + l(u_{i-1}, u_i)$ after the last round of updates.

So now:

$$\begin{aligned}\sum_{i=1}^k dist(u_i) &\leq \sum_{i=1}^k (dist(u_{i-1}) + l(u_{i-1}, u_i)) \\ \sum_{i=1}^k dist(u_i) &= \sum_{i=1}^k dist(u_{i-1}) + \sum_{i=1}^k l(u_{i-1}, u_i)\end{aligned}$$

But:

$$\sum_{i=1}^k dist(u_i) = \sum_{i=1}^k dist(u_{i-1})$$

To see this:

$$dist(u_1) + dist(u_2) + \dots + dist(u_k) = dist(u_0) + dist(u_1) + \dots + dist(u_{k-1})$$

Because $u_0 = u_k$.

So:

$$\sum_{i=1}^k l(u_{i-1}, u_i) \geq 0$$

This is the contradiction.

This proof is “constructive” – it is saying that $dist(u)$ must change (decrease) for some vertex u in a negative weight cycle reachable from s in this additional round of calls to $update(\cdot)$.

Practice

MINIMUM SPANNING TREE 1

Give an example of a connected undirected $G = \langle V, E, l \rangle$ such that the set of edges $\{\langle u, v \rangle\}$: there exists a cut $\langle S, V \setminus S \rangle$ such that $S \subset V$ and $\langle u, v \rangle$ is an edge of smallest weight that crosses $\langle S, V \setminus S \rangle$ does not form an MST.

Let X be that set of edges.

What is the set X for our example graph: complete graph with vertices $V = \{a, b, c\}$.

$$l(a, b) = l(b, c) = l(a, c) = 1.$$

- Is $\langle a, b \rangle \in X$? Yes. It is a light edge that crosses the cut: $\langle \{a\}, \{b, c\} \rangle$.
- Is $\langle b, c \rangle \in X$? Yes. Consider the cut $\langle \{b\}, \{a, c\} \rangle$.
- Is $\langle a, c \rangle \in X$? Yes. Consider the cut $\langle \{a\}, \{c, b\} \rangle$.

So, we are done. Because $X = \{\langle a, b \rangle, \langle b, c \rangle, \langle a, c \rangle\}$ is not an MST of that G .

Because that is not acyclic, i.e., not a tree.

Practice

MINIMUM SPANNING TREE 2

Professor Sabatier conjectures the following converse of what we say under “now the general approach” on page 3 of Lecture 6a:

Let $G = \langle V, E, l \rangle$ be a connected undirected graph. Let: (i) $A \subseteq E$ that is included in some MST of G , (ii) $\langle S, V \setminus S \rangle$ be any cut of G that respects A , and (iii) $A \cup \{\langle u, v \rangle\}$ also be included in some MST of G . Then, $\langle u, v \rangle$ is an edge of smallest weight that crosses the cut $\langle S, V \setminus S \rangle$.

Show that the professor’s conjecture is not necessarily true.

Let $V = \{a, b, c\}$, $E = \{\langle a, b \rangle, \langle a, c \rangle\}$, and $l(a, b) = 1$, $l(a, c) = 1000$.

Now let $A = \emptyset$. Then the cut $\langle \{a\}, \{c, b\} \rangle$ respects A .

And $\langle a, c \rangle$ is not a light edge that crosses that cut but is in a (the) MST of the graph.

