# *Greedy Algorithms and Greedy Problems*

*August 28, 2018*

## *Several Fundamental Problems*

### *Interval Scheduling*

We have a set of requests numbered $1 \leq i \leq n$, each starting at $S_i$ and finishing at $F_i$; we want to schedule a non-overlapping number the maximum number of them. Assuming that all requests are sorted in some order, we proceed from left to right, looking at one request at a time. The greedy strategy here is to accept a request that finishes first, i.e., sorting by $F_i$ for all $i$. For the $O(n \log n)$ implementation, we implement as a while loop scanning the intervals from left to right, recording an optimal $L$ and $R$ which stores the current interval to be added to the set of scheduled intervals.

```
function schedule(A: set of intervals; F(*): Finish time; S(*): Beginning time):
  sort(A) (: By finishing time :)
  let OS, OF := new sets
  let p := 0, pA := 0, lF := 0
  while (pA < S.length) {
    if (S(A[pA]) > lF) {
      OS[p] := S(A[pA]
      OF[p] := F(A[pA]
      p += 1
    } else pA += 1
  }
```

The algorithm runs in time $O(n \log n + n)$. We proceed to prove its correctness. Let $O$ be the optimal set of intervals selected. Let $S$ be the set of solutions. We prove that $|S| = |O|$ . Let $i_1 i_2 ... i_k$ denote the sequence in which requests are added to $S$. Let $j_1 j_2 ... j_m$ denote the sequence in which requests are added to $O$. Note $k = |S|$ and $m = |O|$. It is trivial to first state that all intervals within $S$ don't overlap, as for $i_r$ and $i_x$, $x \geq r$, $S(i_x) \geq F(i_r)$.

**Lemma 1.** *For all $r \leq k$, $F(i_r) \leq F(j_r)$.*

*Proof.* Proceed by induction. For $r = 1$, since $A$ is sorted by interval ends, $F(i_r)$ is minimal for all $1 \leq r \leq |A|$ and selected by the algorithm. For $r > 1$, by induction hypothesis, $F(i_{r-1}) \leq F(j_{r-1})$. Since $F(j_{r-1}) \leq S(j_r)$, $F(i_{r-1}) \leq S(j_r)$; this means that $j_r \in A$ for our algorithm after $r - 1$. Since the greedy algorithm selects the interval with the smallest finish time at the $r$th step, $F(i_r) \leq F(j_r)$. □

**Theorem 2.** *The algorithm* `schedule` *is optimal.*

*Proof.* We prove by contradiction. Assume $m \geq k$. Then, when $r = k$, $F(i_r) \leq F(j_r)$, and since $k \leq m$ an extra request $j_{k+1} \in O$. Since intervals do not overlap, $F(j_k) \leq S(j_{k+1})$ and by Lemma 1, $F(i_k) \leq S(j_{k+1})$ and hence $j_{k+1} \in A$ at the termination of step $k$. Since the condition of our algorithm requires visiting all elements in $A$, we have a contradiction. □

*Scheduling to minimize lateness*

We have a set of $n$ jobs denoted by 2-tuples $(t_i, d_i)$, $1 \leq i \leq n$. $t_i$ is a nonnegative number that gives us the runtime of the task; $d_i$ specifies the deadline that the task must be finished. We introduce two functions, $s(i)$ that denotes the starting time of task $i$, and $f(i)$ that denotes the finishing time of task $i$. Note that by definition $f(i) = s(i) + t_i$. The problem asks us to minimize $\max\{f(i) - d_i\}$ for all $i$. The optimal strategy here is called Earliest Deadline First, which schedules sorted by $d_i$, from earlist to furthest. An algorithm is presented as follows:

```
function edf(N: num of tasks, s: start time,
T: set of runtimes, D: set of deadlines,
S: set of start times, F: set of ending times):


  sort D and exchange members of T using order of D; ordered from smallest to largest
  var f := s, maxD = -infinity
  for(i = 0; i < N; i += 1) {
    S[i] := f
    F[i] := S[i] + T[i]
    maxD := max(maxD, F[i] - D[i])
    f := F[i]
  }
  return maxD
```

We proceed to prove the optimality of Earliest Deadline First. We define *idle time* as follows:

**Definition 3.** Let the set of scheduled jobs $S := \{(s(i), f(i)) | 1 \leq i \leq N\}$.The set of idle times is $I := \{(f(i), s(i+1)) | 1 \leq i < N \wedge s(i+1) - f(i) \geq 0\}$.

