Assignment 1

Saad Ahmed Salim
1712299042

24/11/2020

—

Design and Analysis of Algorithms
SECTION: 2

—

Dr. Sifat Momen

Question 1: Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \log_2 n$ steps. For which values of n does insertion sort beat merge sort?

Answer:

In order to beat merge sort, the time complexity of insertion sort should be less than merge sort.
So, inequality sets: $8n^2 < 64n \log_2 n$

Solving the inequality for n, the number of items must be $n \geq 2$ because if $n = 0$ or 1 it can't identify the inequality.

So, comparing times:

If $n = 2$; $8n^2 = 32$; $64n \log_2 n = 128$ (Condition: TRUE)
If $n = 3$; $8n^2 = 72$; $64n \log_2 n = 304.31$ (Condition: TRUE)
If $n = 5$; $8n^2 = 200$; $64n \log_2 n = 743.01$ (Condition: TRUE)
……………………………………………………………….
If $n = 25$; $8n^2 = 5000$; $64n \log_2 n = 7430.16$ (Condition: TRUE)
…………………………………………………………………
If $n = 35$; $8n^2 = 9800$; $64n \log_2 n = 11489.59$ (Condition: TRUE)
……………………………………………………………………
If $n = 42$; $8n^2 = 14112$; $64n \log_2 n = 14494.54$ (Condition: TRUE)
If $n = 43$; $8n^2 = 14792$; $64n \log_2 n = 14933.08$ (Condition: TRUE)

If $n = 44$; $8n^2 = 15488$; $64n \log_2 n = 15373.75$ (Condition: FALSE)

Now, it is clear that n is between 2 and 43 where insertion sort can beat merge sort as it is faster. But after $n = 43$, merge sort beats insertion sort. Answer is $n \leq 43$.

Question 2: What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Answer:

In order to find the smallest number of n, must be needed an inequality set.

So, the inequality set: $100n^2 \leq 2^n$

Solving the inequality for n, the number of items must be greater or equal to 1.

So, comparing times:

If $n = 1$; $100n^2 = 100$; $2^n = 2$ (Condition TRUE)
If $n = 2$; $100n^2 = 400$; $2^n = 4$ (Condition TRUE)
If $n = 3$; $100n^2 = 900$; $2^n = 8$ (Condition TRUE)
If $n = 4$; $100n^2 = 1600$; $2^n = 16$ (Condition TRUE)
If $n = 10$; $100n^2 = 10000$; $2^n = 1024$ (Condition TRUE)
If $n = 11$; $100n^2 = 12100$; $2^n = 2048$ (Condition TRUE)
If $n = 12$; $100n^2 = 14400$; $2^n = 4096$ (Condition TRUE)
If $n = 13$; $100n^2 = 16900$; $2^n = 8192$ (Condition TRUE)
If $n = 14$; $100n^2 = 19600$; $2^n = 16384$ (Condition TRUE)

If $n = 15$; $100n^2 = 22500$; $2^n = 32768$ (Condition FALSE)
If $n = 16$; $100n^2 = 25600$; $2^n = 65536$ (Condition FALSE)

Now, it is clear that $100n^2$ faster than 2^n after the $n = 15$ and 2^n is faster between 1 to 14. Answer is $n > 14$, 2^n algorithm runs faster. At $n = 15$, 2^n exceeds $100n^2$.

Question 3: Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non-decreasing order.

Answer:

In order to sort into non-increasing instead of non-decreasing order of insertion sort, we have to change the main condition.

while i > 0 and A[i] > key; this **A[i] > key** procedure shifts all the elements in the sorted sub-list in non-decreasing order. If **A[i] < key**, it will resist to form into non-decreased order and make our condition true which is sort into non-increasing order. I will show the procedure with comments below:

Insertion-Sort-Decreasing(A)

1. **for j = 2 to A.length**
2. **key = A[j]** // Store the value of A[j]
3. **i = j – 1** // Initialize i to search for the appropriate position
4. **while i > 0 and A[i] < key** // If it has a value that is less than the key does the following steps
   // as long as you do not find a value that is more than or equal to the current value, copy the smaller value that is found to be less than the current value; one step forward while keeping track of the position of the value that is not less than the current one, 0 if there is no item left.
5. **A[i+1] = A[i]** // Copy the smaller items one position forward
6. **i = i -1** // Point to the previous elements
7. **A[i+1] = key** // Finally, copy the key to the position.

Question 4: Rewrite the MERGE procedure such that it does not use sentinels, instead stopping once either array L or R has had all of its elements copied back to A and then copying the remainder of the other array back into A.

Answer:

Merge (A, p, q, r)

1. **N1 = q – p + 1** // get the sizes.
2. **N2 = r – q**
3. Let **L[1…N1] and R[1...N2]** // create new arrays L for left, L for right.
4. **for i = 1 to N1**
5. do **L[i] = A[p+i-1]**
6. **for j = 1 to N2**
7. do **R[j] = A[q+j]** // i, j to hold the current index of the two arrays.
8. **i = 1** // initialize the indexes
9. **j = 1**
10. **for k = p to r** // copy from L if i is within the range and either
11. **if i ≤ N1 and (j>N2 or L[i] ≤ R[j])** // the left has smaller value or the right index went out of bounds
12. **A[k] = L[i]**
13. **i = i + 1**
14. **else**
15. **A[k] = R[j]**
16. **j = j + 1**

In order to remove the remove the sentinels, I copy the elements from the left array and increment its index if the rights index has exceeded its range.