Practice

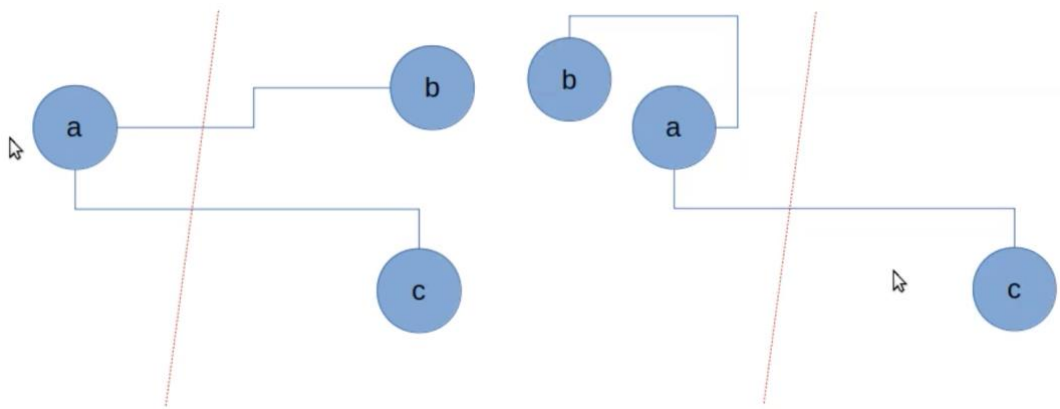
MIN-CUT AND OPTIMAL SUBSTRUCTURE

Consider the min-cut problem: given as input an undirected graph $G = \langle V, E \rangle$, what is the minimum number of edges that cross any cut $\langle S, V \setminus S \rangle$ where $S \subset V$?

Bob claims that the problem has the following optimal substructure. Given a cut $\langle S, V \setminus S \rangle$ that is a min-cut for G , then $\langle S \setminus \{u\}, V \setminus (S \cup \{u\}) \rangle$ is a min-cut for G' , where we get G' from G by removing $u \in V$ and all edges incident on it, provided both $S \setminus \{u\}$ and $V \setminus (S \cup \{u\})$ are non-empty.

Refute Bob's claim.

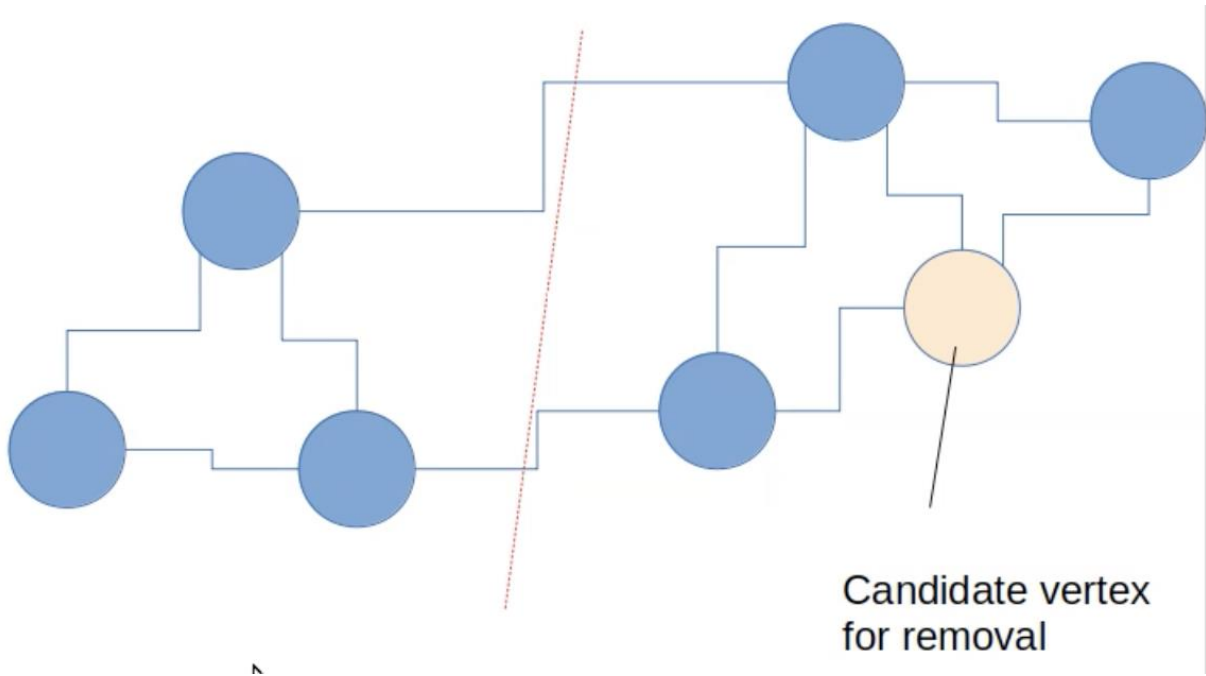
Examples:



Original min-cut $\langle \{a, b\}, \{c\} \rangle$, and indeed, for this min-cut, it turns out:

$\langle \{a\}, \{c\} \rangle$ is indeed a min-cut for G' for this G .

Practice



Counterexample: In this example, the min-cut is 2. However, if the candidate vertex is removed, the graph can be shifted such that the min-cut now becomes 1. Hence, this is a valid counterexample, and Bob's claim is refuted.

