Design and Analysis of Algorithms

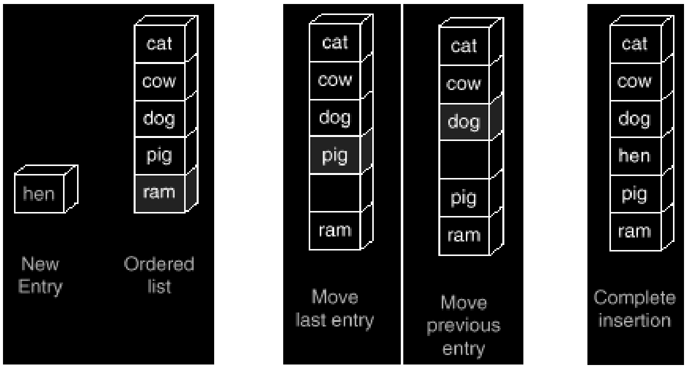
Lecture 7: Sorting

Sorting

- A little old estimate said that more than half the time on many commercial computers was spent in sorting.
- More than 25 different sorting algorithms
- Types of sorting:
  - External vs. Internal

Insertion Sort

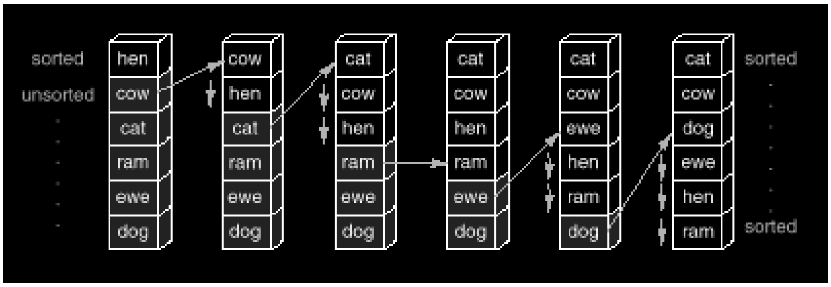
- Insertion in an Ordered List



Sorting by Insertion

- Maintain two lists, one sorted, another unsorted.
- Initially the sorted list has size zero, unsorted list has all the original keys.
- One by one insert the keys from unsorted list to the right position in the sorted list.

Sorting by Insertion (Example)



Insertion Sort (contiguous list)

```
void InsertionSort(List *list)
{
      Position fu; /*first unsorted entry position*/
      Position place; /*searches sorted part of list*/
      ListEntry current; /*holds entry temporarily*/
      for (fu = 1; fu < list->count; fu++)
        if(LT(list->entry[fu].key,list->entry[fu-1].key))
        {
            current = list->entry[fu];
            for (place = fu - 1; place >= 0; place--)
            {
                list->entry[place+1]=list->entry[place];
                if (place==0|| LE(list->entry[place-1].key, current.key))
                break;
             }
             list->entry[place] = current;
         }
}
```

Insertion Sort (linked list)

```
void InsertionSort(List *list)
{
    ListNode *fu; /* the first unsorted node to be inserted */
    ListNode *ls; /* the last sorted node (tail of sorted sublist) */
    ListNode *current, *trailing;
    if (list->head)
    {
        ls = list->head; /* An empty list is already sorted. */
        while (ls->next)
        {
            fu = ls->next; /* Remember first unsorted node. */
             if (LT(fu->entry.key, list->head->entry.key))
             {
                  ls->next = fu->next; fu->next = list->head; list->head = fu; /*Insert first unsorted at the head of sorted list.*/
             }
             else
             { /* Search the sorted sublist. */
                 trailing = list->head;
                 for (current = trailing->next; GT(fu->entry.key, current->entry.key);
                 current = current->next)
                 trailing = current;
                 /* First unsorted node now belongs between trailing and current. */
                 if (fu == current)
                         ls = fu;
                 else
                 {
                         ls->next = fu->next; fu->next = current; trailing->next = fu;
                  }
               }
          }
     }
}
```

