

MATHEMATICAL INDUCTION

- Induction and Deduction
- Mathematical Induction (its strength)
- Examples
 - Sum of first n natural numbers
Prove by induction that the sum of the first n natural numbers is $n(n + 1)/2$. i.e.

$$S(i) : \sum_{i=1}^n i = \frac{n \times (n + 1)}{2}$$

- Sum of powers of two
Prove by induction that the sum of the first n powers of two (0.. n) is $2^{n+1} - 1$. i.e.

$$S(i) : \sum_{i=0}^n 2^i = 2^{n+1} - 1$$

- n th derivative of e^{kx}
Prove by induction that the n th derivative of e^{kx} is $k^n e^{kx}$. i.e.

$$S(n) : \frac{d^n}{dx^n} e^{kx} = k^n e^{kx}$$

- Arithmetic and Geometric progressions.
 - Derivations of general forms
 - Proofs by induction
 - An expression for the sum of the first n odd numbers and prove it using Mathematical Induction.
 - Procedure for deriving an expression for the sum of a polynomial series: $\sum_{i=0}^n p(n_k)$ where $p(n_k)$ is a polynomial in n of order k : $a_1n^k + a_2n^{k-1} + \dots + a_k$

- General template for Induction proofs:

- One more example: Error correcting codes

$S(n)$: C_n is any set of bit strings of length n that is *error detecting*, then C_n contains at most 2^{n-1} strings.

 - Assume $S(n)$
 - Divide C_{n+1} into two sets of strings $1x$ and $0y$
 - x & y obviously be error detecting themselves.
 - By hypothesis, x/y contain no more than 2^{n-1} strings.
 - Thus maximum total number of strings is $2 \times 2^{n-1} = 2^{n+1-1}$

Complete (strong) Vs Weak induction

- Complete (or strong) induction: Uses two or more assumptions from the basis up to n .
- Examples:

Prove that there exist integers a and b such that $\forall n \in \mathbb{Z} \cap n > 0 : n = 2a + 3b$.

Prove that there exist no integers x, y, z such that $\forall n \in \mathbb{Z} \cap n > 2 : x^n + y^n = z^n$.
- Fallacies and getting carried away with induction (All marbles are red, $\forall n \in \mathbb{Z} : a^n = 1$)
- Structural induction: Proving properties about a structure by using a basis case and an inductive assumption.
- Examples:

In a colony of aphids reproducing asexually, let the status of an aphid be represented by the number of its children. Show that the total number of aphids in the colony is one more than the sum of the statuses of all the aphids.

Assume that a particular implementation of Binary trees uses the Node with right and left child pointers. In such a binary tree, prove that the number of NULL pointers is one more than the total number of nodes.

Proving Program Properties

What are loop invariants?

A Selection Sort program:

```
(1) for (i = 0; i < n-1; i++) {  
(2)     small = i;  
(3)     for (j = i+1; j < n; j++)  
(4)         if (A[j] < A[small])  
(5)             small = j;  
(6)     swap(&A[small], &A[i]);  
(7) }
```

$S(k)$: If we reach the test for $j < n$ on line (3) with $j = k$ then *small* indexes smallest of $A[i..k - 1]$.

Basis: $S(i + 1)$: *small* indexes smallest of $A[i..i]$.

Induction: Proof of $S(k + 1)$ assuming $S(k)$.

$S(k)$: When testing for $j < n$ at line (3), *small* indexes smallest of $A[i..k - 1]$.

Now consider what happens in the loop body when $j = k$, specifically in the “if” statement of lines (4) and (5).

if $A[k]$ is less than the smallest of $A[i + 1..k - 1]$ then $small = k$. Then *small* indexes smallest of $A[i + 1..k]$.

If $A[k]$ is not less than the smallest of $A[i + 1..k - 1]$ then the value of *small* is unchanged. So *small* will now be the index of the smallest of $A[i + 1..k]$.