Practice

OPTIMAL SUBSTRUCTURE ALGORITHM

The interval-scheduling problem from “Proof: Greedy Choice” possesses optimal substructure. What is it, and how do we exploit it to realize an algorithm?

We are given as input a set of requests $R = \{r_1, \dots, r_n\}$, where each $r_i = \langle s_i, f_i \rangle$ such that $s_i, f_i \in \mathbb{Z}^+$ and $s_i < f_i$.

For single-source shortest-paths: if $s \rightsquigarrow x \rightsquigarrow y$ is a shortest-path from s to y , then the $s \rightsquigarrow x$ sub-path is a shortest path from s to x .

We could ask: suppose $r_{i_1}, r_{i_2}, \dots, r_{i_k}$ is an optimal sequence of requests that are non-conflicting such that $f_{i_j} \leq f_{i_{j+1}}$. Is there anything I can say about the optimality of $r_{i_1}, \dots, r_{i_{k-1}}$? More specifically, is it an optimal solution to a sub-problem?

I think: the answer is yes. A sub-problem for which $r_{i_1}, \dots, r_{i_{k-1}}$ has to be an optimal solution: suppose in the input, the requests r_1, \dots, r_n are ordered by non-decreasing finish time. Then: $r_{i_1}, \dots, r_{i_{k-1}}$ has to be an optimal solution to all requests that end at or before s_{i_k} .

In fact: we can “lop off” or “eat into” optimal solution from both directions.

Specifically:

- Assume input set of requests r_1, \dots, r_n are sorted non-decreasing by finish time. That is: $f_1 \leq f_2 \leq \dots \leq f_n$.
- Now: suppose $M[i, j]$ is the max # requests I can schedule that start at or after f_i and end at or before s_j .
- Also denote as $R_{i,j}$ the set of requests that start at or after f_i and end at or before s_j .

Then:

$$M[i, j] = \begin{cases} 0, & \text{if } R_{i,j} = \emptyset \\ 1 + \max_{\substack{i < k < j \\ r_k \in R_{i,j}}} \{M[i, k] + M[k, j]\}, & \text{otherwise} \end{cases}$$

Our final solution: $M[0, n+1]$, where we introduce fictitious requests r_0, r_{n+1} with $f_0 < s_1$, $s_{n+1} > f_n$.

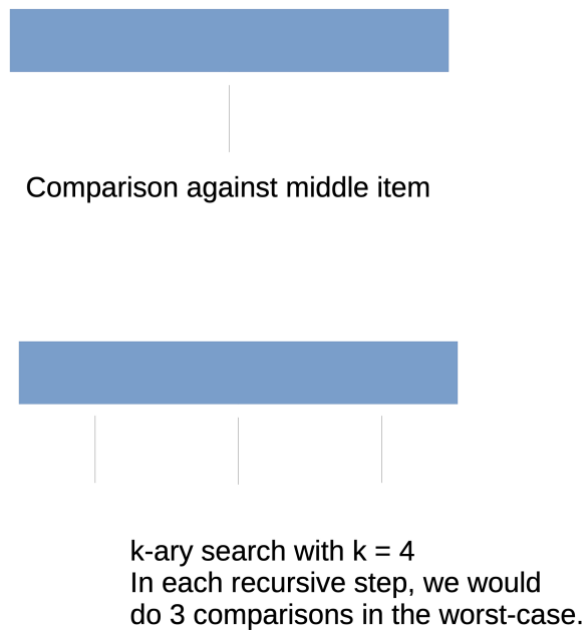
Practice

TIME EFFICIENCY: K-ARY SEARCH

In binary search, we split the input sorted array into two pieces, each of size $n/2$, and recursively search on one of those pieces.

Alice proposes k-ary search, in which we split the array into k pieces, each of size n/k . What is the worst-case time-efficiency of k-ary search as a function of $\langle n, k \rangle$?

Inspired by the eggs-building problem, Alice wonders whether setting $k = \sqrt{n}$ yields a more efficient algorithm than binary search. Does it?



Recurrence for binary search: $T(n) = T(n/2) + 1$.

