Greedy & Priority Queue

Julius January 24, 2024

1 Maximum Number of Non-overlapping Intervals

- **Input:** A set of n requests $\{1, \cdots, n\}$, where request i has a start time $s(i)$ and a finish time $f(i)$.
- **Output:** A largest compatible subset of these requests.

Try 1: Earliest Start Time $s(i)$ First Try 2: Shortest Interval $f(i) - s(i)$ First Try 3: Least Conflicts First

Earliest Finish Time $f(i)$ First:

Sort request in order of finish time and label in $f(i) \leq f(j), i < j$. O(nlgn)
Select requests in order of increasing f(i), always selecting the first request. Iterate through the intervals in this order until reaching the first interval j where $s(j) \geq f(i)$ and pick j. O(n)

```cpp
struct Range{
    int a, b;
    bool operator< (const Range &r)const{
        return b < r.b;
    }
} R[N];
int n;
int main(){
    scanf("%d", &n);
    for(int i = 0; i < n; i++) scanf("%d%d", &R[i].a, &R[i].b);
    sort(R, R+n); // sort the Interval by the end time
    int res = 0, ed = -2e9;
    for(int i = 0; i < n; i++){
        if(ed < R[i].a){ // new interval is compatible with the previous one
            ed = R[i].b;
            res++;
        }
    }
    cout << res << endl;
    return 0;
}
```

Proof of Correctness:

Proof A is a compatible set: Since we always delete all overlapping requests, before choosing the next request, we can never end up with overlapping requests in A. □

Proof A is an optimal set: Say $|A| = m$ and there is an optimal solution O of size m^*. Denote requests in A: $I_1, I_2, \cdots, I_m$, and in O: $I_1^*, I_2^*, \cdots, I_{m^*}^*$.
First, prove that $\forall i \geq 1, f(I_i) \leq f(I_i^*)$ by **Mathematical Induction**:
Base case: Because set A begin with the $min_i(f(i))$, so $f(I_1) \leq f(I_1^*)$.
Inductive Hypothesis: Suppose $\forall 1 \leq i \leq k, k \geq 1, f(I_i) \leq f(I_i^*)$.
Inductive Step: Because I_{k+1}^* is compatible with I_k^*, then it must be also compatible with I_i. While in our algorithm, when we try to pick the $(i+1)^{th}$ interval, there should be at least two choices I_{k+1}, I_{k+1}^*, we chose the one with smallest finish time, so $f(I_{k+1}) \leq f(I_{k+1}^*)$. Therefore, $\forall i \geq 1, f(I_i) \leq f(I_{i^*}^*)$.
Second, prove that $m = m^*$ by **Contradiction**:
Suppose, $m^* \geq (m+1)$, then $f(I_m) \leq f(I_m^*) \leq f(I_{m+1}^*)$, which means I_{m+1}^* is compatible with I_m, so our algorithm should have picked I_{m+1} after I_m. Contradiction. □

2 Scheduling to Minimize Lateness

- **Input:** A set of n requests $\{1, \cdots, n\}$, where request i^{th} has a start time duration $t(i)$ and a deadline $d(i)$.
- **Output:** A schedule that minimizes the maximum lateness, where $l(i) = max(f(i) - d(i), 0)$

Try 1: Smallest Duration $t(i)$ First Try 2: Smallest Slack First

Earliest Deadline $d(i)$ First:

Sort request in order of deadline and schedule them in this order without gaps. O(nlgn)

Proof of Correctness: we can transform the optimal solution into our greedy solution remains optimal.

There is an optional solution with no gaps: Remove the gaps in the optimal solution, the lateness will not increase and the new solution is still optimal. □

Jobs with identical deadlines can be scheduled in any order without affecting maximum lateness: The total time duration of these jobs is the same, and the deadline is identical, so the lateness will not increase after swapping. □

Inversion: Schedule A' has an inversion if $\exists d(i) < d(j)$ while i is scheduled after j.