Thus, in either case, if the loop test is reached subsequently with the value of j incremented from k to $k + 1$, *small* indexes the smallest of $A[i + 1..k]$ which proves $S(k)$ holds for all values of $k \geq i + 1$.

Question: what is the truth value of $S(k)$ when $k > n$)

Now consider the whole SelectionSort function. The following assertion is made about it.

$T(m)$: If we reach the test $i < n - 1$ at line 1, with $i = m$, then

1. $A[0..m - 1]$ are in non-decreasing order.
2. All of $A[m..n - 1]$ are greater than or equal to any of $A[0..m - 1]$.

Basis: $m = 0$.

1. $A[0.. - 1]$ are obviously in sorted order.
2. All of $A[0..n - 1] \geq$ any of $A[0.. - 1]$

Induction: Proof of $T(m+1)$ assuming $T(m)$.

Consider what happens when $i = m$. We know (by the IH) that $A[0..m - 1]$ are in sorted order and no element of $A[0..m - 1]$ is greater than any element of $A[m..n - 1]$.

we exit the loop of lines (3)–(5), the loop variable j would have the value n and so at that point $small$ will index the smallest of $A[m..n - 1]$, by the previous induction.

Now note two things about the situation:

1. By the first condition of the IH, we know that $A[0..m - 1]$ are already in sorted order.
2. By the second condition, we know that $small$ is greater than or equal to any item in $A[0..m - 1]$.

Thus,

1. After swapping $A[m]$ with $A[small]$, incrementing i to $m + 1$, proceeding around the loop, just before the test, we can assert that $A[0..m]$ is sorted which is condition 1.
2. $A[small]$ is the smallest of $A[m..n - 1]$ and that both $A[m]$ and all elements of $A[0..m - 1]$ are less than or equal to any element in $A[m + 1..n - 1]$ which is condition 2.

Note that when $m = n$, we have exited the outer loop and so, $A[0..n - 1]$ are in sorted order.

Proof number 2: Factorial function.

```
(1)  int factorial(int n) {
(2)      int i = 2;
(3)      int fact = 1;
(4)      while (i <= n) {
(5)          fact = fact * i;
(6)          i++;
(7)      }
(7)      return fact;
}
```

First – Proof of termination:

Chose $E = n - i + 1$.

Second — Proof of correctness:

$S(k)$: If we reach the test $i \leq n$ at line (4), with the value of the loop variable i set to k , then $fact$ will contain the value $(k - 1)!$

Basis: The basis is $S(2)$. $i = 2 \rightarrow fact = 1$ by assignment.

Induction: Assume $S(k)$. Prove $S(k + 1)$.

Proving properties of recursive functions:

```
(1) int fact(int n)
(2)     if (n <= 1)
(3)         return 1;
(4)     else
(5)         return n * fact(n-1);
```

We want to prove the following inductive assertion on i .

$S(i)$: If the function fact is called with the argument i , it returns the value $i!$

Basis: Consider $S(1)$. Line (2) performs a test which succeeds and so factorial returns 1 which is indeed 1!

Induction: Assume $S(k)$, i.e. fact(k) returns $k!$. Now consider what will happen if fact() is called with argument $k + 1$ with $k \geq 1$

The test on line (2) fails and so the else part is executed. Thus the call of factorial returns the product of $(k + 1) * fact(k)$. However, by the IH, fact(k) will return the value $k!$. Thus the value returned by the present call will be $(k + 1) * k!$ which by the definition of the factorial function is $(k + 1)!$. QED.

Recursive Selection sort:

```
(1) void recSS(int A[], int i, int n) {
(2)     if (i < n-1) {
(3)         small=i;
(4)         for (j = i; j < n; j++)
(5)             if (A[j] < A[small])
(6)                 small = j;
(7)         swap(&A[small], &A[i]);
(8)         recSS(A, i+1, n);
(9)     }
(10) }
```

$S(k)$: If the function `recSS()` is called with $i = k$, then when it returns it will leave $A[k..n-1]$ in non-descending order.