For k-ary search, recurrence for worst-case time-efficiency: $T(n) = T(n/k) + (k - 1)$. To solve the recurrence:

$$\begin{aligned} T(n) &= T(n/k) + (k - 1) \\ &= T(n/k^2) + 2(k - 1) \\ &= T(n/k^3) + 3(k - 1) \\ &= \dots \\ &= T(1) + \log_k n \cdot (k - 1) \\ &= \Theta(k \cdot \log_k n) \end{aligned}$$

Practice

Where we figure the last term as follows: we ask for what x is $n/k^x = 1$? Answer: $n/k^x = 1 \Leftrightarrow n = k^x \Leftrightarrow \log n = x \cdot \log k \Leftrightarrow x = \log n / \log k = \log_k n$

So, if we set $k = \sqrt{n} = n^{1/2}$, then:

$$T(n) = \Theta(\sqrt{n} \cdot \log_{\sqrt{n}} n) = \Theta(2 \cdot \sqrt{n}) = \Theta(\sqrt{n})$$

And if we do binary search, $T(n) = \Theta(\log n)$. And $\sqrt{n} = \Omega(\log n)$, and $\sqrt{n} \neq O(\log n)$. So, setting $k = \sqrt{n}$ yields a strictly worse performing algorithm than setting $k = 2$.

Practice

TIME EFFICIENCY: BINARY SEARCH

Carry out an expected- (or average-) case analysis of the time-efficiency of binary search.

First off, we should distinguish a successful (binary) search from an unsuccessful search.

Because expected-case time-efficiency of an unsuccessful search is just $\Theta(\log n)$.

For a successful search, the time it takes depends on the item we are looking for. Assume: (i) every item in array is distinct, and (ii) every item in the array is equally likely to be searched for.

Now, if X is a random variable that is the number of comparisons or # iterations or # recursive calls we perform before we find the item we seek is below. As simplification, assume that we have $2^k - 1 = n$ items in the array, for some positive integer k .



If we're searching for this item, then we would return in 1 iteration

If the item I'm searching for is one of those 2 items, then we would return in 2 iterations

Practice

$$\begin{aligned} E[X] &= 1 \times \frac{1}{n} + 2 \times \frac{2}{n} + 3 \times \frac{4}{n} + 4 \times \frac{8}{n} + \dots \\ &= 1 \times \frac{2^0}{2^k - 1} + 2 \times \frac{2^1}{2^k - 1} + 3 \times \frac{2^2}{2^k - 1} + \dots + k \times \frac{2^{k-1}}{2^k - 1} \\ &= \frac{1}{2^k - 1} \cdot (1 \times 2^0 + 2 \times 2^1 + \dots + k \times 2^{k-1}) \\ &= \frac{1}{2^k - 1} \cdot \left((2^0 + 2^1 + \dots + 2^{k-1}) + (2^1 + \dots + 2^{k-1}) + (2^2 + \dots + 2^{k-1}) + \dots + (2^{k-1}) \right) \\ &= \frac{1}{2^k - 1} \cdot \sum_{i=0}^{k-1} \sum_{j=i+1}^k 2^{j-1} \\ &= \frac{1}{2^k - 1} \cdot \sum_{i=0}^{k-1} (2^i + 2^{i+1} + \dots + 2^{k-1}) \\ &= \frac{1}{2^k - 1} \cdot \sum_{i=0}^{k-1} (2^k - 2^i) = \frac{1}{2^k - 1} \cdot \left(\sum_{i=0}^{k-1} 2^k - \sum_{i=0}^{k-1} 2^i \right) \\ &= \frac{1}{2^k - 1} \cdot \left(k \cdot 2^k - \sum_{i=0}^{k-1} 2^i \right) \\ &= \frac{1}{2^k - 1} \cdot (k \cdot 2^k - (2^k - 1)) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot 2^k + 1) = \Theta(k) = \Theta(\log n) \end{aligned}$$

Practice

INTEGER LINEAR PROGRAM: VERTEX COVER

Given an undirected graph $G = \langle V, E \rangle$, a *vertex cover* for it is a set $C \subseteq V$ with the property: $\langle u, v \rangle \in E$ implies at least one of $u, v \in C$.

An optimization problem is: given as input undirected $G = \langle V, E \rangle$, compute the minimum size of a vertex cover. Encode this optimization problem as an Integer Linear Program (ILP).

Adopt as unknowns in our output ILP, $x_1, \dots, x_{|V|}$, where x_i corresponds to a vertex $i \in V$. More specifically, constrain each $x_i \in \{0, 1\}$, with $x_i = 1$ if vertex i is in the vertex cover, and $x_i = 0$ otherwise.