Observe that there exists an optimal solution whose solution set $S$ produces an empty set $I$; in other words, there exists an optimal solution that leaves no idle time.

**Lemma 4.** *There exists an optimal solution $S$ that produces $I = \emptyset$.*

*Proof.* (Sketch) Assume that there exists a set $S'$ with a nonempty set of idle times $I'$ and $s'(i)$, $f'(i)$ denoting the starting and finishing time of the $i$th job in $S'$. Proceeding in the order from 1...N, For each $\{(s'(i), f'(i))|s'(i+1) - f'(i) > 0, 1 \leq i < N\}$ we modify $s'(i+1) := f'(i)$ and $f'(i+1) = s'(i) + t_i$. Then $I' = \emptyset$ and since each modification only moves $s'(i)$ forwards, the newly modified $f'(i)$ will always be less than or equal to the previous $f'(i)$ and hence $\max_{1 \leq i \leq N} f(i) - d_i$ will also be less than or equal to the previous one. □

We introduce an extra definition that helps with the proof:

**Definition 5.** An inversion is a pair of jobs $(i, j)$ with $i < j$ and $d_j > d_i$.

Observe that by definition, our algorithm produces no inversions. We proceed to prove the following lemma:

**Lemma 6.** *There exists an optimal solution $S$ with no idle times and no inversions.*

*Proof.* First, $S$ with no idle times exists by lemma 4. We proceed to prove that $S$ can contain no inversions. Assume that $S$ contains some inversion $(i, j)$. Then after swapping $(i, j)$ $S$ contains one less inversion. By continuing this swapping process we may produce a set $\bar{S}$ with no inversions. Let an interval $I = [s(i), f(j)]$. Then by lemma 4 since we have no idle time, $|I| = \sum_{i \leq k \leq j} t_k$. Note that swapping $i, j$ does not affect $|I|$, as we only change $\bar{s}(j) = s(i)$, $\bar{f}(j) = s(i) + t_j$ and $\bar{s}(i) = s(i) + t_j + \sum_{i < k < j} t_k$, $\bar{f}(i) = \bar{s}(i) + t_i$. Now let $L = \max_{1 \leq k \leq n} f(k) - d_k$, and $\bar{L} = \max_{1 \leq k \leq n} \bar{f}(k) - d_k$. If $L$ lies outside of $[i, j]$, we are done, since $|I|$ is not changed by the swap. If $L$ lies inside $[i, j]$ but is not $j$, we are also done; for if the maximum lateness occurs at $j$, since $i < j$, the swap produces $\bar{L} \leq L$; if the maximum lateness occurs at $(i, j)$ □

*Optimal Caching*

*Single-Source Shortest Path*

*Kruskal's MST*

*Clustering*

*Huffman Codes*

*Min-Cost Arborescenes*

## *Several Additional Problems*

### *Point-Interval Problems*

**Example 7 (UVa11134 Fabled Rooks).** By method of reduction we reduce this problem to two 1-dimentional interval problems. Given $n$ intervals $[i, j] \in I$ we want to assign a unique point to each interval within $[0, n]$. An obvious strategy here is to sort the intervals and start taking by left endpoint.