Basis: $S(n-1)$.

The IH is for $S(k)$ where $0 \leq k < n$. We must show that $S(k-1)$ holds given $S(k)$.

Merge Sort:

```
LIST MakeList() {
    int x;
    LIST pNewCell;

    if (scanf("%d", &x) == EOF) return NULL;
    else {
        pNewCell = (LIST) malloc(sizeof(CELL));
        pNewCell->next = MakeList();
        pNewCell->element = x;
        return pNewCell;
    }
}

LIST MergeSort(LIST list) {
    LIST SecondList;

    if (!list || !list->next) return list;
    else {
        SecondList = split(list);
        return merge(MergeSort(list), MergeSort(SecondList));
    }
}
```

```

LIST merge(LIST list1, LIST list2) {
    if (!list1) return list2;
    if (!list2) return list1;
    if (list1->element <= list2->element) {
        list1->next = merge(list1->next, list2);
        return list1;
    } else {
        list2->next = merge(list2->next, list1);
        return list2;
    }
}

LIST split(LIST list) {
    LIST pSecondCell;

    if (!list || !list->next) return NULL;
    else {
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}

```

Running Times

1. Log means \log_2 in this module.
2. About counting machine instructions...
3. The 90–10 rule: (Eg. Experiment using gprof/gmon under Unix)
4. Why is it pointless to count the actual number of seconds?
5. Benchmarking Vs Analysis
6. Running time is conventionally denoted by $T(n)$ (no units)

A selection sort program fragment:

```
(2)  small = i;  
(3)  for (j = i+1; j < n; j++) {  
(4)      if (A[j] < A[small])  
(5)          small = j;
```

Total time spent in this loop is $4(n - i - 1) + 1$.

What is meant by quadratic and linear time?

Big-Oh notation:

$T(n)$ is $O(f(n))$ if there exists some integer $n_0 > 0$ and constant $c > 0$ such that $\forall n > n_0 : T(n) \leq cf(n)$

Example: Find the Big-oh for $T(n) = (n + 1)^2$ (Hint: 3,2 or 1,4)

Question: Can $T(n)$ be considered $O(n^2/1000)$?

Show that: $\log n$ is \sqrt{n} etc.

1. Constant factors don't matter
2. Low order terms don't matter

Thumbrule:

if $\lim_{n \rightarrow \infty} \frac{h(n)}{g(n)} = 0$ then $O(g(n) + h(n)) = O(g(n))$.

The Transitive Law:

if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Some useful points:

1. if p and q are polynomials in n and $\text{degree}(p) \leq \text{degree}(q)$ then p is $O(q)$
2. if $\text{degree}(p) > \text{degree}(q)$ then p is *not* $O(q)$
3. Every exponential grows faster than any polynomial
4. No exponential is $O(p)$ where p is a polynomial

Some common expressions and their names:

| BIG-OH | INFORMAL NAME |
|---------------|---------------|
| $O(1)$ | constant |
| $O(\log n)$ | logarithmic |
| $O(n)$ | linear |
| $O(n \log n)$ | $n \log n$ |
| $O(n^2)$ | quadratic |
| $O(n^3)$ | cubic |
| $O(2^n)$ | exponential |

Example: What is the time complexity of the following fragment:

```
(1) scanf(“%d”, &n);
(2) for (i = 0; i < n; i++)
(3)     for (j = 0; j < n; j++)
(4)         A[i][j] = 0;
(5) for (i = 0; i < n; i++)
(6)     A[i][i] = 1;
```

Discuss program running times for:

1. Primitive operations
2. for/while loops (simple and complex)
3. if/switch statements
4. function calls (in loops (in init/tests/body))

Drawing a parse tree for a statement and finding the running time. Use as example the Selection sort fragment.