Analysis

- i th entry requires anywhere between 0 to (i-1) iterations. On the average it requires
  - [0+1….+(i-1)]/(i-1)= i/2 iterations
- Each iteration has
  - 1 comparison
  - 1 assignment

Analysis

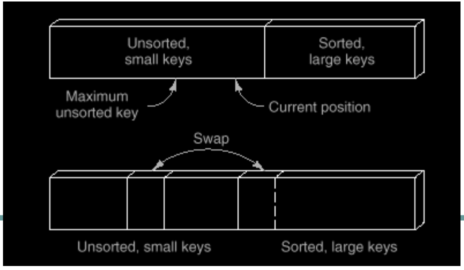
- i th entry requires anywhere between 0 to (i-1) iterations. On the average it requires
  - [0+1….+(i-1)]/(i-1)= i/2 iterations
- Each iteration has
  - 1 comparison
  - 1 assignment
- Outside the loop there are
  - 1 comparison
  - 2 assignments
- i iterates from 2 to n

Comments on Insertion Sort

- Insertion sort is an excellent method to check if a sorted list is still sorted.
- It is also good if a list is nearly in order.
- The main disadvantage of insertion sort is that there are too many moves, even on sorted keys, if just one key is out of place.
- A data which needs to travel at far away location needs to go through many steps.
- One data moves just one position in one iteration.

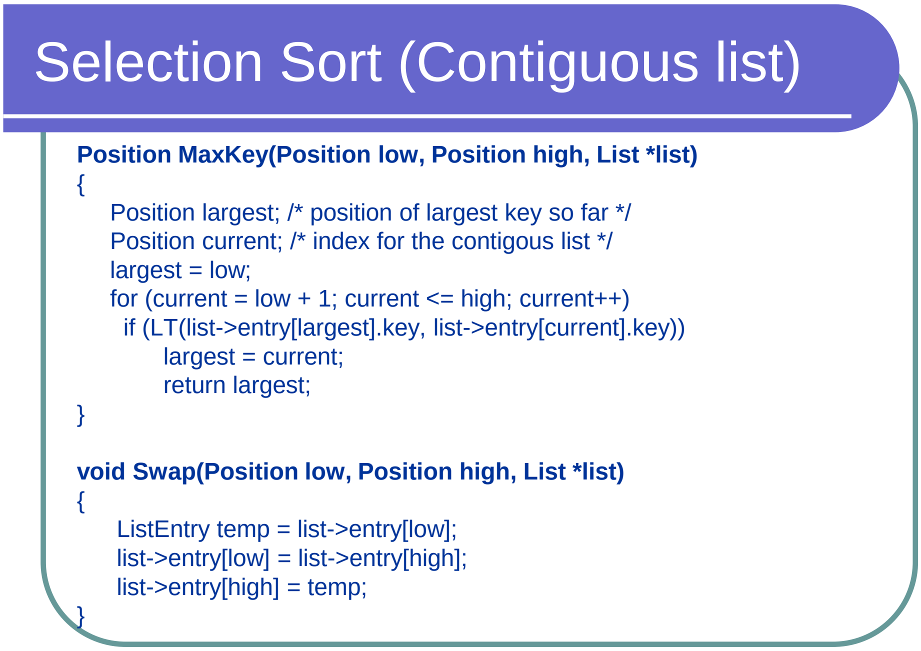
Selection Sort

- Selection sort one by one selects the max (or min) keys from the unsorted list and just appends them at the end of the sorted list.
- Consequently, there is no insertion cost



Selection Sort (Contiguous list)