Consider $a$ =[1,3], $b$ =[1,4], $c$ =[2,2]. Then we would assign 1 to $a$, 2 to $b$, and run out of points at $c$. However, sorting by left endpoints don't always work. Another more sophisticated strategy might be sorting by area (right endpoint - left endpoint); this avoids the pitfalls induced by sorting by left endpoint. However, this doesn't always work either. Consider a long interval $[m, k]$; without loss of generality, let $k - m \geq 2$; further, let $[m, k-1]$ be covered by $k - m$ intervals of length 1. Without loss of generality, let $n - k \geq 2$; let an additional interval of length 2 cover $[k, k+1]$. Then if we sort by area we would consider the $k - m$ intervals of length 1 first (as they are the smallest); since they are of length 1 there is only one assignment for each. Then we consider the interval of length 2, and since $k$ is not taken yet we greedily take $k$, which makes $[m, k]$ fully assigned and fails this greedy strategy.

The only remaining greedy strategy that's obvious is **to sort by right endpoints**. We can immediately see that this works for the two counterexamples above, and it actually works. The intuition is that by ordering our right endpoints we always leave out the rightmost spaces of the intervals consecutively; this is useful when dealing with overlapping ones.

Code:

```
struct entry { unsigned first; unsigned second; unsigned id; };
inline static void void entrysort(vector<entry>& V) {
```

```
  sort(V.begin(), V.end(), [](const entry& a, const entry& b) {
    return a.second < b.second; });
}
bool search(unsigned N, vector<entry>& V, unsigned* SV, unsigned *S) {
  for(unsigned i = 0; i < V.size(); ++i) {
    unsigned& vL = V[i].first;
    unsigned& vR = V[i].second;
    unsigned& id = V[i].id;
    unsigned j;
    for(j = vL; j <= vR; ++j) if (SV[j] == 0) { SV[j] = id; break; }
    if (j > N || j > vR || SV[j] != id)  return false;
    S[id] = j;
  }
  return true;
}
int main() {
  unsigned N;
  while (cin>>N) {
    if (N==0) break;
    vector<entry> X, Y;
    unsigned SX[5001], SY[5001], SL[5001], SR[5001];
    memset(SX, 0, 5001*sizeof(unsigned)); memset(SY, 0, 5001*sizeof(unsigned));
    memset(SL, 0, 5001*sizeof(unsigned)); memset(SR, 0, 5001*sizeof(unsigned));
    for(unsigned i = 0; i < N; ++i) {
      entry xp, yp;
      cin >> xp.first >> yp.first >> xp.second >> yp.second; // xl,yl,xr,yr
      xp.id = yp.id = i+1;
      X.push_back(xp); Y.push_back(yp);
    }
    entrysort(X); entrysort(Y);
    if (!search(N, X, SX, SL) || !search(N,Y,SY,SR)) {
      printf("IMPOSSIBLE \n");
      goto impj;
    }
    for(unsigned i = 1; i <= N; ++i) {
      // IDs are 1-indexed
      cout << SL[i] << " " << SR[i] << endl;
    } impj:; // impossible; skip
  }
  return 0;
```

```
}
```

*Note* 8. Sorting by right endpoint works for a series of interval problems.

*Knapsack Problems*

There are two types of knapsack problems: knapsack problems and knapsack-like problems. The first type of knapsack problems are traditional; the second type of problems only require us to stuff stuffs into a knapsack.

**Example 9** (UVa1149)**.** We are to stuff stuffs into tubes that can stuff two in a box, in order to reach some amount of value. This is also an optimization problem. Code:

```
unsigned N; unsigned K; unsigned L; unsigned A[100001];
void solve() {
  bool assigned = 0;
  unsigned nAssigned = 0, l=0,r=K-1;
  sort(A, A+K,[](const unsigned& a, const unsigned& b){return a<b;});
  while (l<r) {
    if (A[l] + A[r] <= L) {
    // can fit two in a box
    ++nAssigned; --r,++l;
  } else {
    ++nAssigned;
    --r;
  }
  if (l==r) { ++nAssigned; }
  printf("%d\n", nAssigned);
}
int main() {
  cin >> N;
  for(unsigned i = 0; i < N; ++i) {
    cin >> K;
    cin >> L;
    for(unsigned i = 0; i < K; ++i) {
      cin >> A[i];
    }
    solve();
    if (i!=N-1) printf("\n");
```

```
  }
  return 0;
}
```