So immediately, we have the constraints:

- For all $i = 1, \dots, |V|$, $x_i \geq 0$.
- For all $i = 1, \dots, |V|$, $x_i \leq 1$.

To model the constraints of a vertex cover:

For each $\langle u, v \rangle \in E$, a constraint:

- $x_u + x_v \geq 1$.

And finally, our optimization objective:

- Minimize $\sum_{u \in V} x_u$, or Maximize $-\sum_{u \in V} x_u$.

That's it. What is the size of the output ILP instance, given as input $\langle G \rangle$?

Answer: $\Theta(|V| + |E|)$.

Practice

INTEGER LINEAR PROGRAM: ALGORITHM

Consider the following restricted, decision version of ILP, which is known as ZOE.

Given as input $\mathbf{A} \in \{0, 1\}^{n \times m}$, does there exist $\mathbf{x} \in \{0, 1\}^m$ such that $\mathbf{Ax} = \mathbf{1}$?

Suppose we have access to an oracle for this decision version. That is, given any such \mathbf{A} , it outputs **true** if indeed such an \mathbf{x} exist, and **false** otherwise, and it does so in constant-time.

Devise a polynomial-time algorithm that given as input such an \mathbf{A} , outputs a vector \mathbf{x} such that $\mathbf{Ax} = \mathbf{1}$ if indeed such an \mathbf{x} exists, and the string ‘no solution’ otherwise.

Suppose $\mathbf{x} = [x_1, x_2, \dots, x_m]$. And suppose the oracle is denoted $I(\cdot)$.

First invoke $I(\mathbf{A})$. If the output is **false**, then output ‘no solution’ and halt.

Otherwise, we know that such an \mathbf{x} exists, and we need to find it.

First try $x_1 = 0$. Then simplify \mathbf{Ax} . That is, suppose $\mathbf{A} = [a_{i,j}]$. Then, \mathbf{Ax} is:

$$\begin{bmatrix} \sum_{j=1}^m a_{1,j} \cdot x_j \\ \sum_{j=1}^m a_{2,j} \cdot x_j \\ \dots \\ \sum_{j=1}^m a_{n,j} \cdot x_j \end{bmatrix}$$

If we adopt $x_1 = 0$, then \mathbf{Ax} is:

$$\begin{bmatrix} \sum_{j=2}^m a_{1,j} \cdot x_j \\ \sum_{j=2}^m a_{2,j} \cdot x_j \\ \dots \\ \sum_{j=2}^m a_{n,j} \cdot x_j \end{bmatrix}$$

Practice

So $A[1, \dots, n; 2, \dots, m]$, i.e., the original A matrix with the $m - 1$ columns $2, \dots, m$ only is a new instance of ZOE of size $n \times (m - 1)$. Now invoke $I(A[1, \dots, n; 2, \dots, m])$. If it returns **true**, then we know that an \mathbf{x} exists to the original instance \mathbf{A} of ZOE such that $x_1 = 0$. If the return is **false**, then we know that $x_1 = 1$.

Rewritten more carefully:

Suppose we determine that $x_1 \neq 0$, i.e., $x_1 = 1$. Then, in any row i of \mathbf{A} that $a_{i,1} = 1$, suppose for some $j \neq 1$, $a_{i,j} = 1$. Then the corresponding $x_j = 0$. Because that is the only way that $\sum_{j=1}^m a_{i,j} \cdot x_j = 1$. Thus, we can go ahead and adopt 0 for all those x_j 's.

Once we determine x_1 , similarly determine x_2 unless it has already been determined to be by trial-and-error, with at most 2 possibilities for it. Note that the only rows that are useful in determining x_2 are those rows i in which (i) $a_{i,2} = 1$. Also, if x_1 was determined to be 1, then we also should not consider any row in which (ii) $a_{i,1} = 1$. If no such rows exist that satisfy both (i) and (ii), then we know $x_2 = 0$.

So, in the worst case, number of invocations to $I(\cdot)$ is $m + 1$. So, we have constructed a polynomial-time algorithm to determine such an \mathbf{x} if one exists.

So, the point is: within a “polynomial factor” finding such an \mathbf{x} is no more difficult than determining whether an instance of ZOE is **true**.