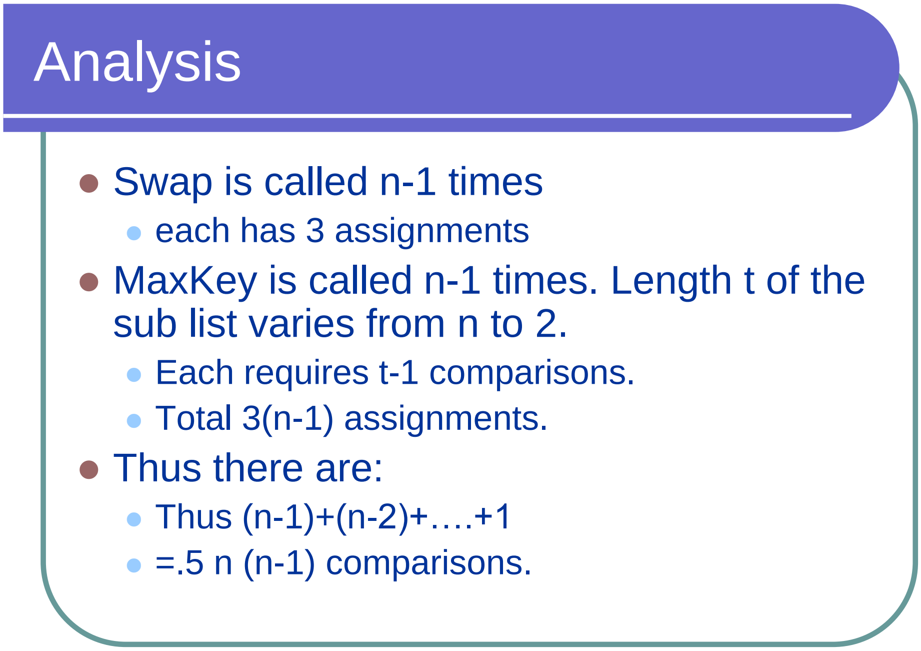
```
void SelectionSort(List *list)
{
     Position current; /*position of place being correctly
    filled*/
     Position max; /*position of largest remaining key */
     for (current = list->count - 1; current > 0; current--)
     {
           max = MaxKey(0, current, list);
           Swap(max, current, list);
      }
}
```



Selection Sort (Contiguous list)

```
Position MaxKey(Position low, Position high, List *list)
{
     Position largest; /* position of largest key so far */
     Position current; /* index for the contigous list */
     largest = low;
     for (current = low + 1; current <= high; current++)
       if (LT(list->entry[largest].key, list->entry[current].key))
             largest = current;
             return largest;
}

void Swap(Position low, Position high, List *list)
{
      ListEntry temp = list->entry[low];
      list->entry[low] = list->entry[high];
      list->entry[high] = temp;
}
```

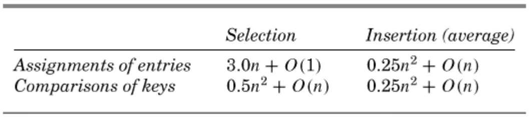


Analysis

- Swap is called n-1 times
  - each has 3 assignments
- MaxKey is called n-1 times. Length t of the sub list varies from n to 2.
  - Each requires t-1 comparisons.
  - Total 3(n-1) assignments.
- Thus there are:
  - Thus (n-1)+(n-2)+….+1
  - =.5 n (n-1) comparisons.

Comparison of Selection and Insertion Sort

- Quiz:
  - What is the best case for selection sort?
  - What is the worst case for selection sort?
  - Which method should we use
    - For large n?
    - If we know, the list is almost sorted?
    - Cost of assignment is large?