*Stacking Problems*

## *Techniques*

### *Choosing the optimal subproblem*

Subproblems in greedy algorithms can also be called finding the right objective function. A great example is 7. Another series differently flavored problems focus more heavily on choosing and writing out a good objective function. These types of problems are usually optimization problems. The following is a contest example; note that besides choosing the optimal strategy, the process of coding up such a problem is also nontrivial; there are many more nuances (edge cases) of the problem to consider.

**Example 10** (Uva10440 Ferry Loading II)**.** We are to schedule cars that come during a day to ferry them across a river. We shall think about the $m + 1$ case at first to reach the conclusion that we are to ferry the remainder of cars at the first run in order to leave out maximal waiting time for the last car. Code:

```
int main() {
  int c;
  cin >> c;
  for(int i = 0; i < c; ++i) {

    int t, m, n, C[1500], total_time = 0, total_rounds = 0;
    cin >> n >> t >> m;
    for(int j = 0; j < m; ++j) cin >> C[j];
    if (m>n) {
      int take = m % n;
      if (take==0) {
        take = n - 1;
        total_time = C[take];
        while (take < m) {
          take += n;
          total_time = max(C[take], total_time+2*t);
          total_rounds++;
        }
        if (C[take-n] >= total_time - 2*t) total_time = C[m-1]+t;
        else total_time -= t;
```

```
      } else {
        --take;
        total_time = C[take];
        while (take < m) {

          total_time = max(C[take], total_time+2*t);
          total_rounds++;
          take += n;
        }
        if (C[take-n] >= total_time - 2*t) total_time = C[m-1] + t;
        else total_time -= t;
      }
    } else {
      total_rounds = 0;
      total_time = C[n-1] + t;
    }
    cout << total_time << " " << total_rounds << endl;
  }
  return 0;
}
```

*Left-Right Scan*

**Example 11** (Uva11054 Wine Trading in Gergovia)**.** We are to give the minimum cost for transporting wines alone an 1D segment. Envision a wine tanker that goes unidirectionally from left to right that is also able to take on negative costs. We simulate that tanker using a loop and sweep from left to right. The minimum cost is independent of the optimal strategy; hence the tanker gives the minimum cost. Code:

```
typedef long long ll;
int main() {
  int N;
  while (cin>>N) {
    if (N<2||N>100000) break;
    ll transfer = 0, accum = 0, village = 0;
    for(int i = 0; i < N; ++i) {
      cin >> village;
      transfer += abs(accum);
      accum += village;
    }
    cout << transfer << endl;
```

```
  }
  return 0;
}
```

*Optimal Construction*

Optimal construction is a method introduced in Rujia Liu's book [cite]. It is similar to brute-force, except (again), only explores the branch that produces the optimal solutions. At each step, the optimal construction method chooses to directly construct the solution. The method is best taught by examples.

**Example 12** (**UVa10340 All in All**)**.** We would like to see if a string is another string's noncontiguous substring. We simply do a mergesort-type scan through the two strings using two indices $i, j$, and see if at each point, the contents of $i, j$ matches each other. Note that the size of strings is not provided in this problem; it is common for UVa problems to not provide a clue for the problem size. Problems of this type are usually small and indicate the solution is obvious. Code:

```
int main() { string s, t;
  while (cin >> s >> t) {
    if(s.size()==0&&t.size()==0) break;
    unsigned i=0,j=0;
    while(i<s.size()&&j<t.size()) {
      if (t[j]==s[i]) ++i; ++j;
    }
    if (i==s.size()) printf("Yes\n");
    else printf("No\n");
  }
  return 0;
}
```

Onto a more nuanced example. The following example may be difficult to think up, or even difficult to code up. The trick lies in **constructing all solutions**. As usual, when encountering the construction-type problems recursion simplifies the structure a lot. We may therefore attempt to use recursion to simplify the amount of code and logic needed.