Analysing recursive functions:

For the factorial function,

Basis: $T(1) = O(1)$

Induction: $T(n) = O(1) + T(n - 1)$

Let the basis take a units of time.

And the inductive step take $b + T(n - 1)$ units of time.

$$\begin{aligned}T(1) &= a \\T(2) &= a + b \\T(3) &= a + b + b = a + 2b \\&\vdots \\T(n) &= a + (n - 1)b\end{aligned}$$

Solving by repeated substitution:

We have the sequence of equations

$$\begin{aligned}T(n) &= b + T(n - 1) \\T(n - 1) &= b + T(n - 2) \\&\vdots \\T(2) &= b + T(1)\end{aligned}$$

Substituting repeatedly, we get $T(n) = (n - 1)b + a$

Consider $T(n) = b + T(n-1)$ to be the first substitution.

We want to prove by induction on the number of substitutions i (something not stated explicitly in the book):

$S(i)$: If $1 \leq i < n$, then $T(n) = ib + T(n - i)$

so that we can assert that after $(n-1)$ substitutions, $T(n) = (n - 1)b + T(1)$.

Basis: $i = 1$, We know $T(n) = b + T(n - 1)$ is true from its definition.

Induction: Consider $i < n - 1$. Then,

$S(i) : T(n) = ib + T(n - i)$.

$S(i + 1)$ is the $(i + 1)$ th substitution. So substitute once more in $S(i)$, i.e. Substitute for $T(n - i)$ in $S(i)$.

$T(n) = ib + b + T(n - i - 1) = (i + 1)b + T(n - (i + 1)) = S(i + 1)$ Q.E.D.

Analysis of Merge sort:

Simple to show that merge() takes $O(n)$ time on a list length n

Consider `split()`

$T(0) = O(1)$ and $T(1) = O(1)$. Let them be a and b resp.

$T(n) = O(1) + T(n - 2)$, i.e. $T(n) = c + T(n - 2)$ Why?

$$\begin{aligned}T(2) &= c + T(0) = a + c \\T(3) &= c + T(1) = b + c \\T(4) &= c + T(2) = a + 2c \\&\vdots\end{aligned}$$

$T(n) = a + cn/2$ for even n and $T(n) = b + c(n - 1)/2$ for odd n .

Proof by induction on the number of substitutions:

$$\begin{aligned}T(n) &= c + T(n - 2) \\&= 2c + T(n - 4) \\&\vdots\end{aligned}$$

$S(i)$: If $1 \leq i < n/2$, then $T(n) = ic + T(n - 2i)$

Substitute once more for $T(n - 2i)$ to get the $(i + 1)$ th substitution.

$$\begin{aligned}T(n) &= ic + c + T(n - 2i - 2) \\&= (i + 1)c + T(n - 2(i + 1)) \\&= S(i + 1) \quad Q.E.D.\end{aligned}$$

Now consider the function MergeSort()

Easy to see that

$$T(n) = 2T(n/2) + O(n), \text{ i.e. } T(n) = 2T(n/2) + bn$$

Let's begin the substitutions

$$\begin{aligned} T(1) &= a \\ T(2) &= 2T(1) + 2b = 2a + 2b \\ T(4) &= 2T(2) + 4b = 4a + 8b \\ T(8) &= 2T(4) + 8b = 8a + 24b \\ T(16) &= 2T(8) + 16b = 16a + 64b \\ &\vdots \end{aligned}$$

Generalising we get, $T(n) = an + bn \log_2 n$

An induction on the number of substitutions we need to make to $T(n)$ before ending up with $T(1)$. This will obviously be $\log_2 n$ since n is halved at each stage. (How many times can we cut a number to half its size before ending up with a number ≤ 1 ?)

To intuit $S(i)$, consider the following substitutions

$$\begin{aligned}T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/4) + bn/2) + bn = 4T(n/4) + 2bn \\ &= 4(2T(n/8) + bn/4) + 2bn = 8T(n/8) + 3bn \\ &\vdots\end{aligned}$$

So in general,

$S(i)$: If $1 \leq i \leq \log_2 n$, then $T(n) = 2^i T(n/2^i) + ibn$

Proof:

Basis: $T(n) = 2T(n/2) + bn$, true from the definition.