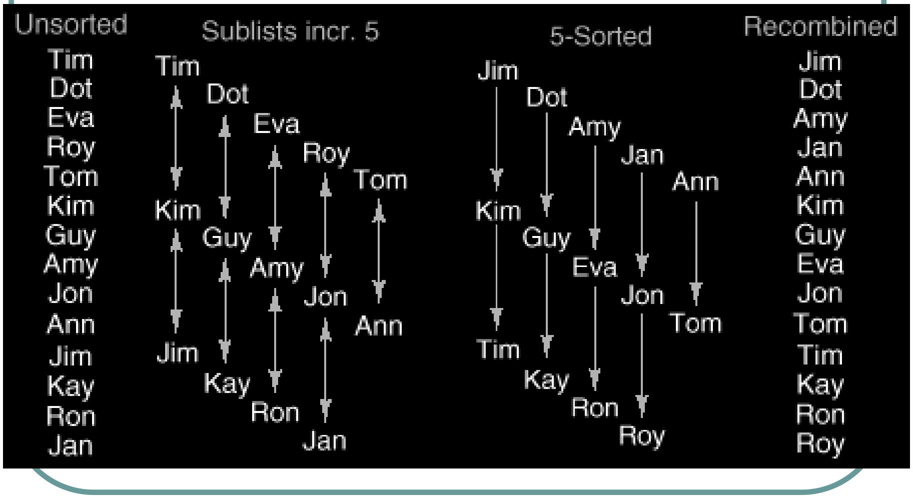


	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

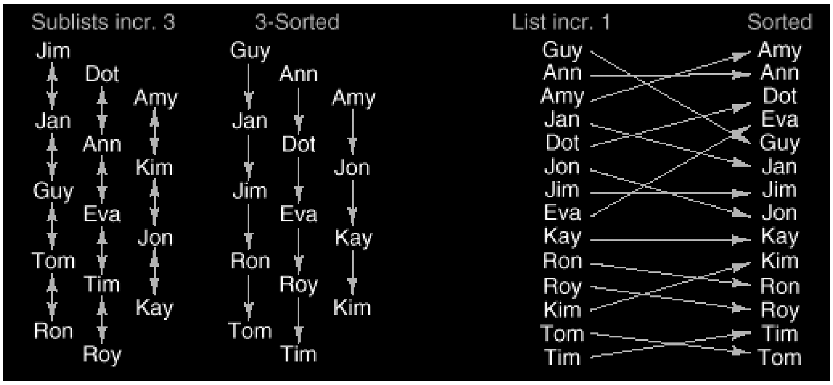
Shell Sort

- The problem with insertion sort is that, if a data needs to move much long distance it have to go through many iterations.
- Solution is Shell Sort!
- Invested by D.L. Shell in 1959.

Shell Sort – Idea (Step-1)



Shell Sort – Idea (Step-2)



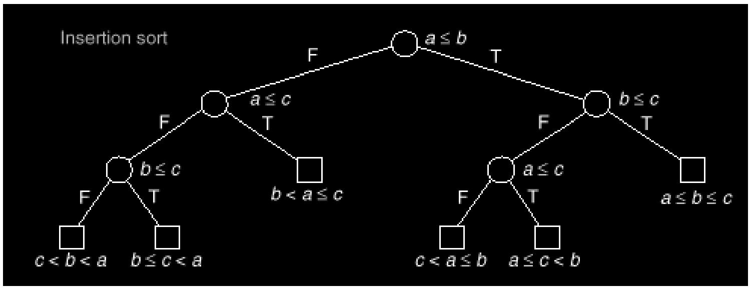
Shell Sort

- How to select the increments?
  - 5,3,1 worked. Many other choices will work also.
- However, no study so far could conclusively prove one choice is better that the other.
- Only requirement is that last round should be of increment 1 (that's an pure insertion sort).
- Probably it in not a good idea to use increments in power's of 2. Why?

- Analysis:
- exceedingly difficult
- for large n it appears the number of moves is in $n^{1.25}$ to $1.6n^{1.25.}$

- Lower Bounds of Sorting

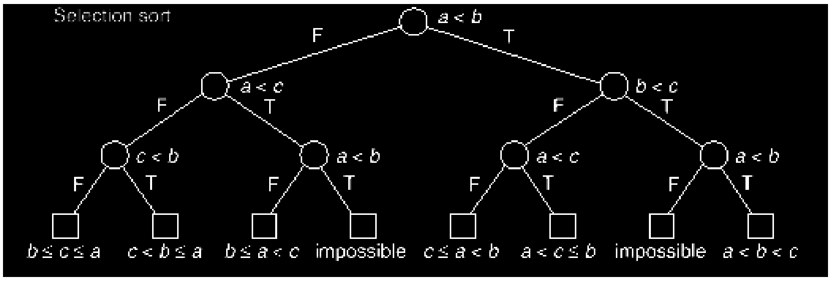
- Comparison Tree of Insertion Sort (a,b,c)



•The worst path is the worst case performance.

•The average path is the average performance.

Comparison Tree of Selection Sort (a,b,c)



•Selection sort tree is more bushy on the average.