**Example 13** (UVa 1610)**.**

```
string out; unsigned outSize=0xffff; bool fin = 0;
void generate(string& x, string& L, string& R) {
```

```
  if (fin) return;
  if (x.size() <= L.size() && x >= L && x < R && x.size() < outSize) {
    cout << x << endl;
    outSize = x.size();
    fin=1;
    return;
  }
  if (x.size() > L.size() || x >= R) return;
  if (x < R) {
    if (x[x.size()-1]<'Z') {
      ++x[x.size()-1];
      generate(x,L,R);
      --x[x.size()-1];
    }
    x += 'A';
    generate(x,L,R);
    x.pop_back();
  }
}
int main() { unsigned d;
  while (scanf("%u", &d)) {
    vector<string> names;
    if (d==0)return 0;
    for(unsigned i = 0; i < d; ++i) {
      string x; cin >> x;
      names.push_back(x);
    }
    sort(names.begin(), names.end(), [](const string& a, const string& b) {
      unsigned i=0,j=0;
      while (a[i]==b[j]) {
        if(i==a.size()) return 1;
        if(j==b.size()) return 0;
        ++i,++j;
      }
      return (int)(a[i]<b[j]);
    }); // lexicographic sorting
    string L = names[(names.size()-1)/2];
    string R = names[(names.size()-1)/2+1];
    string x="A";
    generate(x,L,R);
    fin=0; outSize=0xffff; out="";
```

```
  }
  return 0;
}
```

A yet more obscure example requires us to use hash tables. We can fully take advantage of C++11's `unordered_map`, which is a generic hash table that is of reasonably good performance.

**Example 14** (UVa1152 Sum of Four)**.** We are to figure out the number of $a, b, c, d$ s (with $a \in A, b \in B, c \in C, d \in D$) that sum to zero. If we attempt to do dynamic programming, we immediately find out that there is only a pseudopolynomial-time algorithm available. We resort to a clever use of hashing, hashing $a + b$ and $-(c + d)$ to reduce an $n^4$-time lookup to $\sim n^2$-time. Again, the $n^2$-time lookup is highly dependant on the performance of the integer hashing function. For our purposes the C++ STL hash function seems good enough. Code:

```
int *A, *B, *C, *D;
unsigned S=0;
int main() { int N, k;
  cin >> N;
  for(int i = 0; i < N; ++i) {
    S=0;
    unordered_map<int, unsigned> M;
    scanf("%d",&k);
    A = new int[k];B = new int[k];C = new int[k];D = new int[k];
    for(unsigned i = 0; i < k; ++i) {

      cin >> A[i];cin >> B[i];cin >> C[i];cin >> D[i];
    }
  }
  for(unsigned i = 0; i < k; ++i)
    for(unsigned j = 0; j < k; ++j) {
      if (M.find(A[i]+B[j]) != M.end()) {
        M[A[i]+B[j]] += 1;
      } else M[A[i]+B[j]] = 1;
    }
  for(unsigned i = 0; i < k; ++i)
    for(unsigned j = 0; j < k; ++j) {
      if (M.find(-(C[i]+D[j])) != M.end()) {
        S += M[-(C[i]+D[j])];
      }
```

```
    }
  cout << S << endl;
  delete[] A; delete[] B; delete[] C; delete[] D;
  if (i!=N-1) puts("");
  return 0;
}
```

### *Where Does Greed Suffice?*

Greedy algorithm has two properties it must satisfy: 1. Optimal substructures (Recall dynamic programming) 2. A greedy property (that the objective function is indeed optimal). **We shall view greedy algorithm as a special kind of dynamic programming; one which does not explore all possibilities of the recursion tree, but only a branch of all possibilities that does indeed contain all optimal solutions.** Hence, the recurrence for greedy algorithms runs in polynomial time without memoization. By contrast, the recurrence for dynamic programming must run in exponential time without memoization because of the need to explore all solutions.