There is an optimal schedule with no inversions: First, if there is an non-adjacent inversion (A, B), there must be an adjacent inversion (A', B') between (A, B). Then for these adjacent inversions, swap them will not negatively impact the maximum lateness for these two jobs. So we can swap all the adjacent inversions. □

All schedules with no inversions and no idle time have the same maximum lateness: If all deadlines are different, there is only one such schedule. If there are identical deadlines, then we can swap the jobs with identical deadlines without affecting the maximum lateness. □

From the above proofs, we can conclude that all solutions with no gaps and no inversions are optimal.

3 Covering Points with Minimum Number of Intervals

- **Input:** A set of n points $\{1, \cdots, n\}$, where point i^{th} has a coordinate $x(i)$.
- **Output:** A minimum number of intervals with length L that covers all points.

Try 1: Densest Area First Try 2: Most Points First

Start from left and cover the leftmost point:

Sort points in order of increasing coordinate. O(nlgn)
Set ed as the leftmost uncovered point and add new Interval start from ed.
While there are uncovered points:
    Check if this point is in ed + L, if yes, then label it as covered.
    If not, then set ed as this point and add new Interval start from ed.

```cpp
int P[N];
int n, L;
int main(){
    scanf("%d%d", &n, &L);
    for(int i = 0; i < n; i++) scanf("%d", &P[i]);
    sort(P, P+n); // sort the points by the coordinate
    int cnt = 1, ed = P[0];
    for(int i = 0; i < n; i++){
        if(P[i] > ed + L){
            ed = P[i];
            cnt++;
        }
    }
    cout << cnt << endl;
    return 0;
}
```

Proof of Correctness:

The intervals in A is always to the right of corresponding intervals in Optimal Solution O. The first interval in A is never to be the left of first interval in O. That's because our first interval is the right most interval to cover the first point.
Then, we can induct that the r^{th} interval in A is always to the right of the r^{th} interval in O:
Let's focus on the left end point P of the $(r+1)^{th}$ interval in A, which is not covered by the r^{th} interval in A, so it's not covered by the r^{th} interval in O. So the $(r+1)^{th}$ interval in O can't start after P, otherwise it will not cover P. So the $(r+1)^{th}$ interval in O is to the left of the $(r+1)^{th}$ interval in A. □

O can't have less intervals than A. Suppose there is only k-1 intervals in O, and k interval in A. Then the left end point P of k^{th} interval of A is not covered by the $(k-1)^{th}$ interval of A, and according to the above proof, the $(k-1)^{th}$ interval of O is to the left of $(k-1)^{th}$ interval of A. Point P is not covered by the $(k-1)^{th}$ interval of O, which means O is not a valid solution. □

4 Queuing for Earliest Finish Time

- **Input:** A set of n persons $\{1, \cdots, n\}$, with corresponding swimming, biking and running time $s(i), b(i), r(i)$.
- **Output:** A queuing order for these persons that minimizes the final time for all persons to finish. Biking and running can concurrent while swimming can't.

Try 1: Shortest Swimming Time $s(i)$ First Try 2: Shortest Total Time $s(i) + b(i) + r(i)$ First

Shortest Biking and Running time $b(i) + r(i)$ First:
Order the altheletes in decreasing order of $b(i) + r(i)$. O(nlgn)

Proof of Correctness:
Inversion: Althelete i with $b(i) + r(i) < b(j) + r(j)$ is scheduled before j.
If there is an inversion in one solution, there must be an adjacent inversion. Then for this adjacent inversion (i, j) with s(i) goes just before s(j) and $b(i) + r(i) < b(j) + r(j)$, we can swap them without negatively affecting the total time. So we can swap all the adjacent inversions.

5 Priority Queue

A priority queue perform these two operations fast:
Insert an element into the set and **Find the smallest or largest element in the set**.