Induction: Assume $S(i) : T(n) = 2^i T(n/2^i) + ibn$

Consider the next substitution into $S(i)$,

$$\begin{aligned}T(n) &= 2^i T(n/2^i) + ibn \\ &= 2^i (2T((n/2^i)/2) + bn/2^i) + ibn \\ &= 2^{i+1} T(n/2^{i+1}) + (i+1)bn \\ &= S(i+1) \quad Q.E.D.\end{aligned}$$

Making $\log_2 n$ substitutions to $T(n)$, we end up with the base case: $T(n) = an + bn \log_2 n$ and so $T(n)$ is $O(n \log n)$.

Solving Recurrence Relations

Two methods:

1. Repeated substitution
2. Guessing and proving (Don't freak out!!)

Example of first type (Repeated substitution)

Try $T(1) = a, T(n) = T(n - 1) + g(n)$

We can show by repeated substitution, i times,

$$T(n) = T(n - i) + \sum_{j=0}^{i-1} g(n - j)$$

To get the basis case (where no more subs can be made, we must use $i = n - 1$. This gives:

$$T(n) = T(1) + \sum_{j=0}^{n-2} g(n - j)$$

For factorial, $g(n) = O(1) = b$

For recursive selection sort, $g(n) = O(n) = bn$

Try to derive the complexities of the two functions.

Merge sort:

$$\begin{aligned}T(n) &= 2T(n/2) + g(n) \\&= 4T(n/4) + 2g(n/2) + g(n) \\&= 8T(n/8) + 4g(n/4) + 2g(n/2) + g(n) \\&\vdots \\&= 2^i T(n/2^i) + \sum_{j=0}^{i-1} 2^j g(n/2^j)\end{aligned}$$

The base case when no more subs are to be made is reached after $\log_2 n$ substitutions.

$$\text{So, } T(n) = an + \sum_{j=0}^{\log_2 n - 1} 2^j g(n/2^j)$$

For merge sort, $g(n) = bn$ (recall)

So,

$$\begin{aligned}T(n) &= an + \sum_{j=0}^{\log_2 n - 1} bn \\&= an + bn \log_2 n \\&\text{is } O(n \log n) \quad \text{Q.E.D.}\end{aligned}$$

Example of second type (by guessing)

Try to prove the complexity of merge sort.

Guess that $T(n)$ is bounded by $cn \log n + d$

Now proceed along the following lines.

Prove the following by induction.

$S(n)$: For merge sort, $T(n) \leq f(n)$ where $f(n) = cn \log n + d$ where $n \geq 1$ is a power of 2.

Basis: $S(1)$ is true if $a \leq d$ (Recall that $T(1) = a$ and $T(n) = 2T(n/2) + bn$).

Now assume $S(i)$ for $1 \leq i < n$ and prove $S(n)$ as below:

We want to prove $T(n) \leq cn \log n + d$ for some c, d , that is there exists some value of c and d for which the above inequality is true; can we find them?

$$\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&\leq 2\left(c\frac{n}{2}\log\frac{n}{2} + d\right) + bn \\
&\leq cn(\log n - \log 2) + 2d + bn \\
&\leq cn \log n - cn + 2d + bn \\
&\leq cn \log n + d + n(b - c) + d
\end{aligned}$$

Can we find a b, c and d such that for $n \geq 1$, $n(b - c) + d \leq 0$?

If so, the above inequality will hold.