Implementation	Insert	Find Min
Array	O(1)	O(n)
Sorted Array	O(n)	O(1)
Linked List	O(1)	O(n)
Sorted Linked List	O(n)	O(1)
Binary Tree	O(lgn)	O(1)

Full Binary Tree: A binary tree of depth k which has exactly $2^k - 1$ nodes.
Complete Binary Tree: A binary tree of n nodes and depth k with its nodes corresponding to the nodes, numbered 1 to n, in the full binary tree of depth k.
Binary Heap: a complete binary tree with the property that the value of the key at each node is at least as large as the key values at its children. (Max Heap)

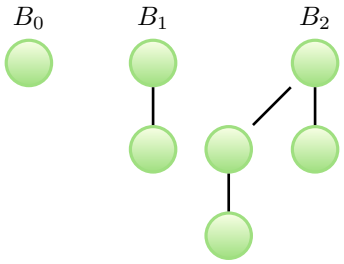
```
int heap[N], len, m;
void down(int x){
    int min = x; // for Min Heap
    if (x * 2 <= len && heap[min] > heap[x * 2]) min = x * 2;
    if (x * 2 + 1 <= len && heap[min] > heap[x * 2 + 1]) min = x * 2 + 1;
    int max = x; // for Max Heap
    if (x * 2 <= len && heap[max] < heap[x * 2]) max = x * 2;
    if (x * 2 + 1 <= len && heap[max] < heap[x * 2 + 1]) max = x * 2 + 1;
    if (min != x) { // if(max != x)
        swap(heap[min], heap[x]); // swap(heap[max], heap[x]);
        down(min); // down(max);
    }
}
int main(){
    scanf("%d%d", &len, &m); // Heap of size len and get the first m smallest elements
    for (int i = 1; i <= len; i++) scanf("%d", &heap[i]); // input the heap data from 1
    for (int i = len / 2; i; i--) down(i); // build the heap
    while(m--){
        cout << heap[1] << ' ';
        heap[1] = heap[len];
        len--;
        down(1);
    }
    return 0;
}
```

Building Heap O(n): Suppose we have n nodes and depth k. For n/2 nodes in depth k, they are all leaf nodes. For n/4 nodes in depth k-1, each needs at most 1 swap. For n/8 nodes in the k-2 depth, each needs at most 2 swaps, and so on. $T = 1 * (n/4) + 2 * (n/8) + 3 * (n/16) + \cdots + lgn * 1$, then divide both sides by 2, we get $T/2 = 1*(n/8)+2*(n/16)+3*(n/32)+\cdots+lgn/2$, then $T - T/2 = T/2 = (n/4)+(n/8)+(n/16)+\cdots+1-lgn/2 = (n - lgn)/2$, so $T = n - lgn$.

Top K elements in an array

Input: An unsorted array of length n.
Output: The k largest elements in the array (k < n).
Constraints: Can't use additional memory and run in O(nlgk) time.
Build a Min-Heap of size k, then iterate through the rest n-k elements in array, if the element is larger than the root of the heap, then swap them and down the root. $O(k) + O((n-k)lgk) = O(nlgk)$

Binomial Tree: an ordered tree defined recursively. B_0 consists of one node. B_k consists of two binomial tree B_{k-1} that are linked together such that root of one is the left most child of the root of the other.



Binomial Heap: a collection of binomial trees that satisfies the following properties:
1. Each binomial tree in the Heap obeys the min-heap property.
2. For any non-negative integer k, there is at most one binomial tree in the Heap whose root has degree k.
We only need lg(n) bits to represent a number n, so we need at most lg(n) trees to represent the binomial heap of size n. For Insert, we can insert a new node as a binomial tree of size 1, then merge it with the other binomial tree in the heap in O(lg(n)). The same time complexity as Delete, Find-Min, Decrease-Key, Merge. But the **Binomial Heap is the base for Fibonacci Heap**, which procastinate and doesn't keep nodes tidy in some operations. Finally, with better amortized time complexity.