Since $n(b - c) + d \leq 0$, $n(b - c) \leq -d$. Also, since $n \geq 1$, this is only true if $b - c \leq -d$. Also, since $a \geq d$, assume $a = d$ then we have $b - c \leq -a$ or $a + b \geq c$. Letting $c = a + b$, and $d = a$, we have

$$T(n) \leq (a + b) \log n + a$$

Thus $T(n)$ is $O(n \log n)$

Some common recurrences and their running times:

| $T(n)$ | BIG-OH |
|---------------------------|-------------------|
| $T(n - 1) + bn^k$ | $O(n^{k+1})$ |
| $cT(n - 1) + bn^k, c > 1$ | $O(c^n)$ |
| $cT(n/d) + bn^k, c > d^k$ | $O(n^{\log_d c})$ |
| $cT(n/d) + bn^k, c < d^k$ | $O(n^k)$ |
| $cT(n/d) + bn^k, c = d^k$ | $O(n^k \log n)$ |

If time permits:

Try to guess and prove an upper bound for the following recurrence:

Basis: $G(1) = 3$

Induction: $G(n) = (2^{\binom{n}{2}} + 1)G(\frac{n}{2})$ for $n > 1$.

Hints:

1. First expand $G(n)$ by repeated substitution to get the dominant term in the expression.
2. Don't forget the constant term (d)

Combinatorics

Counting assignments – Colours to houses, etc. k^n rule
(Also called selections with replacement)

Example: How many bits do you need to uniquely address every byte in a memory bank of 100 MB?
($2^n = 100 * 1000 * 1024$).

Counting permutations – $\prod(n)$

Ordered selections without replacement: $\prod(n, m)$ = number of ways we can select m items from n such that order matters for the selected items only. ($\prod(n, m) = n(n - 1)(n - 2) \cdots (n - m + 1) = \frac{n!}{(n-m)!}$)

Computing logs of large factorials

Unordered selections (also called combinations)

$$\binom{n}{m} = \frac{\prod(n, m)}{\prod(m)}$$

Example: Number of poker hands (5 cards)

Important properties of $\binom{n}{m}$

- for $0 < m < n$, $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$
- $\binom{n}{m} = \binom{n}{n-m}$

Pascal's triangle:

$$\binom{n}{m} = P[n+1][m+1]$$

(m th entry in the n th row, numbering from 0)

Computing ratios of large factorials

Running times of functions to compute permutations and combinations. (Contrast with running times of functions to list all the permutations and combinations, Recursive and iterative versions.)

Why $\binom{n}{m}$ must yield an integer

Shape of the function $f(m) = \binom{k}{m}$

Binomial co-efficients: Consider $(x+y)^n$ for $n = 1, 2, \dots, i$

$$(x + y)^n = \sum_{m=0}^n n \binom{n}{m} x^m y^{n-m}$$

Consider special cases of above when $y = 0, 1$ or $x = y = 1$

Multinomials: Permutations with identical elements.

Distribution of objects to bins

Distributing distinguishable objects in k classes

DIJKSTRA'S SHORTEST-PATH ALGORITHM

This will be the last lecture from me this semester for this course. I will, however, see you again after the Easter break for the mid-semester test.

The plan of this lecture:

1. Attempt to find your own shortest path algorithm.
2. Description and discussion of Dijkstra's algorithm
3. Proof of the algorithm and why it works.
4. Silent admiration and appreciation of the beauty, elegance and simplicity of the algorithm.

Example graph:

$$A \rightarrow B = 15$$

$$B \rightarrow C = 12$$

$$A \rightarrow C = 20$$

$$B \rightarrow D = 28$$

$$D \rightarrow E = 24$$

$$E \rightarrow F = 11$$

$$C \rightarrow F = 13$$

Start off by assuming that all other nodes are at distance infinity from the source node and gradually adjust these distances as we go on.

The shortest distances are discovered from the source node to every other node, in the order of nearness, i.e. nearest nodes are decided before looking at farther nodes.

Some definitions:

- A node is called *settled* if the minimum distance to it from the source is known.
- The *shortest special path* (SSP) to an unsettled node, if it exists, is a path that travels only through settled nodes except at the last step.
- We maintain a value $dist(u)$ for every node u such that if the node gets settled, then $dist(u) =$ length of shortest path to u . If u is not settled, then $dist(u) =$ length of shortest special path to u .

HOW TO SETTLE A NODE:

Adjust the value of $dist(u)$ for all nodes u that remain unsettled and are affected by the settled node.

i.e. To settle node v , for each arc $v \rightarrow u$, if $dist(v) + len(v \rightarrow u) < dist(u)$, then $dist(u) := dist(v) + len(v \rightarrow u)$

At the start of the algorithm, no nodes are settled and there exists a special path of length zero from the source node to itself.

At each pass, settle one unsettled node with the least value of $dist(u)$ as described above.

The algorithm terminates when all nodes have been settled.

IMPORTANT POINT:

At the beginning of any pass over the nodes in the algorithm, the shortest of all special paths is **THE** shortest path to the node it goes to. Note that this is not the case with the other special paths (non-shortest special paths). That is why this is the node that is settled next.

To see why, let the shortest of all special paths go to node v and another special path, (not the shortest one) go to node u .

THERE IS A POSSIBILITY that there might be a path from v to u , such that going to v and thence to u is shorter than going to u directly. However, **THERE IS NO POSSIBILITY** that going to u and thence to v is shorter than going to v directly since going to u is itself longer than going to v . So there is **A RISK** in settling u before v , but there is **NO RISK** in settling v before u . Spend a few minutes on this point and understand this completely and you will feel absolutely comfortable with the inductive proof.

ALTERNATELY, PONDER:

Why should nodes be settled in the order of nearness? This is in fact a critical requirement for the algorithm. Consider the scenario when there exist special paths of length 10 and 20 from the source s to two nodes u and v . What guarantees that the algorithm will work: Settling u first or v ?

PROOF BY INDUCTION:

S(n): When there are k settled nodes,

- a For each settled node, u , $dist(u)$ gives the minimum distance from s to u and the shortest path $s \rightarrow u$ consists entirely of settled nodes.
- b For each unsettled node u , $dist(u)$ is the minimum distance of any special path from $s \rightarrow u$ or ∞ if no such path exists.

First prove that (a) holds after the $(k + 1)$ th node, v is settled: Proof is by contradiction: Assume that $dist(u)$ is not the shortest path to v . Then there must have existed some non-special path that is shorter than the special path since **by the Inductive Hypothesis (b)**, we know that when k nodes were settled, the shortest special path to v is the shortest of all special paths to v . But we can't have an unsettled w be on a non-special path to v because (see Ponder point above) w should have been settled before v . Therefore (a) holds.

Then prove that (b) also holds after v is settled: Consider an arbitrary unsettled node u after v is settled. Either there is no special path to u or there is one. If the former, then $dist(u) = \infty$ is indeed the shortest of special paths to it by definition. If the latter, then either v is part of the special path or not. If v is part of the special path, then it **must** be the penultimate node (Why?). In this case, the $len(s \rightarrow u) = dist(v) + len(v \rightarrow u)$. If v is not part of the special path, then the settling of v has no effect on the length of the path to u . Since the algorithm sets $dist(u)$ to the smaller of the old value of $dist(u)$ and $dist(v) + length(v \rightarrow u)$, if settling v changes the shortest path to u then it **will** reset $dist(u)$ accordingly. Q.E.D.

COMMON POINT OF CONFUSION:

Do not confuse “Shortest of all special paths to A NODE” with “Shortest of all special paths.” The former is used when we have more than one special path to a given node (typically upon settling a new node). We reset the `dist()` for that node to be the shortest of all special paths to it. The latter is the shortest of all special paths to any unsettled node. This points to the next node to be settled.

Final note:

Investigate the following web site:

<http://www.MapsOnUs.com>

Try planning the shortest route from Tom Bradley Intl Airport, LA to Wall street, NY. What about the fastest? Why are they different?